The Microsoft Windows Device Driver Writer's Release Notes

Microsoft Corporation
September, 1988

A Comment On These Notes

These release notes are a partial release of a manual that will be included in future versions of the **Windows OEM/Device Driver Kit**. Although they are incomplete, they do contain helpful information for device driver writers. These notes should be the final authority when they dispute the **Windows Adaptation Guide**. The complete set of notes comprising the finished manual will be available via the **OnLine** service by December 1st, 1988. You should check the bulletin board area after that date. Some references to unfinished chapters are made in these notes. Please excuse them and utilize the information currently included to help you in your development efforts.

How to Use These Notes

These notes serve to update, amplify, and sometimes replace the information in the **Windows Adaptation Guide**. It serves as a tutorial for those of you starting new driver development and as a guide to "tricks of the trade" for those of you wondering why your driver doesn't look or perform like the standard Windows drivers for the EGA and VGA. The information in the booklet is mostly geared toward display driver writers who are trying to adapt Windows for "non-standard" displays such as those based on a coprocessor (TI 34010, Intel 82786, AMD QPDM to mention a few). However, printer driver writers will want to use the booklet as their definitive guide for Windows Escape/Control functions.

The best way to approach this kit is to follow step-by-step the procedures in this booklet and write the modules in the order and using the methods specified here.

What This Kit Is and Isn't

This kit is for the sole use of producing Microsoft Windows Device Drivers or OEM Adaptations. It has everything that you need (besides the assembler and compiler) that you need to build and perform basic functionality tests for your driver. It does not contain the tools necesary to build a Microsoft Windows application, nor does it include the full Microsoft Windows retail kit. You can buy these other materials through your Microsoft dealer.

You cannot develop the virtual device drivers for **Windows/386** by using this kit. However, it is absolutely necesary for you to first write a **Windows/286** driver in order to even start writing a **Windows/386** driver. Therefore, in order to achieve your ultimate goal of adapting **Windows/386** to your device, you must use this kit to adapt **Windows/286** first, and then contact Microsoft to obtain the **Windows/386 Binary Adaptation Kit.**

What You'll Need

This kit does not contain **MASM v5.10** which is an absolute necessity for producing any Windows driver. If you're only producing a display, mouse or keyboard driver, **MASM** will be the only additional tool that you need. If you're going to be producing a printer driver, you'll probably want to write it in **C** (just like the sample drivers included with this kit). In that case, you'll need **C v5.10** as well as **MASM.**

As mentioned above, this kit does not contain a retail **Windows/286** kit. You'll need to obtain one of these to perform final testing of your driver. This kit does contain the so-called **debug version of Windows.** However, you will be able only to test the most basic driver functionality using it. In addition, it's important that you invest in some standard Windows applications since these test the most esoteric areas of your driver. Recommended applications include **Micrografx's Designer, Microsoft's Excel, and Aldus's PageMaker.** You should also have a Windows/286-compatible printer such as a **PCL** or **PostScript** Printer.

Step One: Setup Your Development Environment

One of the most important steps in creating a successful Windows driver is to create a good standard environment. Everything that you need (except the assembler and/or compiler) to create your environment is included in this kit.

It is best to start with a virgin machine. Since you'll be compiling, assembling, and debugging some pretty hefty-sized files, you'd best use a fast (16 MHz or above) development machine. In any case, make sure that you purge your development machine of any old versions of assemblers and compilers. Also, it is an absolute **MUST** to get rid of old versions of Windows-specific tools such as old versions of **RC.EXE, RCPP.EXE, LINK4.EXE, LINK.EXE, MAPSYM.EXE, NEWFON.EXE** and **SYMDEB.EXE!** In addition, purge all libraries and include files that you may have from old Windows Device Driver Kits and/or Software Development Kits. Also, make sure you get rid of old **.ICO, .BMP, .CUR, .FNT, and .FON** files. Some of the drivers that we've received from OEMs **still** contain the old icons from Windows 1.xx!!

Now you're ready to install the device driver kit. In order to use the provided debug version of Windows, you'll need an **EGA Display Card with 256K of memory, a Microsoft Mouse, and at least 512K of memory.** First, you should just bring Windows up with the default debugging version. Create a subdirectory on your hard disk and copy the entire contents from the **WINBOOT** directory on **disk 1** to your hard disk. Now, enter the command **KERNEL**. The debugging version of Windows should start running. If you have some Windows programs, you can run them at this time and become familiar with how the **EGA** display operates with Windows. Especially note the speed in which text is put onto the screen. Your text routines should attempt to produce text at least as fast as the **EGA** driver's. There's a subdirectory on **disk 2** called **TOYS** which contains some trivial Windows programs for your testing and enjoyment. A good program to run is **MUTEXT.EXE** which demonstrates the ability of the driver to draw text onto the screen quickly.

Now that you've determined that the test environment is set up correctly, it's time to install the tools. Follow these steps:

1) Install the "DOS-mode" **MASM v5.10** onto your hard disk. You'll need only the following files:

MASM.EXE
MAKE.EXE
CREF.EXE (optional -- if you like making cross reference files)

2) If you're writing a printer driver, or if you make the fatal mistake of writing your display driver in **C**, install **Microsoft C v5.10** on your hard disk. You'll probably only need the small model libraries.

3) Now, copy the files from **the disk 1, BLDTOOLS** directory. Make sure that you have no other versions of **LINK.EXE, LINK4.EXE, NEWFON.EXE, RC.EXE and RCPP.EXE** existing on your hard disk. The **LINK.EXE** which comes in this kit supercedes both **LINK4.EXE** and versions of **LINK.EXE** that come with the various compilers. The new version of **LINK4** will work perfectly for building Windows drivers as well as for linking your other applications.

4) Now, copy the file **SWINLIBC.LIB** from **the disk 1 LIB** directory. This **.LIB** file is the only one which is needed by all of the drivers that you'll be building. For certain printer drivers and other OEM files, you may need other libraries. These will be provided in the directories relevant to the driver that needs the library. Also, copy the include files from the **disk 1 INCLUDE** directory.

5) Make sure that you've got your **FILES=20 and BUFFERS=40** lines set in **CONFIG.SYS.** Also, make sure that you've got appropriate paths set to the tools in **AUTOEXEC.BAT** (Here's a hint for **MASM** users: For the optimum compile times and the best error checking put the following line in your **AUTOEXEC.BAT** file:

SET MASM=-z-t-n-W2-b63

New to **MASM** is the **-W2** switch. I've found this switch invaluable in getting rid of those pesty **NOPs** caused by using a **JMP** instruction instead of the **JMP SHORT** instruction. The meaning of all of these switches is documented in your **MASM Reference Guide** provided with **v5.10**.). Now you'll want to include the **SET LIB=** and **SET INCLUDE=** lines in your **AUTOEXEC.BAT** file. These tell the assembler, compiler, resource compiler and linker where to find your include and library files. See the compiler/linker documentation for specifics on the syntax of these **SET** commands.

THIS COMPLETES THE SETUP OF YOUR DEVELOPMENT ENVIRONMENT!

Step Two: Choose the Sample Driver That You Want to Base Your Driver On

It's important that you start writing your device driver from a proper model. For printer drivers, this is usually an easy decision. Most printers are based on either dumb raster technology or a high level command language such as **PostScript** or **PCL** (the technology used by the **HP LaserJet** printer). Sample drivers for some of these printer technologies (as well as a sample **plotter** driver) are provided on **disks 8 & 9.**

Display drivers are a bit more complex. There are many displays which are just flat, one plane, one bit per pixel monochrome bitmaps. You are best advised to use the **EGA Monochrome** driver found on **disk 5**. If your display is similar in architecture to the **VGA**, you'll use the **VGA** driver found on **disk 6**. Other monochrome displays are interlaced similar to the **CGA or Hercules Monochrome** display. In that case, use the sample driver on **disk 2 or 3**. For all of the above drivers, you probably need to change only a few equates in such files as **CURSOR.INC, and XXXXX.ASM** where **XXXXX** is the name of the particular driver. You may have to add some code to page memory segments in and out of PC memory space. Also, if your raster display has a hardware cursor (as many do), you will want to modify the code dealing with the cursor as about 1/3 of the time spent in the display driver is spent updating the cursor.

Other types of devices are based on some sort of co-processor or hardware engine such as the **Texas Instruments 34010**. Unfortunately, at the time of shipment, no sample sources are available for such a device. These sources should become available in the near future. Please watch the **OnLine** bulletin board for announcements. Queries on writing drivers for these devices can always be directed to the Windows device driver specialist via the **OnLine Service Request** system.

