Microsoft® Windows
Device Development Kit

development tools for providing Microsoft® Windows device support

Device Driver Adaptation Guide

VERSION 3.0

for the MS-DOS® Operating System

Microsoft Corporation

Printed and bound in the United States of America.

Document No. xxxx
Part No. yyyy
10 9 8 7 6 5 4 3 2 1

Table of Contents

Device Driver Adaptation Guide

Introduction to Device Drivers

This document is intended for device driver writers working as consultants and for Independent Hardware Vendors (IHVs) and computer manufacturers. The information contained herein is proprietary to Microsoft Corporation. Therefore, only those members of your organization directly involved in the development of Microsoft Windows device drivers should have access to this document.

This introduction provides some background information that you should review before using the documentation provided with the *Microsoft Windows Device Development Kit* (DDK). Included here are sections on the following:

- What you need to know or have before you start
- Description of the manuals provided with the DDK
- Notational conventions used throughout the DDK documentation

What Should You Know or Have Before Starting?

You will need to know Windows, MS-DOS ®, MASM, and, if writing a printer driver, the C programming language. Definitions of key terms used in describing device drivers and virtual devices are provided in Appendix A, "Terms and Acronyms," which is located in the *Microsoft Windows Virtual Device Adaptation Guide*.

The *Microsoft Windows Installation and Update Guide* for the DDK provides detailed information on the requirements for setting up your development environment and the contents of the source disks included with the DDK.

You will need to purchase the *Microsoft Windows Software Development Kit* (SDK) and the retail Windows package for testing. You will also need to purchase access to the Microsoft OnLine software support system to get technical support while developing your driver.

The *Microsoft Windows Software Development Kit* contains reference material, a special linker, the Windows Resource Compiler (RC), special versions of the SYMDEB and CodeView debuggers, header files, and several utilities that aid development and testing.

It also provides several INCLUDE and header (.H) files that contain declarations of all the Windows functions, definitions of many macro identifiers that you can use in programming, and structure definitions. Import libraries included in the kit allow LINK to resolve calls to Windows functions and to prepare the program's .EXE file for dynamic linking.

Microsoft OnLine can provide you with the accurate, interactive support you need to remain as productive as possible. Use it to retrieve information (on virtually all of Microsoft's products) from our technical product KnowledgeBase, to search through our

Software Library for sample drivers and source code, or to submit Service Requests (specific questions on writing device drivers) directly to one of our highly qualified customer support engineers. Watch the Exchange Bulletin Board for announcements on the availability of new sample sources for special devices. For more information about Microsoft OnLine, call Microsoft Product Support Services Telemarketing at (800) 443-4672. **(Is the number still correct?)**

DDK Documentation Set

The 3.0 version of the *Microsoft Windows Device Development Kit* has been completely reorganized. It now consists of the following four manuals:

- *Microsoft Windows Device Driver Adaptation Guide*, which covers how to write or modify device drivers for Windows 3.0 when running in either real or standard mode.
- *Microsoft Windows Virtual Device Adaptation Guide*, which covers how to write virtual devices for Windows 3.0 when running in 386 enhanced mode.
- *Microsoft Windows Installation and Update Guide*, which provides information on the DDK source code, test scripts, utilities, and building tools provided with, and the development environments required for, Windows 3.0 when running in either real, standard, or 386 enhanced mode.
- *Microsoft Windows Printers and Fonts Kit*, which includes information on the Printer Font Metrics (PFM) file formats and the new PFM Editor, along with technical notes on the PCL/HP LaserJet and PostScript printer drivers.

We recommend that both novice and advanced device driver and virtual device writers read the *Microsoft Windows Installation and Update Guide*, this introduction, and Chapter 1, "Overview of Windows." After that, you can skip to the appropriate chapter(s) for the particular driver with which you work.

The following sections summarize the contents of each part and chapter in the two main DDK documents.

Microsoft Windows Device Driver Adaptation Guide

Part 1, "Writing Windows Device Drivers," consists of nine chapters that provide information on writing or modifying specific Windows 3.0 device drivers.

Chapter 1, "Overview of Windows," provides information common to both device driver and virtual device writers, such as definitions, time requirements, calling conventions, and INCLUDE file descriptions.

Chapter 2, "Display Drivers," contains information specific to writing or modifying Windows 3.0 display drivers. The major functions are described briefly and examples are given.

Chapter 3, "Display Drivers: New Features," discusses from a device driver standpoint the Windows 3.0 changes to color palette management, protected-mode support, greater than 64K fonts, and device-independent bitmaps (DIBs). More detailed information on each of these new features is provided in the *Microsoft Windows Software Development Kit.*

Chapter 4, "Display Driver Grabbers," contains descriptions of the functions and data structures used by the grabbers that work with Windows 3.0 when running in real and standard mode. Virtual device grabbers are discussed in the *Microsoft Windows Virtual Device Adaptation Guide.*

Chapter 5, "Printer Drivers," contains information specific to writing or modifying Windows 3.0 printer drivers. The major functions and escapes are described briefly and examples are given. The relationship between GDI and printer drivers is also discussed in detail.

Chapter 6, "Network Support," contains descriptions of the new benefits provided to network users, incompatibility problems and solutions, and how to make your network software work well with Windows 3.0.

Chapter 7, "Network Drivers," contains information specific to writing or modifying Windows 3.0 network drivers. The major functions are described briefly and examples are given.

Chapter 8, "Keyboard Drivers," contains information specific to writing or modifying Windows 3.0 keyboard drivers. The major functions are described briefly and examples are given.

Chapter 9, "Miscellaneous Drivers," contains brief descriptions of the communications, sound, and mouse drivers.

Part 2, "General Reference for Device Drivers," consists of six chapters that provide general reference-type information for use in writing or modifying Windows 3.0 device drivers.

Chapter 10, "Common Functions," provides an alphabetical listing with detailed descriptions of the main functions used by most device drivers.

Chapter 11, "Device Driver Escapes," provides an alphabetical listing with detailed descriptions of the escapes used mainly by printer drivers.

Chapter 12, "Data Structures and File Formats," contains detailed descriptions of the major data structures and file formats used by most device drivers.

Chapter 13, "The Font File Format," provides descriptions of the three main data structures used with fonts: TEXTMETRIC, TEXTXFORM, and FONTINFO.

Chapter 14, "Raster Operation Codes and Definitions," provides a table of raster operation codes and their definitions, along with a brief description of reverse Polish notation.

Chapter 15, "Miscellaneous Character Set Tables," contains a brief description of character sets and provides examples of the main ones used by Windows 3.0: ANSI, OEM, and SYMBOL.

Microsoft Windows Virtual Device Adaptation Guide

Part 3, "Writing Virtual Devices," consists of three chapters that provide information on writing Windows 3.0 virtual devices. A more detailed description of each chapter is provided in the introduction to that document.

Part 4, "Virtual Device Services," consists of 23 chapters that provide information on each of the major categories of services used with virtual devices. A more detailed description of each chapter is provided in the introduction to that document.

Part 5, "Appendixes," consists of the following four appendixes that contain information common to both device drivers and virtual devices. The first two provide useful information that can be reviewed quickly before you read the specific device-related chapters. The remaining appendixes deal with topics that may be more useful *after* reading the specific device-related chapters.

- Appendix A, "Terms and Acronyms"
- Appendix B, "Understanding Modes"
- Appendix C, "Creating Distribution Disks for Drivers"
- Appendix D, "Enhanced Windows INT 2FH API"

Notational Conventions

The following notational conventions are used throughout the DDK documentation set.

Convention	Meaning
bold	Bold is used for keywords, such as function, register, macro, and data structure field names. These names are spelled exactly as they should appear in source programs. Notice the bold in the following example: **Disable** (*lpDestDev*) Here, **Disable** is bold to indicate that it is the name of a function.
italics	Italics are used to indicate a placeholder that should be replaced by an actual argument. In the preceding example, *lpDestDev* is italic to indicate that it should be replaced by an argument.
(Parentheses)	Parentheses enclose the parameter or parameters that are to be passed to a function. In the preceding example, *lpDestDev* is the parameter.
`Monospace`	Monospace type is used for program code fragments and to illustrate the syntax of data structures.

Part 1 Writing Windows Device Drivers

This first part of the *Microsoft Windows Device Driver Adaptation Guide* provides information on how to write or modify Windows device drivers; make them compatible and work efficiently with Microsoft Windows 3.0 when running in both real and standard modes; and make them bimodal, i.e., capable of running under either real or protected mode.

Separate chapters are provided for descriptions of each of the major device drivers. Some information that is common to many of the drivers is provided in Chapter 1, "Overview of Windows." However, most of the common reference-type information is provided in Part 2, "General Reference for Device Drivers."

CHAPTERS

Chapter 1 Overview of Windows

This chapter contains information that is common to or used by most of the different Windows 3.0 device drivers. Since these drivers are the basic building blocks for enhanced Windows virtual devices (VxDs), references are also made, where appropriate, to Windows VxDs. The following information is provided here:

- Definitions of device drivers, virtual devices, programs, and libraries
- Description of how Windows device drivers and virtual devices work together
- Estimates on the time required to write a device driver
- Descriptions of the Windows modules and those needed to build a device driver
- Explanation of the Windows calling conventions
- Description of the Windows INCLUDE files

Subsequent chapters will detail how to write specific device drivers. Notice, however, that for some drivers, such as the Mouse and Keyboard drivers, you should be able to use the supplied source code and not need to write a new driver.

1.1 What are Device Drivers and Virtual Devices?

A *device driver* is often called a Windows dynamic-link library or DLL (to distinguish it from a program). It forms the interface between Windows and a particular piece of peripheral hardware (e.g., a printer or a display screen). This DLL contains the Windows Graphics Device Interface (GDI) functions needed to access or drive a specified device or family of devices. It also contains information naming the types of devices it supports.

In other words, a *device driver* is the software that provides the hardware-dependent, low-level interface between the Windows functions and the output device.

A separate driver must be written for each peripheral in the system. However, to avoid using up too much memory, the driver is only loaded when it is installed into the system. **(This sentence sounds funny/strange. Is "installed into the system" the correct phrase here?)**

An enhanced Windows *virtual* device (VxD) is a separately compiled program **(is program the right word considering the definition of program given in the next section?)** that is loaded and linked with the Virtual Machine Manager (VMM) when enhanced Windows is first started. Each VxD is responsible for handling a specific piece of hardware or for providing services used by the Virtual Machine's (VM's) application program. See the *Microsoft Windows Virtual Device Adaptation Guide* for detailed information on VxDs.

In enhanced Windows, the VxD sits between the Windows driver (in the System VM) and the actual hardware. These two pieces of code can communicate via established I/O ports, or they can establish a new interface (e.g., an output string instead of an output character for a parallel port VxD). In addition to this interaction, the VxD must also serialize access to the hardware ports by other VMs running simultaneously with Windows.

1.2 Programs vs. Libraries

From the user's perspective, a Windows program and a Windows library (or device driver, which is a type of library) are very different. The user cannot run a Windows library directly. Windows loads a part of a library into memory only when a program needs to use a function that the library provides. The user can, of course, run any Windows program.

In fact, the user can run multiple instances of the same Windows program. Windows uses the same code segments for the different instances but creates a unique data segment for each. Windows never runs multiple instances of a Windows library.

From the programmer's perspective, a Windows program is a task that usually creates and manages windows on the display. Libraries are modules that assist the task. A programmer can write additional library modules that one or more programs can use. For the programmer, one important distinction between programs and libraries is that a Windows library does not have its own stack; instead, the library uses the stack of the program that calls the function in the library.

When Windows loads a program or a library into memory, it must resolve all the calls the module makes to functions in other modules. Windows does this by inserting the addresses of the functions into the code—a process called *dynamic linking*.

1.3 How the Windows Pieces Fit Together

Windows requires device drivers for the hardware on which it runs, regardless of whether you are running Windows in real, standard, or 386 enhanced mode and in real or protected mode. However, when you are running enhanced Windows, it may also require a virtual device.

The purpose of a Windows device driver (used with the real, standard, and 386 enhanced mode versions) and that of an enhanced Windows virtual device (used only with Windows when running in 386 enhanced mode) is different. A Windows device driver exists to perform actions on its device, such as printing a circle or getting the mouse location. It maps an idealized device API onto limited real devices. An enhanced Windows virtual device exists to virtualize the hardware; it does not, at least not in a visible manner, provide an API and services. Instead of mapping a general API onto specific devices, it simply traps and virtualizes all access to that device.

Explanations of the three versions of Windows and how to write or modify the appropriate driver or virtual device for your hardware are provided in Part 1, "Writing Windows Device Drivers," in this guide and in Part 3, "Writing Virtual Devices," in the *Microsoft Windows Virtual Device Adaptation Guide.*

1.4 How Long Will it Take to Write a Device Driver or Virtual Device?

The development cycle depends on a number of factors, including whether or not you are modifying an existing driver and the complexity of the interface to the hardware. If you are developing an enhanced Windows virtual device, you must also factor in at least two to three weeks to learn all about the enhanced Windows architecture and environment.

If you have already written a Windows device driver and simply want to make it compatible with Windows 3.0 when running in protected mode, modifying and/or writing the necessary code should take only a week or two. If your device driver is simple and only does I/O and Windows function calls, any changes will be minor and can be done in a week.

If the device is complex and not similar to one of the Microsoft-supplied device drivers, the effort could take several months or longer.

You will need to do additional work to develop an enhanced Windows virtual device (VxD) if the hardware can be accessed from non-Windows programs as well as from Windows. Displays, serial communications, pointing devices, and parallel printer ports all fall into this category. The VxD serializes access to the hardware so that program output from the various programs that are running does not get mixed together. Additionally, a VxD can handle asynchronous data transfer more efficiently than a Windows device driver. Therefore, for running under enhanced Windows, you may want to move that functionality out of the device driver and into the VxD. However, the driver still needs the functionality to run under Windows in real or standard mode.

Writing a VxD for serializing access to a piece of hardware will take a week or two. Building additional functionality into a VxD for asynchronous data transfer will take a couple of weeks longer than the time it takes to implement the code for doing the actual data transfer.

If you are writing an enhanced Windows virtual device for a piece of hardware that is only slightly different from one of the standard supplied device drivers, it should only take a few weeks. However, drivers for completely different video adapters or displays may take up to several months to write.

1.5 Core Windows Modules That Interface With Your Driver

Windows has several machine-independent modules that take control of your computer's resources and maintain the user interface for application programs. Microsoft develops these Windows modules, and they are ready for use with your computer.

The following modules form the heart of Windows:

Module	Description
GDI.EXE	The Graphics Device Interface (GDI). It generates the graphics operations needed to create images on the system display and other display devices.
KERNEL.EXE	Controls and allocates all the machine resources for use with Windows. It works with your computer's operating system to manage memory, load the applications, and schedule the execution of programs and other tasks.
USER.EXE	Creates and maintains windows on the display screen. It carries out all user requests to create, move, size, or destroy a window; controls the screen's icons and cursors; and directs mouse, keyboard, and other input to the appropriate application.

You call GDI or KERNEL from your driver to request that they carry out certain functions. USER may call your driver to perform some operations.

1.6 Other Modules in the Windows Environment

In addition to the core Windows modules (i.e., GDI, KERNEL, and USER), there are other modules including device drivers that are necessary to complete the Windows environment. Each module is designed to support a unique function within the system.

The following are brief descriptions of each of these modules and device drivers: **(Lisa, are these names still correct and is the list complete? What about GRABBER.EXE and NETWORK.DRV?)**

Module	Description
COMM.DRV	Supports serial device communications.
DISPLAY.DRV	Supports the system display and pointing device cursor.
FONTS.FON	Contains system font resources.
KEYBOARD.DRV	Supports keyboard input.
MOUSE.DRV	Supports mouse or other pointing device input.
OEMFONTS.FON	Contains terminal font resources for running non-Windows applications.
SOUND.DRV	Supports the sound generation and system speaker.
SYSTEM.DRV	Supports the system timer, information about system disks, and access to OEM-defined system hooks.

Module	Description
WINOLDAP.GRB	Supports data exchange between non-Windows applications and Windows.
WINOLDAP.MOD	Supports the loading and execution of non-Windows applications.

The above generic filenames are reserved. Therefore, do *not* name your display driver DISPLAY.DRV. Instead, use a unique descriptive name with the .DRV extension. For example, the high resolution EGA display driver provided with Windows is called EGAHIRES.DRV. Or you can identify the vendor and device with a name such as V7VGA.DRV for the Video7 VGA driver.

1.7 Compiling and Linking the Driver Modules

(Lisa, I pulled this section out of the old DDK files and just cleaned up some of the terminology. We hadn't discussed this anywhere else. But it needs a good technical review to make sure it's still accurate. Thanks.)

The following files are required to build the device driver module:

Type	Description
Resource file	Defines the dialog box for the **DeviceMode** function.
Source files	Contain the device driver code, including the required functions.
INCLUDE files	Contain the definitions used by the device driver. The files PRINTER.H (C preprocessor definitions) and GDIDEFS.INC (assembly language definitions) should always be included, along with any additional INCLUDE files the device driver supplies and uses.
Libraries	Contain the supporting functions. As a minimum, device drivers must link with the C Windows library (SWINLIBC.LIB), USER.LIB, and GDI.LIB.

1.8 Windows Calling Conventions

You can write Windows device drivers in assembly language or in a Microsoft high-level language. Windows requires specific segment name and calling conventions that all Microsoft high-level languages and MASM provide. However, assembly language programmers should use the CMACROS assembly language macro package since it will provide them with these conventions automatically. They can also use MASM 5.1 and later versions, which also provide some built-in high level language features. (See Section 1.9, "How to Use the INCLUDE Files," for details on using the CMACROS INCLUDE file. See also the *Microsoft Windows Software Development Kit Programmer's Reference* for further details.)

Windows uses the following convention to call and return a device driver function:

- The CS register points to the called driver's code segment, which must not be larger than 64 kilobytes. Drivers can depend on the code segment to remain in any fixed position in physical memory if it is declared as fixed in the .DEF file.

- Whenever you write code that calls a device driver exported function, your code must execute the standard Windows prolog shown in the sample code that follows this description. The **cProc/cBegin** macro pair does this automatically for you. The standard Windows prolog sets the **DS** (data segment) register to point to the called function's DGROUP (which must not be larger than 64K).

  If declared as fixed in the .DEF file, the data segment is not moved and can be depended on to remain in place. A device driver can save data in its data segment in one function with full confidence that it will not be lost or modified by other parts of Windows.

  However, while it is fixed from the point of view of the owning code, it may not be so in the view of code outside of Windows. That is, it might be swapped out to disk or banked in EMS. Therefore, special measures must be taken by code that will be called by memory-resident software.

  However, the data segment can also be written to be moveable to allow for more flexible allocation of memory space. Drivers need to be as small as possible to ensure sufficient memory space for applications.

- **(RonG, pls review per your email on Fonts in High EMS)** When mapping screen fonts into high EMS, the fonts are locked into memory only when they are actually being used. With large-frame EMS, they are mapped into high EMS, causing the mapping out of any discardable code currently occupying the space. Once the font is locked down, the integrity of any discardable code cannot be guaranteed, and the global heap is invalidated, making it impossible to load any new discardable code. Therefore, for display drivers, all the font/text operations in the **ExtTextOut**, **StrBlt**, **GetCharWidths**, and **(perhaps?) Control** functions need to be in the fixed segment.

- The **SS** register points to the caller's (i.e., application's) stack segment, which will be different from the driver's data segment. Dynamic Link Libraries (DLLs), such as device drivers, do not have their own stack segment. They use whatever stack is available (i.e., the application's stack).

- The called function must save and restore any of the following registers that it uses: SS, **SP**, **BP**, **SI**, **DI**, and **DS**. However, if you use CMACROS.INC, the **BP** and **DS** registers are automatically saved and restored.

- The direction flag must be cleared when exiting any function that sets or modifies it. The **DS**, **SI**, and **DI** registers must also be preserved.

- A function call's code must place returned values in **AX** if they are 16-bit, and in **DX:AX** if they are 32-bit.
- Use FAR calls in your code to reach all the exported entries into a function. Each exported entry must execute a FAR return.
- The **cEnd** macro generates the proper epilog code (if necessary) and the return instruction.
- At the time of the call, all parameters for the entry are present on the stack, with the last parameter closest to the stackframe pointer, and the others at offsets deeper in the stack. Thus, CALL OEMFUNC(arg1, arg2, arg3) is implemented:

```
push      arg1
push      arg2
push      arg3
call      far  OEMFUNC
```

  The entry and exit code in the OEMFUNC function is as follows:

```
OEMFUNC         PROC      far
                mov       ax,  ds       ;Windows prolog support
                nop
                inc       bp
                push      bp
                mov       bp,  sp
                push      ds
                mov       ds,  ax
                sub       sp,  <# bytes of local stack space>
                push      si
                push      di
;
;  Now let's get the parameters off of the stack:
;
                mov       ax,  [bp+A]   ;now AX contains arg1
                mov       bx,  [bp+8]   ;similarly, BX contains arg2
                mov       cx,  [bp+6]   ;puts arg3 into CX
                ...
;
;  Body of routine here. ;
                ...
                pop       di
                pop       si
                sub       bp,  2
                mov       sp,  bp
                pop       ds
                pop       bp
                dec       bp            ;Windows epilog support
                ret       # bytes of parameter space, in this case 6
OEMFUNC         ENDP
```

- All pointer arguments are passed as 32-bit quantities, occupying two WORDs on the stack. The segment portion is pushed first, then the offset portion. This allows you to use the LDS or LES instructions to retrieve pointers from the stack.

1.9 How to Use the INCLUDE Files

When writing assembly language drivers, you will need to incorporate at least the following INCLUDE (.INC) files, which can be found in either the SDK or DDK. See the *Microsoft Windows Installation and Update Guide* for the DDK for a list of the files provided with this kit.

- CMACROS.INC
- GDIDEFS.INC
- WINDEFS.INC

Some of these contain both C and ASM definitions and, therefore, can also be used in drivers written in C. Some of the other include files provided with the DDK are WINDOWS.H and SPOOL.H, which is used by printer drivers. **(Lisa, is SPOOL.H the correct name now?)**

1.9.1 CMACROS.INC

The most important INCLUDE file is CMACROS.INC, which contains a set of assembly-language macros that were written to be compatible with the Microsoft Macro Assembler (MASM) v5.1. CMACROS.INC provides a simplified interface to the function and segment conventions of high-level languages, such as C and Pascal. **(Lisa, has it been updated to work with any newer MASM version that might have come out since this was written last year?)**

You must include this file at the beginning of the assembly-language source file by using the INCLUDE directive. You must also give the full pathname if the macro file is not in the current directory or in a directory specified on the command line.

The Cmacros are divided into the following groups:

Group	Description
Segment macros	Give access to the code and data segments that an application can use without any special definition. Medium-, large-, and huge-model applications can define additional segments by using the **createSeg** macro. These segments have the names, attributes, classes, and groups required by Windows.
Storage-allocation macros	Allocate static memory (either private or public), declare externally defined memory and procedures, and allow the definition of public labels.
Function macros	Define the names, attributes, parameters, and local variables of functions.

