WinG Programmer's Reference

Philosophy and Overview of WinG

An introduction to the philosophy and design of WinG: what it is and why it was created.

Why WinG?

A WinG Glossary

Click Here to read the WinG readme.txt release notes.

Programming With WinG

Documentation of the WinG APIs and helpful articles about how best to use them.

WinG API Documentation

Off-screen Drawing With WinG

Mixing GDI and WinG

Halftoning With WinG

Maximizing Performance with WinG

DIB Orientation

Managing Palettes in a WinG Application

Compiling WinG Applications

Debugging WinG Applications

DISPDIB and WinG

Further Reading

The WinG Development Kit

Information on shipping a WinG application and discussion of the sample code included in the WinG Development Kit.

Shipping a Product With WinG

WinG SDK Code Samples

Managing Palettes in a WinG Application

DOS executes only one application at a time, so every DOS application has the freedom to modify the hardware palette on palette devices (such as VGA) without interfering with other active applications. In the multitasking Windows environment, applications must modify the palette through the Palette Manager to ensure friendly interoperability of applications.

The most common programming error in palettized Windows applications stems from the system selecting the stock system palette into devices acquired through GetDC or through a WM_PAINT message. The application must call SelectPalette and RealizePalette to set and activate a palette in these DCs, and the palette information survives in the DC only in the time between GetDC and ReleaseDC calls.

The following code snippet will result in an image mapped to the static colors:

```
HDC hdc = GetDC(hwnd);
SelectPalette(hdc, hpal, FALSE);
RealizePalette(hdc);
ReleaseDC(hwnd, hdc);                  // Palette information lost!!
...
hdc = GetDC(hwnd); // DC with stock palette selected!!
BitBlt(hdc, x,y,width,height, source, 0,0, SRCCOPY);
ReleaseDC(hwnd, hdc);
```

Ron Gery's article "The Palette Manager: How and Why" (Microsoft Technical Article, 23 March 1992) describes the mechanisms of the Palette Manager in depth. The following articles discuss optimizing Palette Manager use in WinG applications.

Using an Identity Palette

Palette Animation With WinG

Accessing a Full Palette Using SYSPAL_NOSTATIC

Why WinG?

Although business applications such as word processors and spreadsheets have moved almost exclusively to Windows, DOS remains the operating system of choice for high-volume, high-performance action games and innovative graphics applications for PCs. These applications have not made the transition to Windows because of restrictions placed on the programmer by GDI's device independence, by the windowed environment, and by the inability of general graphics libraries to provide the necessary speed.

The display techniques used by high-performance graphics applications have two common characteristics. First, the application hides the frame composition process by double-buffering in software or hardware. Second, programmers use knowledge specific to the problem at hand to optimize their graphics routines in ways a general graphics library can not.

Hiding frame composition eliminates flicker by presenting only completed frames to the user. Under DOS, a VGA card can accomplish the display in hardware by page flipping, or the buffer can reside in main memory which is copied to the screen. Some applications further optimize display access by copying only the areas of the buffer that have changed since the last frame, a process called dirty rectangle animation. Today, high-performance DOS games use all of these techniques.

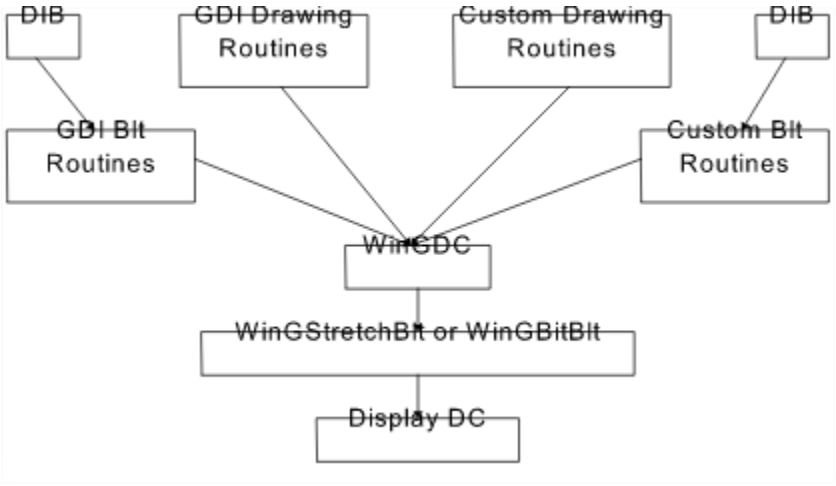
In short, most DOS games programmers use knowledge specific to their application and their hardware to write optimized graphics routines. Until now, Windows programmers could not use such methods because GDI prevents access to device-specific surfaces; programmers can not draw directly onto the surface of a GDI device context.

WinG (pronounced "Win Gee") is an optimized library designed to enable these high-performance graphics techniques under Windows 3.x, Win32s, Windows NT 3.5, Windows 95, and future Windows releases. WinG allows the programmer to create a GDI-compatible HBITMAP with a Device Independent Bitmap (DIB) as the drawing surface. Programmers can use GDI or their own code to draw onto this bitmap, then use WinG to transfer it quickly to the screen. WinG also provides halftoning APIs that use the standard Microsoft halftone palette to support simulation of true color on palette devices.

Off-screen Drawing With WinG

WinG introduces a new type of device context, the WinGDC, that can be used like any other device context. Unlike other DCs, programmers can retrieve a pointer directly to the WinGDC drawing surface, its BITMAPINFOHEADER, and its color table. They can also create and select new drawing surfaces for the WinGDC or modify the color table of an existing surface. DIBs become as easy to use as device-specific bitmaps and compatible DCs, and programmers can also draw into them using their own routines.

Most often, applications will use WinGCreateDC to create a single WinGDC and will use WinGCreateBitmap to create one or more WinGBitmaps into which they compose an image. The application will typically draw into this buffer using DIB copy operations, GDI calls, WinG calls, and custom drawing routines, as shown here.



A double-buffering architecture for WinG

Once drawing, DIB composition, and sprite composition for the current frame is complete, the application will copy the WinGDC buffer to the screen using WinGStretchBlt or WinGBitBlt. This double-buffering architecture minimizes flicker and provides smooth screen updates.

Many games and animation applications draw their characters using sprites. On arcade machines, sprite operations are performed in hardware. Under DOS with a VGA, games simulate sprite hardware using transparent blts into an off-screen buffer. The DOGGIE sample application (in the SAMPLES\DOGGIE directory of the WinG development kit) uses WinG in the same way to perform transparent blts to a WinGDC and includes source code for an 8-bit to 8-bit TransparentDIBits procedure.

Mixing GDI and WinG

There are two ways to use GDI and WinG together. You can mix GDI and custom drawing operations into WinGBitmaps, and you can mix GDI operations and WinG blts to the screen. Both have their caveats.

Drawing into WinGBitmaps

WinG allows drawing onto the DIB surface of a WinGDC with GDI, but there are some anomalies to keep in mind.

1. Most importantly, GDI does NOT regard WinGDCs as palette devices. WinGDCs are actually 256-color RGB devices. You can modify the device color table using the WinGSetDIBColorTable API. The Palette Animation With WinG article describes how to match a given palette to a WinGDC color table.

2. Drawing with GDI on a WinGDC surface does not always produce a pixel-perfect duplicate of the image you would see using GDI on a display device. The images will be acceptably similar, but some stray pixels will remain if you XOR the two images together.

3. Brushes realized in a WinGDC will be aligned to the upper left corner of the WinGDC whereas brushes used in screen DCs are aligned to the upper left corner of the screen. This means that when you blt a WinGDC that has been filled with a pattern into a screen DC that has been filled with the same pattern, the patterns will not necessarily align correctly.

If you have this problem, you can either change the brush origins and re-realize the brushes in either DC (see the section "1.6.8 Brush Alignment" in the Windows SDK Programmer's Reference Volume 1, also available on the Microsoft Developer Network CD) or you can make off-screen brushes align correctly with on-screen brushes by blting the WinGDC to a brush-aligned position on the screen. For example, an 8x8 brush pattern can be correctly aligned to the screen by blting the WinGDC to an x, y position when x and y are both multiples of 8.

4. BitBlt does not blt between DCs owned by different devices. You can't blt from a Printer DC to a Display DC, for example. A WinGDC is a new type of device context, and just as with other DCs, you can't use BitBlt to blt between a WinGDC and any other type of DC (such as a WinGDC to a Display DC or a Memory DC to a WinGDC). To blt from a WinGDC to a DisplayDC, use WinGBitBlt or WinGStretchBlt. Again, you can't use BitBlt to blt from a Display DC to a WinGDC on Windows 3.x or Win32s, and you can only use WinGBitBlt or WinGStretchBlt to blt from a WinGDC to the screen.

To blt from the screen into a WinGDC on Windows 3.x you will need to create a compatible bitmap, BitBlt the screen rectangle into this bitmap, then GetDIBits from the bitmap into the WinGDC's DIB surface. On Windows 95 and Windows NT 3.5, BitBlt into a WinGDC will work properly.

Win32 applications that use GDI and custom drawing routines to draw on the surface of a WinGBitmap should call GDIFlush after calling GDI functions and before calling custom functions. Win32 GDI batches drawing commands, including WinGBitBlt and WinGStretchBlt, and this will guarantee that all GDI drawing is completed before custom drawing begins. Without this call, drawing may be done in an incorrect order.

Drawing to the Screen

Win32 applications should call GDIFlush after blting to the screen from a WinGDC.

Halftoning With WinG

WinG allows applications to simulate true 24-bit color on 8-bit devices through the WinG halftoning support APIs, WinGCreateHalftonePalette and WinGCreateHalftoneBrush.

The halftone palette is an identity palette containing a carefully selected ramp of colors optimized for dithering true color images to 8-bit devices. The WinGCreateHalftonePalette function returns a handle to a halftone palette which applications can select and realize into a display device context to take advantage of the halftoning capabilities offered by WinG.

The brushes returned by the WinGCreateHalftoneBrush API use patterns of colors in the halftone palette to create areas of simulated true color on 8-bit devices into which the halftone palette has been selected and realized. The CUBE sample application (in the SAMPLES\CUBE subdirectory of the WinG development kit) uses halftoned brushes to generate a wide range of shaded colors on an 8-bit display.

The halftone palette gives applications a framework for dithering 24-bit images to 8-bit devices. The palette itself is a slightly modified 2.6-bit-per-primary RGB cube, giving 216 halftoned values. The 256-color halftone palette also contains the twenty static colors and a range of gray values.

Given a 24-bit RGB color with 8 bits per primitive, you can find the index of the nearest color in the halftone palette using the following formula:

```
HalftoneIndex = (Red / 51) + (Green / 51) * 6 + (Blue / 51) * 36;
HalftoneColorIndex = aWinGHalftoneTranslation [HalftoneIndex];
```

The `aWinGHalftoneTranslation` vector can be found in the HALFTONE source code. The HALFTONE sample (in the SAMPLES\HALFTONE subdirectory of the WinG development kit) applies an ordered 8x8 dither to a 24-bit image, converting it to an 8-bit DIB using the WinG Halftone Palette.

Applications should avoid depending on a specific ordering of the halftone palette by using PALETTERGB instead of PALETTEINDEX to refer to entries in the palette.

Maximizing Performance With WinG

Here is the WinG Programmer's Guide to Achieving WinG Nirvana, the Top Ten ways to maximize blt performance under Windows using WinG.

10. Take Out Your Monochrome Debugging Card and Unplug Network Connections

Eight bit monochrome video cards can turn the 16 bit 8 MHz ISA bus into an 8 bit 4 MHz PC bus, cutting your video bus bandwidth by up to 75%. Monochrome cards are an invaluable aid when debugging graphical applications, but when timing or running final tests, remove the card for maximum speed.

Incoming network packets during WinG runtime profiling cause asynchronous interrupts that steal CPU time from the time-sensitive profiling process. This may result in noticeable timing errors that lead to sub-optimal performance. For best results, unplug any network connections before running a WinG application for the first time on a new configuration, then plug the network connection back in when the WinG profiling process is complete.

(NOTE: Microsoft is not responsible for damage caused by incorrectly handling hardware).