Using the Include Files

There are at least three of the include files that you'll be including in one or more of your assembly language modules. These are:

CMACROS.INC
GDIDEFS.INC
WINDEFS.INC

The most important include file is **CMACROS.INC**. It is basically as set of macros written to be compatible with **MASM v5.10** which take care of many of the housekeeping tasks necesary for setting up stack frames, calling between modules written in **C** and those written in assembly, and defining local and global variables. **CMACROS.INC** has no comments in it. If you try to read the file, you'll find that it's kind of a jumbled mess. Therefore, let's explain the some of the functions of the **CMACROS** file that you'll be using for the Windows device driver. It is important that you now open up and read **chapter 7 of the Microsoft Windows Programming Tools** manual which is included in the kit. This is a tutorial and reference on the use of the **CMACROS**.

Setting up of stack frames is the first concept that we need to discuss. The device driver is always called from the portion of Windows called **GDI**. GDI is completely device independent and basically generates the lines, characters, and bitmaps that the device driver is required to draw. GDI passes the parameters for each drawing command to the device driver on the stack. All calls from GDI are **far calls**. When the driver is called, the parameters for the call have been pushed onto the stack at offset **ss:[bp+6]**. How does one retrieve these parameters in an clear, easily documented way? Well, the **CMACROS** does this automatically for you. Following is a skeleton of the **BitBlt** assembly language file which shows the typical use of the **CMACROS** in a device driver.

```
page    ,132
title   BitBlt Skeleton
;
;
.xlist
memS                    ;use small model (this is the default)
?PLM=1                  ;use Pascal calling (this is the default)
?WIN=1                  ;generate prolog-epilog code (this is the default)
?CHKSTK=1               ;call CHKSTK for all procs in this file
include         CMACROS.INC
.list
;
;
sBegin          Data
;
;Define a public data item called MyData:
;
globalB         MyData,0,2
;
;Define a private data item called BitBltData:
;
staticW         BitBltData,0,1
sEnd            Data
;
page
sBegin          Code
assumes         cs,Code
assumes         ds,Data
;
cProc           BitBlt,<FAR,PUBLIC,WIN,PASCAL>,<si,di>
        parmD           lpDstDev
        parmW           DstxOrg
        parmW           DstyOrg
        parmD           lpSrcDev
        parmW           SrcxOrg
        parmW           SrcyOrg
        parmW           xExt
        parmW           yExt
        parmD           Rop
        parmD           lpPBrush
        parmD           lpDrawMode
;
        localB          LocalData
        localW          LocalWordData
        localV          Local20BytesofData,20
cBegin
|
|       Your code goes here.
|
cEnd
;
;
sEnd    Code
end
```

You should refer to chapter 7 of the programming tools manual to make sure that you understand each of the terms used in this example. However, there are a number of additional **CMACROS** features that are not documented in **Chapter 7**. We'll discuss them here.

WIN and PASCAL as used in our example (on the **cProc** line), allow you to overrule the **?PLM and ?WIN** flags. In our example, they of course are redundant. However, the example drivers included in the kit sometimes use them and it is certainly not harmful to use them.

NODATA can be used as a keyword in the **cProc** line. Normally, the prolog and epilog code set the **DS** register to point to the default **Data** segment whenever the process type is **FAR**. This can result in unwanted destruction of the **AX** register since **AX** is used to setup **DS**. It's also wasteful sometimes to have the prolog and epilog code setup **DS**. Therefore, you can use the **NODATA** keyword to prevent the prolog and epilog code from modifying **DS**.

Example:

```
cProc       EnableCursor,<FAR,PUBLIC,WIN,PASCAL,NODATA>,<es>
```

If you are assembling your progam using **MASM's** case sensitivity switch (-MI), some of the names documented in **chapter 7** will not work. Make sure that you use the following syntax for the default segment names:

```
Code        Data        Stack
```

also:

```
CodeOFFSET DataOFFSET StackOFFSET
```

In addition, the **arg** command should be in lower case.

There are a couple of tricks that you can use to define multiple modules using the same stack frame. Let's say that my **BitBlt** process should be logically divided into two modules, one containing hardware independent code, the other containing hardware dependent stuff. Both of them need to share the same variables that are passed on the stack and defined as variable names by **CMACROS.** You'd might think that you'd have to use the following calling sequence in module one:

```
arg     lpDstDev
arg     DstxOrg
arg     DstyOrg
etc.....
cCall   BitBltModuleTwo
```

and the following "receiving sequence" in module two:

```
cProc   BitBltModuleTwo<FAR,PUBLIC,WIN,PASCAL>
        parmD lpDstDev
        parmWDstxOrg
        parmWDstyOrg
etc.....
cBegin
```

This inter-module calling sequence is extremely wasteful both in terms of the setting up of the stack frame in the caller and the far call. Instead, **CMACROS** provides the programmer with a neat trick. What you do is create a dummy **cProc** header in all of the modules that you want to share the same stack frame (except of course the calling module). Then, you use the **<nogen>** qualifier on **cBegin**. **CMACROS** will then create the equates for the stack frame but will not generate any code in the dummy process. Then, you can make near calls to any sub-processes without any wastefulness. Here's an example:

In File 1

```
externNP      Module2
;
cProc  Module1,<FAR,PUBLIC,WIN,PASCAL>,<si,di>
       parmD Param1
       parmWParam2
       parmB Param3
cBegin
|
|      some code.
|
cCall  Module2
|
|
|
cEnd
```

In File 2

```
cProc  ModuleFamilyDummy,<FAR,PUBLIC,WIN,PASCAL>
       parmD Param1
       parmWParam2
       parmB Param3
cBegin        <nogen>          ;don't generate any code -- just equate stack
                               ;offsets for the parameters to symbolic names
cEnd          <nogen>          ;don't generate any code -- just end the process
;
;
cProc  Module2,<NEAR,PUBLIC>
;
cBegin
;
;Now Module2 will be able to use the same stack frame variable as Module1.  The
;far call has been avoided as well as the pushing of the stack frame in Module1.
;
cEnd
```

This concludes the discussion of the **CMACROS.**

Now let's discuss **GDIDEFS.INC**. This file allows you to refer to symbolic constants and structures by their Windows standard names. It is probably good practice to do this. In order to shorten the assembly time and cross reference lists, you can selectively include parts of **GDIDEFS**. You do this by defining equates which tell the assembler which parts of **GDIDEFS** to include. These are listed here:

```
incLogical equ 1        ;include logical pen, brush, and font definitions
incDevice equ 1         ;include the symbolic names for GDIInfo definitions
incFont equ 1           ;include the FontInfo and Textform definitions
incDrawMode equ 1       ;include the DrawMode data strucure definitions
incOutput equ 1         ;include the output style constants
incControl equ 1        ;include the escape number definitions
```

Lastly, we discuss the **WINDEFS.INC** file. This include file contains two very useful macro which is used to turn off the mouse interrupts. The **EnterCrit and LeaveCrit** macros should be used whenver you don't want an asynchronous interrupt reentering an area of code that Windows is executing. The only asynchronous interrupt that occurs in the display driver is the mouse interrupt. Using the mouse interrupt as an example, it is possible that the mouse can generate interrupts faster than your mouse handling code can process them. It's therefore likely that you could be updating a mouse coordinate when another mouse coordinate comes along wanting to be serviced! This would result in an incredible jumble of mouse coordinates. Therefore, as you're about to update your mouse coordinates, use the **EnterCrit** macro. This will correctly stop the mouse interrupts from occurring. After you safely have the new mouse coordinates, you can use the **LeaveCrit** macro to reallow interrupts. Do not use simple **CLI & STI** instructions to accomplish these tasks since they will not correctly restore the states of the flags and interrupts.

Step Three: Fill Out the GDIInfo Data Structure

The first step in producing a successful Windows driver is to properly fill out the **GDIInfo** data structure. You should find the file in your model driver where this structure is located. The **GDIInfo** structure tells Windows about the capabilities of your device. It also tells Windows applications how to expand and contract their bitmaps in order to look WYSIWG on your display. It is important that you follow the calculations in this section exactly in order to assure yourself of a consistent looking display. The structure looks like the following:

Offset	Contents
0	Version number -- always put the number 0201h here
2	Device type -- Plotter=0, Display=1, Printer=2, (other types are found in **GDIDEFS.INC**)
4	Width of display in mm (see below)
6	Height of display in mm (see below)
8	X-resolution in pixels
10	Y-resolution in scanlines
12	Bits per pixel (see below)
14	Number of planes (see below)
16	Number of brushes (see explanation at EnumObj)
18	Number of pens (see explanation at EnumObj)
20	Reserved (must be 0)
22	Number of fonts (see explanation at EnumObj)
24	Number of true, non-dithered colors on device
26	Number of bytes required for PDEVICE structure (see explanation of PDEVICE structure)
28	Curves capabilities (see below)
30	Polyline drawing capabilities (see below)
32	Polygonal capabilities (see below)
34	Text drawing capabilities (see below)
36	Clipping ability for shape drawing **only** (see below)
38	Miscallaneous capabilities (see below)
40	X aspect
42	Y aspect
44	Hypotnuse of X & Y aspect
46	Length of patterned line segments
48	Metric Lo-Res Viewport (see below)
56	Metric Hi-Res Viewport (see below)
64	English Lo-Res Viewport (see below)
72	English Hi-Res Viewport (see below)
80	TWIP Viewport (see below)
88	Pixels per inch in X (see below)
90	Pixels per inch in Y (see below)
92	DC Management (Always 4 for displays)
94	5 words reserved (**must be 0**)

Let's begin the discussion of the **GDIInfo** data structure by talking about the various "screen metrics" entries (such as width and height in mm). By looking at the description here and in the Adaptation Guide you might think that you would obtain the width in mm by getting out your verniercalipers or ruler and measuring your screen size. **ABSOLUTELY NOT SO!!!!** These numbers are intimately tied up with the screen resolution and aspect and the fonts that you wish to use. Windows is supplied with some raster fonts called Courier, Helvetica, and Times Roman. There are five versions of these fonts currently supplied. They are:

COURA, HELVA, TMSRA which are for a 2 to 1, low resolution display such as the **CGA.** (Actual pixels per inch = 96 in X and 48 in Y).