Group	Description
Call macros	Can be used to call **cProc** functions and high-level-language functions. These macros pass arguments according to the calling convention defined by the **?PLM** option, which is defined in the file with the Cmacros.
Special-definition macros	Inform the Cmacros about user-defined variables, function-register use, and register pointers.
Error macros	Allow assertions to be coded into an assembly-language source program. This lets you code optimum instruction sequences for some operations based on the variable allocation or bit position of a flag in a WORD, and assert that the assumptions made are true. They also generate an error message to the console and an error message in the listing. Both the text that caused the error and the result of its evaluation are displayed in the generated error message.

In other words, the Cmacros take care of many of the housekeeping tasks necessary for setting up stack frames, calling between modules written in C and those written in assembly language, and defining local and global variables.

Since the CMACROS.INC file has no comments in it, this section will include explanations of some of the functions that you will use in a Windows device driver. For information on the other functions, detailed descriptions of syntax, and individual examples, refer to the chapter on Assembly-Language Macros in the *Microsoft Windows Software Development Kit Tools* manual. **(Is the reference correct?)**

Setting Up Stack Frames

How to set up stack frames is the first concept to be discussed. The device driver is always called from the device-independent Windows Graphics Device Interface (GDI). 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]. By using the Cmacros, you can automatically retrieve these parameters in a clear and easily documented way. Otherwise, you must refer to these parameters by offsets from ss:[bp+6], which can quickly become confusing. For more information on calling conventions, refer to the *Mixed Language Programming Guide* included with the MASM documentation.

The following is a skeleton of the Bit Block Transfer (**BitBlt**) assembly language file showing how CMACROS.INC is used typically in a device driver.

```
title   BitBlt Skeleton
;
;
.xlist
```

```
memS                    ;use small model (the default)
?PLM=1                  ;use Pascal calling (the default)
?WIN=1                  ;generate prolog-epilog code (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          lpDestDev
        parmW          DestxOrg
        parmW          DestyOrg
        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
```

If you have questions on any of the terms used in the skeleton example, refer to the chapter on Assembly-Language Macros in the *Microsoft Windows Software Development Kit Tools* manual. Several Cmacros features will also be discussed here in more depth. **(Is the reference correct?)**

Keywords

WIN and **PASCAL**, as used on the **cProc** line, allow you to overrule the **?PLM** and **?WIN** flags. In the skeleton example, they are redundant. However, the sample drivers included in the DDK sometimes use them, and they are certainly harmless.

You can also use **NODATA** 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. However, this is sometimes wasteful and can result in an unwanted destruction of the **AX** register since **AX** is used to set up **DS**. Therefore, you can use the **NODATA** keyword to prevent the prolog and epilog code from modifying **DS**. For example:

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

Case Sensitivity

If you are assembling your program using MASM's case sensitivity switch (**-Ml**), some of the names documented in the Assembly-Language Macros chapter of the *Microsoft Windows Software Development Kit Tools* manual will not work. Make sure that you use the following syntax for the default segment names:

```
Code            Data            Stack
```

and:

```
CodeOFFSET      DataOFFSET      StackOFFSET
```

Also notice that the **arg** command should be in lower case.

Defining Multiple Modules

There are some easier ways to define multiple modules using the same stack frame. For example, take the case in which a **BitBlt** process, like the one shown in the skeleton example, should really be logically divided into two modules. One would contain hardware-independent code and the other would contain hardware-dependent code. However, both of them still need to share the same variables that are passed on the stack and defined as variable names by Cmacros. In such cases, some programmers would use the following *calling* sequence in module one:

```
arg     lpDestDev
arg     DestxOrg
arg     DestyOrg
etc . . .
cCall   BitBltModuleTwo
```

and the following *receiving* sequence in module two:

```
cProc   BitBltModuleTwo<FAR,PUBLIC,WIN,PASCAL>
        parmD   lpDestDev
        parmW   DestxOrg
        parmW   DestyOrg
etc . . .
cBegin
```

However, this inter-module calling sequence is extremely wasteful in terms of setting up the stack frame in both the caller and the FAR call. Instead, Cmacros allows you to create a dummy **cProc** header in all the modules that will share the same stack frame (except, of course, the calling module). Then, use the **<nogen>** qualifier on the **cBegin** line. Cmacros will create the equates for the stack frame but will not generate any code in the dummy process. After that, you can make NEAR calls to any subprocesses without any wastefulness, as shown in the following example:

In File 1:

```
externNP        Module2
;
cProc   Module1,<FAR,PUBLIC,WIN,PASCAL>,<si,di>
        parmD Param1
        parmW Param2
        parmB Param3
cBegin
|
|       some code.
|
cCall   Module2
|
|
|
cEnd
```

In File 2:

```
cProc   ModuleFamilyDummy,<FAR,PUBLIC,WIN,PASCAL>
        parmD Param1
        parmW Param2
        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
```

1.9.2 GDIDEFS.INC

GDIDEFS.INC allows you to refer to symbolic constants and structures by their Windows standard names, which is good practice. To shorten the assembly time and cross-reference lists, you can selectively include parts of GDIDEFS.INC by defining equates that tell the assembler which parts to include. These equates are listed as follows:

Equate	Definition
incLogical equ 1	Includes logical pen, brush, and font definitions
incDevice equ 1	Includes the symbolic names for GDIINFO definitions
incFont equ 1	Includes the FONTINFO and TEXTXFORM definitions
incDrawMode equ 1	Includes the DRAWMODE data structure definitions
incOutput equ 1	Includes the output style constants
incControl equ 1	Includes the escape number definitions

1.9.3 WINDEFS.INC

WINDEFS.INC contains two very useful macros that are used to turn off hardware interrupts such as those from the floppy and hard disk controllers, math coprocessor, timer, keyboard, and mouse. Use the **EnterCrit** and **LeaveCrit** macros whenever you do not want an asynchronous interrupt to reenter an area of code that Windows is executing.

Using the mouse interrupt as an example, it is possible for the mouse to generate interrupts faster than your mouse-handling code can process them. Therefore, it is likely that Windows could be updating a mouse coordinate when another mouse coordinate came along wanting to be serviced. Due to this succession of interrupts, special care must be taken to prevent any loss of mouse actions.

To manage this situation, when you are about to update your mouse coordinates, use the **EnterCrit** macro. This will stop the mouse interrupts from occurring. After you get the new mouse coordinates, then you can use the **LeaveCrit** macro to reallow interrupts. Do not use simple CLI and STI instructions to accomplish this since they will not correctly restore the states of the flags and interrupts.

Chapter 2 Display Drivers

(Lisa, do you want to provide me with a marketing-oriented spiel similar to the one you did for printer drivers? I already incorporated your last two printer paragraphs since they seemed rather generic.)

This chapter describes the support you need to provide in your Microsoft Windows display driver. Of course, the extent of the support you provide depends on the type of hardware supported. However, we strongly encourage you to implement all the structures and functions defined in this chapter, if applicable to your device. By doing so, Windows applications will be able to take full advantage of your hardware device.

The DDK includes sample code for display driver sources. These provide you with examples of how the following functions are used by Windows to display output to the screen:

- **Output**
- **Enable and Disable**
- **RealizeObject**
- **ColorInfo**
- **BitBlt**
- **StrBlt/ExtTextOut**
- **Control**

The functions are listed here and described in the following sections in the order in which we recommend you implement them. A few additional functions that you also need to implement are also briefly described here. These can be done in any order after **StrBlt**. Detailed descriptions of all these functions are provided in Chapter 10, "Common Functions."

2.1 Filling Out the GDIINFO Data Structure

The first step toward producing a successful Windows display driver is to fill out properly the GDIINFO data structure. (You can find the file for this data structure in your model driver.) The GDIINFO data structure tells Windows about the capabilities of your device. It also tells Windows applications how to expand and contract their bitmaps to achieve a

WYSIWYG appearance on your display. To ensure yourself of a consistent-looking display, you must follow exactly the calculations in this section.

The GDIINFO data structure is organized as shown in the following table. All the entries are WORDs (2 bytes). Most of the items will be discussed in greater detail in subsequent sections. For additional information on GDIINFO from a printer driver's viewpoint, see Chapter 5, "Printer Drivers."

Value	Offset	Contents
dpVersion	0	Version number (always use the number 0300H)
dpTechnology	2	Device type (Plotter=0, Display=1, Printer=2, other types are found in GDIDEFS.INC)
dpHorzSize	4	Width of display in mm
dpVertSize	6	Height of display in mm
dpHorzRes	8	X-resolution in pixels
dpVertRes	10	Y-resolution in scanlines
dpBitsPixel	12	Bits per pixel
dpPlanes	14	Number of planes
dpNumBrushes	16	Number of brushes
dpNumPens	18	Number of pens
	20	Reserved (Must be 0)
dpNumFonts	22	Number of fonts that the device has
dpNumColors	24	Number of true, non-dithered colors on device (or number of reserved colors for palette-capable devices)
dpDEVICEsize	26	Number of bytes required for PDEVICE structure
dpCurves	28	Curves capabilities
dpLines	30	Polyline drawing capabilities
dpPolygonals	32	Polygonal capabilities
dpText	34	Text drawing capabilities
dpClip	36	Clipping ability for shape drawing *only*
dpRaster	38	Miscellaneous capabilities (**BitBlt**)

Value	Offset	Contents
dpAspectX	40	X aspect
dpAspectY	42	Y aspect
dpAspectXY	44	Hypotenuse of X and Y aspect
dpStyleLen	46	Length of patterned line segments
dpMLoWin	48	Metric Lo-Res Window
dpMLoVpt	52	Metric Lo-Res Viewport
dpMHiWin	56	Metric Hi-Res Window
dpMHiVpt	60	Metric Hi-Res Viewport
dpELoWin	64	English Lo-Res Window
dpELoVpt	68	English Lo-Res Viewport
dpEHiWin	72	English Hi-Res Window
dpEHiVpt	76	English Hi-Res Viewport
dpTwpWin	80	TWIP Window
dpTwpVpt	84	TWIP Viewport
dpLogPixelsX	88	Pixels per inch in X
dpLogPixelsY	90	Pixels per inch in Y
dpDCManage	92	DC Management (always 4 for displays)
	94	5 WORDs that are reserved (must be 0)
dpPalColors	104	Number of simultaneous colors (for palette-capable devices)
dpPalReserved	106	Number of reserved system colors (for palette-capable devices)
dpPalResolut	108	Palette resolution, which equals the number of bits going into video DACS

2.1.1 Screen Metrics

The screen metrics entries include such items as width and height in mm. These values are closely related to the screen resolution, aspect, and fonts that you want to use. **(Lisa, please check the following!)**

Windows provides the following raster fonts:

- Courier - a fixed-width font
- Helv - a proportional font without serifs
- Tms Rmn - a proportional font with serifs
- Symbol - a representation of the AGFA Compugraphics POSTSCRIPT ® math symbols and the Adobe ™ POSTSCRIPT ® symbol sets
- System - a proportional font without serifs
- Fixed System - the Windows 2.0 fixed-width system font
- Terminal
- OEM

Windows also currently provides six versions of these screen and system fonts:

- COURA, HELVA, TMSRA, SYMBOLA, CGASYS, CGAFIX: for a 2 to 1, low-resolution device such as the CGA display. (Actual pixels per inch = 96 in X and 48 in Y.)
- COURB, HELVB, TMSRB, SYMBOLB, EGASYS, EGAFIX: for a 1.33 to 1 device such as the EGA. (Actual pixels per inch = 96 in X and 72 in Y.)
- COURC, HELVC, TMSRC, SYMBOLC: for a 1 to 1.2 device. (Generally used for printing devices.) (Actual pixels per inch = 60 in X and 72 in Y.)
- COURD, HELVD, TMSRD, SYMBOLD: for a 1.66 to 1 device. (Generally used for printing devices.) (Actual pixels per inch = 120 in X and 72 in Y.)
- COURE, HELVE, TMSRE, SYMBOLE, VGASYS, VGAFIX: for a 1 to 1 device such as the VGA display. (Actual pixels per inch = 96 in X and 96 in Y.)
- COURF, HELVF, TMSRF, SYMBOLF, 8514SYS, 8514FIX: for a 1 to 1 device such as the 8514/A display. (Actual pixels per inch = 120 in X and 120 in Y.)

Pixels Per Inch

For the Windows font mapper to match one of these default fonts to your display, you have to "fix" the numbers used in the various screen metrics entries. First, you must decide on what numbers to use in the two entries, "Pixels per inch in X" and "Pixels per inch in Y," which are offsets 88 and 90 in the structure. You should fill in the two entries with the "actual pixels per inch" numbers given in the above-mentioned list of fonts to ensure your device can display these fonts. For example, if the target display card has square pixels, use the closest entry for the "E" fonts or "F" fonts and put a 96 or 120 in offset 88 and a 96 or 120 in offset 90 of the data structure.

Width and Height in mm

Once you have determined the "logical" pixels per inch, you can easily calculate the width and height of the screen in mm. The equation for calculating the width in mm is as follows:

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

You can similarly calculate the height in mm as follows:

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

Feel free to round off these values to even numbers.

The Metric, English, and TWIP Windows and Viewports

Some Windows application programs rely on these numbers to produce printer output with spacing that is proportional to the screen. By using these numbers, an application could show a border or graphic picture that will be proportionately the same size on the printer as it is on the screen.

You must keep all these ratios 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. For example, Windows Write allows the user to choose whether to express the border widths in mm or inches. Therefore, it is up to the device driver to provide the correct numbers.

The Metric Lo-Res Window and Viewport consist of four WORD-length 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 Hi-Res Window and Viewport consist of four WORD-length 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 Lo-Res Window and Viewport consist of four WORD-length 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 Hi-Res Window and Viewport consist of four WORD-length 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 printer's point = 1/72 of an inch) Window and Viewport consist of four WORD-length 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)

Notice that Windows performs a WORD-length, signed calculation on these windows and viewports. Therefore, you cannot calculate numbers bigger than 32K (32,768). However, if your screen is larger than just a few inches wide, you will exceed this limit when you start calculating the English windows and viewports and may even exceed it on the Metric windows and viewports. Fortunately, you can simply scale down the calculated values by dividing them by some fixed amount. You must use the same amount to divide the "width in mm and X-resolution" and the "height in mm and Y-resolution."

For example, assume the following results to your TWIP calculation:

Width in mm = 280
Height in mm = 210
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

A divisor that gives you a number < 32K for the width/X-resolution pair is 512.

A divisor that gives you a number < 32K for the height/Y-resolution pair is 384.

Therefore, the numbers that you should put in your GDIINFO data structure are as follows:

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

The X and Y Aspect Ratios

These metric calculations are based on the aspect ratios that you must know for your display cards. That is, you must know whether your display card has a 1:1 aspect ratio (square pixels), a 1.33:1 aspect ratio (such as the EGA), or some other aspect ratio. You figured this out when you chose which font metric to use. Now you must find whole numbers that are less than 100 and that produce a whole number hypotenuse when put through the Pythagorean theorem equation. The equation is as follows:

$a^2 + b^2 = c^2$
Where c^2 is the hypotenuse.

For example, if you use 10 for both a and b when you have a square pixel display, then you will get the following:

$10^2 + 10^2 = 200$

The square root of 200 rounded to a whole number is 14. Therefore, for this example, you would put the following:

X aspect (offset 40) = 10
Y aspect (offset 42) = 10
Hypotenuse of X and Y aspect (offset 44) = 14

The following is an example for a 1.33:1 display. If you choose 48 for the X-aspect and 38 for the Y-aspect, then the calculation will give a hypotenuse (rounded to a whole number) of 61.

The Length of Patterned Line Segments

The final metric calculation is for the length of patterned (also known as *styled*) line segments. This is simply calculated as follows:

2 * Hypotenuse

Windows uses this number to make the patterned lines that it draws into bitmaps and onto displays appear correct and consistent on different displays and printers.

2.1.2 Bit Planes and Bits Per Pixel

The EGA and VGA drivers included in this kit are *planar* in nature. Therefore, put the number 4 in the "Number of Planes" (offset 14) entry. This means that they have 4 planes and are capable of 16 true, non-dithered colors (two colors per plane to the fourth power = 16). It also has the number 1 in the "Bits Per Pixel" (offset 12) entry.

Many of today's more sophisticated displays allow you to draw onto them using pixel color values, i.e., 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 "Number of Planes" entry for an 8-bit per pixel driver is 1. However, the "Bits Per Pixel" entry in the structure has the number 8, which indicates that it takes 8 bits (one byte) to represent each pixel on the display. Therefore, such a device is capable of displaying 2^8=256 colors on the screen.

The number of planes and bits-per-pixel also define the bitmap format that the device driver must understand. See Chapter 12, "Data Structures and File Formats," for more information on the BITMAP data structure.

Some display devices allow addressing of the board in either *planar* or *packed pixel* mode. However, it is more efficient for both you and Windows to use the packed pixel mode whenever possible. For more information on the packed pixel sources included in the DDK, see the *Installation and Update Notes.*

2.1.3 Supported Capabilities and the Output Function

Now you must decide what capabilities you want your driver to support. You will need to decide on which shapes you will choose to draw using your device's hardware. Windows' only requirement is 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, it may be faster for you and produce better results if you use your display's advanced hardware to draw such shapes as circles, ellipses, and alternate-fill polygons. The Curves, Polylines, and filled figure (Polygonal) capabilities allow you to tell Windows that you want it to call your **Output** function to give you a chance to draw the figure with your hardware.

Windows 2.0 and later versions give you further flexibility in supporting **Output** shapes. Assume that you can draw a polygon with 256 vertices but not with 257. Or, that you can draw ellipses to your screen but do not wish to duplicate the algorithm for ellipse drawing into main memory bitmaps (Windows, however, requires that any figure you claim you can draw onto the screen must also be able to be drawn to a main memory bitmap).

You then say that you have the ability to draw polygons and ellipses. When your driver's **Output** function is called, you can have it return a failure return code and have Windows synthesize the figure for you with the basic building blocks (scanlines, solid polylines, and pixels).

The following is a list of the **Output** function's capabilities and their corresponding offset numbers. These will be discussed in greater detail in the following subsections.

- Curves (offset 28)
- Polyline Drawing (offset 30)
- Polygonal Drawing (offset 32)

- Text Drawing (offset 34)
- Clipping (offset 36)
- Miscellaneous Raster/BitBlt (offset 38)

Curves (Offset 28)

The following table shows what you can draw when you set each bit:

Value	Bit	Capability
CC_CIRCLES	0	Circles
CC_PIE	1	Pie wedges
CC_CHORD	2	Chord arcs
CC_ELLIPSES	3	Ellipses
CC_WIDE	4	Wide, solid-curved borders around curve figures
CC_STYLED	5	Patterned lines surrounding curves
CC_WIDESTYLED	6	Wide, patterned-curved borders around curve figures
CC_INTERIORS	7	Can fill the interiors of curves

NOTE All other bits in the WORD should be set to zero. If the driver doesn't support curves, all bits should be set to zero.

Windows can use an ellipse to draw a circle if circles are not supported by the driver. If your device can fill the ellipse, then you should set the interiors bit. Windows can also use an alternate-fill polygon to draw wide borders (both solid and patterned) just as efficiently as if the driver supported them correctly.

Polyline Drawing (Offset 30)

The following table shows what you can draw when you set each bit:

Value	Bit	Capability
LC_POLYLINE	1	Polylines (all display drivers must set this bit)
LC_MARKER	2	Reserved
LC_POLYMARKER	3	Reserved

Value	Bit	Capability
LC_WIDE	4	Wide lines
LC_STYLED	5	Patterned lines
LC_WIDESTYLED	6	Wide patterned lines
LC_INTERIORS	7	Can fill the interiors of wide lines

NOTE All other bits in the WORD should be set to zero. If the driver doesn't support any line capabilities, all bits should be set to zero.

If your device supports alternate-fill polygons, then Windows can efficiently use the polygons to create wide lines. However, if your device supports wide lines, you might want to support them, since you cannot "fail" on drawing wide lines into a main memory bitmap.

If you support styled lines, make sure that the lengths of the line segments that your hardware draws are the same as those at offset 46 (**dpStyleLen**) of GDIINFO. Also, if you support wide or styled polylines, you must support them to both main memory bitmaps and to your screen.

NOTE If you decide to support styled lines, you must support them to both main memory bitmaps and screen bitmaps. This is because Windows will not let you return a failure from **Output** for any of the line styles.

If your hardware supports styled and wide lines, it is probably worth the effort to implement them, even though they are used rather infrequently. However, Windows also does a fine job of synthesizing them by using pixel draws, which are slow but work. (In the sample drivers, wide lines are not supported, but styled lines are.)

Polygonal Drawing (Offset 32)

The following table shows what you can draw when you set each bit:

Value	Bit	Capability
PC_POLYGON	0	Alternate-fill polygons
PC_RECTANGLE	1	Rectangles
PC-TRAPEZOID	2	Winding number fill polygons
PC_SCANLINE	3	Scanlines (all display drivers must set this bit)
PC_WIDE	4	Wide borders around polygonal figures

Value	Bit	Capability
PC_STYLED	5	Patterned borders around polygonal figures
PC_WIDESTYLED	6	Wide patterned borders around polygonal figures
PC_INTERIORS	7	Can fill the interiors of polygonal figures

NOTE All other bits in the WORD should be set to zero. If the driver doesn't support polygons, all bits should be set to zero.

Again, we do not recommend supporting wide borders since Windows will use alternate-fill polygons (i.e., the kind that most hardware supports) to produce these borders.

There are no drivers currently written that support the winding number fill polygons. Most hardware is incapable of doing winding number fill polygons because it is a fairly complex algorithm.

However, we do recommend patterned border support if the hardware supports it. Great speed increases are possible with polygonal patterned borders.

Text Drawing (Offset 34)

The following table shows what you can draw when you set each bit. The bits marked with an asterisk (*) are the ones you *should* set (if your device has the capability).

Value	Bit	Capability
TC_OP_CHARACTER	0*	Can draw text with pixel justification (required)
TC_OP_STROKE	1*	Can do everything that bit 0 set can, and also rotate text
TC_CP_STROKE	2*	Can clip to a pixel boundary(required for displays)
TC_CR_90	3	Can rotate text 90 degrees (does not work)
TC_CR_ANY	4	Can rotate text to any angle (does not work)
TC_SF_X_YINDEP	5	Can scale a font in X and Y directions independently
TC_SA_DOUBLE	6	Can double the size of the font

Value	Bit	Capability
TC_SA_INTEGER	7	Can scale the font by any integer size (3X, 4X,...)
TC_SA_CONTIN	8	Can scale the font by any amount
TC_EA_DOUBLE	9*	Can bold the font
TC_IA_ABLE	10*	Can italicize the font
TC_UA_ABLE	11*	Can underline the font
TC_SO_ABLE	12*	Can do a "strikeout" on the font
TC_RA_ABLE	13*	Can draw with raster fonts (required for displays)
TC_VA_ABLE	14*	Can draw with vector fonts (not on displays; only for plotters.)
TC_RESERVED	15	Reserved

Clipping (Offset 36)

If your hardware device can "scissor clip" to a rectangular region, then you should put a 1 here and support clipping in your driver's **Output** function.