9. Store WinGBitmap Surface Pointer and BITMAPINFO

WinGCreateBitmap takes a BITMAPINFO, creates an HBITMAP, and returns a pointer to the new bitmap surface. Store the BITMAPINFO and pointer at creation time with the HBITMAP rather than calling WinGGetDIBPointer when you need it.

8. Don't Make Redundant GDI Calls

GDI objects such as brushes, fonts, and pens, take time to create, select, and destroy. Save time by creating frequently used objects once and caching them until they are no longer needed. Move the creation and selection of objects as far out of your inner loops as possible.

7. Special Purpose Code May Be Faster Than GDI

There may be many ways to accomplish a given graphics operation using GDI or custom graphics code in your application. Special purpose code can be faster than the general purpose GDI code, but custom code often incurs associated development and testing overhead. Determine if GDI can accomplish the operation and if the performance is acceptable for your problem. Weigh the alternatives carefully and see number 6 below.

6. Time Everything, Assume Nothing

Software and its interactions with hardware are complex. Don't assume one technique is faster than another; time both. Within GDI, some APIs do more work than others, and there are sometimes multiple ways to do a single operation--not all techniques will be the same speed.

Remember the old software development adage: 90% of your time is spent executing 10% of the code. If you can find the 10% through profiling and optimize it, your application will be noticeably faster.

Timing results may depend on the runtime platform. An application's performance on your development machine may be significantly different from its performance on a different runtime machine. For absolute maximum speed, implement a variety of algorithms, time them at runtime or at installation, and choose code paths accordingly. If you choose to time at installation, remember that changes to video drivers and hardware configuration after your application has been installed can have a significant effect on runtime speed.

This tip extends even to WinG. WinGRecommendDibFormat will tell you the format of the fastest 1:1 identity unclipped blt to the screen, but if your application requires other blt types, such as a 1:2 stretch, time bottom-up and top-down cases separately and decide for yourself which to use.

5. Don't Stretch

Stretching a WinGBitmap requires more work than just blting it. If you must stretch, stretching by factors of 2 will be fastest.

On the other hand, if your application is pixel-bound (it spends more time writing pixels to the bitmap than it does blting), it may be faster to stretch a small WinGBitmap to a larger window than it is to fill and blt a WinGBitmap with the same dimensions as the window. Your application can respond to the WM_GETMINMAXINFO message to restrict the size of your window if you don't want to deal with this problem.

4. Don't Blt

"The fastest code is the code that isn't called." Blt the smallest area possible as seldom as possible. Of course, figuring out the smallest area to blt might take longer than just blting a larger area. For example, a dirty rectangle sprite system could use complex algorithms to calculate the absolute minimum rectangles to update, but it might spend more time doing this than just blting the union of the dirty areas. The runtime environment can affect which method is faster. Again, time it to make sure.

If you must blt, try to align the destination on the screen to DWORD (4-byte) boundaries. On an 8-bit display, this means ensuring that the X value of the upper-left corner of the destination rectangle is evenly divisible by 4. Applications can use the GetDCOrg function to calculate the destination alignment. Or, if you're blting to the entire client area of your window, make sure that the X value of the upper-left corner of the client area is DWORD aligned on the screen. It's fairly easy to make sure you align your blts, and it can noticeably increase their speed.

3. Don't Clip

Selecting GDI clip regions into the destination DC or placing windows (like floating tool bars) over the destination DC can slow the blt speed.

Clip regions may seem like a good way to reduce the number of pixels actually sent to the screen, but someone has to do the work. As number 4 and number 7 discuss above, you may be better off doing the work yourself rather than using GDI.

An easy way to test your application's performance when clipped is to start the CLOCK.EXE program supplied with Windows. Set it to Always On Top and move it over your client area.

2. Use an Identity Palette

WinGBitmaps without identity palettes require a color translation per pixel when blted. 'Nuff said.

See the Using an Identity Palette article for specific information about what identity palettes are, how they work, and how you can use them.

1. Use the Recommended DIB Format

WinG adapts itself to the hardware available at runtime to achieve optimum performance on every platform. Every hardware and software combination can be different, and the best way to guarantee the best blt performance is to use the DIB parameters returned by WinGRecommendDibFormat in calls to WinGCreateBitmap. If you do this, remember that your code must support both bottom-up and top-down DIB formats. See the DIB Orientation article for more information on handling these formats.

DIB Orientation

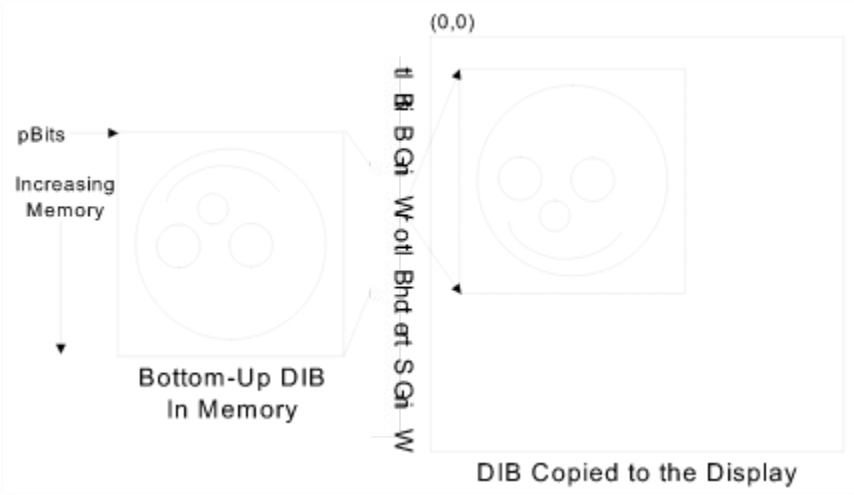
The most frustrating thing about working with DIBs is that DIBs are usually oriented with the bottommost scanline stored first in memory, the exact opposite of the usual device-dependent bitmap orientation. This standard type of Windows DIB is called a bottom-up DIB.

WinG hides the orientation of DIBs from an application unless the application wants to know. Drawing into a WinGDC using GDI functions and blting the WinGDC to the display using either of the WinG DIB copy commands (WinGStretchBlt or WinGBitBlt) results in an image that is almost identical to one created using GDI to draw directly onto a display DC. See the Using GDI With WinGDCs article for more information.

If you don't plan on writing custom drawing routines and will not be using existing Windows 3.1 DIB-to-screen functions (such as StretchDIBits or SetDIBitsToDevice), you can skip the rest of this section

If you do plan on writing custom drawing routines or just want to know how they work, this section will begin to alleviate the confusion. The Microsoft Technical Articles "DIBs and Their Use" by Ron Gery and "Animation in Windows" by Herman Rodent will flesh out the ideas presented here, provide helpful advice, and describe DIBs in depth. The DOGGIE sample in the WinG SDK and the TRIQ sample from Microsoft's GDI Technical Notes show how to draw into a memory DIB. TRIQ is listed in the section Further Reading.

Confusion with bottom-up DIBs inevitably stems from the fact that the bottommost scanline is stored first in memory, giving a coordinate space where (0, 0) is the lower left corner of the image. Windows uses (0, 0) as the upper left corner of the display and of device dependent bitmaps, meaning that the Y coordinates of bottom-up DIBs are inverted. In the diagram below, the smiling face casts its gaze towards the DIB origin, but when translated to the display with WinGStretchBlt or WinGBitBlt, it looks away from the display origin.



Bottom-Up DIBs are flipped when copied to the display

WinGStretchBlt, WinGBitBlt, StretchDIBits, and SetDIBitsToDevice invert the bottom-up DIB as they draw it to the screen.

For an 8-bit bottom-up DIB, the address in memory from which the screen pixel (X, Y) is retrieved can be found with these equations:

```
// Calculate actual bits used per scan line
DibWidthBits = (UINT)lpBmiHeader->biWidth *
    (UINT)lpBmiHeader->biBitCount);
// And align it to a 32 bit boundary
```

```
DibWidthBytes = ((DibWidthBits + 31) & (~31)) / 8;
pPixelXY = DibAddr + (DibHeight - 1 - Y) * DibWidthBytes + X
```

where DibAddr is the base address of the DIB, DibHeight is the height of the DIB, lpBmiHeader is a pointer to a BITMAPINFOHEADER describing the DIB, and DibWidthBytes is the DWORD-aligned offset of bytes in memory from any X in one scanline to any X in the next scanline. In DIBs, every scanline begins at a DWORD-aligned memory address, with extra bytes added as needed at the end of the previous scanline.

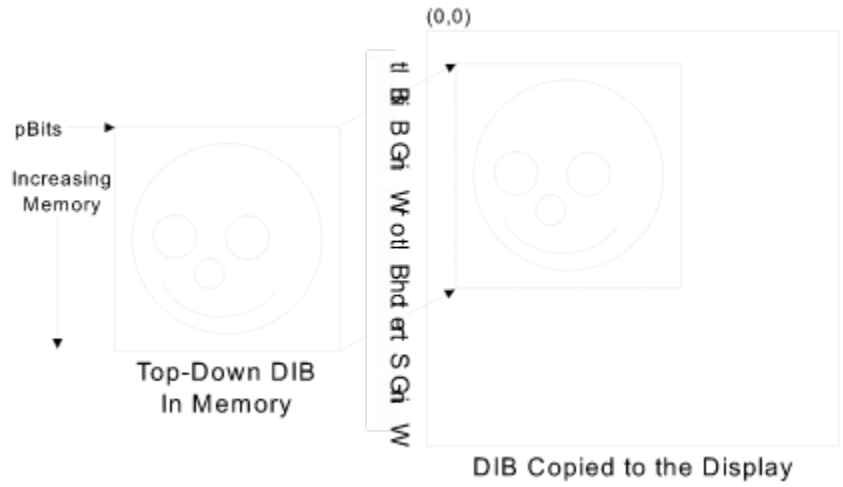
Top-Down DIBs

Another kind of DIB, called a top-down DIB, is stored with the same orientation as most device-dependent bitmaps: the first scanline in memory is the top of the image. Top-down DIBs are identified by a negative biHeight entry in their BITMAPINFOHEADER structures.

Sometimes, WinG can greatly improve the speed of a DIB-to-screen copy by using a top-down DIB because it can avoid inverting the DIB to a device-dependent format. When this is the case, WinGRecommendDIBFormat will return a negative value in the biHeight field of the passed BITMAPINFOHEADER structure.

If you are writing custom DIB drawing routines, you will have to handle top-down DIBs for best performance because there is a good chance that WinG will recommend them with WinGRecommendDibFormat.

WinGStretchBlt and WinGBitBlt recognize top-down DIBs and handle them correctly, but **Windows 3.1 functions such as StretchDIBits and SetDIBitsToDevice will not work properly with top-down DIBs unless you intentionally mirror the image.**



Top-Down DIBs are copied directly to the display

For an 8-bit top-down DIB, the memory address of the pixel (X, Y) can be found with this equation:

```
PixelAddr = DibAddr + Y * DibWidthBytes + X
```

where DibAddr is the base address of the DIB and DibWidthBytes is the DWORD-aligned offset of bytes in memory from the beginning of one scanline to the next.

The PALANIM sample application (in the SAMPLES\PALANIM subdirectory of the WinG development kit) includes a routine to draw horizontal lines into a DIB. To do this, it determines the orientation of the target DIB and performs its calculations accordingly.

The DOGGIE sample application (in the SAMPLES\DOGGIE subdirectory of the WinG development kit)

includes a routine to copy one DIB into another with a transparent color. The assembly function that does this also behaves well with both DIB orientations.

Using an Identity Palette

The Windows Palette Manager, described in depth in Ron Gery's technical article "The Palette Manager: How and Why" (see the Further Reading section for details) arbitrates conflicts between Windows applications vying for color entries in a single hardware palette (known as the system palette). It gives each application it's own virtual 256-color palette, called a logical palette, and translates entries in the logical palette to entries in the system palette as they are needed for blting images to the screen.

An identity palette is a logical palette which exactly matches the current system palette. An identity palette does not require translation of palette entries, so using an identity palette can drastically improve the speed with which you can blt WinGDCs to the screen.

The WinG Halftone Palette is an identity palette. This article describes how to create your own identity palettes for maximum WinG blt speed.

Static Colors

The Palette Manager reserves a number of colors in the palette, called the static colors, which it uses to draw system elements such as window captions, menus, borders, and scroll bars. An identity palette must include the static colors in the appropriate palette entries.