COURB, HELVB, TMSRB which are for a 1.33 to 1 display such as the **EGA.** (Actual pixels per inch = 96 in X and 72 in Y).

COURC, HELVC, TMSRC which are for a 1 to 1.2 display.
(Actual pixels per inch = 60 in X and 72 in Y).

COURD, HELVD, TMSRD which are for a 1.66 to 1 display.
(Actual pixels per inch = 120 in X and 72 in Y).

COURE, HELVE, TMSRE which are for a 1 to 1 display such as the **VGA and 8514/A.** (Actual pixels per inch = 96 in X and 96 in Y).

Now, in order for the Windows font mapper to match one of these default fonts to your display, you have to 'fix up' (in other words "lie" about) the numbers used in the various 'screen metrics' entries in **GDIInfo.** The first numbers that you must decide upon are the **PIXELS PER INCH IN X & Y** (offsets 88 & 90 in the structure). If you want to use one of the default fonts in programs such as Write and PageMaker, you should fill in the **PIXELS PER INCH** entries with the 'actual pixels per inch' entry in the above list of fonts. For example, if the target display card has square pixels, I'd use the entry for the 'E' fonts and put a 96 in offset 88 and a 96 in offset 90 of the structure, even though the actual pixels per inch might not be 96 by 96.

Once we've determined the "logical" pixels per inch, we can easily calculate the width and height of the screen in mm. The equation for calculating the width in mm is:

$$\frac{\textbf{X-resolution in pixels (offset 8)}}{\textbf{Pixels per inch in X (offset 88)}} \textbf{ * 25.4 mm per inch}$$

The height in mm should be similarly calculated as:

$$\frac{\textbf{Y-resolution in scanlines (offset 10)}}{\textbf{Pixels per inch in Y (offset 90)}} \textbf{ * 25.4 mm per inch}$$

You should feel free to round these values to nice even numbers.

Now it's time to calculate the **Metric LoRes, Metric HiRes, English LoRes, English HiRes, and Twip window and viewport.** What are these used for? Programs such as PageMaker rely upon these numbers to produce printer output that has the same spacing proportional to the screen. By using these numbers, PageMaker can show a border which will be proportionately the same size on the printer as it is on the screen. All of these ratios must be the same because it might be preferable for an application to use the metric system rather than the inches/feet (English) system for its calculations.

Windows Write in fact allows the user to choose whether he wants to express his border widths in mm or inches. Thus, it's up to the device driver to provide the correct numbers here.

The **Metric LoRes Viewport** consists of four word lengthed entries:

offset 48	**Width in mm * 10**
offset 50	**Height in mm * 10**
offset 52	**X-resolution in pixels**
offset 54	**- (Y-resolution in scanlines)**

The **Metric HiRes Viewport** consists of four word lengthed entries:

offset 56	**Width in mm * 100**
offset 58	**Height in mm * 100**
offset 60	**X-resolution in pixels**
offset 62	**- (Y-resolution in scanlines)**

The **English LoRes Viewport** consists of four word lengthed entries:

offset 64	**Width in mm * 1,000**
offset 66	**Height in mm * 1,000**
offset 68	**X-resolution in pixels * 254**
offset 70	**- (Y-resolution in scanlines * 254)**

The **English HiRes Viewport** consists of four word-lengthed entries:

offset 72	**Width in mm * 10,000**
offset 74	**Height in mm * 10,000**
offset 76	**X-resolution in pixels * 254**
offset 78	**- (Y-resolution in scanlines * 254)**

The **TWIP (A TWIP is a printer's point = 1/72 of an inch) Viewport** consists of four word lengthed entries:

offset 80	**Width in mm * 14,400**
offset 82	**Height in mm * 14,400**
offset 84	**X-resolution in pixels * 254**
offset 86	**- (Y-resolution in scanlines * 254)**

Unfortunately, the calculations that Windows performs on these viewports is a word lengthed, signed calculation. Therefore the numbers that you calculate cannot be bigger than 32K (32,768). Unfortunately, if your screen is larger than a few inches wide, you'll exceed this limit when you start calculating the English viewports (you may even exceed it on the Metric viewports). Luckily, there's a solution to this. This is to **scale** the calculated values down by dividing by some fixed amount . You must divide the **width in mm and X-resolution** by the same amount and the **height in mm and Y-resolution** by the same amount. For example, let's say that my **TWIP** calculation came out thus:

Width in mm = 280
Height in mm = 280
X-resolution = 1024
Y-resolution = 768

280 * 14,400 = 4,032,000 = Width in mm
210 * 14,400 = 3,024,000 = Height in mm
1024 * 254 = 260,096 = X-resolution
- (768 * 254) = - 195,072 = Y-resolution

Divisor that'll give me a number < 32K for the width/X-resolution pair is 512.
Divisor that'll give me a number < 32K for the height/Y-resolution pair is 384.

So.....

The numbers that I'd put in my GDIInfo Data Structure are:

4,032,000 / 512 = 7875 = Width in mm
3,024,000 / 384 = 7875 = Height in mm
260,096 / 512 = 508 = X-resolution
- (195,072) / 384) = -508 = Y-resolution

Our last metric calculations are the X & Y aspect ratios. These are based on the aspect ratios that you must know for your display cards. That is, you must know whether your display card is a 1 to 1 aspect ratio (square pixels) , a 1.33 to 1 aspect ratio (like the EGA) or some other aspect ratio. You have already decided this when you chose the font metric (which set of COUR, HELV, TRMSE) to use. What you must do is find whole numbers which are less than 100 which will produce a whole number hypotnuse when put through the Pythagorean Theorem equation. This equation is:

$$a^2 + b^2 = c^2$$

Where c^2 is the hypotnuse. Here are two examples:

For a square pixel display use 10 for both **a** & **b**. Then:

$$10^2 + 10^2 = 200$$

The square root of 200 rounded to a whole number is 14. Thus, for this example, we would put:

X aspect (offset 40) = 10
Y aspect (offset 42) = 10
Hypotnuse of X & Y aspect (offset 44) = 14

Our next example is for a 1.33 to 1 display. We choose **48** for the X-aspect, **38** for the Y-aspect. The calculation gives a hypotnuse (rounded to a whole number) of **61.**

The last metric is the length of patterned (synonomous with **styled**) line segments. This is simply calculated as **2 * Hypotnuse**. Windows uses this number to make patterned lines that it draws into bitmaps and onto displays look correct and consistent on different displays and printers.

Now let's discuss the issue of **bit planes and bits per pixel**. The EGA and VGA drivers included in this kit are **planar** in nature. That is, they put the number **3** in the **GDIInfo Number of Planes (offset 14)** entry. This means that they have 3 planes and are therfore capable of 8 colors true, non-dithered colors (two colors per plane to the third power = 8). They also put the number **1** in the **GDIInfo Bits Per Pixel (offset 12)** entry. This was done by the author of the VGA driver because that device requires one pass for each plane when writing onto the screen. Windows will therefore allocate enough memory for each of the three planes when it allocates bitmaps. This is somewhat inefficient because each plane of the bitmap must be allocated on a 16 byte boundry. Therefore, a few bytes are wasted at the end of each plane. Many of today's more sophisticated displays allow you to draw onto them using pixel color values, in other words, to write all of their planes in one pass. These devices are called **packed pixel** devices and include the 8514/A and TI 34010 based devices. The **GDIInfo Number of Planes** entry for the 8514/A driver is **1** meaning that Windows will allocate a single plane bitmap for drawing. However, the **Bits Per Pixel** entry in the structure has the number **8** indicating that it takes 8 bits (one byte) to represent each pixel on the display. The 8514/A driver is therefore capable of displaying 2^8**=256** colors on the screen. The 8514/A display device allows addressing of the board in either **planar** or **packed pixel** mode. It is just more efficient for both Windows and the driver writer to use **packed pixel** mode whenever possible. In earlier versions of Windows and **Excel**, there were bugs in handling packed pixel drivers. These have thankfully been fixed and now a device driver writer can use the packed pixel mode with confidence. You should always have the "other" mode set to 1. If you're using planar mode, set bits per pixel to 1. If you're using packed pixel mode, set number of planes to 1.