This capability is used only by Windows to determine whether or not you can clip **Output** shapes. Text *must* be clipped to a pixel boundary by the driver no matter what is placed in this field.

Output, **StrBlt** (**ExtTextOut**), **BitBlt**, **SetDIBitsToDevice**, and **UpdateColors** (for palatte-capable devices) are the only functions that require any clipping.

Miscellaneous Raster/BitBlt (Offset 38)

Many of these capabilities are required, and not optional, for displays. For example, Microsoft has evaluated some Windows 2.x drivers that do not support huge (>64K) bitmaps. Many applications, such as Windows Paint, depend on this support and will not work correctly if the display driver does not handle them. Also, if you expect to support most of the major applications, your display driver must support **ExtTextOut**.

Since **FastBorder** shares a bit with the **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.

The following table shows what you can draw when you set each bit. The ones marked with an asterisk (*) are required for display drivers.

Value	Bit	Capability
RC_BITBLT	0*	Can do **BitBlt**
RC_BANDING	1	Requires GDI banding support (printers only)
RC_SCALING	2	Requires GDI scaling support (printers only)
RC_BITMAP64	3*	Supports huge, >64K (multi-segment) bitmaps
RC_GDI20_OUTPUT	4*	Supports **ExtTextOut**, **FastBorder**, and **GetCharWidth**
RC_GDI20_STATE	5	State block support (printers only)
RC_SAVEBITMAP	6	**SaveScreenBitmap** capability (*strongly* recommended for displays)
RC_DI_BITMAP	7	Can do **Get** and **Set** DIBs and RLE to and from memory in all the DIB resolutions (1,4,8, and 24 bits-per-pixel). However, if the flag is *not* set, GDI will simulate in monochrome.
RC_PALETTE	8	Can do color palette management
RC_DIBTODEV	9	Can do **SetDIBitsToDevice**
RC_BIGFONTS	10	Can do >64K fonts (set only in protected mode) in the new version 3.0 format. However, if the flag is *not* set, all the fonts will be in the old version 2.0 format.
RC_STRETCHBLT	11	Can do **StretchBlt**
RC_FLOODFILL	12	Can do **FloodFill**

2.2 The Enable and Disable Functions

The following are the call parameters for **Enable** and **Disable**:

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

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

GDI calls the **Enable** function twice for display drivers. The first time, the passed variable is *Style* = 1. This means that GDI wants you to move your GDIINFO data structure into the area pointed to by *lpDestDev*. You must return, in the AX register, the size in bytes (= sizeGDIINFO) of your GDIINFO data structure.

The second time (with *Style* = 0), three things must occur. These are discussed in the following sections.

2.2.1 Initializing Your Graphics-Board Hardware

(Peterbe, does this section just refer to 2.x drivers now? Has this been fixed?)

First, you must initialize your graphics-board hardware to be ready to run Windows. However, there is a special caveat here. For Windows to properly initialize its keyboard, you must set the byte at 40H:49H into an IBM® ROM BIOS-compatible graphics mode. For many high-resolution devices that do not use the ROM BIOS to set up their modes, this may seem unnecessary. However, the Windows keyboard will be locked out unless this is 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. Be sure to save the original byte at 40H:49H so you can restore it at **Disable** time.

The following is 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
```

2.2.2 Initializing Your Other Hardware

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

While initializing, you must call a special function (INT 2FH) to make your Windows driver 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, you must be sure to leave and reenter Windows in an orderly fashion when switching in and out of the Compatibility Box.

Your hardware could get extremely confused if OS/2 switched away from you while you were in the middle of setting up for a draw! If OS/2 calls into the Compatibility Box using

INT 2FH when you are in a state from which you cannot switch in an orderly fashion, you can return a failure code to OS/2. OS/2 will keep trying to call you until it receives a successful code. Then, you should save any states that you need to restore upon reentry and allow the switch to occur.

When you switch back into the Compatibility Box, you reinitialize your hardware and call Windows to repaint the screen. OS/2 uses the INT 2FH functions 4001H (**Notify Background Switch**), to switch from the Compatibility Box to OS/2, and 4002H (**Notify Foreground Switch**), to switch back into the Compatibility Box. (See the following subsections for descriptions of these functions.)

OS/2 also uses the following functions:

Number	Name
4000H	**Enable VM-Assisted Save/Restore**
4003H	**Enter Critical Section**
4004H	**Exit Critical Section**
4005H	**Save Video Register State**
4006H	**Restore Video Register State**
4007H	**Disable VM-Assisted Save/Restore**

Therefore, the **Enable** function must hook INT 2FH and check each call to that interrupt to see if it is one of the above-mentioned functions. You hook the interrupt by using the following MS-DOS functions: 35H to get the old vector and 25H to set the new vector. Be sure to save the address of the old interrupt.

You should look at the SSWITCH.ASM file in one of the sample drivers to see how INT 2FH is hooked. Notice that you must hook INT 2FH, even when running under versions 3.x or 4.x of MS-DOS, because many network systems (including MS-NET) that are running under a "real mode" MS-DOS will want to use the same functionality that OS/2 uses. See the following subsections and Volume 2, Chapter 42, "INT 2FH API," for more detailed information.

However, *before* you hook INT 2FH, you must get the address of a special function in the Windows USER module that forces a repaint of the entire screen. This is because when you switch back from OS/2 to Windows, which is running in the Compatibility Box, you must restore the screen to the state in which it was when you exited. Fortunately, the USER module's function can do this automatically for you.

To get the address for this special function, first call the Windows function **GetModuleHandle**. This returns a special identifier to the USER module called a *handle*.

Once you have the handle to the module, you 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 you a long pointer to the repaint function, which you then save and call when appropriate.

(MarcW, please review carefully my edited text here.)

INT 2FH/AX=4000H - Enable VM-Assisted Save/Restore

A Virtual Machine (VM) application (such as Windows) can issue this call when it is initializing to determine what level of virtualization the Virtual Display Device (VDD) supports and to disable I/O trapping of unreadable registers whenever this VM is in the foreground. The VDD instead relies on the VM's INT 2FH support to save and restore the VM's register state (see functions 4005H and 4006H). If this capability is enabled, the VDD returns in AL a non-zero value, which may be one of the following:

001H -	No modes virtualized in background
002H -	Only text modes virtualized in background
003H -	Only text and single-plane graphics modes virtualized
0FFH -	All supported video modes virtualized

The state of the video adapter at the time this call is made will be the state restored prior to **Notify Foreground Switch** (function 4002H) and requests by **Restore Video Register State** (function 4006H). Also, video memory is no longer saved across screen switches; it is the application's responsibility to completely reinitialize video memory after a **Notify Foreground Switch** request.

INT 2FH/AX=4001H - Notify Background Switch

The VDD issues this call to a VM that is being unconditionally switched to the background. Once this call is complete, the VM can continue to run. However, if it accesses video memory while in an unvirtualized video mode, it will be frozen until brought to the foreground again. The VM must return from this call within 1000ms; otherwise, the screen switch will proceed anyway.

It is expected, though not required, that an application that has enabled **VM-Assisted Save/Restore** (function 4000H) will not access video memory or registers after this notification, to avoid being frozen in a video mode that cannot be virtualized. However, any application that does so can still be detected and frozen if the operation cannot be virtualized. When **VM-Assisted Save/Restore** is not enabled, the VM's registers and memory are completely saved after this call has returned (or timed-out).

INT 2FH/AX=4002H - Notify Foreground Switch

The VDD issues this call to a VM that is being unconditionally switched to the foreground. The VM can assume that it once again has complete access to the physical display hardware. No time-out is enforced on this call.

If the VM has enabled **VM-Assisted Save/Restore**, it is now expected to reinitialize completely the video memory. The state of the adapter will already be restored to the state that existed when function 4000H was issued. If **VM-Assisted Save/Restore** is not enabled, the full state of the adapter (memory and registers) will already be restored, and this call need not be acted upon.

Under certain error conditions, this notification may be issued without a corresponding **Notify Background Switch** (function 4001H); an example is the critical section time-out, discussed in the following two subsections.

INT 2FH/AX=4003H - Enter Critical Section

A VM application (such as Windows) issues this call whenever it is in a critical section and consequently cannot respond to a **Save Video Register State** request (function 4005H). When a **Save** is required (e.g., to reprogram temporarily the video hardware to perform a Clipboard copy operation) and the VM is in a critical section, the required operation is postponed for up to 1000ms or until the **Exit Critical Section** call (function 4004H) is made, whichever comes first. If time-out occurs, then the VDD reprograms the hardware anyway and, when its operation is complete, initiates the **Notify Foreground Switch** request (described earlier in this section), in an attempt to reinitialize the application properly.

A count of **Enter Critical Section** requests is kept, so that nested calls can be made. If the count will overflow, the **Enter** request is ignored.

INT 2FH/AX=4004H - Exit Critical Section

A VM application (such as Windows) issues this call when it has completed its critical section processing. If there is a pending **Save Video Register State** request, then it is performed immediately afterward.

The count of **Enter Critical Section** requests is decremented. If the count will underflow, the **Exit Critical Section** request is ignored.

INT 2FH/AX=4005H - Save Video Register State

The VDD issues this call when it requires access to the video hardware registers (e.g., for a full-screen Clipboard copy operation). The VM receiving this call must save any data necessary to restore effectively its video state when a **Restore Video Register State** request (function 4006H) is issued later. The VM must return this call within 1000ms; otherwise, the required operation will proceed anyway.

This call is issued only if the VM has enabled **VM-Assisted Save/Restore** (see function 4000H). It is not issued prior to **Notify Background Switch** calls (function 4001H); it is issued only at times when the hardware must be reprogrammed for what are essentially brief and non-visible operations.

INT 2FH/AX=4006H - Restore Video Register State

The VDD issues this call when it relinquishes to a VM the access to the video registers. The VM receiving this call should restore any register states necessary to continue uninterrupted foreground operation. No time-out is enforced on this call.

This call is issued only if the VM has enabled **VM-Assisted Save/Restore** (see function 4000H). Whatever registers the VDD modified are restored to the state saved at the time of function 4000H. In other words, before this call is issued, every register is guaranteed to be either unchanged or reset to the state at the time of function 4000H; precisely which registers may be reset is undefined, but the set is restricted to those Sequencer and Graphics Controller registers that do not affect the display.

INT 2FH/AX=4007H - Disable VM-Assisted Save/Restore

A VM application (such as Windows) issues this call when it is terminating to re-enable I/O trapping of unreadable registers whenever this VM is in the foreground. The INT 2FH functions that save and restore the VM's register state (4005H and 4006H, defined earlier in this section) are no longer issued for this VM, and the enter/exit critical section services (4003H and 4004H, also defined earlier) are ignored.

2.2.3 Copying Your PDEVICE Data Structure

The last thing to do while in your **Enable** function is to copy the PDEVICE structure you want to the area pointed to by *lpDestDev*. This call to **Enable** should return a 1 in **AX** if all was successful. Otherwise, it will return a 0.

The PDEVICE data structure defines so-called *physical* objects used solely by the device driver to identify to itself such things as bitmaps, pens, and brushes. Therefore, the contents of this data structure are normally determined by the device driver writer.

The PDEVICE structure has only one WORD-length field that 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 *lpDestDev* 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 the other fields in the PDEVICE structure may or may not be used by the driver in whatever way it wants to.

The "bitmapped" displays (such as the CGA and VGA) duplicate a BITMAP data structure into the PDEVICE structure. This is because, in most cases, their drawing functions 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 just want to store some appropriate states in it. This is totally up to you.

2.2.4 Comments on the Disable Function

This is a simple function. When the **Disable** function is called, first return your device to the state in which it was when you started Windows, and then restore the byte at 40H:49H to its original state and unhook yourself from INT 2FH.

In protected mode, the device should now ask enhanced Windows to start I/O trapping again.

The **Disable** function is called whenever the Windows graphics mode is about to terminate. That is, whenever the user wishes to leave Windows or switch to a "badly behaved" non-Windows application (one that cannot run in a window since it relies on 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 a non-Windows application, the **Enable** function (with *Style* = 0) is called.

Over a non-Windows 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 accordingly any data that it needs to have saved. Also, your **Enable** process should reinitialize any flags relating to screen states since these will probably be destroyed by the exit to the non-Windows application.

2.3 The RealizeObject Function

The following are the call parameters and return values for the **RealizeObject** function:

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

Returns:

If *lpPhysicalObj* = 0, it returns the size required for a physical object in AX.

If *lpPhysicalObj* <> 0, it returns 1 if successful and zero if unsuccessful.

2.3.1 Background Information

The Windows Graphics Device Interface (GDI) is a device-independent graphics drawing engine. It communicates with a Windows application through the Windows Application Programming Interface (API), which is documented in the *Microsoft Windows Software Development Kit*. 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:

- Pen
- Brush
- Font

The *pen* is used to draw polylines and borders around objects drawn by the **Output** function. It has three attributes:

- Color
- Style (or pattern, such as dotted lines)
- Width

The second object is called a *brush* (or pattern). This object is used to fill figures drawn by **Output**, and to fill (with some logical operation) rectangular areas drawn by **BitBlt**.

For example, the rectangular areas that make up a Windows screen are all drawn by **BitBlt** using a brush. The brush has the following attributes:

- Pattern (an 8-pixel by 8-pixel repeating block pattern)
- Color(s)
- Hatch (predefined patterns that use an explicit foreground and background color that is assigned to them)

The last object is the *font* that is used to draw text by the **StrBlt** and **ExtTextOut** functions. Display drivers generally do not realize fonts and should fail with an error return code of zero if asked to do so.

Hardware rarely uses the exact same representation of a Windows object that Windows does. For example, it is inconvenient for the IBM 8514 display adapter 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** function is where the translation between device-independent (or logical) and device-optimal (or physical) objects takes place.

But 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 pretranslated objects and might use this same object in hundreds of different draws. The translation is done only once for many draws.

2.3.2 General Attributes

The following is a brief discussion of the general attributes of the **RealizeObject** function and how best to use it for display drivers.

First, we must 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.

However, some displays support many of the complex drawing functions that Windows allows the driver to support. For example, assume there is a certain display device that supports all sorts of patterned line drawing with any width of line.

Normally, the device driver writer would register all of these capabilities into his GDIINFO data structure and, then, write a polyline routine with a physical pen that supports wide and styled lines. Everything seems to work well. The writer then runs some of the toy applications included in this kit and everything works really fast.

Unfortunately, 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 the algorithm, but it also makes the device driver unnecessarily large. Therefore, the device driver writer should often allow GDI to support the more complex pen styles and widths, even though there is a sacrifice of some speed when drawing with these pens.

2.3.3 The Pen Object

The logical pen has the following structure:

Value	Offset	Description
lopnStyle	0	Pen style
lopnWidth	2	Width of pen in pixels
	4	Height of pen in scanlines
lopnColor	6	RGB pen color (doubleword: high byte is 0)

The following are the possible styles that can be passed in offset 0 of the logical pen structure:

Value	Style code	Description
LS_SOLID	0	Solid line
LS_DASHED	1	Dashed line

Value	Style code	Description
LS_DOTTED	2	Dotted line
LS_DOTDASHED	3	Dot-dashed line
LS_DASHDOTDOT	4	Dash-dot-dotted line
LS_NOLINE	5	NULL. Draw no line.

Be sure to support the NULL style in your drawing code. If you get a pen with this style, you should draw no line and return a success code.

You might want to *not* support wide and/or styled lines. If you do support them, you must make sure that you support the same styling algorithms when drawing to the screen and main memory. However, GDI is able to synthesize correctly both wide and styled lines quite efficiently. Therefore, you may not want to support them in your first pass and add their support later only if necessary. To do so, just set the correct bits in your GDIINFO data structure to tell Windows that you do not 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 have told GDI that you do not 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 and wide-line activities itself.

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

2.3.4 The Brush Object

The logical brush has the following structure:

Value	Offset	Description
lbStyle	0	Brush style (0=Solid, 1=Hollow, 2=Hatched, 3=Patterned)
lbColor	2	For solid brushes: RGB color (high byte = 0) For palette-capable devices, if the high byte is not zero, then the low WORD is an index and not an RGB. For hatched brushes: RGB foreground color (high byte = 0) For patterned brushes: pointer to pattern BITMAP structure
lbHatch	6	Hatch style (not used for other brush styles)

Value	Offset	Description
lbBkColor	8	Physical color for hatch background (*not* RGB. See Section 2.4, "The ColorInfo Function," for a description.)

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

Solid brushes can be dithered. If your *Style* is solid and does not match exactly the color that you have in your palette, then you probably want to dither it. For an example of how to dither, refer to the sample code included with the DDK.

The following are the possible hatch styles that can be passed in offset 6 of the logical brush structure:

Value	Style code	Description
HS_HORIZONTAL	0	Horizontal (——)
HS_VERTICAL	1	Vertical (||||)
HS_FDIAGONAL	2	Forward diagonal (////)
HS_BDIAGONAL	3	Backward diagonal (\\\\\\\\)
HS_CROSS	4	Cross (++++)
HS_DIAGCROSS	5	Diagonal crosshatch (XXXX)

2.3.5 Using the RealizeObject Parameters

It is important to understand how to use the **RealizeObject** parameters. When your display is called, you must first determine whether the caller (GDI) actually wants you to realize an object, or if it is 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 zero. 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 zero.

If GDI is asking you to 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 or not you are to realize a pen (=1), brush (=2), or font (=3). You would then do the translation and return **AX**=1 as a success code or **AX**=0 as a failure code. For example, if you do not support the realization of fonts, you would return **AX**=0.

The last parameter is "dual-purpose." If you are asked to realize a font, it is a pointer to the TEXTXFORM data structure. (See Chapter 13, "The Font File Format," for more information.)

If you are asked to realize a brush or patterned pen, this parameter is *not* a pointer. It is actually two WORDs — the starting coordinates in X and Y of the window in which the application (the one that called GDI) is running.

Therefore, it is essential 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 the X-coordinate (X-origin+8), the pattern begins to repeat itself.

During **BitBlt**, the pattern reference point must be at the beginning of the window in which the application is running. Therefore, you must rotate any patterns so that they begin their repetition relative to the application's window. If you do not rotate them, you 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.

2.4 The ColorInfo Function

The following are the call parameters and return values for the **ColorInfo** function:

```
cProc           ColorInfo,<FAR,PUBLIC>,<si,di>
        parmD           lpDestDev
        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 zero.

If *lpPhysicalColor* is not NULL, **DL:AX** contains the RGB value of the device's color that most closely matches the color passed in *ColorIn*. **DH** must be zero.

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. However, for palette-capable devices only, if the high byte is not zero, then an index (WORD) is passed and not an RGB color.

Since the **RealizeObject** function also requires the translation from logical to physical colors when creating physical pens and brushes, you may have already written most of this function when you wrote the **RealizeObject** function. Simply follow the instructions found above and in the description for the **ColorInfo** function given in Chapter 10, "Common Functions."

NOTE The high byte of any doubleword RGB color returned by your device driver must be zero.

2.5 The BitBlt Function

The following are the call parameters for the **BitBlt** function:

```
cProc           BitBlt,<FAR,PUBLIC>,<si,di>
        parmD           lpDestDev
        parmW           DestxOrg
        parmW           DestyOrg
        parmD           lpSrcDev
        parmW           SrcxOrg
        parmW           SrcyOrg
        parmW           xExt
        parmW           yExt
        parmD           Rop3
        parmD           lpPBrush
        parmD           lpDrawMode
```

2.5.1 Background Information

BitBlt is perhaps the most important function used in Windows. You might want to implement it first so that you can go into the debugger and watch Windows take shape on your screen. It is **BitBlt** that actually draws on the screen the rectangles that comprise the Windows desktop. It also draws the icons and other bitmaps, but not the cursor.

When Windows first starts up, the following occurs:

1. **Enable** is called twice.
2. **ColorInfo** is called a number of times.
3. **RealizeObject** is called to create the default Windows pens and brushes (black solid pen, white solid pen, black brush, white brush, etc.).

(Ask Chip how to change this paragraph.)

Then the Windows MS-DOS Executive begins execution and calls **ColorInfo** and **RealizeObject** to create the brushes for its screen. Notice that *no* pens are used for the MS-DOS Executive screen, and that no Polylines or Scanlines are drawn. **BitBlt** and **StrBlt** do all the drawing on this screen. Therefore, these are the only two drawing functions that you need to implement to see your driver running the MS-DOS Executive. You can write stub functions for all the rest.

The first thing that **BitBlt** draws is the colored screen background. To do this, it uses the **BrushCopy** raster operation to draw a rectangle. Depending on how you treated this brush in your **RealizeObject** function, it will be either a solid or dithered brush.

Next, **BitBlt** draws a number of borders and rectangles, also using various brushes that you already realized.

NOTE When debugging your driver for the first time, it is a good idea to set SYMDEB or WDEB386 breakpoints at **ColorInfo, RealizeObject**, and **BitBlt** so that you can see how brushes are realized and used in Windows.

Lastly, **BitBlt** puts up the corner icons. These are monochrome bitmaps that are first drawn to a main memory bitmap that is maintained by USER. This composite bitmap is created using **BitBlt** during the early part of Windows initialization. (This one bitmap contains all the bitmaps that are part of the driver's resources.) These monochrome bitmaps are then transferred to the screen.

2.5.2 The BitBlt Parameters

This section discusses in detail how to use each of the passed parameters for **BitBlt**. Additional information on the function and its parameters is provided in Chapter 10, "Common Functions."

lpDestDev

This is a long pointer to a PDEVICE data structure. If it is a main memory bitmap (i.e., WORD 0 of the structure = 0), it will be a BITMAP data structure. (See Chapter 12, "Data Structures and File Formats," for the documentation on these structures.)

If the destination is the device (i.e., screen, printer, etc., and WORD 0 of the structure = 2000H), then the structure is whatever you defined the PDEVICE data structure to be at **Enable** time. You should determine the characteristics of the destination bitmap from this structure. Such things as its color format, width, and height can be extracted from the structure.

Remember that the destination can be either a bitmap in main PC memory or the device. There is *always* an *lpDestDev* passed to **BitBlt**.

DestxOrg and DestyOrg

These are the starting X and Y coordinates for the draw on the destination bitmap (or device).