The display driver defines the actual RGB values of the static colors, so they must always be determined at run time. The GetSystemPaletteEntries will retrieve the colors currently in the system palette, and you can isolate the static colors using the SIZEPALETTE and NUMCOLORS capability indices with GetDeviceCaps and a little knowledge of how the Palette Manager works.

The static colors are split in half and stored at either end of the system palette. If there are *nColors* possible entries in the system palette and there are *nStaticColors* static colors, then the static colors will be found in entries 0 through *nStaticColors*/2 - 1 and entries *nColors* - *nStaticColors*/2 through *nColors*-1 in the system palette. Typically, there are 20 static colors, found in indices 0-9 and 246-255 of a 256-color palette. The peFlags portion of these PALETTEENTRY structures must be set to zero.

The SetSystemPaletteUse API turns use of the static colors on and off for the system. Using SYSPAL_STATIC, 20 entries will be reserved in the palette. SYSPAL_NOSTATIC reserves only 2 entries, which must be mapped to black and white. See the Accessing a Full Palette Using SYSPAL_NOSTATIC article for more information.

Other Colors

The remaining non-static colors in the logical palette may be defined by the application, but they must be marked as PC_NOCOLLAPSE or PC_RESERVED (see the PALETTEENTRY documentation for a description) to ensure an identity palette.

A palette containing the static colors in the appropriate entries with the remaining entries marked PC_NOCOLLAPSE, selected and realized into a DC, becomes an identity palette (with the exception of a palette with duplicates of the high-intensity colors as mentioned below). Because no translation to the system palette is required, the Palette Manager can step aside gracefully and leave you to achieve maximum blt bandwidth.

If your palette contains duplicates of the high-intensity static colors (the colors at the top of the palette) you need to set the duplicate entries to PC_RESERVED so GDI doesn't map the upper colors in your logical palette into those slots. PC_RESERVED tells GDI to not only fail to map the current entry into another, but it also keeps other entries (the high-intensity colors in this case) from mapping into the current entry. PC_NOCOLLAPSE only stops the current entry from mapping into other entries, but other entries can still map into the current entry making the palette non-identity.

Creating an Identity Palette

The CreateIdentityPalette() function below shows how to create an identity palette from an array of RGBQUAD structures. Before you realize an identity palette for the first time, it is a good idea to clear the system palette by realizing a completely black palette, as the ClearSystemPalette() function below does. This will ensure that palette-managed applications executed before your application will not affect the identity mapping of your carefully constructed palette.

To make sure that you have successfully created and are using an identity palette, you can tell WinG to send debugging messages to the standard debug output, as described in the Debugging With WinG article.

The PALANIM sample (in the SAMPLES\PALANIM subdirectory of the WinG development kit) uses these routines to create a 256-entry identity palette filled with a wash of color.

Click Here to copy the CreateIdentityPalette() code sample to the clipboard.

Click Here to copy the ClearSystemPalette() code sample to the clipboard.

```
HPALETTE CreateIdentityPalette(RGBQUAD aRGB[], int nColors)
{
    int i;
    struct {
        WORD Version;
        WORD NumberOfEntries;
        PALETTEENTRY aEntries[256];
    } Palette =
    {
        0x300,
        256
    };

    //*** Just use the screen DC where we need it
    HDC hdc = GetDC(NULL);

    //*** For SYSPAL_NOSTATIC, just copy the color table into
    //*** a PALETTEENTRY array and replace the first and last entries
    //*** with black and white
    if (GetSystemPaletteUse(hdc) == SYSPAL_NOSTATIC)
    {
        //*** Fill in the palette with the given values, marking each
        //*** as PC_NOCOLLAPSE
        for(i = 0; i < nColors; i++)
        {
            Palette.aEntries[i].peRed = aRGB[i].rgbRed;
            Palette.aEntries[i].peGreen = aRGB[i].rgbGreen;
            Palette.aEntries[i].peBlue = aRGB[i].rgbBlue;
            Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
        }

        //*** Mark any unused entries PC_NOCOLLAPSE
        for (; i < 256; ++i)
        {
```

```
            Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
        }

        //*** Make sure the last entry is white
        //*** This may replace an entry in the array!
        Palette.aEntries[255].peRed = 255;
        Palette.aEntries[255].peGreen = 255;
        Palette.aEntries[255].peBlue = 255;
        Palette.aEntries[255].peFlags = 0;

        //*** And the first is black
        //*** This may replace an entry in the array!
        Palette.aEntries[0].peRed = 0;
        Palette.aEntries[0].peGreen = 0;
        Palette.aEntries[0].peBlue = 0;
        Palette.aEntries[0].peFlags = 0;
    }
    else
    //*** For SYSPAL_STATIC, get the twenty static colors into
    //*** the array, then fill in the empty spaces with the
    //*** given color table
    {
        int nStaticColors;
        int nUsableColors;

        //*** Get the static colors from the system palette
        nStaticColors = GetDeviceCaps(hdc, NUMCOLORS);
        GetSystemPaletteEntries(hdc, 0, 256, Palette.aEntries);

        //*** Set the peFlags of the lower static colors to zero
        nStaticColors = nStaticColors / 2;
        for (i=0; i<nStaticColors; i++)
            Palette.aEntries[i].peFlags = 0;

        //*** Fill in the entries from the given color table
        nUsableColors = nColors - nStaticColors;
        for (; i<nUsableColors; i++)
        {
            Palette.aEntries[i].peRed = aRGB[i].rgbRed;
            Palette.aEntries[i].peGreen = aRGB[i].rgbGreen;
            Palette.aEntries[i].peBlue = aRGB[i].rgbBlue;
            Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
        }

        //*** Mark any empty entries as PC_NOCOLLAPSE
        for (; i<256 - nStaticColors; i++)
            Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;

        //*** Set the peFlags of the upper static colors to zero
        for (i = 256 - nStaticColors; i<256; i++)
            Palette.aEntries[i].peFlags = 0;
    }

    //*** Remember to release the DC!
    ReleaseDC(NULL, hdc);
```

```
    //*** Return the palette
    return CreatePalette((LOGPALETTE *)&Palette);
}


void ClearSystemPalette(void)
{
    //*** A dummy palette setup
    struct
    {
        WORD Version;
        WORD NumberOfEntries;
        PALETTEENTRY aEntries[256];
    } Palette =
    {
        0x300,
        256
    };

    HPALETTE ScreenPalette = 0;
    HDC ScreenDC;
    int Counter;

    //*** Reset everything in the system palette to black
    for(Counter = 0; Counter < 256; Counter++)
    {
        Palette.aEntries[Counter].peRed = 0;
        Palette.aEntries[Counter].peGreen = 0;
        Palette.aEntries[Counter].peBlue = 0;

        Palette.aEntries[Counter].peFlags = PC_NOCOLLAPSE;
    }

    //*** Create, select, realize, deselect, and delete the palette
    ScreenDC = GetDC(NULL);
    ScreenPalette = CreatePalette((LOGPALETTE *)&Palette);
    if (ScreenPalette)
    {
        ScreenPalette = SelectPalette(ScreenDC,ScreenPalette,FALSE);
        RealizePalette(ScreenDC);
        ScreenPalette = SelectPalette(ScreenDC,ScreenPalette,FALSE);
        DeleteObject(ScreenPalette);
    }
    ReleaseDC(NULL, ScreenDC);
}
```

Palette Animation With WinG

Palette animation creates the appearance of motion in an image by modifying entries in the system palette, resulting in color changes in the displayed image. Carefully arranged and animated palette entries can produce motion effects such as running water, flowing lava, or even motion of an object across the screen.

The Windows AnimatePalette function replaces entries in the logical palette and the system palette. The app does not need to re-realize the palette after a call to AnimatePalette.

Because every DIB and WinGBitmap has an associated color table which is translated to the system palette when the image is copied to the screen, DIBs blted after the palette is animated will not appear animated because their colors are translated to the new palette.

The Using an Identity Palette article discusses the creation of an identity palette which removes the need for color translation when blting. If a palette animating application went through the trouble to create the identity palette, it should maintain the identity mapping between the palette and the WinGDC by matching the WinGBitmap color table to the animated palette before blting. To do this, use WinGSetDibColorTable to keep the WinGBitmap color table synchronized with changes in the system palette.

Remember that any entries in a palette which are to be animated must be marked with the PC_RESERVED flag. PC_RESERVED is a superset of PC_NOCOLLAPSE flag, so these entries can be used in an identity palette.

The PALANIM sample (in the SAMPLES\PALANIM subdirectory of the WinG development kit) performs a simple palette animation with an identity palette, making sure to update the WinGDC color table to match the palette before it blts using the following code, which copies the current logical palette (hpalApp) into the color table of the WinGDC (hdcOffscreen). Of course, if you create the palette yourself from an array of colors, there will be no need to call GetPaletteEntries because you could update the color table from the array you already have in memory. Also, in a palette animation that does not animate the complete palette, an application would not need to modify the entire palette and color table, as this code snippet does:

```
int i;
PALETTEENTRY aPalette[256];
RGBQUAD aPaletteRGB[256];

//*** BEFORE BLTING, match the DIB color table to the
//*** current palette to match the animated palette
GetPaletteEntries(hpalApp, 0, 256, aPalette);
//*** Alas, palette entries are r-g-b, rgbquads are b-g-r
for (i=0; i<256; ++i)
{
    aPaletteRGB[i].rgbRed = aPalette[i].peRed;
    aPaletteRGB[i].rgbGreen = aPalette[i].peGreen;
    aPaletteRGB[i].rgbBlue = aPalette[i].peBlue;
    aPaletteRGB[i].rgbReserved = 0;
}
WinGSetDIBColorTable(hdcOffscreen, 0, 256, aPaletteRGB);
```

Accessing a Full Palette Using SYSPAL_NOSTATIC

The Palette Manager usually reserves twenty static colors in the palette for use in drawing captions, menus, text, scroll bars, window frames, and other system elements. These static colors ensure a common color scheme across all applications, but this leaves only 236 palette entries available to each application. A Windows graphics application requiring a full palette of 256 colors has two options, outlined here.

The first option is to incorporate the static colors into the palette at runtime, knowing that the RGB values of the colors may change slightly from display driver to display driver. This means that the palette will vary slightly when the application runs on different platforms, but it ensures the consistent look and feel between the application and coexisting applications in the system.

The static colors are defined as follows:

Index	Color	Index	Color
0	Black	246	Cream
1	Dark Red	247	Light Gray
2	Dark Green	248	Medium Gray
3	Dark Yellow	249	Red
4	Dark Blue	250	Green
5	Dark Magenta	251	Yellow
6	Dark Cyan	252	Blue
7	Light Gray	253	Magenta
8	Money Green	254	Cyan
9	Sky Blue	255	White

If you can accept the limitation of including these colors in your palette and determining their exact RGB values at runtime (using GetSystemPaletteEntries), you can skip the rest of this article.

The second option is to tell the Palette Manager to make 18 of the twenty static colors available to the application, with entry 0 remaining black and entry 255 remaining white. However, choosing to control those palette entries means you'll have some more intimate relations with the Palette Manager.

To change the use of the static colors in the system palette, you use the SetSystemPaletteUse API, passing either SYSPAL_STATIC or SYSPAL_NOSTATIC. Setting the palette use to SYSPAL_NOSTATIC gives you access to palette entries 1 through 254. Your palette must map entry 0 to RGB(0, 0, 0) and entry 255 to RGB(255, 255, 255), but black and white are standard in most palettes anyway.

Ordinarily, Windows uses entries 0-9 and 246-255 to draw captions, borders, menus, and text, and it will continue to do so after you've changed the RGB values of those palette entries unless you tell it to do otherwise. If you do not inform the operating system of your changes, your application and all others in the system will become very messy and your application will be condemned by its peers as unfriendly.

You want your application to be friendly to the operating system and to the other active applications. You can handle this in two ways: you can make your application a full-screen window with no controls, thereby taking over the entire screen and the full palette, or you can tell the operating system to use different palette entries to draw its captions, borders, menus, and text so that other visible windows do not appear completely strange. In either case, you must restore the static colors when your application becomes inactive or exits.