Now it's time to decide what capabilities that you want your driver to support. The major decisions that you'll have to make are on which shapes that you'll choose to draw using your devices hardware. Windows requires only that you be able to draw solid or patterned single pixel wide horizontal lines (**scanlines**) and solid single pixel wide lines in any direction (**polylines**). However, with the advent of new hardware, it may be faster and prettier to use your display's advanced hardware to draw such shapes as circles, ellipses, and alternate-fill polygons. The curves, polylines, and filled figure capabilities allow you to tell Windows that you want it to call your **Output** routine to give you a chance to draw the figure with your hardware. Windows 2.00 and above gives you further flexibility in supporting **Output** shapes. Let's say that you can draw a polygon with 256 vertices but not with 257. Or, perhaps you can draw ellipses to your screen but do not wish to duplicate the algorithm for ellipse drawing into main memory bitmaps (an unfortunate requirement of Windows, any figure that you claim that you can draw onto the screen must also be able to be drawn to a main memory bitmap). You would then say that you have the abilities to draw polygons and ellipses. Then, at the time when your driver's **Output** module is called, you have the option of returning a failure return code and having Windows synthesize the figure for you with the basic building blocks (**scanlines, solid polylines, and pixels**).

Let's list the **Output** capabilities and their bit numbers now.

Curves capabilities (offset 28):

Bit	means I can draw
Bit 0 set	--> circles
Bit 1 set	--> pie wedges
Bit 2 set	--> chord arcs
Bit 3 set	--> ellipses
Bit 4 set	--> wide solid curved borders around curve figures
Bit 5 set	--> patterned lines surrounding curves
Bit 6 set	--> wide patterned curved borders around curve figures
Bit 7 set	--> can fill the interiors of curves
All other bits in the word should be set to 0	

Discussion:

It is rather worthless in terms of time to develop and size or driver to support circles and wide borders. Windows will draw a circle with an ellipse in place of a circle if it's not supported by the driver. Unless you can draw circles but not ellipses or your device draws circles significantly faster than ellipses, it is useless to support them. Wide borders (both solid and patterned) are just as efficiently drawn by Windows using an alternate fill polygon as if the driver supported them correctly. Therefore, we advise having the ability to draw arcs and ellipses if possible. Also set the interiors bit if your device can fill the ellipse.

Polyline Drawing capabilities (offset 30):

Bit	means I can draw

Bit 0 is reserved and should be 0
Bit 1 set --> polylines (all display drivers are required to set this bit)
Bit 2 set --> markers
Bit 3 set --> polymarkers
Bit 4 set --> wide lines
Bit 5 set --> patterned lines
Bit 6 set --> wide patterned lines
Bit 7 set --> can fill the interiors of wide lines
All other bits in the word should be set to 0

Discussion:

Do not support markers or polymarkers. They are misdocumented in the Adaptation Guide and are not currently supported by GDI. The support of wide lines is desirable if the device supports them. Remember though that you cannot "fail" on drawing wide lines into a main memory bitmap. If your device supports alternate fill polygons, then wide lines are somewhat redundant as Windows can efficiently use the polygons to create wide lines. If you support styled lines, you'd best make sure that the length of the line segment that your hardware draws are the same as those at offset 46 of **GDIInfo**. Also, if you support wide or styled polylines you will have to support them to both main memory bitmaps and to your screen. **Windows will not let you return a failure from Output for any of the line styles! If you decide to support them, you must support them to both main memory bitmaps and screen bitmaps!** If your hardware supports styled lines it probably is worth the effort to implement them. However, styled and wide lines are used rather infrequently. Windows does a fine job of synthesizing them for you when they are used (using pixel draws -- it's slow but it works). In the sample drivers, wide lines aren't supported although styled lines are.

Polygonal Drawing capabilities (offset 32):

Bit	means I can draw
Bit 0 set	--> alternate fill polygons
Bit 1 set	--> rectangles
Bit 2 set	--> winding number fill polygons
Bit 3 set	--> scanlines (all display drivers are requires to set this bit)
Bit 4 set	--> wide borders around polygonal figures
Bit 5 set	--> patterned borders around polygonal figures
Bit 6 set	--> wide patterned borders around polygonal figures
Bit 7 set	--> can fill the interiors of polygonal figures
All other bits in the word should be set to 0	

Discussion:

Again, the support of wide borders is not recommended since Windows will use alternate fill polygons (the kind that most hardware supports) to produce these borders. There are a few caveats in the support of rectangles and polygons. Make sure your read the appropriate sections about them in this booklet. No driver has been written which supports the winding number fill polygons. It is not known if they work or not. Thus, you may not wish to support them. Most hardware is incapable of doing winding number fill polygons anyway as it's a fairly complex algorithm. Patterned border support is a pretty good idea here if the hardware supports them. Great speed increases are possible for polygonal patterned borders.

Clipping capabilities (offset 36):

If your device can "scissor clip" to a rectanglular region then you should put a **1** here and support clipping in your driver's **Output** module. This capability is only used by Windows to determine whether you can clip **Output** shapes. Text **must** be clipped to a pixel boundry by the driver no matter what is placed in this field. No other routines besides **Output and StrBlt (ExtTextOut)** require any clipping.

Text Drawing Capabilities (offset 34):

Although the Windows Adaptation Guide gives all sorts of options for character drawing capabilities, you'd best stick to the ones used by the sample display drivers. Following are the bits fields. Ones marked with a * are the ones that you should set (if your device has the capability) in your driver. Most of the others either don't work or haven't been tested.

	Bit	Capability
*	0	Can draw text with pixel justification. (required)
	1	Can do everything that bit 0 set can and can also rotate text.
*	2	Can clip to a pixel boundry (required for displays)
	3	Can rotate text 90 degrees (doesn't work)
	4	Can rotate text to any angle (doesn't work)
	5	Can scale font in X & Y directions independently
	6	Reserved
	7	Can double the size of the font
	8	Can scale the font by any integer size (3X, 4X ...)
	9	Can scale the font by any amount
*	10	Can bold the font
*	11	Can italisize the font
*	12	Can underline the font
*	13	Can do a "strike out" on the font
*	14	Can draw with raster fonts (required for displays)
	15	Can draw with vector fonts (don't do this on displays)

Miscallaneous Capabilities (offset 38)

Many of these capabilities are really not optional for displays. For example, some Windows 2.xx drivers that Microsoft has evaluated don't support huge (> 64K) bitmaps. Many applications such as **Paint** depend on this support and will not work correctly if the display driver doesn't handle them. Also, if you expect to support **PageMaker, Excel, and Write** you'd better support ExtTextOut. These are **not** optional capabilities! Windows will not "cover" for you if you don't support them. A quick note on the FastBorder capability. Since this shares a bit with ExtTextOut capability, you must set the bit. However, you can return a failure code from FastBorder for which Windows will compensate. The 8514/A driver does this. Here are the bit fields. The ones marked with a * are required for display drivers:

	Bit	Capability
*	0	Can do BitBlt
	1	Requires GDI banding support (printers only)
	2	Requires GDI scaling support (printers only)
*	3	Supports huge (multi-segment) bitmaps
*	4	Supports ExtTextOut and FastBorder
	5	State block support (printers only)
	6	SaveScreenBitmap capability (**strongly** recommended for displays)

This wraps up our discussion of the GDIInfo Data Structure.