There are a number of calculations that you need to do if you are drawing into a bitmap. Because the bitmap is arranged as a series of addresses (called *linear addresses*), you must convert the X and Y coordinates into these linear addresses. The following is an example of how to do this for a monochrome (1 bit-per-pixel) bitmap. Notice that by varying this macro somewhat, you can do the calculation for any color format.

```
include CMACROS.INC
include GDIDEFS.INC
ConvertXYToLinear       macro
local   GetStartingLineAddress
        mov     ax,DestxOrg         ;;get starting X
        mov     dx,ax               ;;copy this for bit offset calc

;;Get the byte containing the starting X-coordinate into AX:

        shr     ax,3                ;;divide by 8 (8 bits per byte)

;;The remainder of the divide by 8 is the bit offset into the byte of
;;the starting X-coordinate.

        and     dl,07h              ;;this is the way we get a remainder

;;Now AX & DL contain the linear byte and bit offset of the starting
;;X-coordinate on each line. Save them for use in the BLT loop:

        mov     DestxOrgByteOffset,ax
        mov     DestxOrgBitOffset,dl

;;Now it's time to obtain the linear address of the starting-Y. Since we
;;may have a huge bitmap, we need to get the line offset into the proper
;;segment:

        mov     ax,DestyOrg         ;;get starting Y
        xor     bx,bx               ;;initialize huge bitmap segment adder
        xor     dx,dx               ;;initialize nbr of lines per segment
        lds     si,lpDestDev                ;;get pointer to BITMAP structure
                                            ;;into DS:SI
        mov     cx,[si].bmSegmentIndex  ;;get the huge bitmap flag

;;The huge bitmap flag will be 0 if the bitmap is a small one.
;;Skip huge bitmap processing if bitmap is small (CX = 0):

        jcxz    GetStartingLineAddress

;;We have a huge bitmap. Given the DestyOrg in AX, we can find which
;;segment the bitmap is in and the line's offset within that segment.

        mov     bx,[si].bmScanSegment   ;;get nbr of scanlines per segment
                                        ;;from BITMAP
        div     bx

;;After the divide, AX will have the segment offset of the DestyOrg, and
;;DX will have the starting line within that segment. By multiplying
;;the result in AX by 1000H (64K), we will get the number of segments to
;;add on to our starting segment to get to the segment containing the
;;starting Y-coordinate.

        mov     di,dx               ;;save line offset from multiply
        mul     cx                  ;;multiply by 64K (remember, CX has huge
                                    ;;increment; value depends on mode of processor)
        mov     bx,ax           ;;save this result in BX
        mov     ax,di           ;;restore saved line offset to AX

GetStartingLineAddress:

;;The following code applies to both small and huge bitmaps.
;;At this point:
;;      AX contains the line offset into the segment of the starting line.
;;      BX contains the amount to be added to the segment address to get
;;      to the starting Y-coordinate's segment.

        mov     cx,[si].bmWidthBytes    ;;get number of bytes per line from
                                        ;;BITMAP
        mul     cx

;;Now AX contains the linear address within the segment of DestyOrg.
;;It's now time to add everything together to get to the starting byte
;;of the BLT.

        lds     si,[si].bmBits          ;;now DS:SI points to the bitmap's start
        mov     dx,ds                   ;;get to correct segment in the bitmap
        add     dx,bx
        mov     ds,dx                   ;;now DS points at the correct segment
        add     si,ax                   ;;now DS:SI points at starting line
        add     si,DestxOrgByteOffset

;;Now DS:SI points to the byte containing the starting (X,Y) coordinate
;;in the bitmap.

endm
```

lpSrcDev

This pointer may point to the source PDEVICE data structure or it may be NULL. To determine whether or not this parameter means anything, you must interpret the *Rop3* parameter to see if a source is involved in the block transfer (see the next subsection).

If the *Rop3* parameter does not include a source, then this pointer points to NULL. If there is a source, this is the PDEVICE of that source.

SrcxOrg and SrcyOrg

Again, if the *Rop3* parameter indicates that there is a source operand in the block transfer, then these two parameters will contain the starting X and Y coordinates of the source of the block transfer. If there is no source involved in the block transfer, these two parameters will be ignored.

The **BitBlt** function can be difficult in that *SrcxOrg* and *SrcyOrg* may be *negative* (due to scaling done by GDI). Therefore, we recommend you include the following code fragment in your **BitBlt** function:

```
ClipBitBltSource        macro
local   CheckYClip
local   DoneClipping
        mov     ax,SrcxOrg      ;;get starting X
        or      ax,ax           ;;is it negative?
        jns     CheckYClip      ;;no, continue

;;If the starting X-coordinate is negative, we must adjust the SrcxOrg
;;to 0, bump down the xExt by the amount that we clipped, and advance
;;the DestxOrg by the amount that we clipped.

        neg     ax              ;;get amount clipped in X
        sub     xExt,ax         ;;adjust the xExt
        add     DestxOrg,ax
        mov     SrcxOrg,0

CheckYClip:

;;Now, check the Y-clipping in a similar manner:

        mov     ax,SrcyOrg      ;;get starting Y
        or      ax,ax           ;;is it negative?
        jns     DoneClipping    ;;no, continue

;;If the starting Y-coordinate is negative, we must adjust the SrcyOrg
;;to 0, bump down the yExt by the amount that we clipped, and advance
;;the DestyOrg by the amount that we clipped.

        neg     ax              ;;get amount clipped in Y
        sub     yExt,ax         ;;adjust the yExt
        add     DestyOrg,ax
        mov     SrcyOrg,0

DoneClipping:
endm
```

xExt and yExt

These are the width and height of the block transferring area. By adding them to the X and Y origins and subtracting 1, you can get the ending X-coordinate of a line. Notice that these apply both to the source and the destination.

Rop3

This parameter is crucial to your understanding of **BitBlt**. (See Chapter 14, "Raster Operation Codes and Definitions," for a list of the codes as well as a detailed description of how to read *Rop3* codes and reverse Polish notation.)

The *Rop3* parameter (known in the *Windows 2.0 Adaptation Guide* as *Rop*) is a ternary (three operand) raster operand. That is because there can be three operands (i.e., source, destination, and pattern) involved in the block transfer. The *Rop3* parameter describes which of the three operands is involved in the block transfer and what you must do with each operand. In contrast, the **Output** and **Pixel** functions use a binary raster operation (*Rop2*), which involves only the brush (or pen) and the destination. No source is involved.

The first thing that you should do in your **BitBlt** function is to "decode" the *Rop3* parameter. What you must learn from the *Rop3* code is the number of operations to do in the block transfer, the operands involved, and the actual operation script. For memory-mapped boards (those similar in architecture to the CGA, EGA, VGA, and Hercules), you can use the prototype functions included in the sample drivers. For hardware such as the 8514/A or TI-34010-based boards, you should probably construct a table of the 256 possible *Rop3* parse strings, the number of operations involved in the block transfer, and the operands involved. Such a table is given here as an example:

```
OperationSource        equ     1
OperationPattern       equ     2
OperationDest          equ     3
OperationUnaryNOT      equ     4
LogOpBLACK             equ     0 SHL 4
LogOpWHITE             equ     10h
LogOpReplace           equ     20h
LogOpAND               equ     30h
LogOpOR                equ     40h
LogOpXOR               equ     50h
;
;An example parse string for Rop3 number 0BH (PSDnaon)
;would then be
;
RopB    db      OperationUnaryNOT
        db      OperationSource + LogOpAND
        db      OperationPattern + LogOpOR
        db      OperationUnaryNOT
RopBLength
equ
        $-RopB
```

In addition to the table of parse strings, you must also create a lookup table so you can locate the given parse string for decoding the *Rop3*. This lookup table can have the following efficient format:

Location	Description
WORD 0	Offset of the parse string
Byte 2	Number of operations in the *Rop3* (in the case of RopB, it would be 4)
Byte 3	Operands-present flag (bit 0 = source present, bit 1 = brush present, bit 2 = destination must be saved)

Then, at the beginning of the block transfer, you can do the lookup based on the passed *Rop3*:

```
mov     bx,DataOFFSET RasterOpLookupTable
mov     di,seg_Rop3
shl     di,2                           ;each entry has 4 bytes
```

```
mov     ax,[bx+di]                      ;get address of parse string
mov     ParseStringAddress,ax           ;save it
mov     ax,[bx+di+2]                    ;get nbr of operations in Rop3
                                        ;in AL, operands present flags
                                        ;in AH
mov     ParseLoopCounter,al             ;save it
mov     OperandsPresentFlags,ah         ;
```

You will then have taken all the information necessary for doing the block transfer from the *Rop3* parameter. By testing bits in the operands-present flag, you can make determinations such as whether or not there is a source or a brush involved, and whether or not the destination must be saved.

The concept of saving the destination is an important one. Assume you are told to execute *Rop3* code 8BH. Its reverse Polish string is DSPDxoxn. Its parse string would be represented as follows:

```
Rop8B   db OperationPattern+LogOpXOR    ;this destroys the destination
        db OperationSource+LogOpOR
        db OperationDestination+LogOpXOR;the destination has
                                        ;already been destroyed
        db OperationUnaryNOT            ;
```

As you can see, this *Rop3* could not be done unless the destination was saved from being destroyed by the first operation! Therefore, in the operands-present flag, you should reserve a bit that tells you that you must pre-save the destination from destruction before you begin performing the block transfer.

The last *Rop3* concept is actually a shortcut. As you can imagine, typing in parse strings for all 256 *Rop3s* is quite tedious, not to mention the huge memory requirement for all 256 parse strings and lookup table entries! However, the Rop codes 128 through 255 are simply the Rop codes 0 through 127 with a NOT added on to the back. Therefore, you only need 128 parse strings. If the *Rop3* code is >=128, you can add on a NOT to the end of the *Rop3* operation. Also, if the original *Rop3* parse string ends with a NOT operation, you can cancel the two NOTs and save two operations.

For example:

```
Rop code ØFH is a NOT Pattern
Rop code FØH is a Pattern copy
```

Since Rop F0H is normally >128, you would take its inverse (0FH) and add on a NOT to the end. However, since Rop 0FH has a NOT as its last operation, the two NOTs cancel and F0H becomes a simple Pattern copy.

lpPBrush

If the *Rop3* code indicates that a pattern is involved in the block transfer, this points to a PBRUSH structure that was realized in **RealizeObject**.

If there is a brush operand, it can be put onto the screen by simply drawing a solid or patterned rectangle bounded by *DestxOrg*, *DestyOrg*, *xExt*, and *yExt*.

If a hollow brush is passed to **BitBlt**, the brush portion of the operation should *not* be done. However, all the other operations (such as Source and NOT operations) must still be done.

lpDrawMode

The *lpDrawMode* parameter is used only for mono-to-color and color-to-mono conversions. If your device is color, then Windows can ask you to convert a monochome (1 plane, 1 bit-per-pixel) bitmap into a bitmap matching your board's and driver's color format. The colors contained in the *lpDrawMode* parameter (background color at byte offset 4 and foreground color at byte offset 8 in the structure) allow you to do this conversion. The following holds true:

1. Mono-to-color conversion:
   - Bits that are 1 in the monochrome bitmap become background color.
   - Bits that are 0 in the monochrome bitmap become foreground color.
2. Color-to-mono conversion:
   - Color bytes that match the background color become background color, which is 1 (for white).
   - Anything that does not match the background color is foreground color, which is 0 (for black).

NOTE Every color device must support the transferring of monochrome (Black/White) bitmaps to the screen as well as color bitmaps.

Even though the *lpDrawMode* parameter has other fields such as **OpaqueFlag** (at byte offset 2) and **Rop2** (at byte offset 0), these are totally disregarded by **BitBlt**. All block transfers are opaque no matter what the *lpDrawMode* parameter says, and *Rop3* is used instead of **Rop2**.

2.6 The StrBlt/ExtTextOut Functions

StringBlt (**StrBlt**) is the old Windows 1.x name for **ExtTextOut**. The two names are now often used synonymously since **StrBlt** is used as an entry point to the **Extended Text Out** (**ExtTextOut**) function. It is documented in more detail in Chapter 10, "Common Functions."

The function is still needed for compatibility with the earlier version. However, only a few old applications still call **StrBlt**, while most new ones do not. They are simply mapped by GDI to **ExtTextOut**. You will need to put in only one piece of code (shown in Chapter 10, "Common Functions") and then jump into **ExtTextOut**.

The following are the call parameters for the **StrBlt** function. They are the same as the first nine parameters of the **ExtTextOut** function.

```
cProc   StrBlt,<FAR,PUBLIC>, <si, di>
        parmD   lpDestDev
        parmW   DestxOrg
        parmW   DestyOrg
        parmD   lpClipRect
        parmD   lpString
        parmW   Count
        parmD   lpFontInfo
        parmD   lpDrawMode
        parmD   lpTextXForm
```

2.6.1 The ExtTextOut Function

ExtTextOut is one of the text drawing functions. It works in conjunction with **BitBlt** to do the drawing on the screen. To see your driver running on the screen, all you need to implement are these two functions.

This function replaces the **StrBlt** function for Windows version 2.0 and later. The following are the call parameters for the **ExtTextOut** function:

```
cProc   ExtTextOut,<FAR,PUBLIC,WIN,PASCAL>,<si, di>
        parmD   lpDestDev
        parmW   DestxOrg
        parmW   DestyOrg
        parmD   lpClipRect
        parmD   lpString
        parmW   Count
        parmD   lpFontInfo
        parmD   lpDrawMode
        parmD   lpTextXForm
        parmD   lpCharWidths
        parmD   lpOpaqueRect
        parmW   Options
```

2.6.2 The ExtTextOut Parameters

This section provides supplemental information on how to use each of the passed parameters for **ExtTextOut**. For more details, refer to the description for this function given in Chapter 10, "Common Functions."

lpDestDev

The *lpDestDev* parameter tells you what you are drawing onto. For displays, this is the display device or bitmap.

DestxOrg and DestyOrg

The *DestxOrg* and *DestyOrg* parameters give the origins of the top-left corner of the string.

lpClipRect

The *lpClipRect* parameter is a long pointer to the clipping rectangle.

Clipping is the hardest part of **ExtTextOut**. If you did not have to clip, then the **ExtTextOut** function would be small and easy to implement. You would simply take each character's bitmap from the Font bitmap, transfer it to the screen, and then expand it for the number of bits per pixel and planes you have.

However, you must clip, and there are some rules to remember. One of them is that for all clip rectangle or rectangle (RECT) data structures or scanlines, the ending coordinates are always one greater than the actual pixel number at which you are to stop drawing. Therefore, if the clip rectangle has the coordinates 0, 0, 10, 10 (where the 0s are the starting X and Y, and the 10s are the ending X and Y), you only draw pixels 0 to 9 and do not draw through the 10th pixel.

The RECT data structure contains the following two points:

```
typedef struct {
        short left, top;
        short right, bottom;
        }RECT;
```

Where:

Left, top are the coordinates that specify the upper-left corner of the rectangle.

Right, bottom are the coordinates that specify the lower-right corner of the rectangle.

lpString

The *lpString* parameter is a long pointer to the string itself. Each character in the string is a byte length.

Count

Count can have one of three meanings.

First, if *Count* is positive (i.e., greater than zero), then it is the number of characters to display from the string.

Second, if *Count* is negative (i.e., less than zero), then you just return the length of the string as if there were no clipping rectangle. You run the string blt with no characters and return the height in **DX** and the X-length in **AX**. Then run the text justification and character spacing algorithm described in the DRAWMODE data structure.

Third, if *Count* is zero, then check the *Options* flag. If the 2s bit is set in the *Options* flag, then it infers that you have to draw an opaque rectangle.

If *lpOpaqueRect* is zero (even if the *Options* flag is 2), do nothing and return success.

If *lpOpaqueRect* is not zero, then you do the following:

1. Get the opaquing rectangle (described by *lpOpaqueRect*).
2. Intersect it with the clipping rectangle.
3. Intersect the rectangle with the bitmap.
4. Check to make sure that the rectangle is valid.
5. Draw the opaquing rectangle.

Logically, this is rather difficult due to the lack of sufficient variables. Therefore, you need to do lots of checks to make sure that the clipping and opaquing rectangles are valid.

In summary, if *Count* is zero, then it can infer one of two things:

> There is nothing to do, so you can get out. Or, you should only draw the opaquing rectangle pointed to by *lpOpaqueRect*.

lpFontInfo

This is a long pointer to the FONTINFO data structure and represents the physical font in use. (See the FONTINFO data structure fields in Chapter 12, "Data Structures and File Formats.") Notice that you can be presented with characters that are not in the character set or do not fall within the range of **dfFirstChar** to **dfLastChar**. In such a case, you should use the **dfDefaultChar** field.

lpDrawMode

This is a long pointer to the DRAWMODE data structure that includes the current text color, background mode, background color, text justification, and character spacing. (See the DRAWMODE data structure in Chapter 12, "Data Structures and File Formats.")

If the background mode (or **Transparent/Opaque** flag) is 1, then you draw the string transparently. If it is 2, then you draw the string opaquely.

If the string is to be drawn opaquely (e.g., if the foreground color is red and the background color is green), first draw the green and, then, draw the red character on top of it. If it is to be drawn transparently, just draw the red foreground and forget the background. However, you must get the *total* length of the string first.

If there is a break character, use the text justification and character spacing algorithm described below. As you are running the string (either making the count or putting the string on the page), you should get the **TBreakExtra** flag from the DRAWMODE data structure. If it is zero, then it is not a justified string and you can disregard the information given here. However, you must always add the **CharacterExtra** in the DRAWMODE data structure to the width of the character.

If no justification is required, **TBreakExtra** will be set to zero. To enable justification, an application must set **TBreakExtra** and **BreakCount** to the desired values. The other justification fields are evaluated using these values and **BreakErr** is set to **BreakCount/2+1**.

It is expected that **StrBlt** will be implemented as described below, but any implementation that spreads the excess pixels across the character breaks satisfies the requirements of text justification.

```
width = width of char
if TBreakExtra <> 0 and char = BreakChar then
        width = width + BreakExtra
        BreakErr = BreakErr - BreakRem
        if BreakErr <= 0 then
                width = width + 1
                BreakErr = BreakErr + BreakCount
        endif
endif
width = width + CharacterExtra move over by width
```

lpTextXForm

The TEXTXFORM data structure is used by devices that have hardware font capability. (See Chapter 13, "The Font File Format," for a description of the TEXTXFORM data structure.) If the hardware can italicize, then *lpTextXForm* tells you that this is an italic font and you look up your font in there. However, we are not currently aware of any display devices with smart font capabilities that use this structure.

lpCharWidths

If not NULL, then *lpCharWidths* is a long pointer to an array of words. Each word specifies the width from the start of the current character origin to the next character origin.

For example, assume the first character is an A, the next one is a B, and the *normal* width of A is 4 pixels. However, if *lpCharWidths* is 5, then you would move the B over by one pixel. If *lpCharWidths* is only 3, then the B would be almost on top of the A. Notice, though, that you cannot have a negative character width. It would always be changed to zero since you cannot go backwards.

lpOpaqueRect

The *lpOpaqueRect* parameter, if not NULL, is a long pointer to the opaquing rectangle. See the earlier discussion on the *Options* flag, under the *Count* parameter, for further information on the actions required by this parameter.

Options

The *Options* parameter is an integer with bits set to indicate **ExtTextOut** options. See the earlier discussion under the *Count* parameter for additional information on this parameter.

Notice also that whenever the clipping or opaquing rectangles intersect, those rectangles also need to be intersected with the bitmap to make sure the boundaries of the clipping rectangle are within the confines of the bitmap. You should look at the height and width of the bitmap (which are in the BITMAP data structure at offsets 2 and 4) to make sure that the clipping rectangle passed is intersected with those. That way, you will get the smallest possible rectangle composed of the three items: the bitmap, the clipping rectangle, and (if specified) the opaquing rectangle.

2.7 Stub Functions

The following are two stub functions to which you need to set up calls. They are not currently supported in GDI and should always return a failure code. However, they may be supported in the future. Simply copy verbatim the code reproduced here to your driver and you will be finished with their support.

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

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

You must also add a termination function called WEP(*bSystemExit*), or Windows Exit Procedure, to accommodate the support of dynamic-link libraries (DLLs). This function indicates whether all of Windows is shutting down or just the single DLL. More detailed descriptions are provided in Chapter 10, "Common Functions," and in the SDK *Guide to Programming*.

2.8 The Move and Check Cursor Functions

The **MoveCursor** function moves the cursor to the given screen coordinates. However, that can be rather difficult if you do not have a hardware cursor and have to move your own cursor on the screen when it is saved in memory or cached on the board.

The following is a sample procedure showing how to move the cursor:

1. Clear the interrupt flags using the **EnterCrit** macro. Stopping them will stop the cursor from moving.
2. Obtain the X and Y coordinates for the position at which they want you to place the cursor. These are passed to you by USER.

3. Put them into a variable called *UndoneXandY*. Save this in case you are unable to draw the cursor because something else is happening.
4. Get the old X and Y coordinates.
5. Set a **DrawBusy** flag saying that you are busy drawing and the driver should not try to draw another cursor at this time.
6. Use the **LeaveCrit** macro to allow interrupts again.

If the **DrawBusy** flag is on, then you have the undone coordinates saved in *UndoneXandY*. When you are able to draw again, you can then go ahead and do it. Otherwise, you disable the old cursor and put the new cursor on.

A more complete description of this function is available in Chapter 10, "Common Functions."

2.8.1 Excluding Cursors

Sometimes you may want to exclude or get rid of the cursor from the screen before doing something like a **BitBlt, StrBlt**, or drawing function. Or you may not want to read the cursor back when reading from the screen because it is difficult and time consuming.

How do you tell where your cursor was or from what area to exclude it? Do a check to see where the X and Y coordinates are and exclude them if they lie within your exclude region.

- For **BitBlt**, just disable the cursor within the rectangle that you are transferring.
- For **Output** for Scanlines, exclude the whole scanline.
- For Polylines, exclude the clip rectangle.
- For Polygons, exclude the clip rectangle.
- For Ellipse, Circle, Rectangle, and other drawing functions, exclude it from the bounding rectangle.
- For **StrBlt**, exclude it from the bounding rectangle and/or the opaquing rectangle.

To make the cursor run smoothly for these excludes, just before you start dealing with the board hardware, do the following:

1. Set the **DrawBusy** flag, which disallows cursor movement. **MoveCursor**, however, is still registering these movements.
2. Check out your exclude region.

   If the cursor falls within that exclude region, you exclude or turn off the cursor. That removes the cursor from the screen and restores whatever was there before.

3. Do your draw.
4. After you are finished with the draw, call **UnexcludeCursor**.
5. See if the cursor was excluded.

   If it was not, just get out and you are finished.

   If it was, take *UndoneX* and *UndoneY*, which are the movements that were registered while you were in the process of drawing, and enable the cursor at those coordinates.

2.8.2 The CheckCursor Function

This function is called on every timer interrupt. It allows the cursor to be displayed if it is no longer excluded. A description of this function is available in Chapter 10, "Common Functions."

However, you can also call the **UnexcludeCursor** routine described above since it does the same thing.

2.9 The Control Function

The **Control** function is required for display drivers. However, they need to support only the following two escapes:

- QUERYESCSUPPORT
- GETCOLORTABLE

Windows 2.x drivers also needed to support SETCOLORTABLE. However, for Windows 3.0, that escape is no longer required due to color palette management considerations. See Chapter 3, "Display Drivers: New Features," for more information on color palette management.

You will use QUERYESCSUPPORT to tell a calling application that you support a subset or none of the **Control** subfunctions. For more detailed information on escapes in general and complete descriptions of these escapes, see Chapter 11, "Device Driver Escapes."

2.10 Additional Functions

Display drivers also need to include the following additional functions:

(Gunter please add missing values.)