The following procedure handles the switch between SYSPAL_STATIC and SYSPAL_NOSTATIC for you, managing the mapping and remapping of the system colors for you through the Windows functions GetSysColor and SetSysColors. It stores the current mapping of the system colors before switching to SYSPAL_NOSTATIC mode and restores them after switching back to SYSPAL_STATIC mode.

To use the AppActivate() function in an application, call AppActivate((BOOL)wParam) in response to a WM_ACTIVATEAPP message and call AppActivate(FALSE) before exiting to restore the system colors. This will set the system palette use and remap the system colors when your application is activated or deactivated.

The PALANIM sample (in the SAMPLES\PALANIM subdirectory of the WinG development kit) uses this function to take over the static colors at run time and clean up before it exits.

Click Here to copy this code sample to the clipboard.

```
#define NumSysColors (sizeof(SysPalIndex)/sizeof(SysPalIndex[1]))
#define rgbBlack RGB(0,0,0)
#define rgbWhite RGB(255,255,255)

//*** These are the GetSysColor display element identifiers
static int SysPalIndex[] = {
    COLOR_ACTIVEBORDER,
    COLOR_ACTIVECAPTION,
    COLOR_APPWORKSPACE,
    COLOR_BACKGROUND,
    COLOR_BTNFACE,
    COLOR_BTNSHADOW,
    COLOR_BTNTEXT,
    COLOR_CAPTIONTEXT,
    COLOR_GRAYTEXT,
    COLOR_HIGHLIGHT,
    COLOR_HIGHLIGHTTEXT,
    COLOR_INACTIVEBORDER,
    COLOR_INACTIVECAPTION,
    COLOR_MENU,
    COLOR_MENUTEXT,
    COLOR_SCROLLBAR,
    COLOR_WINDOW,
    COLOR_WINDOWFRAME,
    COLOR_WINDOWTEXT
};

//*** This array translates the display elements to black and white
static COLORREF MonoColors[] = {
    rgbBlack,
    rgbWhite,
    rgbWhite,
    rgbWhite,
    rgbWhite,
    rgbBlack,
    rgbBlack,
    rgbBlack,
    rgbBlack,
    rgbBlack,
    rgbWhite,
    rgbWhite,
    rgbWhite,
    rgbWhite,
    rgbBlack,
    rgbWhite,
    rgbWhite,
```

```
    rgbBlack,
    rgbBlack
};

//*** This array holds the old color mapping so we can restore them
static COLORREF OldColors[NumSysColors];

//*** AppActivate sets the system palette use and
//*** remaps the system colors accordingly.
void AppActivate(BOOL fActive)
{
    HDC hdc;
    int i;

    //*** Just use the screen DC
    hdc = GetDC(NULL);

    //*** If the app is activating, save the current color mapping
    //*** and switch to SYSPAL_NOSTATIC
    if (fActive && GetSystemPaletteUse(hdc) == SYSPAL_STATIC)
    {
        //*** Store the current mapping
        for (i=0; i<NumSysColors; i++)
            OldColors[i] = GetSysColor(SysPalIndex[i]);

        //*** Switch to SYSPAL_NOSTATIC and remap the colors
        SetSystemPaletteUse(hdc, SYSPAL_NOSTATIC);
        SetSysColors(NumSysColors, SysPalIndex, MonoColors);
    }
    else if (!fActive)
    {
        //*** Switch back to SYSPAL_STATIC and the old mapping
        SetSystemPaletteUse(hdc, SYSPAL_STATIC);
        SetSysColors(NumSysColors, SysPalIndex, OldColors);
    }

    //*** Be sure to release the DC!
    ReleaseDC(NULL,hdc);
}
```

Debugging WinG Applications

WinG will report runtime errors and helpful debugging messages through standard Windows debug output devices (the serial port or applications such as DBWIN.EXE) if you so desire. If you want WinG to send error messages to the debug output, make sure the following entry appears in your WIN.INI file. If there is already a [WinG] section (there will be if WinG has previously profiled your display), just add the Debug=1 line under that heading:

```
[WinG]
Debug=1
```

If you specifically do not want debug messages to appear, set this to:

```
[WinG]
Debug=0
```

If neither debug level is specified in the [WinG] section of your WIN.INI file, debugging will be turned ON if you're using the Windows debug kernel and OFF if you're using the Windows retail kernel. Setting the Debug level explicitly in your WIN.INI will always override this default behavior.

WinG can also provide you with a detailed log of palette translations between the color table of a WinGBitmap and the logical palette and system palette. This will help to identify discrepancies in the palette which prevent WinG from using faster blts with an identity palette. To receive these palette notifications through the debug output, add the following line to the [WinG] section of win.ini:

```
DebugPalette = 1
```

Debugging WinG applications is no different than debugging normal Windows applications, but there are some minor known problems and workarounds for the following debuggers:

Borland Turbo Debugger

There are known problems with debugging WinG applications using Turbo Debugger for Windows. If you have problems debugging with TDW, try running one of the samples first and leave it minimized so that WinG.DLL will already be in memory when you debug your application. There seem to be no problems with the Borland IDE.

Just-in-Time Debuggers

When WinG profiles a display it installs a fault handler to catch any GP Faults caused by WinG or the display driver. Some Just-in-Time debuggers (like CodeView JIT) catch these faults before WinG and kill the profiling window, so WinG doesnt load. These faults should be passed to WinG so it can deal with them appropriately. Either instruct your Just-in-Time debugger to chain the faults through to WinG, or do not run the debugger when WinG is profiling. See Display Profiling.

Compiling WinG Applications

WinG.DLL is a standard 16 bit DLL with PASCAL calling convention exports; WinG32.DLL is a standard Win32 DLL with _stdcall exports. You should be able to write WinG applications with any compiler in any language that can generate Windows 16 bit or Win32 executables that call DLLs.

16 bit Applications

Most 16 bit Windows compilers and linkers are standardized, so you should have no trouble linking to the wing.lib file provided in the WinG SDK. If you do have trouble, try using your compilers import librarian if possible, import the WinG functions directly in your .def file or its equivalent, or dynamically link to WinG.DLL using LoadLibrary and GetProcAddress.

32 bit Applications

Compiling and linking for Win32 can be tricky with some compilers. Several major compilers require various incantations to successfully build a Win32 application. Hopefully, the following hints will aid in the successful compilation and linking of WinG applications using these compilers.

The WinG32.lib file in the WinG SDK is a Common Object File Format (COFF) library file. Most non-Microsoft Win32 compilers support only Object-Module Format (OMF) library files, so you may have problems linking to WinG32 functions. If you encounter link problems, first see if your compiler has an "import librarian," a program that creates a library file from a DLL. Run this program on WinG32.dll and try linking with the generated library file. If this doesn't work or you don't have an import librarian, try explicitly importing the functions in your .def file. As a last resort you can dynamically link to WinG32 using LoadLibrary and GetProcAddress.

Borland

Run implib on WinG32.dll or modify the .def file.

Symantec

Run implib on WinG32.dll or modify the .def file.

Watcom

The Watcom linker will have trouble linking to the wing32.lib export file included with the WinG Development Kit. This export file lists undecorated names, using the conventions established for Alpha, MIPS, and x86 cross-assembly. To create a new wing32.lib file linkable with the Watcom tools, run wlib on win32.dll as follows:

```
wlib wing32 @wing32.lbc
```

where wing32.lbc is a text file containing the following description of the WinG functions:

```
++'_WinGBitBlt'.'WING32.DLL'..WinGBitBlt
++'_WinGCreateBitmap'.'WING32.DLL'..WinGCreateBitmap
++'_WinGCreateDC'.'WING32.DLL'..WinGCreateDC
++'_WinGCreateHalftoneBrush'.'WING32.DLL'..WinGCreateHalftoneBrush
++'_WinGCreateHalftonePalette'.'WING32.DLL'..WinGCreateHalftonePalette
++'_WinGGetDIBColorTable'.'WING32.DLL'..WinGGetDIBColorTable
++'_WinGGetDIBPointer'.'WING32.DLL'..WinGGetDIBPointer
++'_WinGRecommendDIBFormat'.'WING32.DLL'..WinGRecommendDIBFormat
++'_WinGSetDIBColorTable'.'WING32.DLL'..WinGSetDIBColorTable
++'_WinGStretchBlt'.'WING32.DLL'..WinGStretchBlt
```

DISPDIB and WinG

Programmers that do not want to take advantage of the Windows user interface, want to run full screen in a low-resolution mode, or want direct access to the VGA hardware should consider using DISPDIB, a DLL that provides direct VGA hardware access under Windows through the DisplayDib function. Some applications may use both DISPDIB and WinG, giving the user the option of running in a full-screen VGA mode or a high-resolution window.

Ordinarily, DisplayDib displays 256-color bitmaps in 320x200 or 320x240 modes until a key or mouse button was pressed, but you can take over the video indefinitely using the DISPLAYDIB_NOWAIT flag or the DISPLAYDIB_BEGIN and DISPLAYDIB_END flags. Given these flags, DisplayDib immediately returns control to your application, leaving it in a full-screen VGA mode and allowing your application to access the VGA hardware directly.

While it has control of the screen, DISPDIB disables the Windows video driver. GDI has no effect on the display device, but all other Windows system components remain active. Your application can still use memory DCs (including WinGDCs) to modify images displayed using DISPDIB, and it continues to run as usual and to receive messages as usual, including timer, keyboard, and mouse messages (although the mouse pointer is not visible).

Using DISPDIB, an application can take control of the display in a VGA mode. It can allocate an off-screen buffer using WinG, draw into the buffer using GDI, and take advantage of Windows memory management, networking, sound, timers, mouse handling, and other system services.

Mouse coordinates received by an application using DISPDIB will be in the coordinate system of the original display driver. That is, if an application running in a 1024x768 Windows session switches to a 320x200 full-screen mode using DISPDIB, the system continues to return mouse coordinates as though the screen were 1024x768. The application must scale the mouse coordinates appropriately.

Also note that your application using DISPDIB must be the active application when it calls DisplayDib.

DISPDIB prevents other tasks from running while it is active. This causes problems with some applications and with the high-level MCI multimedia APIs, which use a background task to function. You can use the DISPDIB_DONTLOCKTASK flag (0x0200) to prevent DISPDIB from locking out background tasks, but you run the risk of another application gaining control.

Documentation, sample code, and information about obtaining DISPDIB.DLL is available throught the Microsoft Developer Network, the MSDN CD, and Microsoft Developer Relations.

Shipping a Product With WinG

Microsoft grants you the royalty-free right to distribute any application you create using WinG, along with the WinG runtime support files listed below.

If your application uses WinG and the WinG runtime files are not present in the \SYSTEM subdirectory of the target system's Windows directory, your setup program will have to install the WinG runtime. The following files should be installed in the user's SYSTEM directory:

WING.DLL
WING32.DLL
WINGDE.DLL
WINGDIB.DRV
WINGPAL.WND
DVA.386

If DVA.386 was not previously installed on the user's system (for example, by Video for Windows), your setup program should add the following line to the [386Enh] section of the SYSTEM.INI file, after which the user must reboot Windows for optimal WinG performance:

```
device=dva.386
```

This file must be installed properly as shown in the WING.MST setup script or Windows will not boot.

Setup Toolkit

The Windows Software Development Kit includes the Setup Toolkit for Windows, which allows you to create and run setup scripts for installing Windows applications. Documentation for the toolkit comes with the Windows SDK and is also available through the Microsoft Developer Relations Group and the Microsoft Developer Network CD.

The WinG Development Kit setup program installs the WinG runtime files using Microsoft setup exactly as they should be installed on a target user's system. Look at the WING.MST script on the WinG installation diskette to see how this is done. WING.MST calls SETUPHLP.DLL to aid in installing the runtime files. You can use this DLL in your own application setup. SETUPHLP.DLL will version check Win32 DLLs (for installing WinG32.DLL) and check for the presence of dva.386, among other things.

Customizing the WinG Profiler

The first time a WinG application runs in a new video mode or after a new display driver has been installed, WinG performs a performance test to determine the fastest way to blt a WinGBitmap to the display. The user will only see this performance test once for any given video mode, but on slower machines the test may last several minutes.