Step Four: The Enable and Disable Modules

Here are the call parameters for **Enable** and **Disable**:

```
cProc       Enable,<FAR,PUBLIC,WIN,PASCAL>,<si,di>
        parmD       lpDstDev
        parmW       Style
        parmD       lpDstDevType
        parmD       lpOutputFile
        parmD       lpData


cProc       Disable,<FAR,PUBLIC,WIN,PASCAL>,<si,di>
        parmD       lpDstDev
```

The **Enable** routine is called twice by **GDI** for display drivers. The first time that it's called, the passed variable **Style = 1**. This means that **GDI** wants you to move your **GDIInfo** data structure into the area pointed to by **lpDstDev**. **You must return the size in bytes of your GDIInfo data structure in AX (Currently this is 106).**

The second time that **Enable** is called (with **Style = 0**), three things must occur. First of all, you must initialize your graphics board hardware so that you're ready to run Windows. There's a special caveat here. In order for Windows to properly initialize its keyboard, you must set the byte at **40:49** into an IBM ROM BIOS compatible graphics mode. For many high resolution devices that don't use the ROM BIOS to setup their modes, this may seem unnecessary. However, the Windows keyboard will be locked out unless this is explicitly done. There are two possible ways to do this. Either move a **6** (CGA graphics mode) into the byte at **40h:49h** or use the ROM BIOS call to set **mode 6**. Remember to save the original byte at **40h:49h** so you can restore it at **Disable** time. Here's an example of how to do this using the ROM BIOS:

```
mov     ax,0f00h            ;save the current display mode
int     10h
mov     DisplayModeSave,al  ;make sure this is in your default Data segment
mov     ax,6                ;set to IBM display mode 6
int     10h
```

Next, you should perform any initialization of your hardware. You do not need to clear the screen at this time. Windows will call **BitBlt** to do that for you.

The next step in initialization is to call a special routine so that your Windows driver will work in the **OS/2 Compatibility Box**. Because Windows is so graphic in nature, and because the cursor operates asynchronously from the rest of Windows, we must assure that Windows is left and reentered in an orderly fashion when switching in and out of the Compatibility Box where Windows is running. Your hardware could get extremely confused if OS/2 switched away from you while you were in the midst of setting up for a draw! If OS/2 calls into the compatibility box using **INT 2FH** when we're in a state where we can't switch in an orderly fashion, we can return a failure code to OS/2. OS/2 will keep trying to call us until it receives a success code. Then, we save any states that we need to restore upon reentry and allow the switch to occur. When the compatibility box is switched back into, we reinitialize our hardware and call Windows to repaint the screen (see next paragraph). OS/2 uses **INT 2F, function 4001H** to switch from the compatibility box to OS/2 and **function 4002H** to switch back into the compatibility box. Therefore, the **Enable** routine must hook **INT 2FH** and check each call to that interrupts to see if it's one of the above function. You hook the interrupt by going down to vector **2FH** at segment 0. Remember to save the address of the old interrupt.

Before you hook interrupt 2FH, you must get the address of a special routine in Window's **USER** module which forces a repaint of the entire screen. The reason for this is obvious. When you switch back from OS/2 to Windows (which is running in the compatibility box), you must restore the screen to the state that it was when you exited. Luckily, the routine in **USER** can do this automatically for you! How do we get the address of this special routine? First, we call the Windows function **GetModuleHandle** whch returns a special identifier to the USER module. This identifier is called a **handle**. Once we have the **handle** to the module, we simply call the Windows function **GetProcAddress** giving the special process identifier for the **repaint** function (this identifier is always the number **275 decimal**). **GetProcAddress** returns to us a long pointer to the repaint function which we save away and call when appropriate. You should look at the file **SSWITCH.ASM** in one of the sample drivers to see how **INT 2F** is hooked. Note that you must hook **INT 2FH**, even when you're running under version 3.xx or version 4.xx of **DOS** because many network systems including **MS-NET** running under "real mode" **DOS's** will want to utilize the same functionality that **OS/2** uses.

The last thing to do in your **Enable** routine is to copy your desired **PDEVICE** structure to the area pointed to by **lpDstDev**. This call to **Enable** should return a **1** in **AX** if all was successful. Otherwise, return a **0**.

The PDevice Data Structure

There are several data structures whose contents are almost totally left up to the device driver writer's whim. Among them are the **PDevice**, **PBrush**, and **PPen** data structures. These structures define so-called **physical** objects which are solely used by the device driver to identify things such as bitmaps, pens and brushes to itself. We'll discuss the **PBrush** and **PPen** structures at an appropriate time.

The **PDEVICE** structure has only one word lengthed field which is required by Windows. That field is the first word in the structure. For displays, it must hold the number **2000H**. This number will be put into the first word of any **BITMAP** data structure (always pointed to by the **lpDstDev** parameter passed in the call) that Windows asks the device driver to draw onto the device. If the bitmap is to be drawn into a "main memory" bitmap, the first word of the **BITMAP** data structure will always be **0**. In this way, the device driver can tell where to draw the bitmap. All other fields in the **PDEVICE** structure(up to 32 total bytes in length) may be used by the driver in any way it sees fit. The "bitmapped" displays (such as the CGA & VGA) duplicate a **BITMAP** data structure into the **PDEVICE** structure. This is because in most cases, their drawing routines work exactly the same for drawing onto the device as they do for main memory draws. In the case of high resolution (non-bitmapped) devices, you probably only need to use the required first word of the **PDEVICE** structure. Since **PDEVICE** is passed to you on almost every call, you may wish to simply store some appropriate states in it. It's totally up to you.

The Disable Module

This is a very simple module. When called, you should first return your device to the state that it was in upon Windows startup, restore the byte at **40H:49H** to its original state (see explanation in **Enable**), and unhook yourself from **INT 2FH**.

Some General Comments

Disable is called whenever the Windows graphics mode is about to terminate. This means that it is called whenever the user wishes to leave Windows or switch to a badly behaved "old application" (an application which cannot run in a Window since it relies upon being able to call the screen hardware directly). It is not called when switching in and out of the OS/2 compatibility box (OS/2 takes care of hardware reinitialization). When returning from an old application, the **Enable** routine (with **Style = 0**) is called. Over an old application call and during the "switch-out" from the compatibility box to OS/2, your driver's **Data** segment will be saved intact. Other segments will be thrown out. Therefore, your driver should treat any data that it needs saved accordingly. Also, your **Enable** process should reinitialize any flags relating to screen states since these will probably be destroyed by the exit to the old application.

This concludes the discussion of the Enable and Disable processes.

Step Five: RealizeObject

```
cProc       RealizeObject,<FAR,PUBLIC>
        parmD       lpDstDev
        parmW       Style
        parmD       lpLogicalObj
        parmD       lpPhysicalObj
        parmD       lpTextXForm (or WindowOrigin)
```

Returns: Size required for a physical object in AX (if lpPhysicalObj = 0).
1 if successful, 0 if unsuccessful (if lpPhysicalObj <> 0).

The Windows **GDI** is a device independent graphics drawing engine. It communicates through the Windows programming interface (which is documented in the **Microsoft Windows Software Development Kit**) with a Windows application. The GDI then calls your device driver to translate its device independent graphics order into a real picture on a screen or printer page. Windows recognizes three types of objects at the device driver level. The first of these is the **pen** which is used to draw polylines and borders around objects drawn by the **Output** module. The pen has three attributes. They are the **color, style** (or pattern -- dotted lines etc.), and **width**. The second object is called a **brush** (or pattern). This object is used to fill figures drawn by **Output** and by **BitBlt** to fill (with some logical operation) rectangular areas. For example, the rectangular areas that make up a Windows screen are all drawn by **BitBlt** using a brush. The brush has attributes of **pattern** (an 8 pixel by 8 pixel repeating block pattern), **color**, and **hatch** (predefined patterns that use an explicit foreground and background color assigned to them). The last object that Windows supports is the **font** which is used by the **StrBlt** and **ExtTextOut** routines to draw text. Display drivers do not in general realize fonts and should fail with an error return code of 0 if asked to do so.

It is very rare that a piece of hardware uses the exact same representation of a Windows object that Windows does. For example, it is very inconvenient for the IBM 8514 display adaptor to deal in terms of **RGB** 24 bit colors. It prefers to look at colors as 8 bit quantities. However, the most device independent way for an application to pass down its desired color is by using the **RGB** representation. The **RealizeObject** module is the place where the translation between device independent (**or logical**) and device-optimal (**or physical**) objects takes place. You may wonder why a special translation routine is needed. Why not simply use GDI's logical objects and translate them into device-optimal objects at the actual time of drawing? The answer is that GDI tries to be economical. It stores many pre-translated objects and may use this same obect in hundreds of different draws. The translation is done only once for many draws.

Let's discuss the general attributes of the **RealizeObject** call and how to best utilize it for display devices. First of all, it's important to reiterate the most important concept of the Windows display driver interface:

Whatever you do on the screen, you must also be able to do into a main (host) memory bitmap.

This concept complicates the life of the display driver writer, especially when he's writing to a display that supports many complex drawing functions which Windows allows the driver to support. Let's say for example that there's a certain display device which supports all sorts of patterned line drawing with any width of line. On first inclination, the device driver writer would register all of these fabulous capabilities into his **GDIInfo** data structure and proceed to write a polyline routine with a physical pen that supports wide and styled lines. Everything seems to work great. He runs some of the toy applications included in this kit and everything's real fast. Unfortunately however, Windows requires the same abilities (the ability to write wide and styled polylines) for drawing into an arbitrary main memory bitmap. The device driver writer would have to duplicate all of his board's wide and styled line drawing capability into an 8086 routine running on the PC. Not only is this quite often a huge task in terms of algorithm, but it also makes the device driver unduly huge. Therefore, it quite often behooves a device driver writer to let GDI support the more complex pen styles and widths, even though it sacrifices some speed when drawing with these pens.