Function	Ordinal value
EnumDFonts	@ 6

Function	Ordinal value
Pixel	@ 9
ScanLR	@ 12
DeviceMode	@ 13
Inquire	@
FastBorder	@
EnumObj	@
GetCharWidth	@ 15
StretchBlt (optional)	@ 27
SetCursor	@ 102

To determine when your driver needs these and for more detailed information on each of them, refer to their descriptions in Chapter 10, "Common Functions."

2.11 How to Build Display Driver Resources

Display drivers contain most of the cursors, icons, and bitmaps that are used by Windows. They are supplied by the display driver to take advantage of all the capabilities of the driver (e.g., color icons). Also, the definitions of certain system parameters (e.g., default colors and border widths) are supplied by the display driver. All of this information is supplied as resources added to the driver by the resource compiler (RC.EXE).

Windows 3.0 has changed a number of these resources, most notably the bitmaps required to implement the 3-D effect. We recommend that you use the set of bitmaps supplied in the DDK that best matches the resolution/capabilities of your display. Since many existing Windows applications expect the old images to still exist, the driver must supply these as well. See the DDK's *Installation and Update Notes* for a detailed list of the required resources and for more information.

The resource file (.RES) is built from the following pieces:

1. A FONTS.ASM file that contains information about font stock objects.
2. A CONFIG.ASM file that contains information about default system colors, line widths, cursor/icon sizes, etc.
3. A COLORTAB.ASM file that contains the color table for Control Panel's Color Tuner dialog.
4. A set of icons, cursors, and bitmaps.
5. A .RC file that is used by the resource compiler to build the binary resource file (.RES).

2.11.1 Creating the FONTS.ASM File

The FONTS.ASM file tells Windows the characteristics of certain fonts that Windows uses as stock objects.

This file also defines these same characteristics for the two fonts minimally required to run such programs as Windows Write. These are the ANSI fixed-pitch (Courier) and variable-pitch (Helvetica ®) fonts. Normally, you will not have to create these fonts; you may use the ones supplied with Windows at the aspect ratio closest to that of your display or other device.

The FONTS.ASM file consists of three data structures that describe these same characteristics plus one for the Terminal font. Each data structure is of type LOGFONT (see Chapter 12, "Data Structures and File Formats," for more information on that structure). The order of the three LOGFONT structures in the FONTS.ASM file *must* be as follows:

1. OEM font (of facename "Terminal")
2. ANSI fixed-pitch font (usually Courier)
3. ANSI variable-pitch font (usually Helvetica)

2.11.2 Creating the CONFIG.ASM File

The CONFIG.ASM file tells Windows about many of the default characteristics of the screen, such as:

- Colors
- Line widths, both horizontal and vertical
- Scroll bar "thumb" sizes
- Cursor and icon compression ratios

The following is a prototype CONFIG.ASM file:

```
OEM segment public

;Machine dependent parameters

        dw      ?       ;Height of vertical thumb (in pixels)
        dw      ?       ;Width of horizontal thumb (in pixels)
        dw      ?       ;Icon horiz compression factor (can be 1 or 2)
        dw      ?       ;Icon vert compression factor (can be 1 or 2)
        dw      ?       ;Cursor horz compression factor (can be 1 or 2)
        dw      ?       ;Cursor vert compression factor (can be 1 or 2)
        dw      ?       ;Reserved
        dw      ?       ;cxBorder (thickness of vert lines) (usually 1 pixel)
        dw      ?       ;cyBorder (thickness of horiz lines) (usually 1 pixel)
```

```
RGB     macro   R, G, B
        db      R,G,B,0
        endm

;Default system color values

        RGB     130,130,130     ;clrScrollbar
        RGB     192,192,192     ;clrDesktop
        RGB     000,064,128     ;clrActiveCaption
        RGB     255,255,255     ;clrInactiveCaption
        RGB     255,255,255     ;clrMenu
        RGB     255,255,255     ;clrWindow
        RGB     000,000,000     ;clrWindowFrame
        RGB     000,000,000     ;clrMenuText
        RGB     000,000,000     ;clrWindowText
        RGB     255,255,255     ;clrCaptionText
        RGB     128,128,128     ;clrActiveBorder
        RGB     255,255,255     ;clrInactiveBorder
        RGB     255,255,255     ;clrAppWorkspace
        RGB     000,000,000     ;clrHiliteBk
        RGB     255,255,255     ;clrHiliteText
        RGB     192,192,192     ;clrBtnFace
        RGB     128,128,128     ;clrBtnShadow
        RGB     192,192,192     ;clrGrayText
        RBG     000,000,000     ;clrBtnText

OEM ends
```

NOTE The values shown in the example above are the default colors shipped with the VGA color display. These are the recommended values.

The following are detailed descriptions of each field:

Field	Description
cnVertThumHeight	A 2-byte value specifying the height in pixels of the vertical scroll bar thumb.
cnHorizThumWidth	A 2-byte value specifying the width in pixels of the horizontal scroll bar thumb.
cnIconXRatio	A 2-byte value specifying the ratio by which the icon width is to be reduced before displaying.
cnIconYRatio	A 2-byte value specifying the ratio by which the icon height is to be reduced before displaying.
cnCurXRatio	A 2-byte value specifying the ratio by which the cursor width is to be reduced before displaying.

Field	Description
cnCurYRatio	A 2-byte value specifying the ratio by which the cursor height is to be reduced before displaying.
Reserved	A 2-byte reserved field that should be set to zero.
cnXBorder	A 2-byte value specifying the thickness in pixels of vertical lines.
cnYBorder	A 2-byte value specifying the thickness in pixels of horizontal lines.
cnScrollBarColor	A 4-byte RGB value specifying the default color of the scroll bar.
cnDesktopColor	A 4-byte RGB value specifying the default color of the Windows background.
cnActiveCapColor	A 4-byte RGB value specifying the default color of the caption in the active window.
cnInactiveCapColor	A 4-byte RGB value specifying the default color of the caption in an inactive window.
cnMenuBackgndColor	A 4-byte RGB value specifying the default color of the menu background.
cnWindowBackgndColor	A 4-byte RGB value specifying the default color of a window's background.
cnCaptionColor	A 4-byte RGB value specifying the default color of the caption.
cnMenuTextColor	A 4-byte RGB value specifying the default color of the text in a menu.
cnWindowTextColor	A 4-byte RGB value specifying the default color of the text in a window.
cnCaptionTextColor	A 4-byte RGB value specifying the default color of the text in a caption.
cnActiveBorderTextColor	A 4-byte RGB value specifying the default color of the text in an active border.
cnInactiveBorderTextColor	A 4-byte RGB value specifying the default color of the text in an inactive border.
cnWorkSpaceTextColor	A 4-byte RGB value specifying the default color of the application workspace (MDI background).

Field	Description
cnHilightBk	An RGB value specifying the highlight color used in menus, edit controls, listboxes, etc.
cnHilightText	An RGB value specifying the text color for highlighted text.
cnBtnFace	An RGB value specifying the color of the 3-D button face shading.
cnBtnShadow	An RGB value specifying the color of the 3-D button edge shadow.
cnGrayText	An RGB value specifying the color of Solid Gray to be used for drawing disabled items. (Must be zeros if no solid gray is available.)
cnBtnText	An RGB value specifying the text color in Windows 3.0 pushbuttons.

2.11.3 Creating the COLORTAB.ASM File

The COLORTAB.ASM file contains a list of the colors that are to appear in the Control Panel's Color Tuner dialog. This table should contain all the solid colors that are representable as RGB values as well as all the good looking dithers. The table may contain up to 48 RGB values.

The following is a prototype of a COLORTAB.ASM file that contains the suggested RGB values for 4-plane EGA or VGA drivers.

```
RGB     macro  R, G, B
        db     B,G,R,0
        endm

COLORTABLE  segment public

        dw      48              ; # colors in table
        RGB     80h,80h,0FFh
        RGB     80h,0FFh,0FFh
        RGB     80h,0FFh,80h
        RGB     80h,0FFh,00h
        RGB     0FFh,0FFh,80h
        RGB     0FFh,80h,00h
        RGB     0C0h,80h,0FFh
        RGB     0FFh,80h,0FFh
        RGB     00h,00h,0FFh
        RGB     00h,0FFh,0FFh
        RGB     00h,0FFh,80h
        RGB     40h,0FFh,00h
        RGB     0FFh,0FFh,00h
```

```
                RGB      0C0h,80h,00h
                RGB      0C0h,80h,80h
                RGB      0FFh,00h,0FFh
                RGB      40h,40h,80h
                RGB      40h,80h,0FFh
                RGB      00h,0FFh,00h
                RGB      80h,80h,00h
                RGB      80h,40h,00h
                RGB      0FFh,80h,80h
                RGB      40h,00h,80h
                RGB      80h,00h,0FFh
                RGB      00h,00h,80h
                RGB      00h,80h,0FFh
                RGB      00h,80h,00h
                RGB      40h,80h,00h
                RGB      0FFh,00h,00h
                RGB      0A0h,00h,00h
                RGB      80h,00h,80h
                RGB      0FFh,00h,80h
                RGB      00h,00h,40h
                RGB      00h,40h,80h
                RGB      00h,40h,00h
                RGB      40h,40h,00h
                RGB      80h,00h,00h
                RGB      40h,00h,00h
                RGB      40h,00h,40h
                RGB      80h,00h,40h
                RGB      00h,00h,00h
                RGB      00h,80h,80h
                RGB      40h,80h,80h
                RGB      80h,80h,80h
                RGB      80h,80h,40h
                RGB      0C0h,0C0h,0C0h
                RGB      40h,00h,40h
                RGB      0FFh,0FFh,0FFh

COLORTABLE      ends
```

2.11.4 Creating Icons, Cursors, and Bitmaps

ICBs acceptable for use by many display resolutions and aspect ratios are provided in various subdirectories of the resource file directories on the disks provided with the DDK. See the DDK's *Installation and Update Guide* for detailed lists of the subdirectories.

If you want to create your own ICBs, you can do so by using the SDK Paint application. In creating ICBs, you should meet the criteria given in the following table on Cursor, Icon, and Bitmap files.

NOTE The maximum allowable cursor and icon sizes are 64x64 pixels.

Resource Name	Type	Filename	Purpose
<u>CURSORS</u>			
OCR_NORMAL	cursor	NORMAL.CUR	An upward diagonal arrow used as the default mouse cursor.
OCR_IBEAM	cursor	IBEAM.CUR	An I-beam shaped cursor used in edit control windows.
OCR_WAIT	cursor	WAIT.CUR	An hourglass that is used while carrying out lengthy operations.
OCR_SIZENWSE	cursor	SIZENWSE.CUR	A two-headed arrow used when sizing windows. Arrows point NW and SE.
OCR_SIZENESW	cursor	SIZENESW.CUR	A two-headed arrow used when sizing windows. Arrows point NE and SW.
OCR_SIZEWE	cursor	SIZEWE.CUR	A two-headed arrow used when sizing windows. Arrows point W and E.
OCR_SIZENS	cursor	SIZENS.CUR	A two-headed arrow used when sizing windows. Arrows point N and S.

NOTE The following cursors are no longer used by Windows, but must be provided for compatibility with existing Windows applications that may expect them to be available.

OCR_CROSS	cursor	CROSS.CUR	An upright cross used as a selection marker.
OCR_UP	cursor	UP.CUR	An upward arrow.
OCR_SIZE	cursor	SIZE.CUR	A box shape formerly used when sizing tiled windows.
OCR_ICON	cursor	ICON.CUR	An empty box formerly used when the mouse was in the icon area.
<u>ICONS</u>			
OIC_HAND	icon	HAND.ICO	A stop sign used to indicate an error condition that halts operation.

Resource Name	Type	Filename	Purpose
OIC_QUES	icon	QUES.ICO	A question mark used when querying for a reply.
OIC_BANG	icon	BANG.ICO	An exclamation mark used to emphasize the consequences of an operation.
OIC_NOTE	icon	NOTE.ICO	An asterisk used to indicate non-critical situations.
OIC_SAMPLE	icon	SAMPLE.ICO	The default icon used when no other icon to an operation can be found.

BITMAPS

The following 7 shapes have two forms: the normal image and the depressed image. These are used to create the 3-D effect of pushing in a button.

Resource Name	Type	Filename	Purpose
OBM_RESTORE OBM_RESTORED	bitmap	RESTORE.BMP RESTORED.BMP	Images used as the restore button on the title bar.
OBM_REDUCE OBM_REDUCED	bitmap	MIN.BMP MIND.BMP	Images used as the minimize button on the title bar.
OBM_ZOOM OBM_ZOOMD	bitmap	MAX.BMP MAXD.BMP	Images used as the maximize button on the title bar.
OBM_RGARROW OBM_RGARROWD	bitmap	RIGHT.BMP RIGHTD.BMP	A right-pointing arrow for scroll bars.
OBM_LFARROW OBM_LFARROWD	bitmap	LEFT.BMP LEFTD.BMP	A left-pointing arrow for scroll bars.
OBM_UPARROW OBM_UPARROWD	bitmap	UP.BMP UPD.BMP	An up-pointing arrow for scroll bars.
OBM_DNARROW OBM_DNARROWD	bitmap	DOWN.BMP DOWND.BMP	A down-pointing arrow for scroll bars.
OBM_CLOSE	bitmap	SYSMENU.BMP	A double-wide image that contains system menu shapes for both main windows and MDI windows.
OBM_CHECK	bitmap	OCHECK.BMP	A check mark used to check menu items.
OBM_CHECKBOXES	bitmap	OBUTTON.BMP	A box used for check boxes in dialogs.

Resource Name	Type	Filename	Purpose
OBM_COMBO	bitmap	COMBO.BMP	An arrow used in combo boxes.
OBM_MNARROW	bitmap	MNARROW.BMP	An arrow used in multi-level menus.

NOTE the following bitmaps are no longer used by Windows, but must be supplied for compatibility with applications that expect them to be available.

OBM_BTNCORNERS	bitmap	OBTNCORN.BMP	A circle formerly used to draw round-cornered pushbuttons.
OBM_SIZE	bitmap	OSIZE.BMP	A size box formerly used on tiled windows.
OBM_BTSIZE	bitmap	OBTSIZE.BMP	A size box used at the intersection of vertical and horizontal scroll bars.
OBM_OLD_RESTORE	bitmap	OREST.BMP	Restores the bitmap used for Windows 2.x.
OBM_OLD_REDUCE	bitmap	ORED.BMP	Minimizes the bitmap used for Windows 2.x.
OBM_OLD_ZOOM	bitmap	OZOOM.BMP	Maximizes the bitmap used for Windows 2.x.
OBM_OLD_RGARROW	bitmap	ORIGHT.BMP	A right-arrow bitmap used for Windows 2.x.
OBM_OLD_LFARROW	bitmap	OLEFT.BMP	A left-arrow bitmap used for Windows 2.x.
OBM_OLD_UPARROW	bitmap	OUP.BMP	An up-arrow bitmap used for Windows 2.x.
OBM_OLD_DNARROW	bitmap	ODOWN.BMP	A down-arrow bitmap used for Windows 2.x.
OBM_OLD_CLOSE	bitmap	OCLOSE.BMP	The system-menu bit-maps used for Windows 2.x.

2.11.5 Assembling and Linking FONTS.ASM, CONFIG.ASM, and COLORTAB.ASM

To create FONTS.BIN, CONFIG.BIN, and COLORTAB.BIN, follow this procedure:

```
masm fonts;
link fonts;
exe2bin fonts;
```

(Substituting “config” and “colortab” for “fonts” as appropriate.)

2.11.6 Using RC to Create the .RES File

Once you have completed all the preceding steps, you must create a script for the resource compiler/editor (RC). We recommend that you use a .RC file from one of the resource file directories.

Issue the following command to compile your resources:

```
rc -r filename.rc
```

where “filename” is the name of your RC script. The output from this operation will be your completed .RES file.

2.12 Display Drivers Checklist

The following checklist is a summary of the major points made in this chapter. An additional checklist for updating 2.x display drivers to the 3.0 requirements is provided in Chapter 3, “Display Drivers: New Features.”

- ☐ To display output to the screen, you must first implement at least the following functions:
  - ☐ **Output**
  - ☐ **Enable and Disable**
  - ☐ **RealizeObject**
  - ☐ **ColorInfo**
  - ☐ **BitBlt**
  - ☐ **StrBlt/ExtTextOut**
  - ☐ **Control**
  - ☐ **SetDIBits and GetDIBits**
- ☐ In the GDIINFO data structure, you must support the following capabilities:

Value	Offset #	Contents	Bit Value	Bit #
dpLines	30	Polylines	LC_POLYLINE	1
dpPolygonals	32	Scanlines	PC_SCANLINE	3

Value	Offset #	Contents	Bit Value	Bit #
dpRaster	38	DIBs	RC_DI_BITMAP	7
			RC_DIBTODEV	9
			RC_BITBLT	0
			RC_GDI20_OUTPUT	4

- ❑ Display drivers must support at least the following escapes in the **Control** function:
  - ☐ QUERYESCSUPPORT
  - ☐ GETCOLORTABLE
  - ☐ SETCOLORTABLE (do *not* use with palette-capable devices)
- ❑ To build a resource file (.RES), you need the following items:
  - ☐ A FONTS.ASM file
  - ☐ A CONFIG.ASM file
  - ☐ A COLORTAB.ASM file
  - ☐ A set of icons, cursors, and bitmaps
  - ☐ A .RC file

Chapter 3 Display Drivers: New Features

This chapter provides information on the new Windows 3.0 features that affect display drivers and how to work with them. These include the following:

- Color Palette Management
- Protected Mode Support
- > 64K Fonts
- Device Independent Bitmaps

3.1 Color Palette Management

Display devices that are capable of displaying at least (and possibly more than) 256 simultaneous colors out of a palette may need to provide a color palette management interface. (See the *Microsoft Windows Software Development Kit* for a complete description of color palette management.)

Color palette management requires the following functionality from the driver:

- An interface to get (read) and to set (write) the hardware palette
- An interface to get/set the driver-maintained color translate table

Depending on the capabilities of your hardware, there will be some additional requirements as explained in the following subsections.

3.1.1 The Hardware Palette Calls

For the get/set hardware palette calls, the following parameters are passed to the driver in this order:

```
cProc   SetPalette, <FAR, PUBLIC, WIN, PASCAL>
                parmW   nStartIndex
                parmW   nNumEntries
                parmD   lpPalette
```

The parameters passed have the following meaning:

Parameter	Description
nStartIndex	The zero-based (color) index into the palette Look Up Table (LUT) specifying where to put the first RGB triplet. Subsequent RGB triplets are placed in subsequent palette LUT entries (in increasing order).
nNumEntries	The total number of entries to set in the device's hardware palette.
lpPalette	A far pointer to the RGB colors to be set into the palette. They are stored as Red, Green, Blue, and Flags, occupying one doubleword. The flags have no meaning for the driver and should be ignored.

For a **GetPalette** call, the same parameters are passed but the driver fills the RGB array pointed to by *lpPalette*. A zero should be written in the *Flags* field.

3.1.2 The Color Translate Table

The driver has to maintain a color translate table to translate the logical color indices, passed to it by GDI, into the actual physical color indices. The color translation has to occur before any raster operation (ROP) is performed (i.e., ROPs are always applied to physical colors).

The following data structures contain logical colors that may need to be translated to physical colors before they can be used:

```
lpDrawMode     ;translate foreground (text) and background colors
lpPen          ;translate pen color
lpPBrush       ;(structure is device specific) translate all the
               ;color indices
memory bitmaps ;translate all the bitmap bits
```

Notice that the application has to perform color translation *only* when the physical device is involved. In other words, if a line is drawn into a memory bitmap or a bitmap is transferred (blt'ed) into another memory device, no color translation is required. On the other hand, if a bitmap is blt'ed to or from the screen into a memory bitmap or a line is drawn directly onto the screen, color translation is required. In the case of a block transfer from the screen to the screen (where the physical device is both the source and destination of the block transfer), color translation is not needed since all the color indices are already translated into physical indices.

Color specifications are passed to the palette-managing display drivers in two forms:

- 0xFF00*iiii*, where *iiii* is the index to use
- 0x00RRGGBB, which gives the explicit RGB color to use. Match this color as closely as possible among the 20 reserved colors. In the case of a brush, the color may be dithered with the 16 VGA colors.

Since the reserved colors will always have fixed indices, their logical and physical indices will be identical.

Color translation hooks have to be placed in the following functions:

- **Output** (translates pen, brush, and draw mode)
- **Pixel** (translates pen)
- **ExtTextOut** and **StrBlt** (translate draw mode)
- **BitBlt** (translates brush and draw mode)

3.1.3 The Palette Translate Table

For the get/set palette translate table calls, a far pointer to an array of indices (i.e., WORDS) of a size specified in **dpPalColors** is passed to the driver for the logical-to-physical color index mapping (for more information on **dpPalColors**, see Section 3.1.6, "Changes to GDIINFO"). In the case of **GetPalTrans**, the driver copies its translate table into the array that GDI passed to it.

The functionality for the **SetPalTrans** call is a little more complicated. If the pointer to the color table is not NULL, the driver has to copy the translate table into its own data segment and also has to construct an inverse of the table it was passed. The inverse table is needed for block transfers from the screen to a memory bitmap.

In constructing the inverse table, the driver may come across ambiguities because different logical colors can map to the same physical color. It is up to the driver to decide how to resolve these cases since the net result will look the same no matter how such ambiguities have been resolved.

If the pointer to the color table is NULL, the driver needs to construct identity translate tables. It can also set accelerator bits to bypass the various translations outlined above. For **BitBlt**, bypassing color translation results in substantial performance improvements.

3.1.4 DIBs with Color Palette Management

The color table for a device-independent bitmap (DIB) consists of WORD indices to be used as the "colors" for the bitmap. For **SetDIBitsToDevice**, they are physical indices; for **SetDIBits**, they are logical indices.

(RonG, please review this carefully to be sure I haven't misinterpreted you! Thanks.)

In the **GetDIBits**, **SetDIBits**, and **SetDIBitsToDevice** functions, the final parameter, *lpConversionInfo*, provides information that is useful only for palette-capable devices. However, all devices need to include this parameter.

During a Get operation, it provides a long pointer to a translate table with the following results:

Bitcount	Result
1	A palette-sized array of bytes, each one either 00H or FFH during the Get. The array is used to determine if the index in the bitmap corresponds to a zero or a 1 in the DIB.
4	A palette-sized array of bytes, each one containing a value between 00H and FFH. Each index in the bitmap will map to the corresponding 4-bit values in the DIB.
8	*lpConversionInfo* will equal an identity table that can be ignored.
24	A palette-sized array of DWORD values, each one containing the RGB value (and unused high byte) corresponding to the index in the bitmap.

During a Set operation, this long pointer will contain something meaningful only when setting from a bitmap with a bitcount of 24. It will then point to some data maintained by GDI. For every RGB value in the DIB, the device will call a function in GDI as follows:

DeviceColorMatch(*RGBvalue, lpConversionInfo*)

where *RGBvalue* is a DWORD containing the RGB value to be color matched (with the high byte ignored).

This GDI function will then return an index to use to represent that color in the device-dependent bitmap.