By default, the profiling window displays a message to the user that says "Your program is testing for optimal display performance so that it will look its best on your system. This will take a few minutes, but you will only see it again if you change to new display settings." However, you might want WinG to display something more informative or specific to your application. You can customize this message by placing a ProfileMessage= entry in the [WinG] section of the user's WIN.INI file. For example:

```
[WinG]
ProfileMessage=Hold on to your seats! WindowsSuperApp is turbo-charging your display!
```

Your custom profile message must be 255 characters or less, and you must place it in WIN.INI immediately before WinG loads. After loading, WinG deletes this entry to avoid conflicts with other WinG applications.

If you choose to provide a custom profile message, your application will have to change the ProfileMessage entry before WinG loads, either by a separate loader application that sets the entry and

runs your application using WinExec or by setting the ProfileMessage entry and dynamically loading WinG using LoadLibrary.

Code Samples

The WinG development kit contains a variety of code samples to help you develop fast applications quickly using WinG.

Snippets

The following code samples appear in this help file:

Setting up an off-screen buffer with WinG.

Calculating the memory address of a scanline in a WinGBitmap.

Creating an Identity Palette.

Clearing the System Palette.

Maximizing palette availability using the SYSPAL_NOSTATIC setting.

Copying a logical palette to a WinGBitmap color table.

Matching an RGB color to a halftone palette entry.

Sample Applications

The WinG Development Kit also contains source code for several sample applications, installed in the \SAMPLES subdirectory. The following applications are available:

DOGGIE allows the user to drag a sprite around the screen with the mouse, demonstrating off-screen composition, dirty rectangle animation, and custom blt routines. Includes source code for a sample 8-bit DIB to 8-bit DIB blt with one transparent color.

CUBE displays a halftoned rotating cube in a window that the user can manipulate with the mouse. It demonstrates off-screen composition, double-buffering, and using the halftone palette and halftone brushes with GDI to draw into a WinGDC.

TIMEWING tests and compares blt speeds of existing GDI functions with the WinGStretchBlt function. This sample will give you an idea of how WinG will compare to standard GDI blts.

HALFTONE converts 24-bit RGB DIBs to 8-bit DIBs by dithering them to the WinG Halftone Palette. The source code implements a standard 8x8 dither and color matching to the halftone palette.

PALANIM performs simple palette animation with an identity palette using WinG. This application uses all of the sample code appearing in this help file.

Balloon Doggie Sample

The Balloon Doggie sample application, found in the SAMPLES\DOGGIE subdirectory of the WinG development kit, demonstrates a simple dirty rectangle animation system. It creates a WinGDC and a WinGBitmap, which it uses as an off-screen buffer, and uses WinGBitBlt to update the screen.

Balloon Doggie includes source code for TransparentDIBits (in TBLT.C and FAST32.ASM), a fast DIB-to-DIB blt with transparency. TransparentDIBits demonstrates the use of custom drawing routines with WinG to provide functions not present or unacceptably slow in GDI.

Note that DOGGIE.EXE requires MASM 5.1 compatibility to compile FAST32.ASM. If you do not own MASM 5.1, use MASM 6.x with the /Zm option or link to the precompiled FAST32.OBJ, included with the DOGGIE sample.

Spinning Cube Sample

The CUBE.EXE sample application, found in the SAMPLES\CUBE subdirectory of the WinG development kit, demonstrates the use of Halftoning to create the appearance of more than 256 colors on an 8-bit palletized display device. Using WinGCreateHalftonePalette and WinGCreateHalftoneBrush, the spinning cube application halftones the faces of the cube to create lighting effects.

When appropriate, the application uses the GDI Polygon function to draw into the off-screen buffer then calls WinGBitBlt to copy the buffer to the screen.

The CUBE sample uses a simple floating-point vector and camera C++ class library (in DUMB3D.HPP and DUMB3D.CPP) that can be used as a starting point by those interested in generating 3D graphics.

WinG Timing Sample

The timing sample, TIMEWING.EXE, found in the SAMPLES\TIMEWING subdirectory of the WinG development kit, times and compares the blt and stretching speeds of StretchBlt, StretchDIBits, and WinGStretchBlt. The application provides a summary you can use to compare the speeds of these techniques on various video configurations and a framework you can use for your own timing tests.

Note that StretchDIBits and WinGStretchBlt operate on device-independent bitmaps whereas StretchBlt and BitBlt operate on device-specific bitmaps, which require no translation and can sometimes be stored in the local memory of the graphics card itself.

Timewing.exe can be built as a Win32 app by compiling with the timewing.m32 makefile. You may need to modify timewing.m32 for your compiler environment.

WinG Halftoning Sample

HALFTONE.EXE, found in the SAMPLES\HALFTONE subdirectory of the WinG development kit, dithers 24-bit DIBs to the WinG Halftone Palette using an 8x8 ordered dither.

The main function, DibHalftoneDIB in HALFTONE.C, does the real work in the dithering. The process of calculating an ordered dither is too complex to describe here, but a description of the techniques involved can be found in "Computer Graphics: Principles and Practice" by Foley, van Dam, Feiner, and Hughes. See the Further Reading article for more information on this book.

The aWinGHalftoneTranslation array found in HTTABLES.C converts a 2.6-bit-per-pixel computed halftone index into an entry in the halftone palette. To calculate the nearest match of an RGB color to the halftone palette, HALFTONE uses the following formula:

```
HalftoneIndex = (Red / 51) + (Green / 51) * 6 + (Blue / 51) * 36;
HalftoneColorIndex = aWinGHalftoneTranslation [HalftoneIndex];
```

See also the documentation for the WinGCreateHalftoneBrush function and the Halftoning With WinG article.

WinG Palette Animation Sample

The PALANIM.EXE application, found in the SAMPLES\PALANIM subdirectory of the WinG development kit, performs simple palette animation using AnimatePalette and WinGSetDIBColorTable as described in the Palette Animation With WinG article.

PALANIM gives the user the option of using the static colors in the palette to create a 254-color ramp or a 236-color ramp in an identity palette for fast blting.

The PALANIM sample uses all of the code samples found in this help file to perform its WinG functions.

WinG readme.txt Release Notes

Microsoft WinG version 1.0

This file describes known bugs, gotchas, and helpful hints for the WinG Version 1.0 final release.

ISVs may want to distribute portions of this readme file that describe configuration bugs along with shipping products that use WinG.

WinG version 1.0 provides fast DIB-to-screen blts under Windows 3.1, Windows for Workgroups 3.11, Windows 95, and Windows NT version 3.5. WinG will not run on Windows NT version 3.1 or on earlier versions of Windows.

WinG requires a 386 or better processor to run. WinG will not run on a 286.

If you have problems with WinG, please run the wingbug.exe file in the bin directory of the SDK and send the generated report to wingbug@microsoft.com. Updated information on WinG is available on CompuServe in the WINMM forum and on ftp.microsoft.com.

Known Bugs And Limitations

The following are known problems with or useful tidbits of information about WinG version 1.0.

- On Windows 3.1, WinGBitmaps must be 8 bits per pixel and must be created with full 256 entry color tables.

- WinGDCs are NOT palette devices. You must change their color tables using WinGSetDIBColorTable, not SelectPalette.

- WinGBitBlt and WinGStretchBlt only support bltting from WinGDCs to the screen.

- Using BitBlt and StretchBlt to blt from one WinGDC to another can be very slow when a clipping region has been selected into the destination.

- WinGBitBlt and WinGStretchBlt may return different values than StretchDIBits for identical blts.

- A few GDI APIs do not work correctly with WinGDCs:

  StretchDIBits will not blt 24bpp and 16bpp DIBs into an 8bpp WinGDC.
  FloodFill with a NULL brush draws incorrectly
  FloodFill outside of the bounds of a WinGBitmap can flood the entire image
  Brushes created with CreatePatternBrush with a WinGBitmap fault when selected into a WinGDC on Win3x - use CreateBitmap(8,8,1,8,0)
  DrawIcon will crash
  WinGBitBlt and WinGStretchBlt will not always blt to the correct location when you have changed the Viewport and/or Window origins

using SetViewportOrg or SetWindowOrg.

- You cannot change the origin of halftone brushes created by WinG.

- Noticeable timing differences have been found while running the WinG profiler on a computer connected to a network. For accurate results, disconnect your computer from the network the first time you run a WinG application. After the profile is complete, you can plug the net in again. NOTE: Microsoft is not liable for any damage you may incur by incorrect handling of your computer hardware.

- WinGBitBlt and WinGStretchBlt use a slightly different color matching algorithm than StretchDIBits when blting an 8-bit image to a 4-bit planar display such as a standard VGA. Mixing WinG and StretchDIBits on these displays may produce odd results.

- WinG relies on the mmsystem timer drivers to determine display performance. If mmsystem.dll and timer.drv are not installed correctly, the results of the performance test may be incorrect. mmsystem.dll should appear on the drivers= line of the [boot] section of system.ini, and timer=timer.drv should appear in the [drivers] section of system.ini.

- WinG version 1.0 does not yet use standard DCI because of time constraints.

- "Just In Time" debuggers install a fault handler in a chain along with the WinG display performance profiler. If your debugger reports a fault during the WinG display performance test, pass the fault on to Windows instead of invoking your debugger.

Driver-Specific Problems

WinG depends on Windows display drivers written by independant hardware manufacturers for much of its speed. Bugs or performance problems in third-party display drivers may cause problems in WinG. In many cases, the video card manufacturer has already fixed the bug, and upgrading your display driver will often clear away problems.

There are some specific "bugs" in display drivers of which you should be aware. The list below is not intended to slight the manufacturer of any particular card or driver.

A list of stress-tested configurations is available on the CompuServe WINMM forum and ftp.microsoft.com.

Some names in this list are trademarks of the respective manufacturer.

- Early drivers for Diamond Viper cards included a "Power Palette" option that is no longer supported by Diamond. They recommend that you upgrade your drivers if you have this option. WinG may be slower when power palette is enabled.

- IBM no longer supports the IBM ThinkPad 720c. There are some problems using WinG with the ThinkPad 720c display drivers.

- Cirrus drivers before version 1.43 have many known bugs which have been fixed in the more recent drivers. Be sure to upgrade your drivers if you are still running with this version.

- Some ATI drivers offer a "Crystal Fonts" option. Turning Crystal Fonts on in 8-bit modes sets up a non-palettized driver that can slow WinG significantly.

- The ATI mach8 Radical drivers cause a number of problems in both WinG and in Windows with some versions of the ATI chipset. Be aware.

- The ATI VGA Wonder drivers (W31-*.drv) will crash during a call to StretchDIBits in the profiler. Users can run the SVGA256.DRV driver that shipped with Windows.

- Many miro Crystal drivers have problems with StretchDIBits, so they crash during profiling.

- Early ATI Mach 32 PCI cards have a hardware timing problem and will hang while blting. ATI will replace these cards for no cost.

- WinG is incompatible with the #9GXE "TurboCopy" mode. Use the #9 control panel to disable TurboCopy (it is off by default).

- WinG uses a GetPixel to synchronize with display hardware when writing directly to the screen. The ATI Mach 32 driver's GetPixel does not work properly, so it is possible to use GDI to draw to the screen, then use WinG to blt to the screen and have them overwrite each other. Be careful mixing GDI drawing commands and WinG blts to the display.

- The Orchid mmtllo.drv driver for the Prodesigner IIs has duplicate system colors which prevents applications from getting an identity palette and greatly reduces the WinG blt speed. A workaround is to set SYSPAL_NOSTATIC mode or use standard the Tseng ET4000 drivers instead of the mmtllo drivers.

A Note on Speed

WinG is designed to be the absolute fastest way to blt DIBs under Windows. The goal, achieved in many cases, is to blt at memory bandwidth to the display device.

On most 8bpp devices, if you use the recommended DIB format (returned by WinGRecommendDIBFormat) and set up correct identity palette, you should get 1:1 blt speeds comparable to BitBlt, which blts device dependent bitmaps (DDBs) to the display. The timewing sample application will show you various blt speeds on your display.

The WinGRecommendDIBFormat API will tell you whether to use top-down

or bottom-up DIBs for fastest unclipped 1:1 identity palette blts. If you plan on using other types of blt (such as stretching or complex clipping), your application may want to time top-down versus bottom-up blts itself at run time. See the WinG help file for more information.

WinG Programmer's Reference

This is a standard Windows 3.1 or Win32 API, structure, or constant. Documentation can be found in the Help files installed with the Windows SDK.