The Pen Object

The logical pen structure has the following structure:

offset	Description
0	**Pen style**
2	**Width of pen in pixels**
4	**Height of pen in scanlines**
6	**RGB pen color (doubleword -- high byte is 0)**

Following are the possible styles that may be passed in offset 0 of the logical pen structure:

style code	Description
0	**solid line**
1	**dashed line**
2	**dotted line**
3	**dot-dashed line**
4	**dash-dot-dotted line**
5	**null -- draw no line**

Be very careful and make sure you support the **null** style in your drawing code. You should draw no line and return a success code if you get a pen with this style.

It may behoove you to not support wide and/or styled lines. If you do, you must make sure that you support the same styling algorithms when drawing to the screen and main memory. GDI is able to correctly synthesize both wide and styled lines quite efficiently. Therefore, you may wish to not support them in your first pass and add their support later if necessary. You simply need to set the correct bits in your **GDIInfo** data structure to tell Windows that you don't support wide and/or styled lines. Under certain conditions GDI may pass you a logical pen with a wide or styled line, even if you've told GDI that you don't support them. In that case, realize the pen into a physical object with a solid, one-pixel wide (**nominal**) pen. GDI will be smart enough to still do the styling/wide line stuff itself.

The **physical** pen structure may be anything that you like. You may wish to put special case flags into the physical pen to communicate special case drawing enhancements to the actual drawing routines.

The Brush Object

The logical brush structure has the following structure:

offset	Description
0	**Brush style (0=Solid, 1=Hollow, 2=Hatched, 3=Patterned)**
2	**For solid brushes: RGB color (high byte = 0)** **For hatched brushes: RGB foreground color (high byte = 0)** **For patterned brushes: pointer to pattern BITMAP structure**
6	**Hatch style (unused for other brush styles)**
8	**Physical color for hatch background (NOT RGB -- see description of the ColorInf routine!)**

Hollow brushes should be realized. If they are passed to a drawing routine, no fill should be done at all. However, if the raster operation which is passed to **BItBlt is NOT Destination**, you should logically **NOT** the entire rectangular area passed, even if the passed brush is hollow.

Following are the possible hatch styles that may be passed in offset 6 of the logical brush structure:

style code	Description
0	**horizontal (----)**
1	**vertical (||||)**
2	**forward diagonal (////)**
3	**backward diagonal (\\\\)**
4	**cross (++++)**
5	**diagonal crosshatch (XXXX)**

The RealizeObject Routine

It is important to understand the ebb and flow of the **RealizeObject** routines. When you're called, you must first determine whether the caller (**GDI**) wants you to actually realize an object or whether it's asking for how much space to allocate for the realized (**physical**) object. If GDI is asking for the size of the physical object, the parameter **lpPhysicalObj** will be 0. You then return in **AX**, the size (in bytes) of your physical pen, brush, or font. If you do not support the realization of fonts, simply return a 0. If GDI is asking to you realize a logical object into a physical one, **lpPhysicalObj** will be pointing to the memory location where you must put your completed physical object and **lpLogicalObj** will be pointing to the logical object that GDI wants you to translate. The **Style** parameter tells you whether you are to realize a pen (=1), brush (=2), or font (=3). You would then proceed to do the translation and return **AX=1** as a success code or **AX=0** as a failure code (e.g. if you do not support the realization of fonts, you'd return **AX=0**).

The last parameter is "dual-purpose". If you're asked to realize a font, it's a pointer to the **TextXForm** data structure. This is documented in the **Windows Adaptation Guide** included in this kit. If you're being asked to realize a brush or patterned pen, this parameter is **not** a pointer. It is actually two words which are the starting coordinates in **X and Y** of the Window that the application (which called GDI in the first place) is running in. It is seemingly quite unimportant for the display driver to receive this information. However, it is essential in order to establish a **pattern reference point**. Most displays use a pattern reference point starting at the same location as the starting point of the draw. In other words, for an 8 bit repeating pattern, the first bit of the pattern is at the X-origin of the draw. Then at X-coordinate (X-origin+8), the pattern begins to repeat itself. Now, in the case of Windows, the pattern reference point must be at the beginning of the window where the application is running. Therefore, we must rotate any patterns so that they begin their repetition relative to the application's window. If we don't bother to rotate, we run the danger of the patterns not "meshing" if the user decides to move the window containing the application to a different place on the screen. Windows will sometimes actually BitBlt part of the application's window to the new location and then repaint the rest. The part that's repainted will have patterns that are drawn relative to the wrong place on the screen and will not "mesh" correctly with the already existing pattern. It is therefore necesary to rotate all patterned brushes to the pattern reference point. The sample display drivers have good examples of how to do this.

This concludes the discussion of the RealizeObject module.

Step Six: The ColorInfo Function

```
cProc       ColorInfo,<FAR,PUBLIC>,<si,di>
        parmD       lpDstDev
        parmD       ColorIn
        parmD       lpPhysicalColor
```

Returns:

If lpPhysicalColor is null: DL:AX will contain the RGB color corresponding to the physical color passed in ColorIn. DH must be 0.

If lpPhysicalColor is not null, DL:AX contains the RGB value of the closest match color that the device has to the color passed in ColorIn. DH must be 0.

The next step in writing your device driver is the **ColorInfo** function. This function is closely related to **RealizeObject** since it deals with translations between **logical colors** which are passed as doubleword RGB values (with the high byte of the doubleword = 0) and **physical colors** which are those recognized and used most readily by your device. You may have already written the majority of this routine when you wrote your **RealizeObject** module since it also requires the translation from logical to physical colors when creating physical pens and brushes. Simply follow the instructions found above and in the description of **ColorInfo** in the Windows Adaptation Guide. Remember, some applications (**EXCEL** in particular) require that the high byte of any doubleword RGB color returned by your device driver to be 0.

Step Seven: The Inquire and Stub Functions

```
cProc       Inquire,<FAR,PUBLIC>,<si,di>
    parmD       lpCursorInfo
```

Returns: AX=4 (which is the number of bytes in the CursorInfo data structure).

The **Inquire** function is a very simple one. It is called before the **Enable** function (once per initialization) and simply requires you to return the desired **Mickey-to-Pixel** ratio for your screen. See the Windows Adaptation Guide for full details.

Following are two functions that you need to set up calls to. They are not supported in the GDI however and should always return a failure code since they may be supported in the future. Simply copy verbatim the code reproduced here to your driver and you're finished with their support.

```
cProc       SetAttribute,<FAR,PUBLIC>
        parmD       lpDstDev
        parmW       StateNum
        parmW       Index
        parmW       Attribute
cBegin
xor         ax,ax                   ;always return AX = 0
cEnd

cProc       DeviceBitmap,<FAR,PUBLIC>
        parmD       lpDstDev
        parmW       Command
        parmD       lpBitmap
        parmD       lpBits
cBegin
xor         ax,ax           ;always return AX = 0
cEnd
```

Windows Escape Information

Windows includes a number of calls which have been standardized but which fit outside of the device independent functionality of GDI, Kernel, and User. These functions are basically direct communication between an application and a cooperating device driver. On the application programmer's side, these functions are grouped together under one call called **Escape**. On the device driver side, they are grouped under a process called **Control**. The **Control** module is required for all device drivers. However, you may choose support only a few of the "sub-functions" documented here or only the minimal functionality required (You can use **QueryEscSupport** which tells the calling application that you support a subset or none of the control functions). It is important to note that most of these subfunctions are applicable to printers only. Most applications will not have the support to call these functions for display devices. In fact, the only sub-functions that are recommended for support by displays are **QueryEscSupport, GetColorTable, and SetColorTable**. If you have any ideas on new **Control** sub-functions, you should contact the Microsoft Windows OEM Manager (whose name appears on the cover letter enclosed with this kit) at (206)-882-8080. You can also include this in a TAR or Service Request. We will then review your suggestion and contact you with any questions. This mechanism is the only method by which an escape/control function can be added to Windows.

The generalized stack frame to expect on a call to **Control** is:

```
cProc       Control,<FAR,PUBLIC>,<si,di>
        parmD       lpDstDev
        parmW       SubFunction
        parmD       lpInData
        parmD       lpOutData
```

Return codes:

All return codes are returned as signed integers in the AX register. Following are the generalized return codes used by the banding printer driver control functions. They also are referred to in the **Microsoft Windows Adaptation Guide** by their symbolic names. For all control subfunctions, a positive number indicates success and a zero return code indicates a failure. It is best however, to use the specific return codes documented here whenever it is indicated that they are appropriate:

SP_ERROR (-1) --> General error in banding.
SP_APP_ABORT (-2) The job was aborted because the application's callback returned false (0).
SP_USER_ABORT (-3) The job was aborted by the user by the spooler's "abort job" function.
SP_OUTOFDISK (-4) The job was aborted due to lack of disk space.
SP_OUTOFMEMORY (-5) The job was aborted due to lack of memory.