3.1.5 The UpdateColors Function

An **UpdateColors** function/entry point needs to be added to the driver for all palette-capable devices. **UpdateColors** is called to redraw a region directly in place and on screen using the translate table passed in this call. That means that it performs read pixel (color index), translates a pixel's color index, and writes the translated index back in place for all the pixels in the region specified in the call.

The following parameters are passed:

```
cProc   UpdateColors, <FAR, PUBLIC>
        parmW   wStartX        ;starting column (in screen pixels)
        parmW   wStartY        ;starting row (in screen pixels)
        parmW   wExtX          ;X extent of region (in screen pixels)
        parmW   wExtY          ;Y extent of region (in screen pixels)
        parmD   lpTranslate    ;array of words, log->phys translate
```

The origin is assumed to be in the upper-left corner of the screen.

3.1.6 Changes to GDIINFO

To support palette devices, three WORD entries have been appended to the GDIINFO structure, which is defined in GDIDEFS.INC:

```
dpPalColors,      dw  ?, ; total number of simultaneous colors
dpPalReserved,    dw  ?, ; # of reserved colors
dpPalResolut,     dw  ?, ; palette resolution (in bits) of video DAC
```

The number of reserved colors on the palette is always 20, with 16 corresponding to the VGA colors and 4 special colors. Half of the reserved palette colors are placed at the beginning and half at the end of the palette. (See the sample palette-device code provided with the DDK for the exact colors to use.)

The driver also has to set the RC_PALETTE bit for the **dpRaster** entry in GDIINFO (RC_PALETTE equ 0000000100000000b). When set, this bit means that the driver can do palette management.

The driver's version number (**dpVersion**) must be updated to 0300H as well.

3.1.7 Ordinal Reference Numbers

The following is a list of the ordinal reference numbers of the entry points required for color palette management support:

SetPalette	@22
GetPalette	@23
SetPalTrans	@24
GetPalTrans	@25
UpdateColors	@26

These lines go into the display driver's definition file under EXPORTS.

3.2 Protected-Mode Support

Windows 3.0 drivers have to be bimodal, i.e., they have to run in protected mode as well as 8086 real address mode. (See the *Microsoft Windows Software Development Kit* for more information on Windows memory management.) To enable the device driver writer to write into the code segment (normally not allowed in protected mode) and to perform segment (actually selector) arithmetic to advance in 64K blocks, some help from KERNEL is needed.

Some of these functions (with their ordinal reference numbers given in parentheses) may need to be imported from KERNEL:

Function (ordinal value)	Description
AllocSelector() (@175)	Allocates and returns an uninitialized selector for the driver's use in AX.
FreeSelector(*wSel*) (@176)	Frees a selector allocated by **AllocSelector**.
PrestoChangoSelector(*wSrcSel*, *wDestSel*)(@177)	Makes the destination selector the same as the source selector, except that: DATA—>CODE or CODE—>DATA.
AllocCSToDSAlias(*wSel*) (@170)	Creates a data selector alias for the code selector passed (returned in AX).
AllocDSToCSAlias(*wSel*) (@171)	Creates a code selector alias for the data selector passed (returned in AX).
__AHIncr (@114)	Performs selector/segment arithmetic. This is an absolute value that has a value of 1000H in real address mode and a selector to next selector increment in protected mode.
LongPtrAdd (@180)	Performs segment arithmetic on a long pointer and a DWORD offset.

The following actions cause a general protection (GP) fault if you do them while in protected mode:

- Accessing (reading or writing) an array beyond its limit.
- Having an offset wrap-around (going from 0FFFFH to 0 using a string instruction).
- Loading an invalid selector into a segment register.
- Updating code segment variables.
- Doing segment arithmetic (except as described above) for selector registers.
- Comparing segment (selector) registers to see which is lower in memory.
- Doing CLIs (Clear Interrupts) and STIs (Set Interrupts).
- Using undocumented MS-DOS calls. Use Windows calls whenever possible.

It should be noted that loading a segment register or making a far call takes substantially longer in protected mode. Therefore, you should minimize changing selectors and the number of far calls, particularly in loops.

3.3 Greater Than (>) 64K Font Support

The font structure has been enhanced for drivers capable of supporting >64K fonts. The changes are as follows:

- The version number of the fonts in the **dfVersion** field in the font structure will now be 0300H.
- The **CharTable** array in the font structure now has 6 bytes per character. Each entry consists of a WORD followed by a DWORD. The first WORD is the width of the character. The following DWORD is the byte offset from the beginning of the FONT-INFO structure to the character's bitmap. With 32-bit offsets, the size of the font structure, including the bitmaps, is no longer limited to 64K.
- Additional fields have been added to the font header between **dfDBFiller** and **dfCharOffSet**.

The following are changes that need to be made in the display driver code:

- The driver must let GDI know that it can support > 64K fonts. To do this, the driver must set the RC_BIGFONT (0000010000000000b) bit of the raster capabilities WORD in the GDIINFO structure. Once this is done, GDI will ensure that all the fonts that the driver gets to handle are in the new format.
- To handle the 32-bit offset, the device driver code has to make use of the extended register set of the 80386. It will have to use **ESI**, **EDI**, **EAX**, **EBX**, **ECX**, and **EBP**. The existing code can be modified easily by using just the corresponding extended registers for the register used by drivers that work with 16-bit offsets.
- Due to the additional fields in the header (which, however, are not currently being used), the driver may need to recompute the offset to the character offset table.

3.4 Device-Independent Bitmaps

Color bitmaps have always been susceptible to problems involving device dependencies. To alleviate these problems, Windows 3.0 now supports device-independent bitmap (DIB) formats and new API calls to handle these maps.

The following API calls have been introduced to handle DIBs:

Call	Description
SetDIBits	Copies the information from a DIB into a device- dependent bitmap format.
GetDIBits	Does the inverse conversion, copying out the bits from a device-dependent bitmap into a DIB format.

Call	Description
SetDIBitsToDevice	Allows an application to block transfer (blt) any portion of a DIB directly onto the screen. However, in this case, a direct copy of the source is done with no other raster operations being supported.

The driver also has to support a dummy function, **CreateDIBitmap**, that should just return zero in **AX** indicating that the creation of a DIB is not supported at the driver level.

The following sections discuss the structure of each of these functions and their implementation at the device driver level. For a more detailed discussion of the BITMAPINFO and RGBQUAD data structures, see the *Microsoft Windows Software Development Kit.*

3.4.1 SetDIBits and GetDIBits

The entry point in the driver for these two functions is the same. GDI passes an extra flag to the driver indicating whether to execute the Set or the Get call. In the following examples, the "entry point function" for these two functions is called **DeviceBitmapBits.** This function does some validations and, then, calls either the **Set** or the **Get** subfunctions to do the actual task.

For a modular design, you should handle all the Run Length Encoded (RLE) DIBs in a separate function. This entry point function checks to see whether or not the DIB is in RLE format. If it is, then it calls **RLEBitBlt,** a function local to the driver, to do the bit transfer. We will discuss the structure of **RLEBitBlt** after describing the local functions that handle unencoded DIBs.

Since the **SetDIBits** and **GetDIBits** functions are not used as often as the other important functions such as **BitBlt** or **ExtTextOut,** you should group them in a separate discardable code segment.

The structure of this function is as follows:

```
createSeg       _DIMAPS,DIMapSeg,public,CODE
sBegin          DIMapSeg

externNPSetDeviceBitmapBits     ; if defined in a separate file
externNPGetDeviceBitmapBits     ; if defined in a separate file

cProc   DeviceBitmapBits,  <FAR,PUBLIC,WIN,PASCAL>,<si, di>

parmD   lp_dst_device       ; a long pointer to a device-dependent bitmap
                            ; device
parmW   set_or_get          ; 0 => set, 1 => get bits call
parmW   iStart              ; start scan number in the source map
parmW   num_scans           ; number of scans to copy
parmD   lp_DI_bits          ; long pointer to the actual bits in the DIB
parmD   lp_DIB_header       ; long pointer to the DIB header block
parmD   lp_DrawMode         ; long pointer to the GDI DRAWMODE structure,
                              ; only used to decode an RLE
parmD   lp_ConversionInfo  ; long pointer to Conversion Translate Table
```

[Any local variable definitions]

```
cBegin

;

;
```

[Do validations on the input parameters and return with **AX** set to 0 if the parameters are not valid.]

```
;

;
```

[Test to see if the DIB is in RLE format. For RLE DIBs, the **biStyle** field of the DIB header block will have the bit named **RLE_FORMAT_8** set. For RLE DIBs, do the following:

1. Prepare a clipping rectangle. For **SetDIBits**, the extents of the clipping rectangle are the width and height of the DIB. However, for **GetDIBits**, the extents are the width and height of the surface from which the bits are to be obtained.
2. Calculate the X and Y extents of the block to be transferred. These are identical to the X and Y extents of the clipping rectangle.

   The destination rectangle top-left corner is treated as being at (0,0).

   The parameters to the call are as follows:

   ```
   RLEBitBlt (lp_dst_dev,  /*bitmap descriptor */
           0,0,            /*distance to left corner*/
           xExt,yExt,      /*extents of blt*/
           iStart,         /*start scan number*/
           num_scans,      /*no. of scans to blt*/
           set_or_get,     /*direction of blt*/
           lp_clip_rect,   /*lptr to clip rect*/
           lp_DrawMode,    /*lptr to the structure*/
           lp_DI_Bits,     /*lptr to DIB buffer*/
           lp_DIB_header   /*lptr to header*/ )
   ```

3. Return from **DeviceBitmapBits** with the status code returned by **RLEBitBlt** in **AX**.]

```
;

;
```

[Prepare variables that might be needed by the **Set** or **Get** subfunctions.]

```
;

;
```

[If **Set or Get** is 0, call **SetDeviceBitmapBits** to copy the bits from the DIB format into the device-specific format. If **Set or Get** is 1, call **GetDeviceBitmapBits** to do the copy in the reverse direction.]

;

;

[Preserve in **AX** the status code returned by the **Set** or **Get** subfunctions.]

(The return value in **AX** is actually the number of scanlines copied.)

```
cEnd

cProc   CreateDIBitmap,<PUBLIC, FAR>

; the dummy procedure

cBegin
                xor     ax, ax
cEnd

sEnd    DIMapseg
end
```

Check the input parameters to make sure the following is true:

- The long pointers are all valid (NULL pointers can be detected here and errors reported).
- **Set or Get** should be either a 0 or a 1.
- The bits-per-pixel in the DIB format must be 1, 4, 8, or 24 and the number of planes must be 1.
- The width of the device-dependent bitmap should be the same as that of the DIB.

Next, decide whether or not the DIB is run length encoded. If it is, prepare parameters for **RLEBitBlt** and call it to complete the transfer.

For normal DIBs, once you know that the input parameters are valid, you are ready to do the actual job by calling the **Set** or **Get** subfunctions as appropriate. However, this is also a convenient time to compute any variables that have to be derived from the input parameters and that are used by both subfunctions. For example, the following information might be computed at this point:

- Flags specifying whether the device-dependent bitmap is in a color or monochrome format and whether or not it spans a 64K boundary. Notice that the DIB could be a color map with the device-dependent bitmap in a monochrome format. If that were the case, you would then have to do the appropriate color conversions.

- Other variables that would be of use during the actual conversion can also be calculated at this stage. These would include, for example, the width of a source and destination scan in bytes, and the starting byte offsets in the source and the destination maps.

SetDeviceBitmapBits

This is a NEAR function that is called from the **DeviceBitmapBits** function. It converts and copies the bits in the DIB format into the device-dependent format. Because most of the initial calculations have already been done in the calling function, the parameters to this function would be the derived variables and the pointers to the source and destination bitmap arrays. Also included here is the RGBQUAD data structure for the color table.

The structure of this function would be something similar to the following example:

[The function should be put in the _DIMAPS segment.]

```
anpfnPreProc    labelword       ; preprocessor jump table

        dw      eti_1           ; preprocessor for 1 bit/pixel DIB
        dw      eti_4           ; preprocessor for 4 bits/pixel DIB
        dw      eti_8           ; preprocessor for 8 bits/pixel DIB
        dw      eti_24          ; preprocessor for 24 bits/pixel DIB

; 'eti_' can be read as 'external_to_internal_'

cProc   SetDeviceBitmapBits, <NEAR, PUBLIC>

        parmD   lp_bits_start   ; long pointer to the start scan of the
                                ;   destination bitmap
        parmD   num_scans       ; number of scans to copy
        parmD   iStart          ; start scan number
        parmD   lp_DI_bits_start; long pointer to DIB bits start scan
        parmD   lp_DIB_header   ; long pointer to DIB header
        parmD   fbsd            ; flag byte passed by DeviceBitmapBits
```

[Other derived parameters such as width of scans, flags, etc.]

[Local and other temporary variables]

```
localW  full_byte_proc          ; address of full-byte conv routine
localW  partial_byte_proc       ; address of partial-byte conv
                                ;   routine
localV  color_translate,256     ; color translation table

cBegin
```

[Depending on the number of bits/pixel, do format-specific initializations. Decide which local routine to use to convert bits from the source that yield one byte for the destination, and which routine to use for the partial bytes at the end of the destination (if any).]

[Convert the source bitmap area one scan at a time. For every scan, use "call **full_byte_proc**" with the number of complete destination bytes that are to be generated and "call **partial_byte_proc**" with appropriate end-byte masks for the partial bytes near the ends of the scans in the destination map (if any). While updating the pointers to the next scan and copying the bytes in a scan, take care of cases when the scans may cross the 64K boundaries.]

[Set AX to 1, to indicate success.]

```
cEnd

; organize the local byte conversion routines here
;-------------------------------------------------------------------------;
e1_icolor_full          proc near

        ; code for external_1_bits/pixel_to_internal_color_format
        ; yields one or more complete destination bytes

e1_icolor_full  endp


e1_icolor_partial       proc near

        ; code for external_1_bits/pixel_to_internal_color_format
        ; yields the last few bits on the destination map

e1_icolor_partial  endp
;--------------------------------------------------------------------------
e4_icolor_full          proc near
        ; code for external_4_bits/pixel_to_internal_color_format
        ; yields one or more complete destination bytes

e4_icolor_full endp


e4_icolor_partial        proc near

        ; code for external_4_bit/pixel_to_internal_color_format
        ; yields the last few bits on the destination map

e4_icolor_partial  endp
;--------------------------------------------------------------------------
e8_icolor_full          proc near

        ; code for external_8_bits/pixel_to_internal_color_format
        ; yields one or more complete destination bytes

e8_icolor_full endp


e8_icolor_partial       proc near

        ; code for external_8_bits/pixel_to_internal_color_format
        ; yields the last few bits on the destination map

e8_icolor_partial  endp
```

```
;------------------------------------------------------------------------------;
24_icolor_full          proc near

        ; code for external_24_bits/pixel_to_internal_color_format
        ; yields one or more complete destination bytes

e24_icolor_full  endp


e24_icolor_partial      proc near

        ; code for external_24_bits/pixel_to_internal_color_format
        ; yields the last few bits on the destination map

e24_icolor_partial endp

;------------------------------------------------------------------------------
```

[A similar set of functions would be defined for the 4 DIB formats to do the conversion into a monochrome device-dependent bitmap format.]

By having a separate conversion routine for each DIB format and for each of the two types of device-dependent bitmap formats (i.e., color and monochrome), a lot of conditional checks can be avoided in the body of the main conversion loop, which improves the response.

Each entry in the color table for non-palette devices, which is passed through the BITMAPINFO header, consists of an RGB Quad. The following is the RGBQUAD data structure:

```
typedef struct {
        BYTE    rgbBlue;
        BYTE    rgbGreen;
        BYTE    rgbRed;
        BYTE    rgbReserved;
        } RGBQUAD;
```

Format-Specific Initializations

There are three reasons for having format-specific initializations:

1. To store the addresses of the appropriate full and partial byte procedures in the respective variables used in the "call" instructions in the body of the loop.
2. To calculate the number of complete source bytes that result from the conversion and the alignment of the partial bits in the last byte.
3. To prepare the color table holding the color in the device-specific format for each of the colors in the DIB format. (The color mapping for 24 bits/pixel should be done during the conversion to avoid having to store a huge table.)

The **biClrUsed** field of the DIB header block is important here. If it is zero, the DIB uses the format-specific default color table size. That is, a 4-bits per pixel DIB will have 16 colors, an 8-bits per pixel DIB will have 256 colors, etc. However, if **biClrUsed** is nonzero, it specifies the size of the table and cannot be more than the default size.

Another point to notice here is that some of the DIB formats may never yield a partial byte. For example, since the source scans are always DWORD aligned, the 1, 4, and 8 bits-per-pixel DIB formats will always yield an integral number of destination bytes if the destination format is 1 bit/pixel (and may have multiple planes). In these cases, we can use the **full_byte_proc** alone to convert the entire scan (including the filler bits at the end, if there are any). However, we can still define partial-byte conversion routines for these cases and use them for those scans that span a 64K boundary.

Doing the color mapping from the independent to the device-specific format speeds up the conversion process substantially. If you did it the other way, you could end up doing the mapping for the same source color as many times as it appears in the picture. You could, of course, store the converted colors on the stack. However, in the 24 bits/pixel case, the converted table size would become huge (a maximum of 16 megabytes). Then, you would have to do the color mapping at the same time as the transfer.

GetDeviceBitmapBits

This function has a structure exactly the same as its Set counterpart, except for the color mapping and the fact that the direction of copy is now reversed.

Here you actually need to do an inverse color mapping, from the device-specific format to one of the DIB formats. You also have to create the logical color table that resides in the DIB header block. The device driver, if it is *not* a palette device, can fill up the logical color table with whatever color it supports and, then, use the corresponding indices in the bitmap. It must also set the number of colors it is using in the **biClrUsed** field of the header block.

(This example and table need to be changed to a 4-plane version. Ron or Gunter?)

Consider an example in which the display device is a 3-plane EGA device; the first plane is for red, the second for green, and the third for blue; and in which the DIB has 8 bits/pixel. The logical color table for the DIB has provisions for 256 colors, but the 3-plane driver can deal with only 8 colors. Actually, since the device represents each color component in only 1 bit, a zero bit can be thought of as representing a logical color of 0 and a 1 bit as representing a color of 255. The driver would prepare a color table for the DIB that looked like the following:

Entry No.	Red	Green	Blue
0	0	0	0
1	255	0	0

Entry No.	Red	Green	Blue
2	0	255	0
3	255	255	0
4	0	0	255
5	255	0	255
6	0	255	255
7	255	255	255

The device driver may fill in just 8 colors and set **biClrUsed** = 8, or it may fill up entries 8 through 255 with zeros and set **biClrUsed** = 0.

While doing the conversion, a pixel for the destination DIB can be prepared by storing a bit from each plane of a pixel in the lower significant 3 bits out of the 8, with the other bits all being 0.

The color mapping tables for each of the DIB formats are predefined for a particular driver and should be copied into the DIB header during the format-specific initialization.

RLEBitBlt

This function handles bit transfers both between a screen or a bitmap and a Run Length Encoded (RLE) DIB and vice versa. This function is organized as follows:

```
FetchFromBuffer label word

        dw      DIMapSegOFFSET          color_get_pixel_from_buffer
        dw      DIMapSegOFFSET          mono_get_pixel_from_buffer

FetchIntoBuffer label word

        dw      DIMapSegOFFSET          mem_color_get_pixels_masked
        dw      DIMapSegOFFSET          mem_mono_get_pixels_masked
        dw      DIMapSegOFFSET          dev_color_get_pixels_masked
        dw      DIMAPSegOFFSET          dev_mono_get_pixels_masked

cProc RLEBitBlt,<FAR,PUBLIC>,<si, di, es>

        parmD  lpPDevice       ;bitmap/screen descriptor
        parmW  DstX            ;top left corner x on destination
        parmW  DstY            ;top left corner y on destination
        parmW  DstXE           ;x extent of blt rectangle
        parmW  DstYE           ;y extent of blt rectangle
        parmW  StartScan       ;start of the band of RLE with regards to whole
        parmW  NumScans        ;number of scans in RLE band
        parmW  SetGet          ;0=>set RLE, 1=>get RLE, 2=>get RLE length
        parmD  lpClipRect      ;lptr to clipping rectangle
        parmD  lpDrawMode               ;lptr to draw mode structure
        parmD  lpDIBInfo                ;lptr to RLE DIB info block
        parmD  lpRLEbits                ;lptr to RLE buffer

        localW X
        localW Y                        ;current position for RLE

        localB SurfaceFlags             ;defines the following flags

RLE_MONOequ 01h                         ;monochrome bitmap/screen
RLE_DEVICE      equ 02h                 ;display surface is device
RLE_HUGE        equ 04h                 ;display surface >64k bitmap

;define some variables to hold addresses of device-format specific routines.

        localW  set_partial_pixels      ;set a masked byte
        localW  set_full_pixels         ;unmasked set bytes
        localW  set_get_start_offset    ;translate (x,y) to offset on
                                        ;   display surface
        localW  fill_pixel_buffer       ;function to fill buffer
```

[Define other local variables.]

```
cBegin
```

[Set up the flag bits in **SurfaceFlags** and a long pointer to point to the start byte of the display surface.]

[The Y orientation of RLE DIB is the inverse of the Windows convention. You start encoding from the bottom left of the display surface and work up. So, set the current position at X = DstX and Y = DstY + DstYE - 1.]

[If a "get" from the surface into the RLE DIB is called for, go to **GetRLEBits.**]

[If a "get length" is called for, do the same as **GetRLEBits**, but only return the size of the necessary output buffer. Put this length into the info block as well.]

```
SetRLEBits:
```

[Create a color translate table, converting the logical colors specified in the RLE DIB header block into color indices in the device format. Notice that, if the **biClrUsed** field is nonzero, it specifies the number of colors contained in the color table.

Also, for monochrome display surfaces, set the "color index byte" to be 00H or 0FFH depending on whether the color is to be treated as black or white.]

[If the display surface is EGA/VGA, do the following:

1. Exclude the cursor from within the clip rectangle.
2. For color devices, initialize the adapter registers to be in write mode 2. Enable all the planes for color devices and just plane 0 for monochrome displays. Leave the address

of the bit-mask register in the graphics controller
address register.

3. Set up the following memory variables to the corresponding function addresses:

set_partial_pixels	<— dev_set_pixels_partial
set_full_pixels	<— dev_set_pixels_full
set_get_start_offset	<— set_small_start_offset]

[For memory bitmaps, set up the address variables as follows:

1. For color memory bitmaps:

set_partial_pixels	<— mem_set_pixels_partial
set_full_pixels	<— mem_set_pixels_full

2. For monochrome memory bitmaps:

set_pixels_partial	<— monomem_set_pixels_partial
set_full_pixels	<— monomem_set_pixels_full

3. For small bitmaps:

set_get_start_offset	<— set_small_start_offset

4. For huge bitmaps:

set_get_start_offset	<— set_huge_start offset]

[If the draw mode in the DRAWMODE structure is *opaque*, then the intersection of the clip rectangle and the rectangle encoded by the RLE must now be flooded with the background color.]

[Load **DS:SI** and point to start of RLE buffer. Load **ES** with display surface selector.]

```
set_decode_RLE:
```

[If the **Opaque** flag is set in DRAWMODE, fill the client rectangle with **bkColor**. Use ROP2 throughout.]