This is a standard Win32 API, structure, or constant. Documentation can be found in the Help files installed with the Win32 SDK.

WinG API

The WinG API is a small set of functions for manipulating DIBs in the same way that device dependent bitmaps are manipulated in Windows.

WinGDCs and WinGBitmaps

WinGCreateDC	Create a new WinGDC
WinGCreateBitmap	Create a new WinGBitmap
WinGGetDIBPointer	Return the DIB pointer to a WinGBitmap
WinGRecommendDIBFormat	Recommend an optimal DIB format for memory-to-screen blts
WinGGetDIBColorTable	Return the DIB color table of a selected WinGBitmap
WinGSetDIBColorTable	Set the DIB color table of a selected WinGBitmap

Blts

WinGBitBlt	Copy a WinGDC to the screen DC
WinGStretchBlt	Copy a WinGDC to the screen DC with stretching

Halftoning

WinGCreateHalftoneBrush	Create a halftone brush
WinGCreateHalftonePalette	Create a copy of the WinG halftone palette
WING_DITHER_TYPE	Dither types for halftone brushes

WinGBitBlt

Copies an area from a specified device context to a destination device context. **WinGBitBlt** is optimized to copy WinGDCs to display Dcs. It does not operate on any other types of Device Contexts.

BOOL WinGBitBlt(HDC *hdcDest*, **int** *nXOriginDest*, **int** *nYOriginDest*, **int** *nWidthDest*, **int** *nHeightDest*, **HDC** *hdcSrc*, **int** *nXOriginSrc*, **int** *nYOriginSrc*)

Parameters

hdcDest	Identifies the destination device context.
nXOriginDest	X coordinate of the upper-left corner of the destination rectangle in MM_TEXT client coordinates.
nYOriginDest	Y coordinate of the upper-left corner of the destination rectangle in MM_TEXT client coordinates.
nWidthDest	Width of the source and destination rectangle.
nHeightDest	Height of the source and destination rectangle.
hdcSrc	Identifies the source device context.
nXOriginSrc	X coordinate of the upper-left corner of the source rectangle in MM_TEXT client coordinates.
nYOriginSrc	Y coordinate of the upper-left corner of the source rectangle in MM_TEXT client coordinates.

Return Value

The return value is non-zero if the function is successful. Otherwise, it is zero.

Comments

WinGBitBlt requires both DCs to use MM_TEXT mapping mode at the time of the call or the results may be unpredictable. At other times, any mapping mode may be used in either DC.

WinGBitBlt does not handle non-zero viewport or window origins set by SetViewportOrg or SetWindowOrg.

WinGBitBlt will fail under Windows 3.1 and Win32s if the source DC does not contain an 8-bit-per-pixel WinGBitmap.

On Win32, calls to **WinGBitBlt** may be batched by the operating system. To ensure that a blt occurs immediately upon calling **WinGBitBlt**, be sure to call GDIFlush just after every call to **WinGBitBlt**.

WinGBitBlt only copies from a WinGDC to a Display DC. To copy from a WinGDC to a WinGDC, use custom blt functions or BitBlt. You can not copy from a Display DC to a WinGDC on Windows 3.x or Win32s.

Maximizing Performance

You will get the highest performance from **WinGBitBlt** if you select a WinGBitmap created from header information supplied by a call to WinGRecommendDIBFormat.

WinGBitBlt is optimized for copying WinGDCs to the screen.

Aligning the destination rectangle to DWORD boundaries (4-pixel boundaries on an 8-bit display) can

help **WinGBitBlt** to achieve maximum speed.

Clipping can slow **WinGBitBlt** down. In general, don't select clipping regions into or blt outside the boundaries of the source or destination DCs and avoid blting to an overlapped window if possible.

See Also

WinGStretchBlt WinGCreateDC WinGCreateBitmap WinGRecommendDIBFormat Maximizing Performance With WinG

WinGCreateBitmap

Creates a WinGBitmap for the given WinGDC using the specified header information.

HBITMAP WinGCreateBitmap(HDC *hWinGDC*, BITMAPINFO far **pHeader*, void far *far **ppBits*)

Parameters

hWinGDC	Identifies the WinG device context.
pHeader	Points to a BITMAPINFO structure specifying the width, height, and color table for the new WinGBitmap.
ppBits	If not 0, points to a pointer to receive the address of the new WinGDC DIB surface.

Return Value

Returns a handle to the new WinGBitmap DIB surface or 0 if it is unsuccessful.

Comments

Currently, under Windows 3.1 and Win32s, **WinGCreateBitmap** will only create 8-bit-per-pixel surfaces.

If ppBits is 0, the address of the newly created bitmap will not be returned. WinGGetDIBPointer will return this information if you choose to ignore it here.

pHeader must point to enough memory to hold a BITMAPINFOHEADER and a complete color table of RGBQUAD entries. The biClrUsed field of the BITMAPINFOHEADER specifies the number of colors in the color table; if it is zero, the maximum number of colors according to biBitCount are used if biBitCount is less than 24. For example, if biBitCount is 8 and biClrUsed is 0, 256 palette entries are expected. See the BITMAPINFOHEADER description in the Windows 3.1 SDK Reference for more information.

When an application has finished using a WinGBitmap, it should select the bitmap out of its WinGDC and remove the bitmap by calling DeleteObject.

The pointer to the WinGBitmap DIB surface returned by **WinGCreateBitmap** must not be freed by the caller. The allocated memory will be freed by a call to DeleteObject.

WinGCreateBitmap uses *pHeader* and the subsequent color table to create the drawing surface. WinG ignores the biClrImportant, biXPelsPerMeter, biYPelsPerMeter, and biSizeImage fields. WinG expects biCompression to be BI_RGB.

If the biHeight field of the passed BITMAPINFOHEADER is negative, **WinGCreateBitmap** will create a top-down DIB as the bitmap surface. See the article on DIB Orientation for a discussion of top-down and bottom-up DIBs.

An HBITMAP can only be selected into one device context at a time, and a device context can only have a single HBITMAP selected in at a time.

WinGCreateBitmap is very similar to the new CreateDIBSection API. CreateDIBSection is a Win32 API available on Windows 95 and Windows NT 3.5.

Maximizing Performance

To create a WinGBitmap that will maximize WinGBitBlt performance, use

WinGRecommendDIBFormat to fill in the entries of *pHeader* before calling **WinGCreateBitmap**, remembering to modify the height and width to suit your needs.

Larger WinGBitmaps take longer to blt to the screen. Also, if the screen DC is clipped, for example by an overlapping window or by a selected clip region, the WinGDC will take longer to blt to the screen.

Using an identity palette that exactly matches the WinGBitmap's color table will greatly increase performance.

In 16 bit code, the pointer to the bits can be accessed with 32 bit offsets in assembly, like all huge objects in Windows.

Example

The following code fragment shows how an application could create a WinGDC with an optimal 100x100 WinGBitmap selected for drawing, then delete it when it is no longer needed. Note that the WinGBitmap will initially have garbage in its color table--be sure to call WinGSetDIBColorTable before using the WinGDC.

The PALANIM sample (in the SAMPLES\PALANIM subdirectory of the WinG development kit) uses these routines, modified to create a 256x256 WinGDC, to allocate and free its drawing buffer.

Click Here to copy this code sample to the clipboard.

```
HBITMAP ghBitmapMonochrome = 0;

HDC Create100x100WinGDC(void)
{
    HDC hWinGDC;
    HBITMAP hBitmapNew;
    struct {
        BITMAPINFOHEADER InfoHeader;
        RGBQUAD ColorTable[256];
    } Info;
    void far *pSurfaceBits;

    // Set up an optimal bitmap
    if (WinGRecommendDibFormat((BITMAPINFO far *)&Info) == FALSE)
        return 0;

    // Set the width and height of the DIB but preserve the
    // sign of biHeight in case top-down DIBs are faster
    Info.InfoHeader.biHeight *= 100;
    Info.InfoHeader.biWidth = 100;

    // Create a WinGDC and Bitmap, then select away
    hWinGDC = WinGCreateDC();
    if (hWinGDC)
    {
        hBitmapNew = WinGCreateBitmap(hWinGDC,
            (BITMAPINFO far *)&Info, &pSurfaceBits);
        if (hBitmapNew)
        {
            ghBitmapMonochrome = (HBITMAP)SelectObject(hWinGDC,
                hBitmapNew);
        }
```

```
            else
            {
                DeleteDC(hWinGDC);
                hWinGDC = 0;
            }
        }

        return hWinGDC;
    }

    void Destroy100x100WinGDC(HDC hWinGDC)
    {
        HBITMAP hBitmapOld;

        if (hWinGDC && ghBitmapMonochrome)
        {
            // Select the stock 1x1 monochrome bitmap back in
            hBitmapOld = (HBITMAP)SelectObject(hWinGDC,
                ghBitmapMonochrome);
            DeleteObject(hBitmapOld);
            DeleteDC(hWinGDC);
        }
    }
```

See Also

WinGCreateDC WinGRecommendDIBFormat CreateBitmapCreateCompatibleBitmap BITMAPINFO BITMAPINFOHEADER WinGGetDIBPointer CreateDIBSection Code Samples Off-screen Drawing With WinG Maximizing Performance With WinG

WinGCreateDC

Creates a WinG device context with the stock 1x1 monochrome bitmap selected.

HDC WinGCreateDC(void)

Return Value

Returns the handle to a new WinGDC if successful. Otherwise, **WinGCreateDC** returns 0.

Comments

Device contexts created using **WinGCreateDC** must be deleted using the DeleteDC function. All objects selected into the WinGDC after it was created should be selected out and replaced with the original objects before the device context is deleted.

When a WinGDC is created, WinG automatically selects the stock 1x1 monochrome bitmap as its drawing surface. To begin drawing on the WinGDC, select a WinGBitmap created by the WinGCreateBitmap function into the WinGDC.

Maximizing Performance

WinGCreateDC has a fairly high overhead and is usually used to create a single off-screen DC. In general, programs will call **WinGCreateDC** once at startup then select new WinGBitmaps on WM_SIZE messages to the double-buffered window. Applications can use the WM_GETMINMAXINFO message to restrict the size of their window if necessary.

Compose frames into WinGDCs, then use WinGStretchBlt or WinGBitBlt to copy the WinGDC to the screen.

Example

See the WinGCreateBitmap API for sample code that uses **WinGCreateDC**.

See Also

WinGCreateBitmap CreateDC DeleteDC WM_SIZE WM_GETMINMAXINFO WinGStretchBlt WinGBitBlt CreateDIBSection Off-screen Drawing With WinG Maximizing Performance With WinG Code Samples

WinGCreateHalftoneBrush

Creates a dithered pattern brush based on the WinG halftone palette.

HBRUSH WinGCreateHalftoneBrush(HDC *hdc*, **COLORREF** *Color*, **enum WING_DITHER_TYPE** *DitherType***)**

Parameters

hdc	Specifies the DC with which the brush should be compatible.
Color	Specifies the color to be approximated by the brush.
DitherType	Specifies the dither pattern for the brush. Can be one of: WING_DISPERSED_4x4 WING_DISPERSED_8x8 WING_CLUSTERED_4x4

Return Value

Returns a handle to a GDI brush if successful. Otherwise, **WinGCreateHalftoneBrush** returns 0.

Comments

This API is intended for simulating true color on 8-bit devices. It will create a patterned brush using colors from the halftone palette regardless of the color resolution of the target device. If *hdc* refers to a 24-bit device, **WinGCreateHalftoneBrush** will not return a solid brush of the given color, it will return a colored dither pattern using colors that appear in the halftone palette. On true-color devices, creating a solid brush that exactly matches the desired color is simple; **WinGCreateHalftoneBrush** lets you use the halftone patterns instead if you so desire.

A halftone brush approximates the requested *Color* using combinations of colors in the halftone palette. Larger dither patterns give a better approximation of the desired color but require more area to show the approximation. Quality is subjective, so programmers should experiment with different dither types to find the one that suits their needs.

If the target DC is a palette device, the WinG halftone palette must be selected into it and realized as an identity palette for correct visual results (See ClearSystemPalette). Use the WinGCreateHalftonePalette function to create a copy of the halftone palette, then select and realize it before using a halftone brush on a palette device.