Following is a list of the sub-functions and a short description of their desired action. In all cases, you should refer first to this list and then to **Appendix E** in the **Microsoft Windows Adaptation Guide** for full details on each subfunction. Note that there has been some ambiguity and redundancy in the numbering and description of these functions and that this list is definitive for Windows/286 and Windows/386 version 2.10.

NewFrame (SubFunction=1)

This is mainly for use by banding printer drivers to indicate to the printer driver that application wants the driver to perform a page break algorithm or form feed and that the application is finished outputting to the page. The generalized return codes documented above should be used by the driver. This function is documented in the **Windows Adaptation Guide.**

AbortDoc (SubFunction=2)

This again is for banding printer drivers and is called when the application orders the printer driver to abort its print job. It is called **AbortPic** in some earlier printer drivers. It returns a positive number if successful, a negative number if unsuccessful. This function is documented in the **Windows Adaptation Guide.**

NextBand (SubFunction=3)

Another one for banding printer drivers. It tells the device driver that the drawing into the banding bitmap is now complete and that the printer driver should now dump the band to the printer and reinitialize the banding bitmap for processing the next band. The driver should use the generalized return codes documented above. This function is documented in the **Windows Adaptation Guide.**

SetColorTable (SubFunction=4)

This function can be used by both printer and display drivers although the EGA and VGA color drivers do not support it. It is used by applications which desire to change the palette used by the display driver. In general, the function works as documented in the **Adaptation Guide.** However, since the driver's color mapping algorithms will probably no longer work with a different palette, an extension has been added to this function. If the color index pointed to by **lpColorEntry** is FFFFH, the driver is to leave all color mapping functionality to the calling application. The application will necesarily know the proper color mapping algorithm and takes resposibility for passing the correctly mapped **physical** color to the driver (instead of the **logical RGB color**) in routines such as **RealizeObject** and **ColorInfo.** For example, if the device supports 256 colors with palette indices of 0 through 255, the application would determine which index contains the color that it wants to use in a certain brush. It would then pass this index in the low byte of the doubleword logical color passed into **RealizeObject.** The driver would then use this color exactly as passed instead of performing its usual color mapping algorithm. If the application wishes to reactivate the driver's color mapping algorithm (ie: if it restores the original palette when switching from its window context), the color index pointed to by **lpColorEntry** should be FFFEH.

GetColorTable (SubFunction=5)

Display drivers will probably wish to support this function. It returns **RGB** colors which are mapped to the physical color index passed in. This function is documented in the **Windows Adaptation Guide.**

FlushOutput (SubFunction=6)

This subfunction is used by banding printer drivers. When called, they should reinitialize the banding bitmap (ie: get rid of any partially drawn stuff in the bitmap). This function is documented in the **Windows Adaptation Guide.**

DraftMode (SubFunction=7)

If supported, the printer driver should force the device into its lowest resolution mode. This function is documented in the **Windows Adaptation Guide.**

QueryEscSupport (SubFunction=8)

For each call to this subfunction, the driver should determine whether it supports the function (and return **AX=1**) or does not support the function (and return **AX=0**). **All device drivers must return success if queried whether they support the QueryEscSupport subfunction.** This function is documented in the **Windows Adaptation Guide.**

SetAbortProc (SubFunction=9)

This subfunction is used by printer drivers to set the address of a subroutine which allows the printing to be aborted. This function is documented in the **Windows Adaptation Guide.**

StartDoc (SubFunction=10)

This subfunction is used to initialize the printer for a new print job. This function is documented in the **Windows Adaptation Guide.**

EndDoc (SubFunction=11)

This subfuction is called when a print job is ended normally (not by an abort). This function is documented in the **Windows Adaptation Guide.**

GetPhysPageSize (SubFunction=12)

This subfunction returns the dimensions of a page in device units ("pixels" wide by "scanlines" high). This function is documented in the **Windows Adaptation Guide.**

GetPrintingOffset (SubFunction=13)

This subfunction returns the offset from location (0,0) (The top left corner of a page) of the first location where printing should occur. Again, the coordinates are returned in device units. This function is documented in the **Windows Adaptation Guide.**

GetScalingFactor (SubFunction=14)

This subfunction communicates to GDI the factor by which it needs to stretch bitmaps when drawing them to the printer. This function is documented in the **Windows Adaptation Guide.**

MFComment (SubFunction=15)

The application can add a comment to a metafile created by the device driver. This function is documented in the **Windows Adaptation Guide.**

GetPenWidth (SubFunction=16)

This control function is undefined at this time and is reserved. If you need to use it, please contact the Windows OEM Manager.

SetCopyCount (SubFunction=17)

This subfunction lets the caller pass the number of copies of each page that it desires to print to the device driver. This function is documented in the **Windows Adaptation Guide.**

SelectPaperSouce (SubFunction=18)

This control allows the application to pass the desired paper source to the device driver. This function is documented in the **Windows Adaptation Guide.**

DeviceData (also called PassThrough) (SubFunction=19)

This control allows the application to pass data directly to the device driver without Windows intervention. This function is documented in the **Windows Adaptation Guide.**

GetTechnology (also called GetTechnolgy) (SubFunction=20)

This control is not described in the **Windows Adaptation Guide.** It is used by applications to query general technology type for printers. This allows an application to implement technology specific functionality. There are no input parameters. The parameter **lpOutData** points the place where the device driver should copy a zero terminated ASCII string containing the printer technology type. At this point, applications such as **PageMaker** recognize strings such as **"PostScript".**

Functions 21, 22, & 23 are fully documented in the PostScript Language Reference Manual published by Addison Wesley Co. Please refer to this manual for full definitions of the concepts described here.

SetLineCap (also called SetEndCap) (SubFunction=21)

This control is used to tell the driver how an application wishes to "cap" a wide line. LineCap is a PostScript concept which is defined as either **Butt Cap (=0), Round Cap (=1), or Projecting Square Cap (=2).**

SetLineJoin (SubFunction=22)

This control is used to tell the driver how an application wishes to join two lines at an angle. LineJoin is a PostScript concept which is defined as either **MiterJoin (=0), RoundJoin(=1), or BevelJoin(=2).**

The parameter **lpInData**, if non-null points to the word lengthed linejoin value that should be set by the driver. If **lpInData** is null, then no setting of the linejoin should be done. If the parameter **lpOutData** is not null, it points to a word lengthed value in which the previous (or current in **lpInData** is null) linejoin should be returned.

SetMiterLimit (SubFunction=23)

This control is used to tell the driver how an application wishes toclip off miter type LineJoins when they become objectionably long. The definition of how the **SetMiterLimit** command works is documented in the **PostScript Language Reference Manual (see above).**

The parameter **lpInData**, if non-null points to the word lengthed miter limit value that should be set by the driver. If **lpInData** is null, then no setting of the miter limit should be done. If the parameter **lpOutData** is not null, it points to a word lengthed value in which the previous (or current in **lpInData** is null) miter limit value should be returned.

12 ✓ **BandInfo (SubFunction=24)**

This allows a calling application to obtain information from the driver about its banding bitmap. It is documented in the Windows Adaptation Guide.

DrawPatternRect (SubFunction=25)

This escape allows an application to draw a patterned rectangle on capable printers (such as the **HP PCL Series Printers**). It is documented in the Windows Adaptation Guide.

GetVectorPenSize (SubFunction=26)

This control subfunction is used in plotter drivers only. GDI calls it when drawing shapes. **EXCEL** in particular uses this control function. Typically, plotters have pen widths which can be many device units wide. GDI uses the information returned by this function to prevent hatched brush patterns from overwriting the border of a closed figure.

The parameter **lpInData** points to a **Physical Pen** data structure. The data contained in this structure allows the device driver to determine which pen that the caller wishes data to be returned for. The driver returns the width of the pen (in device units) in the second word of the **POINT** data structure pointed to by **lpOutData**.

GetVectorBrushSize (SubFunction=27)

This control function is used in plotter drivers only. GDI calls it when drawing shapes. **EXCEL** in particular uses this control function. Typically, plotters have pen widths which can be many device units wide. GDI uses the information returned by this function to prevent the filling of closed figures (e.g. rectangles, ellipses etc.) from overwriting the borders of the figure.

The parameter **lpInData** points to a **Physical Brush** data structure. The data contained in this structure allows the device driver to determine which brush that the caller wishes data to be returned for. The driver returns the width of the brush (which in the case of plotters is the width of the pen used to do the filling) in device units in the second word of the **POINT** data structure pointed to by **lpOutData**.