[Get the type of RLE record and select one of the following cases.]

[case: absolute mode:]

- Get the number of pixels in the segment.

- Save **SI** and the number of pixels in the segment.
- Call **set_get_segment** (defined later, it figures out the position of the segment relative to the clip rectangle and returns the number of pixels within the clip rectangle in **CX**, the abscissa of the start of the segment in **DI**, and updates **DS:SI** in case of clipping to the first visible pixel's color byte.)
- If CX=0, the segment is totally clipped.
- If CX != 0, call **set_multi_pixel_segment** to decode and transfer the pixels.
- Restore SI and update it by the number of pixels in the segment.
- Jump back to **set_decode_RLE.**

[case: encoded run:]

- Save SI and the number of pixels in the segment.
- Call **set_get_segment** to get the count of pixels within the clip rectangle and the start abscissa in **DI**. Ignore the returned value of SI and have **SI** pointing to the color for the pixel in the segment.
- If CX=0, the segment is totally clipped.
- If CX != 0, call **set_one_pixel_segment** to decode and transfer the pixels.
- Update **SI** by the number of pixels in the segment.
- Jump back to **set_decode_RLE.**

[case: delta encoded record:]

- Update X and Y by the delta_x and delta_y specified in the record.
- Jump back to **set_decode_RLE.**

[case: end-of-line:]

- Set X = DstX and Y = Y - 1.
- Jump back to **set_decode_RLE.**

[case: end-of-segment:]

- Update **DS:SI** to the start of the next 64K segment.

[case: end-of-frame:]

- You are done with the transfer. If the surface is a device, then do the reinitializations and unexclude the cursor.
- Set **AX**=1 (success).
- Jump to **RLEend.**

```
GetRLEBits:
```

[Fill up the color table with the colors that the device supports and set **biClrUsed.**]

[If the surface is a device, then exclude the cursor from within the clip rectangle.]

[Depending on whether the surface is color or monochrome, get the appropriate routine from the FetchFromBuffer and FetchIntoBuffer tables and put it into **get_pixel_from_buffer** and **save_get_pixel_addr**, respectively.]

[Set up **ES:DI** to point to the start of the RLE buffer and load **DS** with the surface selector.]

[Based on the type of the surface, load in **DS:SI** the offset that corresponds to the start point (X,Y).]

```
get_rect_code_loop:
```

[Save **SI** and set the number of buffered pixels fetched from the display surface to be 0.]

[Load the address saved in **save_get_pixel_addr** into the **fill_pixel_buffer** variable.]

[Call **get_next_pixel** to get the pixel at (X,Y).]

[Remember the pixel value as **last_pixel** and set the count of pixels to be 1.]

[Encode the pixels in one scan. This piece of code can be implemented based on a state transition model.

The states are as follows:

State	Description
initial_state	You start here for every scanline and, at this point, remember the value of the last pixel.
initial_to_encode or **encode_to_encode**	You come here when the next pixel matches the first pixel value that you remembered.
encode_absolute	When the new pixel differs from the last pixel, you switch to the absolute encoding mode.
encode_overflow	When the number of pixels in an encoded run reaches 255, you have to close the record. Remember the new pixel as the last pixel, set the count to 1, and go back to the initial state.

State	Description
initial_to_absolute	You get here when you start from an initial state with the first two pixels being different.
absolute_absolute	You stay here as long as you keep getting different pixels.
absolute_overflow	Similar to **encode_overflow**; you have to close this run as the number of pixels reaches 255.
encode_to_scan_end or **absolute_to_scan_end**	At each of the states, you count the number of pixels left in the scan. When this turns to 0, you get to one of these states, depending on whether you were in an encoded or an absolute run.]

[After converting every scan, set the code for end-of-line.]

[Decide if the worst-case encoding of the next scan is going to fit in the remainder of the current segment. If it does not, then add an end-of-segment code and update **ES:DI** to the next segment.]

[Until there are no more scans to convert, loop back to **get_rect_code_loop**.]

```
; you have reached the end of encoding.
```

[Find out the size of the encoded image from the current and initial value of **ES:DI** and save this in the **biSizeImage** field of the DIB header block.]

[If the display surface was a device, bring back the cursor.]

[Set **AX** = 1, for success.]

```
RLEend:
```

[Return back to caller.]

```
cEnd

; the support routines come here.
;-----------------------------------------------------------------------;
set_get_segment proc near
```

[Entry:

(X,Y)	- start of segment on display surface
lpClipRect	- clip rectangle on the surface
AX	- number of pixels in the segment
DS:SI	- points to color byte for first pixel

Returns:

[Entry:

DI	- abscissa of the first unclipped pixel
CX	- number of unclipped pixels
DS:SI	- color byte for first unclipped pixel (ignore this for encoded runs)]

```
set_get_segment endp
;------------------------------------------------------------------------;

set_multi_pixel_segment proc near
```

[Decodes an absolute segment.]

[Gets the bitmask for the first pixel (**DI**,Y).]

```
; the above routine uses the function address in the set_get_start
; offset to translate (DI,Y) to an offset
```

[If the surface is a device, load the address of the bitmask register in **DX**. Otherwise, load the width of a scan in **DX**.]

[For each pixel in the segment, get the color byte pointed to by **DS:SI** and translate it to a device color index. Then, call through the memory variable **set_partial_pixels** to transfer the pixel. Update the bitmask to the next bit.]

```
set_multi_pixel_segment endp
;------------------------------------------------------------------------;

set_one_pixel_segment   proc near
```

[Get a start and end byte mask for the segment and the number of intermediate bytes.]

[Transfer the partial bytes at the end using the address of the function in **set_partial_pixels** and the intermediate bytes using the function whose address is in **set_full_pixels.**]

```
set_one_pixel_segment endp
;------------------------------------------------------------------------;

        ; the following routines all take the following parameters:
        ; ES:DI - destination byte
        ; AL    - color value
        ; AH    - bitmask for the byte
        ; DX    - address of bitmask register for device or offset to next
        ;   scan for bitmask.

dev_set_pixels_partial proc near
```

[Writes a masked byte to the device.]

```
dev_set_pixels_partial  endp

dev_set_pixels_full     proc near
```

[Writes a complete byte to the device without masks.]

```
dev_set_pixels_partial endp

;likewise

mem_set_pixels_partial          ; masked byte for color bitmaps
mem_set_pixels_full             ; complete byte for color bitmap
monomem_set_pixels_partial      ; partial byte for monochrome memory
monomem_set_pixels_full         ; complete byte for monochrome memory
;-----------------------------------------------------------------------;
; Then come the two routines to calculate the start offset on the
; display surface.

set_get_small_offset - for device or small bitmaps
set_get_huge_offset  - for > 64k bitmaps
```

[These routines take **(DI,Y)** as the coordinate and return the offset in **ES:DI.**]

```
;-----------------------------------------------------------------------;

; The strategy that you use for obtaining the next pixel is as follows:
;       At any point, you get all the pixels from the current byte in the
;       surface and buffer them.
;
;       If the first byte has a mask, you throw off the unused portion of
;       the byte and maintain a count of the pixels in the buffer.
;
;       As long as you have pixels in the buffer, you get the pixels from
;       there and this routine is device-format independent.
;
;       When the buffer is empty, you get the next set of bits from the
;       surface. For this, you have separate routines for screen and
;       bitmap, and for color and monochrome.

; Since you have to define the following routines, the parameters to
; them would be the mask and current byte.

; For color devices;

dev_color_get_pixels_masked     - masked pixels from color screen
dev_color_get_pixels            - complete byte fetch

; For monochrome displays:

dev_mono_get_pixels_masked      - masked byte fetch
dev_mono_get_pixels             - complete byte fetch

; correspondingly, you have four routines for bitmap:
```

```
; two for color and two for monochrome:

mem_color_get_pixels_masked
mem_color_get_pixels
mem_mono_get_pixels_masked
mem_mono_get_pixels

; finally, the structure of the get_next_pixel routine is as follows:

get_next_pixel proc near
```

[If you still have pixels in the buffer, call a routine to fetch pixels from the buffer. You may have one routine to fetch color pixels and another to fetch monochrome pixels.]

[If the buffer is empty, call a device-format specific routine to fetch the next byte from the display surface and buffer them. Return with the first pixel in the buffer and update the count of pixels in the buffer.]

```
get_next_pixel endp

;------------------------------------------------------------------------------;
```

The reason for organizing the structure of the routine in the above format is that there are two distinct parts. The main part is device independent and controls the transfer process. This takes help from the second part which is a collection of device-type specific routines.

3.4.2 SetDIBitsToDevice

This third API call in the set is for block transferring (blt'ing) a portion of a DIB directly onto the screen. This call saves you the trouble of first converting the DIB into the device-dependent format and, then, transferring it onto the screen. However, only a direct copy of the DIB is provided. Should you want to use the other raster operations that **BitBlt** supports, you must first convert the DIB into the internal format. Moreover, only one direction of copy (DIB to screen) is provided.

The process of copying out the bits is similar to the one adopted in the **SetDeviceBitmapBits** function except that for some devices, such as the EGA/VGA, the nature of the hardware might make it advantageous to copy one pixel at a time. Here you would need a mask for the current pixel in a byte and, in these cases, you could do away with the **partial_byte** conversion routines and continue working with a mask that is rotated and aligned for every pixel that you copy out.

The idea of having a color translate table and format-specific initialization remains the same. The structure of this function is outlined below. (Please notice the different calling parameters.)

[The function should be defined in the_DIMAPS segment as was done for the others.]

```
cProc SetDIBitsToDevice,<FAR,PUBLIC,WIN,PASCAL,<si, di>

parmD   lp_dest_device    ; long pointer to screen device descriptor
```

```
parmW   ScreenXOrigin     ; top left corner x coordinate
parmW   ScreenYOrigin     ; top left corner y coordinate
parmW   StartScan         ; start scan number in the DIB buffer
parmW   NumScans          ; number of scans to copy
parmD   lp_clip_rect      ; long pointer to clip rect on screen
parmD   lp_DrawMode       ; long pointer to GDI drawmode structure
parmD   lp_DIB_bits       ; long pointer to the DIB buffer
parmD   lp_DIB_header     ; long pointer to DIB header block
parmD   lp_ConversionInfo ; long pointer to Conversion Translate Table
```

[Notice that the DIB bit buffer may not start at the start scan number 0, but could start at the number contained in *StartScan*. However, DIBYOrigin is the actual Y origin relative to start scan 0. Also remember that the DIB bit is actually inverted in the Y direction.]

[Define local and temporary variables here.]

```
cBegin
```

[If the **biStyle** field of the DIB header has the RLE_FORMAT_8 bit set, then call **RLE-BitBlt** to do the transfer. The set of parameters for the calls is similar to what was used in the **DeviceBitmapBits** function, except that the extents of the rectangles are the extents of the DIB. The **Set** or **Get** code is zero to imply a set, and the top-left corner of the destination is set to ScreenXOrigin and ScreenYOrigin.]

[Validate the input parameters and return with **AX** set to 0 if any of the parameters are not valid.]

[Clip the blt rectangle in the DIB against the clipping rectangle on the screen. Return with **AX** set to 0 (i.e., 0 scans copied) if the rectangle is clipped totally and nothing shows.]

[Exclude the display of the cursor from the blt area on screen.]

[Calculate the required parameters such as length of scan, start bitmask, number of pixels to blt on a scan, and number of scans to blt.]

[Start offset of the first bytes on the screen and in the DIB. Remember that the source rectangle is going to be copied upside down onto the rectangle on the screen.]

[Perform format-specific initializations depending upon the number of bits/pixel in the DIB. This includes storing the address of the routine that converts the source pixels one pixel at a time, while adjusting the pixel mask every time.]

[Transfer the source rectangle out one scan at a time, each time calling the conversion routine whose address has been computed during initialization. While the pointers are being updated to the next scan start and the pixels are being copied, take care of scans that span a 64K segment boundary.]

[Return with **AX** set to the number of scans copied.]

```
cEnd

; at this point, organize the format-specific initialization and
; conversion routines.
```

[These routines are similar to the ones discussed for **SetDeviceBitmapBits** except for the following differences.

- Partial-byte procedures are not necessary, but segment crossing should be taken care of in the conversion routines for scans that do span a segment.
- The conversion routine takes as input a start mask and the number of pixels to convert. It adjusts the mask after copying each color-translated pixel. This is repeated until the complete scan is covered.]

[As in the case of the **SetDeviceBitmapBits** function, you need to have separate functions depending upon whether the driver is a monochrome or color device. You can make this function be specific only to a color or a monochrome driver and save some code.]

This completes the discussion on the implementation of the three API calls. The next section discusses the extra bit settings in the GDIINFO block and how to declare these functions in the .DEF file.

3.4.3 Changes in the GDIINFO Block and .DEF File

You must use the GDIINFO block to inform GDI that the driver can now handle DIBs. Do this at offset number 38 of the block (Miscellaneous Raster Capabilities), using the equate RC_DI_BITMAP (defined as 0000000010000000b in GDIDEFS.INC) for DIB-to-memory bitmap capabilities and the equate RC_DIBTODEV (defined as 0000001000000000b in GDIDEFS.INC) for DIB-to-device transfer capabilities. However, if you do not set the RC_DI_BITMAP flag, GDI will simulate in monochrome.

The three DIB entry point functions (at the driver level) should be exported by defining entries in the export section of the .DEF file for the driver. These entries should be made right after the **ExtTextOut** and attribute functions, before the polyline drawing function, and in the following order:

DeviceBitmapBits	**@19**
CreateDIBitmap	**@20**
SetDIBitsToDevice	**@21**

Moreover, the new code segment that holds the code for these functions should also be defined as a discardable code segment. The definition in the .DEF file should be as follows:

```
SEGMENTS
_DIMAPS      MOVEABLE  DISCARDABLE   SHARED
_other segments
      .
      .
      .
```

3.5 Checklist For Updating 2.x Display Drivers To 3.0

The following is a list of the major new Windows 3.0 features that will affect drivers:

- Color Palette Management
- Protected Mode Support
- > 64K Fonts
- Device Independent Bitmaps

Sections briefly highlighting the changes that you will need to make to your drivers were provided in this chapter. However, we strongly recommend that you also review the corresponding sections in the *Microsoft Windows Software Development Kit (SDK)*. They contain more information that might be useful to you while reading this chapter and its checklist.

While rebuilding a Windows 2.x driver with 3.0 tools, you should use the following checklist, which summarizes the points made in this chapter, to keep track of each supported feature as added. When done, thoroughly retest the driver to ensure compatibility.

- ☐ Display devices capable of displaying at least 256 simultaneous colors out of a palette may require a color palette management interface. To support palette management, the device driver must provide the following:
  - ☐ An interface to get (read) and to set (write) the hardware palette.
  - ☐ An interface to get/set the driver-maintained color translate table.
  - ☐ An **UpdateColors** function/entry point.
  - ☐ An updated version number (0300H).
  - ☐ A setting for the RC_PALETTE bit for the **dpRaster** entry in GDIINFO.
  - ☐ Color translation in all draw mode, pen, and physical brush structures.

- ☐ All Windows 3.0 drivers have to be bimodal, i.e., they have to run in protected mode as well as 8086 real address mode. To do so, you may need to import some of these functions from KERNEL:
  - ☐ **AllocSelector**() (@175) — to get a selector
  - ☐ **FreeSelector**(*wSel*) (@176) — to free a selector
  - ☐ **PrestoChangoSelector**(*wSrcSel, wDestSel*) (@177) — for code <—> data selector conversion
  - ☐ **AllocCSToDSAlias**(*wSel*) (@170) — to get a data alias of the code selector
  - ☐ **AllocDSToCSAlias**(*wSel*) (@171) — to get a code alias of the data selector
  - ☐ **__AHIncr** (@114) — to do selector huge increments

- ☐ **LongPtrAdd** (@180) — to do "segment" arithmetic

❑ To avoid general protection faults, *do not* do any of the following:

- ☐ Access (read or write) an array beyond its limits.
- ☐ Have an offset wrap-around (going from 0FFFFH to 0 using a string instruction).
- ☐ Load an invalid selector into a segment register.
- ☐ Update code segment variables.
- ☐ Do segment arithmetic (except as described) for selector registers.
- ☐ Compare segment (selector) registers to see which is lower in memory.
- ☐ Do CLIs and STIs.
- ☐ Use undocumented MS-DOS calls. You should use Windows calls whenever possible.

❑ To support > 64K fonts, make the following code changes:

- ☐ The driver must set the RC_BIGFONT bit of the raster capabilities (**dpRaster**) WORD in the GDIINFO data structure.
- ☐ When addressing character bitmaps, modify the existing code by using the corresponding extended (32-bit) register for the register used by drivers that work with 16-bit offsets.
- ☐ Due to the additional (but currently unused) fields in the header between **dfDBFiller** and **dfCharOffset**, the driver may need to recompute the offset to the character offset table or use an appropriately updated structure.

❑ To handle device-independent bitmaps (DIBs), use the following new functions:

- ☐ **SetDIBits** (has the same entry point as **GetDIBits**)
- ☐ **GetDIBits** (has the same entry point as **SetDIBits**)
- ☐ **SetDIBitsToDevice**
- ☐ **CreateDIBitmap** (a dummy function)

❑ In the GDIINFO block, define as discardable the new code segment for these DIB functions.

Chapter 4 Display Driver Grabbers

The "grabber" is that portion of the non-Windows application support layer that allows the video subsystem to be shared between Windows and non-Windows applications. The grabber implements the video subsystem-specific logic necessary to save and restore video context when switching applications. The grabber also supports the capture of data from non-Windows application screens.

Microsoft supplies a number of grabbers for common video subsystems with the retail version of Windows. Specialized, or less commonly used, video subsystems will require a grabber module customized for that particular hardware if non-Windows applications are to be properly supported. The source code for the Microsoft grabbers is included with this Device Development Kit (DDK).

A number of changes have been made to the grabbers for the real and standard mode versions of the Windows 3.0 release. Specifically, grabber functions no longer used by Windows 3.0 have been removed. In general, grabbers compatible with Windows 2.1 should work with Windows 3.0, but the 3.0 grabbers will not work with Windows 2.1. The grabber source code included with the DDK still contains the unused functions. However, they are excluded via IFDEF statements in the source.

Windows 3.0 can operate in either protected or real mode on an 80286 and later processors. Regardless of the mode in which Windows is running, the grabber is always invoked in the processor's real mode, making it possible to use the grabber with protected-mode and real-mode Windows. For information on the grabbers used with the enhanced mode version of Windows 3.0, see Chapter 18, "The VDD and Grabber DLL," in the *Microsoft Windows Virtual Device Adaptation Guide*.

This chapter documents the interface between Windows running in standard and real mode and the video-specific grabber module. Only the functions actually used by Windows 3.0 are documented; descriptions of functions used by previous Windows releases are not included.

Some portions of the grabber documentation, such as the data structure descriptions, are specific to the implementation of the Microsoft-supplied grabbers. This information is supplied under the assumption that these grabbers will be the starting point for anyone who wants to create a new grabber module.

Familiarity with the Microsoft-supplied grabber source code is assumed throughout this chapter.

4.1 Naming Conventions

The grabber files supplied in the retail version of Windows 3.0 have been renamed to reflect the version with which they are to be used. Instead of VGA.GRB, we now have VGA.GR2 for real and standard mode Windows and VGA.GR3 for enhanced mode Windows. In some cases, several drivers may share one grabber (e.g., EGACOLOR.GR2 is used for both EGAHIRES.DRV and EGAHIBW.DRV).

4.2 Grabber Entry Points

The grabber module is loaded by Windows when the user runs a non-Windows application. Windows then calls a number of grabber entry points to determine parameters, such as the video save buffer size, and to initialize the video subsystem for the application. Windows also makes other calls into the grabber to perform functions such as saving the screen before doing an application context switch and restoring the screen after the context switch.

4.2.1 Standard Function Dispatch Table

Windows loads the grabbers as a binary image and, then, transfers control to them by means of a jump table at offset 0 of the grabber code segment.

Windows computes the offset of the desired entry point using the knowledge that a near jump is 3 bytes long. However, as Windows makes a far call to the jump table, the grabber must do a far return even though the functions are near. Windows will always set **DS** equal to the grabber's CS before making the call.

Windows checks for the existence of an optional jump opcode at offset 015H and, if it exists, assumes that the **InitScreen** entry point is present. If present, **InitScreen** will be called when a non-Windows application starts up and, subsequently, after every context switch to that application.

The following is an example of a function dispatch table:

```
        org 0

StdFuncTable label word
        jmp     InquireGrab     ;Func 00001h
        jmp     Obsolete        ;Func 00002h
        jmp     Obsolete        ;Func 00003h
        jmp     Obsolete        ;Func 00004h
        jmp     InquireSave     ;Func 00005h
        jmp     SaveScreen      ;Func 00006h
        jmp     RestoreScreen   ;Func 00007h
        jmp     InitScreen      ;Func 00008h
```

NOTE With the exception of **InitScreen**, which is an optional entry point, the format of this table *must* remain fixed and *must* reside at offset 0.

4.2.2 Extended Function Dispatch Table

The Windows 2.03 release added a number of extended functions to the grabber interface. Because of the nature of the table in the preceding section, extensions to the grabber interface are made by means of subfunction calls to an existing standard function entry point. The standard function used for this purpose is the **InquireGrab** call. **InquireGrab** must dispatch control to additional function handlers if, upon entry, AX contains a function number in the indicated range.

The following is an example of the extended function dispatch table. Notice that this table differs from the standard table because it contains offsets, not jump instructions.

```
ExtFuncTable    label   word
        dw      Obsolete        ;Func 0FFF4h
        dw      Obsolete        ;Func 0FFF5h
        dw      Obsolete        ;Func 0FFF6h
        dw      Obsolete        ;Func 0FFF7h
        dw      GetBlock        ;Func 0FFF8h
        dw      Obsolete        ;Func 0FFF9h
        dw      GetVersion      ;Func 0FFFAh
        dw      DisableSave     ;Func 0FFFBh
        dw      EnableSave      ;Func 0FFFCh
        dw      SetSwapDrive    ;Func 0FFFDh
        dw      GetInfo         ;Func 0FFFEh
        dw      Obsolete        ;Func 0FFFFh
```

4.3 Data Structures

The following are examples of the most commonly used Windows grabber data structures. This is the way Microsoft does it. However, you may elect to create your own data structures or modify the following ones.