The WING_DISPERSED_nxn dither types create nxn patterns that approximate *Color* with a dispersed dot ordered dither.

The WING_CLUSTERED_4x4 dither type creates a 4x4 pattern that approximates *Color* with a clustered dot ordered dither.

Always free GDI objects such as brushes by calling DeleteObject when the object is no longer needed.

Maximizing Performance

Avoid redundant creation, selection, and deletion of identical brushes as much as possible. If an application will be using the same brush repeatedly, it should create the brush once and save it for later use, deleting it when the application is complete.

Example

The CUBE sample application (in the SAMPLES\CUBE directory of the WinG Development Kit) allows the user to select the dither type for creating shaded brushes and provides a good experiment in using the different dither types.

See Also

WinGCreateHalftonePalette WING_DITHER_TYPE CreateDIBPatternBrush CreateSolidBrush Halftoning With WinG Using GDI With WinGDCs Code Samples

WinGCreateHalftonePalette

Creates an 8-bit palette used for halftoning images.

HPALETTE WinGCreateHalftonePalette(void)

Return Value

Returns the handle of a logical palette containing the colors of the WinG halftone palette palette if successful. Otherwise, **WinGCreateHalftonePalette** returns 0.

Comments

The halftone palette should be selected into any DC into which the application will use WinG to halftone.

The WinG halftone palette is an identity palette: the logical palette indices and physical device indices are the same.

The halftone palette inverts correctly, so bitwise XORs invert colors properly.

See the Using an Identity Palette article for a discussion of identity palettes.

Maximizing Performance

Call **WinGCreateHalftonePalette** once at the beginning of your application. Select and realize the palette on WM_QUERYNEWPALETTE, WM_PALETTECHANGED, and WM_PAINT messages.

Example

The HALFTONE sample application (in the SAMPLES\CUBE directory of the WinG Development Kit) uses the halftone palette to dither 24-bit images to 8-bits using an 8x8 ordered dither.

See Also

WinGCreateHalftoneBrush WinGStretchBlt WinGBitBlt RealizePalette WM_QUERYNEWPALETTE WM_PALETTECHANGED Halftoning With WinG Using an Identity Palette Code Samples

WinGGetDIBColorTable

Returns the color table of the WinGBitmap currently selected into a WinGDC.

UINT WinGGetDIBColorTable(HDC *hWinGDC*, **UINT** *StartIndex*, **UINT** *NumberOfEntries*, **RGBQUAD far ***pColors **)**

Parameters

hWinGDC	Identifies the WinG device context whose color table should be retrieved.
StartIndex	Indicates the first palette entry to be retrieved.
NumberOfEntries	Indicates the number of palette entries to retrieve.
pColors	Points to a buffer which receives the requested color table entries.

Return Value

Returns the number of palette entries copied into the given buffer or 0 if it failed.

Comments

The *pColors* buffer must be at least large enough to hold *NumberOfEntries* RGBQUAD structures.

Note that *StartIndex* indicates an entry in a palette array, which is zero-based. *NumberOfEntries* indicates a one-based count. If *NumberOfEntries* is zero, no color table entries will be retrieved.

WinGGetDIBColorTable will return 0 for WinGBitmaps with more than 8 bits per pixel.

See Also

WinGSetDIBColorTable WinGCreateBitmap

WinGGetDIBPointer

Retrieves information about a WinGBitmap and returns a pointer to its surface.

void far *WinGGetDIBPointer(HBITMAP *hWinGBitmap*, **BITMAPINFO far ***pHeader***)**

Parameters

hWinGBitmap	Identifies the WinGBitmap whose surface should be retrieved.
pHeader	If not 0, points to a buffer to receive the attributes of the WinGDC.

Return Value

Returns a pointer to the bits of a WinGBitmap drawing surface if possible. Otherwise, **WinGGetDIBPointer** returns 0.

Comments

If it is supplied, the memory block indicated by *pHeader* must be large enough to hold a BITMAPINFOHEADER, which will be filled in by **WinGGetDIBPointer** with the attributes of the WinGDC. For DIB surfaces using BI_BITFIELDS compression, *pHeader* must contain space for an additional 3 DWORDs, in which the color masks for the DIB will be returned. See the Win32 API documentation for a discussion of BI_BITFIELDS DIBs.

If *hWinGBitmap* is not a WinGBitmap handle, this function will return 0 and **pHeader* will remain unchanged.

Maximizing Performance

WinGCreateBitmap uses or returns the information returned by **WinGGetDIBPointer** as part of the creation process. If possible, applications should store the data when the WinGBitmap is created rather than calling **WinGGetDIBPointer** every time the information is required.

The address of a WinGBitmap surface will remain the same for the life of the WinGBitmap.

In 16 bit code, the pointer to the bits can be accessed with 32 bit offsets in assembly, like all huge objects in Windows.

See Also

WinGCreateDC WinGCreateBitmap BITMAPINFO BITMAPINFOHEADER

WinGRecommendDIBFormat

Fills in the entries of a BITMAPINFO structure with values that will give maximum performance for memory-to-screen 1:1 identity blts using WinG.

BOOL WinGRecommendDIBFormat(BITMAPINFO far *_pHeader_)

Parameters

pHeader	Points to a BITMAPINFO structure to receive the recommended DIB format.

Return Value

Returns non-zero if successful. Otherwise, returns zero.

Comments

pHeader must point to enough memory to hold a BITMAPINFOHEADER and 3 DWORDs for BI_BITFIELDS. **WinGRecommendDIBFormat** will not return a color table.

For any combination of hardware and software, there will be one DIB format that WinG can copy fastest from memory to the screen. **WinGRecommendDIBFormat** returns the optimal format for blting DIBs with no stretching and no complex clipping using an identity palette. If your application makes heavy use of another blt type (such as 1:2 stretching or complex clipping regions), you may want to perform top-down and bottom-up timing comparisons yourself at run-time to determine the most efficient method for your needs.

In many cases, WinG will find that it can copy a DIB to the screen faster if the DIB is in top-down format rather than the usual bottom-up format. **WinGRecommendDIBFormat** will set the biHeight entry of the BITMAPINFOHEADER structure to -1 if this is the case, otherwise biHeight will be set to 1. See the DIB Orientation article for more information about these special DIBs.

Code that uses **WinGRecommendDIBFormat** should never assume that it will recommend an 8-bit format, as this may change depending on the run-time platform.

Example

See the WinGCreateBitmap API for sample code that uses **WinGRecommendDIBFormat**.

See Also

WinGCreateBitmap BITMAPINFO BITMAPINFOHEADER Code Samples

WinGSetDIBColorTable

Modifies the color table of the currently selected WinGBitmap in a WinGDC.

UINT WinGSetDIBColorTable(HDC *hWinGDC*, **UINT** *StartIndex*, **UINT** *NumberOfEntries*, **RGBQUAD far ***pColors **)**

Parameters

hWinGDC	Identifies the WinG device context whose color table should be modified.
StartIndex	Indicates the first palette entry to be changed.
NumberOfEntries	Indicates the number of palette entries to change.
pColors	Points to a buffer which contains the new color table values.

Return Value

Returns the number of palette entries modified in the specified device context or 0 if it failed.

Comments

The *pColors* buffer must hold at least *NumberOfEntries* RGBQUAD structures.

If you want to update the display immediately (for example, in palette animation), use AnimatePalette to modify the system palette and then call **WinGSetDIBColorTable** to match it or the WinGDC will be remapped when it is blted. See the Palette Animation With WinG article for more information and sample code that does this.

Note that *StartIndex* indicates an entry in a palette array, which is zero-based. *NumberOfEntries* indicates a one-based count. If *NumberOfEntries* is zero, no color table entries will be modified.

Maximizing Performance

It is not necessary to call **WinGSetDIBColorTable** every time you call AnimatePalette. Only call this API if you are about to blt and the destination palette has changed since the last call to **WinGSetDIBColorTable**.

Example

See the section titled Palette Animation With WinG for sample code and discussion of using **WinGSetDIBColorTable** to perform palette animation.

The PALANIM sample, in the SAMPLES\PALANIM subdirectory of the WinG Development Kit, performs simple palette animation and maintains an identity palette throughout.

See Also

WinGGetDIBColorTable WinGCreateBitmap Palette Animation With WinG

WinGStretchBlt

Copies an area from the source specified device context to an area of the destination device context, resizing if necessary to fill the destination rectangle. **WinGStretchBlt** is optimized to copy WinGDCs to display DCs. It does not operate on any other types of Device Contexts.

BOOL WinGStretchBlt(HDC *hdcDest*, **int** *nXOriginDest*, **int** *nYOriginDest*, **int** *nWidthDest*, **int** *nHeightDest*, **HDC** *hdcSrc*, **int** *nXOriginSrc*, **int** *nYOriginSrc*, **int** *nWidthSrc*, **int** *nHeightSrc*)

Parameters

hdcDest	Identifies the destination device context.
nXOriginDest	X coordinate of the upper-left corner of the destination rectangle in MM_TEXT client coordinates.
nYOriginDest	Y coordinate of the upper-left corner of the destination rectangle in MM_TEXT client coordinates.
nWidthDest	Width of the destination rectangle..
nHeightDest	Height of the destination rectangle..
hdcSrc	Identifies the source device context.
nXOriginSrc	X coordinate of the upper-left corner of the source rectangle in MM_TEXT client coordinates.
nYOriginSrc	Y coordinate of the upper-left corner of the source rectangle in MM_TEXT client coordinates.
nWidthSrc	Width of the source rectangle.
nHeightSrc	Height of the source rectangle.

Return Value

Returns non-zero if successful, otherwise returns zero.

Comments

WinGStretchBlt requires both DCs to use MM_TEXT mapping mode at the time of the call or the results may be unpredictable. At other times, any mapping mode may be used in either DC.

WinGBitBlt does not handle non-zero viewport or window origins set by SetViewportOrg or SetWindowOrg.

WinGStretchBlt uses the STRETCH_DELETESCANS mode when expanding or shrinking an image.

WinGStretchBlt will fail under Windows 3.1 or Win32s if the source DC does not contain a monochrome or an 8-bit-per-pixel WinGBitmap.

On Win32, calls to **WinGBStretchBlt** may be batched by the operating system. To ensure that a blt occurs immediately upon calling **WinGStretchBlt**, be sure to call GDIFlush just after every call to **WinGStretchBlt**.

WinGStretchBlt only copies from a WinGDC to a Display DC. To copy from a WinGDC to a WinGDC, use BitBlt. You can not copy from a Display DC to a WinGDC.

Maximizing Performance

You will get the highest performance from **WinGStretchBlt** if you use a WinGBitmap created from header information supplied by a call to WinGRecommendDIBFormat.

WinGStretchBlt is optimized for copying WinGDCs to the screen.

Aligning the destination rectangle to DWORD boundaries (4-pixel boundaries on an 8-bit display) can help **WinGStretchBlt** to achieve maximum speed.

Stretching by integer ratios is faster than arbitrary ratios. 1 to 2 stretching is fastest.

Clipping can slow **WinGStretchBlt** down. In general, don't select clipping regions into or blt outside the boundaries of the source or destination DCs and avoid blting to an overlapped window if possible.

See Also

WinGBitBlt WinGCreateDC WinGCreateBitmap WinGRecommendDIBFormat Maximizing Performance With WinG

WING_DITHER_TYPE

Dither types for halftone brushes.