EnableDuplex (SubFunction=28)

This subfunction tells printers which can print on both sides of the page to enable this capability. The parameters are described in the **Windows Adaptation Guide**.

GetSetPaperBins (SubFunction=29)

This subfunction, due to an inconsistency between the PostScript and HPPCL drivers is subfunction number 30 in the Windows/286 Version 2.1 HPPCL driver and subfunction 29 in the PostScript driver. This will be corrected in the next release of Windows and the numbering will be consistent with this documentation.

This function allows the application to query the number of available paper bins on the printer and to communicate the desired paper bin that it wishes the device driver to use. There are 3 possible actions for this escape depending on the value of the passed values in **lpInData** and **lpOutData**. The following table indicates what action is to be taken for each condition.

lpInData	lpOutData	Expected Action
null	lpBinInfo	return nbr of bins and current bin ID
lpBinInfo	lpBinInfo	set bin to passed bin number and return nbr of bins and previous bin ID
lpBinInfo	null	set bin to passed bin number
null	null	return error

The **BinInfo** data structure is as follows:

```
dw      BinNumber
dw      NbrofBins
dw      Reserved
dw      Reserved
dw      Reserved
```

The parameter **lpInData**, if not null, will point to the **BinInfo** data structure. The **BinNumber** parameter will have the bin number which the device driver should set. The parameter **lpOutData** points to the **PaperBins** data structure if it is not null. The driver should return the current bin number (if just set, it should return the previous bin number) and number of bins available in the appropriate fields in the data structure.

GetSetPaperOrientation (also called GetSetPrintOrient) (SubFunction=30)

This subfunction, due to an inconsistency between the PostScript and HPPCL drivers is subfunction number 29 in the Windows/286 Version 2.1 HPPCL driver and subfunction 30 in the PostScript driver. This will be corrected in the next release of Windows and the numbering will be consistent with this documentation.

This function allows the application to query and set either portrait or landscape mode on the printer. There are two possible actions depending on the value of **lpInData:**

lpInData	Expected Action
null	Return current orientation in AX
lpOrient	Set to the orientation value and return previous orientation in AX

The **Orient** data structure is as follows:

```
dw      Orientation
dw      Reserved
dw      Reserved
dw      Reserved
dw      Reserved
```

lpInData, if non-null, will point to the **Orient data structure.** It will be set to either **PortraitMode (=1)** or **LandscapeMode(=2).** The parameter **lpOutData** is not used and can be ignored. The driver should **always** return the current orientation of the printer, even if **lpInData** is null (if just set, it should return the **previous** orientation of the printer) or **-1** if the orientation change ordered was unsuccessful.

EnumPaperBins (SubFunction=31)

This subfunction requires the device driver to inform the caller of the attributes of the various paper bins that it has available. The calling application will probably perform a **GetSetPaperBins** to find out the number of bins available on the printer and then will call **EnumPaperBins** to find out their attributes.

Upon entry to the driver, **lpInData** will point to a **NumItems** data structure which is defined as follows:

```
dw    Number of paper bins
dw    Reserved
dw    Reserved
dw    Reserved
dw    Reserved
```

Also, **lpOutData** will point to a data structure which has its size defined by the **Number of paper bins** entry in the **NumItems** data structure. Its structure (in pseudo C array format) is:

```
short  BinList [Number of paper bins]
char   PaperNames [Number of paper bins] [Bin string length]
```

The current value for **Bin string length** is **24.**

The driver must return an array of bin identification numbers and paper names depending on the number of paper bins passed by the caller.

GetExtendedTextMetrics (SubFunction = 256)

This control allows the application to query the availability of extended text metrics for a font. It is documented in the **Windows Adaptation Guide.**

GetExtentTable (SubFunction = 257)

This control returns the width of code points in a font. It is documented in the **Windows Adaptation Guide.**

GetPairKernTable (SubFunction = 258)

Returns the character pair kerning table for the currently selected font. It is documented in the **Windows Adaptation Guide.**

GetTrackKernTable (SubFunction = 259)

Returns the track kerning table for the currently selected font It is documented in the **Windows Adaptation Guide.**

ExtTextOut (SubFunction = 512)

Allows the driver to take over all GDI functionality of **ExtTextOut.** It is documented in the **Windows Adaptation Guide.**

EnableRelativeWidths (SubFunction = 768)

Allows the application to enable and disable driver relative width handling. It is documented in the **Windows Adaptation Guide.**

EnablePairKerning (SubFunction = 769)

Allows the application to enable and disable driver pair kerning. It is documented in the **Windows Adaptation Guide.**

SetKernTrack (SubFunction = 770)

Allows the application to set the desired kern track. It is documented in the **Windows Adaptation Guide.**

SetAllAdjustValues (SubFunction = 771)

Allows the application to set the text justification table for capable drivers. It is documented in the **Windows Adaptation Guide.**

StretchBLT (SubFunction = 2048)

Allows capable **printer** drivers to perform their own StretchBLT. It is documented in the **Windows Adaptation Guide.**

Escapes Used By The Japaneese Version of Windows

Following is a list of escapes reserved by Microsoft Japan and a short description of each. Microsoft Japan should be contacted to reserve other escape functions if necessary. Their telephone number in Japan is (3) 221-7071.

TTYMode (SubFunction = 2560)

Note: In Windows 1.xx, this escape was SubFunction number 15. Since this conflicts with the standard Microsoft escape 15 (MFComment), it was relocated to the current SubFunction number. Kanji Windows 2.xx's GDI checks for function 15 whenever called by an application and remaps it to the correct number.

If called by an application, this call signal Kanji Windows printer drivers to disregard the passed **FontInfo** data and print using the printer's default hardware font. No attempt should be made.to match the passed font (in **FontInfo**). The printer's hardware font must have the width of Roman characters equal to half the width of Kanji characters.

GaijiFontSize (SubFunction = 2576)

The driver must return the standard Kanji font size in **lpOutData**. **lpOutData** points to a **POINT** data structure. **lpInData** is undefined and can be ignored.

GaijiAreaSize (SubFunction = 2577)

The driver must return the number of Gaiji fonts that can be registered at once as a short integer. Both **lpInData** and **lpOutData** are undefined and can be ignored.

GaijiSystemGetFont (SubFunction = 2578)

The driver returns the gaiji's font pattern in a monochrome bitmap format. the low word of **lpInData** contains (it is not a pointer) the handle to the bitmap. **lpOutData** points to a buffer which contains the Shift JIS code of the Gaiji font pattern to be retreived. Should return **AX = TRUE** on success.

GaijiSystemSetFont (SubFunction = 2579)

The driver sets the font pattern contained in the bitmap whose handle is provided in the low word of **lpInData**. **lpOutData** points to a buffer which contains the Shift JIS code of the Gaiji font. Return **AX = TRUE** on success.

GaijiItToCode (SubFunction = 2580)

The driver returns the Shift JIS code from the index to the Gaiji area. The high word of **lpInData** contains (it is not a pointer) an index to the system reserve area. **lpOutData** points to a buffer which will contain the Shift JIS code of the gaiji upon return. The driver should return **AX = TRUE** upon success.

GaijiLocalOpen (SubFunction = 2581)

The driver should get ownership of the gaiji area. Both **lpInData** and **lpOutData** are undefined and can be ignored. If successful, the driver should return **AX = TRUE**.

GaijiLocalClose (SubFunction = 2582)

The driver releases ownership of the gaiji area. The high word of **lpInData** will contain (it is not a pointer) the handle which was returned to the caller by **GaijiLocalOpen**. **lpOutData** is undefined and can be ignored. Return **AX = TRUE** upon success.

GaijiLocalSetFont (SubFunction = 2583)

The driver sets a gaiji font and retrieves the Shift JIS code for it. Upon entry, the low word of **lpInData** contains a handle to the monochrome bitmap containing the gaiji font. The high word of **lpInData** contains the handle returned by **GaijiLocalOpen**. **lpOutData** contains a pointer to a buffer which will contain the Shift JIS code of the gaiji upon return.

GaijiLocalSave (SubFunction = 2584)

Saves the current gaiji area into global memory in hardware specific format. The low word of **lpInData** contains the flags to be passed by the driver to **GlobalAlloc**. The high word contains the handle returned by **GaijiLocalOpen**. **lpOutData** is undefined and can be ignored. This function should return the handle of the global object in **AX**.

GaijiLocalRestore (SubFunction = 2585)

Restores the gaiji area with the global object saved in **GaijiLocalSave**. Upon entry, the low word of **lpInData** contains the global memory handle which was returned by **GaijiLocalSave**. The high word contains the handle returned by **GaijiLocalOpen**. **lpOutData** is undefined and can be ignored. Upon return, **AX** should contain the number of free areas in the gaiji area +1.