WING_DITHER_TYPE

Values

WING_DISPERSED_4x4
WING_DISPERSED_8x8
WING_CLUSTERED_4x4

See Also

WinGCreateHalftoneBrush WinGCreateHalftonePalette Halftoning With WinG CUBE

```
HPALETTE CreateIdentityPalette(RGBQUAD aRGB[], int nColors)
{
    int i;
    struct {
        WORD Version;
        WORD NumberOfEntries;
        PALETTEENTRY aEntries[256];
    } Palette =
    {
        0x300,
        256
    };

    //*** Just use the screen DC where we need it
    HDC hdc = GetDC(NULL);

    //*** For SYSPAL_NOSTATIC, just copy the color table into
    //*** a PALETTEENTRY array and replace the first and last entries
    //*** with black and white
    if (GetSystemPaletteUse(hdc) == SYSPAL_NOSTATIC)
    {
        //*** Fill in the palette with the given values, marking each
        //*** as PC_NOCOLLAPSE
        for(i = 0; i < nColors; i++)
        {
            Palette.aEntries[i].peRed = aRGB[i].rgbRed;
            Palette.aEntries[i].peGreen = aRGB[i].rgbGreen;
            Palette.aEntries[i].peBlue = aRGB[i].rgbBlue;
            Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
        }

        //*** Mark any unused entries PC_NOCOLLAPSE
        for (; i < 256; ++i)
        {
            Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
        }

        //*** Make sure the last entry is white
        //*** This may replace an entry in the array!
        Palette.aEntries[255].peRed = 255;
        Palette.aEntries[255].peGreen = 255;
        Palette.aEntries[255].peBlue = 255;
        Palette.aEntries[255].peFlags = 0;

        //*** And the first is black
        //*** This may replace an entry in the array!
        Palette.aEntries[0].peRed = 0;
        Palette.aEntries[0].peGreen = 0;
        Palette.aEntries[0].peBlue = 0;
        Palette.aEntries[0].peFlags = 0;
    }
    else
    //*** For SYSPAL_STATIC, get the twenty static colors into
    //*** the array, then fill in the empty spaces with the
    //*** given color table
```

```
    {
        int nStaticColors;
        int nUsableColors;

        //*** Get the static colors from the system palette
        nStaticColors = GetDeviceCaps(hdc, NUMCOLORS);
        GetSystemPaletteEntries(hdc, 0, 256, Palette.aEntries);

        //*** Set the peFlags of the lower static colors to zero
        nStaticColors = nStaticColors / 2;
        for (i=0; i<nStaticColors; i++)
            Palette.aEntries[i].peFlags = 0;

        //*** Fill in the entries from the given color table
        nUsableColors = nColors - nStaticColors;
        for (; i<nUsableColors; i++)
        {
            Palette.aEntries[i].peRed = aRGB[i].rgbRed;
            Palette.aEntries[i].peGreen = aRGB[i].rgbGreen;
            Palette.aEntries[i].peBlue = aRGB[i].rgbBlue;
            Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
        }

        //*** Mark any empty entries as PC_NOCOLLAPSE
        for (; i<256 - nStaticColors; i++)
            Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;

        //*** Set the peFlags of the upper static colors to zero
        for (i = 256 - nStaticColors; i<256; i++)
            Palette.aEntries[i].peFlags = 0;
    }

    //*** Remember to release the DC!
    ReleaseDC(NULL, hdc);

    //*** Return the palette
    return CreatePalette((LOGPALETTE *)&Palette);
}
```

```
void ClearSystemPalette(void)
{
    //*** A dummy palette setup
    struct
    {
        WORD Version;
        WORD NumberOfEntries;
        PALETTEENTRY aEntries[256];
    } Palette =
    {
        0x300,
        256
    };

    HPALETTE ScreenPalette = 0;
    HDC ScreenDC;
    int Counter;

    //*** Reset everything in the system palette to black
    for(Counter = 0; Counter < 256; Counter++)
    {
        Palette.aEntries[Counter].peRed = 0;
        Palette.aEntries[Counter].peGreen = 0;
        Palette.aEntries[Counter].peBlue = 0;

        Palette.aEntries[Counter].peFlags = PC_NOCOLLAPSE;
    }

    //*** Create, select, realize, deselect, and delete the palette
    ScreenDC = GetDC(NULL);
    ScreenPalette = CreatePalette((LOGPALETTE *)&Palette);
    if (ScreenPalette)
    {
        ScreenPalette = SelectPalette(ScreenDC,ScreenPalette,FALSE);
        RealizePalette(ScreenDC);
        ScreenPalette = SelectPalette(ScreenDC,ScreenPalette,FALSE);
        DeleteObject(ScreenPalette);
    }
    ReleaseDC(NULL, ScreenDC);
}
```

```
HBITMAP ghBitmapMonochrome = 0;

HDC Create100x100WinGDC(void)
{
    HDC hWinGDC;
    HBITMAP hBitmapNew;
    struct {
        BITMAPINFOHEADER InfoHeader;
        RGBQUAD ColorTable[256];
    } Info;
    void far *pSurfaceBits;

    // Set up an optimal bitmap
    if (WinGRecommendDIBFormat((BITMAPINFO far *)&Info) == FALSE)
        return 0;

    // Set the width and height of the DIB but preserve the
    // sign of biHeight in case top-down DIBs are faster
    Info.InfoHeader.biHeight *= 100;
    Info.InfoHeader.biWidth = 100;

    //*** DON'T FORGET A COLOR TABLE! ***
    //*** COLOR TABLE CODE HERE ***

    // Create a WinGDC and Bitmap, then select away
    hWinGDC = WinGCreateDC();
    if (hWinGDC)
    {
        hBitmapNew = WinGCreateBitmap(hWinGDC,
            (BITMAPINFO far *)&Info, &pSurfaceBits);
        if (hBitmapNew)
        {
            ghBitmapMonochrome = (HBITMAP)SelectObject(hWinGDC,
                hBitmapNew);
        }
        else
        {
            DeleteDC(hWinGDC);
            hWinGDC = 0;
        }
    }

    return hWinGDC;
}

void Destroy100x100WinGDC(HDC hWinGDC)
{
    HBITMAP hBitmapOld;

    if (hWinGDC && ghBitmapMonochrome)
    {
        // Select the stock 1x1 monochrome bitmap back in
        hBitmapOld = (HBITMAP)SelectObject(hWinGDC,
        ghBitmapMonochrome);
        DeleteObject(hBitmapOld);
```

```
            DeleteDC(hWinGDC);
        }
    }
```

```
#define NumSysColors (sizeof(SysPalIndex)/sizeof(SysPalIndex[1]))
#define rgbBlack RGB(0,0,0)
#define rgbWhite RGB(255,255,255)

//*** These are the GetSysColor display element identifiers
static int SysPalIndex[] = {
     COLOR_ACTIVEBORDER,
     COLOR_ACTIVECAPTION,
     COLOR_APPWORKSPACE,
     COLOR_BACKGROUND,
     COLOR_BTNFACE,
     COLOR_BTNSHADOW,
     COLOR_BTNTEXT,
     COLOR_CAPTIONTEXT,
     COLOR_GRAYTEXT,
     COLOR_HIGHLIGHT,
     COLOR_HIGHLIGHTTEXT,
     COLOR_INACTIVEBORDER,
     COLOR_INACTIVECAPTION,
     COLOR_MENU,
     COLOR_MENUTEXT,
     COLOR_SCROLLBAR,
     COLOR_WINDOW,
     COLOR_WINDOWFRAME,
     COLOR_WINDOWTEXT
};

//*** This array translates the display elements to black and white
static COLORREF MonoColors[] = {
     rgbBlack,
     rgbWhite,
     rgbWhite,
     rgbWhite,
     rgbWhite,
     rgbBlack,
     rgbBlack,
     rgbBlack,
     rgbBlack,
     rgbBlack,
     rgbWhite,
     rgbWhite,
     rgbWhite,
     rgbWhite,
     rgbBlack,
     rgbWhite,
     rgbWhite,
     rgbBlack,
     rgbBlack
};

//*** This array holds the old color mapping so we can restore them
static COLORREF OldColors[NumSysColors];

//*** AppActivate sets the system palette use and
//*** remaps the system colors accordingly.
```

```
void AppActivate(BOOL fActive)
{
    HDC hdc;
    int i;

    //*** Just use the screen DC
    hdc = GetDC(NULL);

    //*** If the app is activating, save the current color mapping
    //*** and switch to SYSPAL_NOSTATIC
    if (fActive && GetSystemPaletteUse(hdc) == SYSPAL_STATIC)
    {
        //*** Store the current mapping
        for (i=0; i<NumSysColors; i++)
            OldColors[i] = GetSysColor(SysPalIndex[i]);

        //*** Switch to SYSPAL_NOSTATIC and remap the colors
        SetSystemPaletteUse(hdc, SYSPAL_NOSTATIC);
        SetSysColors(NumSysColors, SysPalIndex, MonoColors);
    }
    else if (!fActive)
    {
        //*** Switch back to SYSPAL_STATIC and the old mapping
        SetSystemPaletteUse(hdc, SYSPAL_STATIC);
        SetSysColors(NumSysColors, SysPalIndex, OldColors);
    }

    //*** Be sure to release the DC!
    ReleaseDC(NULL,hdc);
}
```

WinG Glossary

Bottom-Up DIB: A DIB in which the first scan line in memory corresponds to the bottommost scanline when the DIB is displayed. This is the standard Windows DIB format.

Color Table: The table of RGB color values referenced by a color-indexed DIB.

Dirty Rectangle Animation: A double-buffering technique in which only the areas on the screen which have changed are updated from frame to frame.

Double Buffering: An animation technique in which images are composed entirely off-screen then copied in whole or in part to the display.

Halftone Palette: An identity palette carefully filled with an array of colors optimized for dithering images to 8 bits per pixel.

Halftoning: A technique for simulating unavailable colors using special patterns of available colors. Also called dithering.

Identity Palette: A logical palette that is a 1:1 match to the system palette.

Logical Palette: A palette object created by an application using the CreatePalette function.

Palette: A table of RGB colors associated with a GDI Device Context.

Palette Animation: An animation technique in which palette entries are shifted to create the appearance of movement.

Static Colors: Reserved colors in the system palette that ordinarily can't be changed by an application. Under normal circumstances, twenty colors are so reserved.

System Colors: The colors used by Windows to draw captions, menu bars, text, and other Windows display elements.

System Palette: A copy of the hardware device palette maintained by the Palette Manager.

Top-Down DIB: A DIB in which the first scan line in memory corresponds to the topmost scanline when the DIB is displayed.

WinGBitmap: A special HBITMAP with a DIB as its drawing surface created for use in a WinGDC.

WinGDC: A device context with a DIB as its drawing surface.

A double-buffering technique in which only the areas on the screen which have changed are updated from frame to frame.

Reserved colors in the system palette that ordinarily can't be changed by an application. Under normal circumstances, twenty colors are so reserved.

The colors used by Windows to draw captions, menu bars, text, and other Windows display elements.

A copy of the hardware palette maintained by the Palette Manager.

A palette object created by an application using the **CreatePalette** function.

The table of RGB color values referenced by a color-indexed DIB.

A logical palette that is a 1:1 match to the system palette.

An identity palette carefully filled with an array of colors optimized for dithering images to 8 bits per pixel.

An animation technique in which palette entries are shifted to create the appearance of movement.

A standard Windows DIB in which the bottommost scanline is stored first in memory.

A special DIB in which the topmost scanline is stored first in memory.

Nirvana has ever been a difficult concept to explain. It is a state of mind achieved through diligence and meditation, a state of complete tranquility and awareness in which conscious thought ceases to exist as such, the final step in the struggle for the cessation of suffering.

Further Reading

The following collection of books, articles, and sample code may help clarify the use of DIBs, provide insight into custom drawing routines, or generally ease the transition from device-dependent bitmaps to WinGDCs. All of these are available on the Microsoft Developer Network CD or through the Microsoft Developer Relations Group. Some are included with the Windows SDK.

Foley, van Dam, Feiner, and Hughes, Computer Graphics: Principles and Practice, Second Edition, Addison-Wesley, 1991

Gery, Ron, "Using DIBs with Palettes," Microsoft Technical Article, 3 March 1992

Gery, Ron, "DIBs and Their Use," Microsoft Technical Article, 20 March 1992

Gery, Ron, "The Palette Manager: How and Why," Microsoft Technical Article, 23 March 1992

Petzold, Charles, "The Device-Independent Bitmap (DIB)," Programming Windows 3.1, Microsoft Press, 1992, pp. 607-619

Rodent, Herman, "Animation In Windows," Microsoft Technical Article, 28 April 1993

"How To Use a DIB Stored as a Windows Resource," Microsoft PSS Article Q67883, 26 April 1993

"Multimedia Video Techniques," Microsoft Technical Article, 20 March 1992

Windows 3.1 Software Development Kit samples: DIBIT, DIBVIEW, CROPDIB, WINCAP, SHOWDIB, FADE

Microsoft Technical Samples, TRIQ, draws triangles or boxes directly into device-independent bitmap memory.