4.3.1 Grabber Information Structure

The extended grabber call GRAB_GETINFO fills a structure provided by Windows with the current video state information. All non-dimensionless quantities are one-based.

```
GrabInfo struc
        giDisplayId     db      ?               ;Display ID Code
        giScrType       db      ?               ;see below
        giSizeX         dw      ?               ;X raster size in .1mm units
        giSizeY         dw      ?               ;Y raster size in .1mm units
        giCharsX        db      ?               ;# X char cells (columns)
        giCharsY        db      ?               ;# Y char cells (rows)
        giMouseScaleX   db      ?               ;X transform for MS-MOUSE
        giMouseScaleY   db      ?               ;Y transform for MS-MOUSE
        giReserved      db      38 dup (?)
GrabInfo ends
```

The following are the bitmaped codes for the **giScrType** field in the GRABINFO structure:

```
ST_TEXT         = 00000000b ;screen is alphanumeric
ST_GRPH         = 00000001b ;screen is graphics
ST_LARGE        = 00000010b ;screen too big to switch
ST_SPECGRAB     = 00000100b ;reserved
```

4.3.2 Grabber Request Packet Structure

Upon entry, all the block functions expect **ES:DI** to point to a GRABREQUEST structure that is formatted as follows. Since not all the fields are used by some functions, field usage is detailed in the header for each function.

```
GrabRequest struc
        grlpData        dd      ?       ;long ptr to I/O buffer
        grXorg          db      ?       ;x origin (unsigned)
        grYorg          db      ?       ;y origin (unsigned)
        grXext          db      ?       ;x extent (unsigned)
        grYext          db      ?       ;y extent (unsigned)
        grStyle         db      ?       ;style flags
        grChar          db      ?       ;char code for fill ops
        grAttr          db      ?       ;attribute for fill ops
GrabRequest ends
```

The following are the codes for the **fScreenOps** field of the GRABREQUEST data structure's **grStyle** field:

```
SCR_OP_MASK     = 00000111b

F_BOTH          = 000h          ;fill w/ single char and attr
F_CHAR          = 001h          ;fill w/ single char only
F_ATTR          = 002h          ;fill w/ single attr only
C_BOTH          = 003h          ;copy chars and attrs from lpData
C_CHAR          = 004h          ;copy chars only from lpData
C_ATTR          = 005h          ;copy attrs only from lpData
C_CHAR_F_ATTR   = 006h          ;copy chars from lpData,fill w/ attr
C_ATTR_F_CHAR   = 007h          ;copy attrs from lpData,fill w/ char
```

The following are the codes for the **fFormat** field of the GRABREQUEST data structure's **grStyle** field:

```
FORMAT_MASK     = 10000000b     ;mask to extract fFormat
FMT_NATIVE      = 00000000b     ;use format native to mode
FMT_OTHER       = 10000000b     ;use clipbrd or fScreenOps
```

The following are the error codes for block operations:

```
ERR_UNSUPPORTED = 0FFh ;block op not supported
ERR_BOUNDARY    = 0FEh ;src or dest range error
```

4.3.3 Grab Buffer Structure

The following is the format of the GRABST data structure: **(Amit, what else can we say about this structure?)**

```
GrabSt struc
        gbType          dw      ?       ;see below
        gbSize          dw      ?       ;length (not including 1st 4 bytes)
        gbWidth         dw      ?       ;width of bitmap in pixels
        gbHeight        dw      ?       ;height of bitmap in raster lines
        gbPlanes        dw      ?       ;# of color planes in the bitmap
        gbPixel         dw      ?       ;# of adj color bits on each plane
        gbWidth2        dw      ?       ;width of bitmap in 0.1 mm units
        gbHeigh2        dw      ?       ;height of bitmap in 0.1 mm units
        gbBits          dw      ?       ;the actual bits
GrabSt ends
```

The following are the codes for the **gbType** field of GRABST:

```
GT_TEXT         = 1
GT_OLDBITMAP    = 2
GT_NEWBITMAP    = 3
GT_RESERVED4    = 4
GT_RESERVED5    = 5
```

4.3.4 Information Context Structure

INFOCONTEXT is a global structure shared by all the grabbers and may be considered the non-Windows application version of the GDIINFO structure for Windows drivers. It provides information needed by functions at all levels of the grabber source directory tree. While storage for this structure is always allocated in the leafnode directory module for each grabber, it is typically validated by the **GetMode** function. Notice that it is essentially a superset of the GRABINFO structure returned by the **GetInfo** entry point. GRABINFO has sufficient reserved space so that this information may be made available to the Windows layer in the future as well as allow for more fields being added if needed.

```
InfoContext struc
        icDisplayId     db      ?       ;Display ID code
        icScrType       db      ?       ;Screen type code
        icSizeX         dw      ?       ;Horz raster size in .1mm units
        icSizeY         dw      ?       ;Vert raster size in .1mm units
        icCharsX        db      ?       ;Number of character columns
        icCharsY        db      ?       ;Number of character rows
        icMouseScaleX   db      ?       ;Mouse to grabber coord xform in X
        icMouseScaleY   db      ?       ;Mouse to grabber coord xform in Y
        icPixelsX       dw      ?       ;Number of pixels in X
        icPixelsY       dw      ?       ;Number of pixels in Y
        icWidthBytes    dw      ?       ;Width in bytes of a row/scanline
        icBitsPixel     db      ?       ;Number of adjacent bits/pixel
        icPlanes        db      ?       ;Number of planes per pixel
        icInterlaceS    db      ?       ;Interlace shift factor
        icInterlaceM    db      ?       ;Interlace mask factor
        iclpScr         dd      ?       ;Long pointer to screen
        icScrLen        dw      ?       ;Current screen page length
InfoContext ends
```

4.3.5 Device Context Structure

The DEVICECONTEXT structure is a device-dependent structure private to the low-level grabber modules. Its layout, content, and/or length will vary from grabber to grabber in other branches of the grabber source directory tree.

The following is the DEVICECONTEXT structure for CGAHERC grabbers:

```
DeviceContext struc
        dcScrMode       db      ?           ;BIOS screen mode
        dcScrStart      dw      ?           ;regen start position
        dcCursorPosn    dw      ?           ;cursor position in CRTC format
        dcCursorMode    dw      ?           ;cursor start/stop scanlines
        dcModeCtl       db      ?           ;3x8 mode reg data
        dcExModeCtl     db      ?           ;3xx extended mode reg data
        dcColorSelect   db      ?           ;3D9 color select reg data
        dcCrtcParms     dw      ?           ;->CRTC parms for non-BIOS modes
        dcfSwitchGmt    db      ?           ;switch graphics/multiple text
DeviceContext ends
```

The following is the DEVICECONTEXT structure for EGA grabbers:

```
DeviceContext struc
        dcScrMode       db      ?           ;BIOS screen mode
        dcScrStart      dw      ?           ;regen start position
        dcCursorPosn    dw      ?           ;cursor position in CRTC format
        dcCursorMode    dw      ?           ;cursor start/stop scanlines
        dcAddrPatch     db      ?           ;3Dx / 3Bx patch byte
        dcfSwitchGmt    db      ?           ;Switch graphics/multiple text
        dcFileNum       dw      ?           ;->random number in swapfile
        dcFontBank      db      4  dup (?)  ;ERI font info
        dcSwapPath      db      64 dup (?)  ;full swap path
DeviceContext ends
```

The following is the DEVICECONTEXT structure for VGA grabbers:

```
DeviceContext   struc
        dcScrMode       db      ?          ;BIOS screen mode
        dcScrStart      dw      ?          ;regen start position
        dcCursorPosn    dw      ?          ;cursor position in CRTC format
        dcCursorMode    dw      ?          ;cursor start/stop scanlines
        dcAddrPatch     db      ?          ;3Dx / 3Bx patch byte
        dcfSwitchGmt    db      ?          ;Switch graphics/multiple text
        dcFileNum       dw      ?           ;->random number in swapfile
        dcFontBank      db      8  dup (?) ;VGA has 8 font banks
        dcSwapPath      db      64 dup (?) ;full swap path
DeviceContext   ends
```

4.4 Coordinate System

All the grabber block functions operate using a left-hand coordinate system that is based on character cells in a manner similar to that used by the PC's BIOS. The origin of the display surface (0,0) is located in the upper-left corner of the screen. Positive X direction is to the right and positive Y direction is downward. The coordinate space consists of the set of integers that range from 0 to (**icCharsX** - 1) in the X direction and from 0 to (**icCharsY** - 1) in the Y direction. The quantities **icCharsX** and **icCharsY** are found in the grabber's INFOCONTEXT structure, which may be obtained at any time by calling the extended grabber entry point **GetInfo**. Notice that points in this space represent the upper-left corner of the character cells, not their center.

As indicated by their names, the block functions operate on blocks of character cells. A block is fully specified by its origin (relative to the screen origin) and its X and Y extents in the GRABREQUEST structure. The extents are unsigned one-based quantities. A block is defined to be the set of character cells in the 2-D range ([**grXorg**, **grXorg** + **grXext**], [**grYorg**, **grYorg** + **grYext**]). Notice that specifying an extent of 0 on either axis has the same effect as specifying an extent equal to the maximum screen extent on that axis. In other words, the entire screen may be easily specified by setting:

grXorg = **grYorg** = **grXext** = **grYext** = 0

For graphics mode, the same cell-based convention applies so that Windows need not discern differences between graphics and text screens. The size of a character cell in graphics mode is defined as the same size cell that BIOS would use to display text using INT 010H functions. Although the current grabbers do not yet support the specification of arbitrary block regions in graphics mode, it is still possible to request a full-screen **GetBlock** operation by using the preceding procedure. An attempt to specify any other block in graphics mode will return the ERR_UNSUPPORTED error code.

Although the ERR_BOUNDARY error code is provided in GRABBER.INC for blocks that violate the screen boundaries, none of the block functions can check currently for this condition. Thus, the operation proceeds with undefined results.

4.5 Buffer Size Calculations

The low-level functions of all the grabbers define a number of equates used to calculate the size of various data buffers needed to take screen snapshots and context switch the display subsystem. Since these calculations vary widely among display adapters, the motivations for many of the assumptions may not be clear. Therefore, the explanation provided in the following subsections is an attempt to clarify this process.

4.5.1 MAX_GBTEXTSIZE and MAX_GBGRPHSIZE

These two equates define the maximum size of the GRABST header (defined in GRABBER.INC) when used to hold text and graphics, respectively, during a screen grab (snapshot). The size of the data portion is display dependent and defined by MAX_VISTEXT and MAX_VISGRPH.

4.5.2 MAX_CDSIZE

This equate defines the minimum size of the context data buffers needed to support a video context switch (**SaveScreen/RestoreScreen**). It does not include the size of buffers required to save the actual screen data, which is defined by MAX_TOTTEXT and MAX_TOTGRPH. The definition of this equate is split across 3 lines to make it fit within 80 columns.

All grabbers include the size of the Device Context (DC) and Information Context (IC) structures as well as the size of the video BIOS data area in this equate. The OEM may additionally include the size of OEM-specific data structures that must be saved. Also, a given display device may require storage for such things as device-specific BIOS areas. For the EGA, this includes the size of the EGA BIOS data area as well as the 4 bytes comprising the EGA **SavePtr**.

4.5.3 MAX_VISTEXT and MAX_VISGRPH

For most supported display devices, the size of the visible portion of a text or graphics page is less than the total size of the page. Therefore, some video RAM is unused. Screen grabs are defined to copy only that portion of the screen that is visible. Unfortunately, the size of the visible page varies among display modes. Since **InquireGrab** can return only one size for text pages and one size for graphics pages, we must set MAX_VISTEXT and MAX_VISGRPH to the size of the largest visual page we want to capture.

Notice that the page size indicated for text is actually somewhat larger than a visual page since the screen grab buffer format specifies that text grabs must have carriage-return/linefeed pairs at the end of each line except the last, which must be terminated by a WORD of zeros.

The character generator on the EGA is programmable and, therefore, the size of a visual page is unknown at the time of **InquireGrab**. One could simply specify a giant buffer, but that would be wasteful most of the time. Thus, we have chosen the EGA's 43-line mode as the largest text screen we will attempt to handle. Anything larger will require that the user select the PIF settings for graphics. However, the VGA grabber will allow 50-line screen grabs.

For graphics grabs, the maximum size we support is 16K for the EGA, which is enough for one page of 640x200x1 or 320x200x2 graphics. We do not support the hires/multicolor modes due to their large memory requirements (grabs are asynchronous and cannot be swapped to disk).

4.5.4 MAX_TOTTEXT and MAX_TOTGRPH

According to settings in the PIF file, a context switch must save either the entire current page of text or "graphics/multiple text." The former case is well defined and includes both the visual and offscreen portions of the current text page. The latter, however, is open to interpretation by the implementer. For example, any graphics mode on the CGA adapter requires the entire frame buffer, allowing for only one page of graphics. Therefore, saving the graphics page completely saves all the possible text pages as well.

Some display adapters, such as EGA and Hercules, differ in this respect in that they contain multiple graphics pages. Here, you could interpret the PIF setting to mean "multiple graphics/multiple text" and allocate space for the union of all text and graphics pages. You could also elect to save the user some memory and allocate space for only the largest single graphics page (interpreting it as "single graphics/multiple text").

Unfortunately, both interpretations usually require prohibitive amounts of memory. On the EGA or VGA, the first method would require that you save all 256K. The second would require saving 128K, since the largest graphics page is in mode 010H (640x350x4) and requires four planes of 32K (including offscreen memory). Therefore, the situation must be examined for each display adapter and a compromise reached.

As a compromise on the EGACOLOR or VGACOLOR grabbers, you must interpret the PIF setting as "single page of lores graphics/some of the text pages." This method requires an allocation of only 16K, which is enough for one page of lores graphics or four of the 80x25 text pages. Since it is increasingly important to save hires graphics also, the EGA grabber is designed to save one page of hires graphics using virtual memory by swapping the page to disk. This method is much slower than saving to a memory buffer, but uses less of the user's memory. Unlike screen grabs, this technique is safe to implement because Windows must also swap to disk, and thus ensures that you do not reenter MS-DOS for file I/O.

The only compromise required on the EGAMONO or VGAMONO grabbers is the interpretation of the PIF setting as "single page of graphics/all text pages." This means it is possible to lose a graphics page if the non-Windows application is maintaining two pages of graphics. However, this is a rare enough occurrence that we choose not to penalize the user with a 64K data buffer requirement "just to be safe."

As a compromise on the HERCULES grabber, you must interpret the PIF setting as "single page of graphics/no text pages." The logic here is based on the assumption that when graphics is active, text is not and, therefore, does not need to be saved. Furthermore, it would take 64K to save both graphics pages, which is far too much memory to request. It is rare that the user runs an application that maintains images on both graphics pages, so it is not necessary to require a 64K buffer.

No compromise is required on the CGA adapter since we save all 16K of the adapter's memory.

4.5.5 GrabTextSize and GrabGrphSize

The **GrabTextSize** variable holds the minumum buffer size needed to support a text grab. The **GrabGrphSize** variable holds the minimum buffer size needed to support a graphics grab. These variables are initialized to the sum of MAX_GBTEXTSIZE and MAX_VISTEXT, and MAX_GBGRPHSIZE and MAX_VISGRPH, respectively. They are maintained as variables instead of constants so that the grabber initialization functions may modify their size at run time if other features are detected that need consideration.

4.5.6 SaveTextSize and SaveGrphSize

The **SaveTextSize** variable holds the minimum buffer size needed to support a text context switch, while the **SaveGrphSize** variable holds the minimum buffer size needed to support a graphics context switch. These variables are initialized to the sum of MAX_CDSIZE and MAX_TOTTEXT, and MAX_CDSIZE and MAX_TOTGRPH, respectively. They are maintained as variables instead of constants so that the grabber initialization functions may modify their size at run time if other features are detected that need consideration.

If the EGA Register Interface (ERI) is present for the EGA, the size of the ERI's context buffer is added to these variables during **DevInit** so that data may be saved as well. This is important since, in many cases, some applications (such as Microsoft Word) use the ERI to modify EGA registers, in which case this context data will be more accurate than the DC and IC structures alone.

4.6 Function Reference

The following are alphabetically organized descriptions of the grabber functions used for Windows 3.0 when running in real and standard modes.

DisableSave

Purpose Disables video context switching.

This entry point gives the grabber the chance to remove any hooks installed by **EnableSave.**

Entry **DS = CS**

Exit None

EnableSave

Purpose Enables video context switching.

This entry point gives the grabber the chance to install any hooks needed for context switching.

Entry DS = CS

Exit None

GetBlock

Purpose Copies a rectangular area of the screen to the buffer.

GetBlock copies a rectangular block of screen data to a specified buffer in a specified format.

Entry DS = CS
ES:DI -> GRABREQUEST structure

In GRABREQUEST, the fields are as follows:

Field	Description
lpData	If **lpData** = NULL, **GetBlock** does not actually perform the transfer; it simply returns the number of bytes the operation would have required. Otherwise, **lpData** is assumed to point to the buffer that receives the screen data.
Xorg	Specifies the unsigned X coordinate of the origin of the transfer in alpha coordinate space.
Yorg	Specifies the unsigned Y coordinate of the origin of the transfer in alpha coordinate space.
Xext	Specifies the unsigned X extent of the rectangular block to transfer as measured from the origin specified by **Xorg** and **Yorg**. If **Xext** is zero, the entire width of the current screen is assumed.
Yext	Specifies the unsigned Y extent of the rectangular block to transfer as measured from the origin specified by **Xorg** and **Yorg**. If **Yext** is zero, the entire height of the current screen is assumed.
Style	The only field of **Style** used for **GetBlock** operations is the **fFormat** field.

Field	Description
	If **fFormat** = FMT_NATIVE, the data is copied in the native screen format for the current screen mode. Thus, text screens result in character/attribute pairs being copied, while graphics screens result in a bitmap. Since the native format is screen dependent, we recommend that it be used for save/restore purposes only; no interpretation of the data should be attempted.
	If **fFormat** = FMT_OTHER, the data will be copied in a variant of the Windows Clipboard format defined by the grab buffer structure GRABST in GRABBER.INC. This buffer will have **gbType** GT_TEXT for text screens and **gbType** GT_NEWBITMAP format for graphics screens.
Char	Not used for **GetBlock** operations.
Attr	Not used for **GetBlock** operations.

Exit AX = number of bytes transferred

Error Exit **AX** = error code
CF = 1

Comments In text mode, the display adapter may contain multiple character sets or allow downloadable character sets. The data returned in Clipboard format will be translated to the "standard" OEM set if it can be determined that another set is in use, which character set it is, and if a translation table is available for the job. The EGA is an example of an adapter that makes it hard to determine that a set has been downloaded and practically impossible to determine which one it is.

GetInfo

Purpose Returns the GRABINFO structure.

GetInfo fills a buffer pointed to by **ES:DI** on entry with data in the GRABINFO format. Notice that the INFOCONTEXT structure is copied since it is a superset of the GRABINFO structure (although INFOCONTEXT is not as big as GRABINFO, because of the existence of reserved fields in the latter).

Although Windows knows nothing about the reserved fields, you end up filling some of them with the extra data in INFOCONTEXT because it is convenient for debugging purposes. Because these fields are still reserved, *they may change at any time.*

Notice that the block functions also check the current mode and return ERR_UNSUPPORTED to indicate non-support for the request, which is the preferred way to determine support for a given mode.

Entry	**ES:DI** -> GRABINFO structure to fill
Exit	**AX** = 0 if block ops not supported on current mode (graphics) **AX** = 1 if block ops supported on current mode (text)

GetVersion

Purpose	Returns the grabber version number.
Entry	None
Exit	**AX** = version number

InitScreen

Purpose

(Amit, please review carefully.) Initializes the screen to a known text mode and a requested number of lines per screen.

This function is called by Windows once before the non-Windows application starts up. Windows also calls this function after the non-Windows application's screen has been saved and the **DisableSave** function has been called during a context switch away from the application. The function is called a third time after the non-Windows application finishes execution. The last two calls to this function ensure that the display is left in a known text mode before putting up Windows' preview switcher screen or going back to Windows.

Entry

DS = **CS**

AX = Number of lines per screen
(If the number of lines requested cannot be supported by the display device, it will be rounded down to a figure that the device can support. Notice that this parameter is new for Windows 3.0. Earlier grabber versions do not take any parameter.)

Exit

None

InquireGrab

Purpose Returns the size of the grab buffer needed or dispatches an extended function call.

In Microsoft Windows 1.0x, this routine handled only grab buffer size requests.

Microsoft Windows 2.0x added the extended functions dispatched by **InquireGrab**. For an explanation of the new subfunctions mentioned in Section 4.2.1, "Standard Function Dispatch Table," and Section 4.2.2, "Extended Function Dispatch Table," refer to their descriptions in this section.

Entry

DS = **CS**
AX = *n*

where *n* is either:

1	Inquire text grab buffer size
2	Inquire graphics grab buffer size
n > 2	Extended subfunction request

Exit

If the function number is 1 or 2,
DX:AX = size in bytes for the grab buffer

else
exit status depends on the extended call.

Comments

To get any useful work from the grabber, Windows must call either **InquireGrab** or **InquireSave** before any other call (**InitScreen** is a possible exception). This is an opportunity to initialize the module if it is the first time one of these entry points has been called.

The grabbers supplied by Microsoft also perform first-time internal initialization when either **InquireGrab** or **InquireSave** are called for the first time.

See also: **InquireSave**, GRABBER.INC, ENTRY.ASM

InquireSave

Purpose

Returns the necessary size of the screen save buffer.

Entry

DS = **CS**
AX = *n*

where *n* is either:

1	Inquire text context save buffer size
2	Inquire graphics context save buffer size

Exit

DX:AX = size in bytes for save buffer

RestoreScreen

Purpose	Restores a previously saved display context.
Entry	AX = size in bytes of screen save area DS = CS ES:DI -> save area
Exit	CF = 0 (screen was successfully restored)
Error Exit	CF = 1 (unable to restore screen)
Comments	Windows guarantees that the offset portion of the screen save area will always be zero.

SaveScreen

Purpose	Saves the current display context.
Entry	AX = size in bytes of screen save area DS = CS ES:DI -> screen save area
Exit	CF = 0 (screen was successfully saved)
Error Exit	CF = 1 (unable to save screen)
Comments	Windows guarantees that the offset portion of the screen save area will always be zero.

SetSwapDrive

Purpose

Sets the current swap drive and path.

The next **SaveScreen** call following this call uses the given swap drive letter to open the swapfile, if needed. Failure to call this function at least once before **SaveScreen** may result in a failure to context switch. This drive and path information is stored as a static value and need not be set before every call to **SaveScreen**. If it is known that the swap drive will remain constant for the "life" of the grabber instance, then we recommend that this call be done once, after Windows' decision to allow context switching has been made.

Entry	**BL** = ASCII drive letter for swap (A, or B, or C, ...) = 0FFH if no swap drive available **ES:DI** -> d:pathname template for Windows temporary file format **DS** = **CS**
Exit	None

Chapter 5 Printer Drivers

Microsoft Windows 3.0 printer drivers provide the interface necessary to obtain device independence between the Windows environment and the printer hardware. The driver gives Windows and Windows applications information on support for fonts, paper trays and sizes, printer orientation, graphics capabilities, color, and other advanced features that may be available on the printer. This information may be used by applications in creating the desired printed output.

Since printed output can come in a variety of different mediums by different types of devices, you must have a unique driver to support your hardware's technology. Such types of technology could be categorized as raster devices (e.g., dot matrix, most laser, and inkjet printers and some film recorders), vector devices (e.g., plotters), or devices with higher-level languages, such as POSTSCRIPT.

The printer driver interacts with the Graphics Device Interface (GDI), Print Manager, and, indirectly, with Windows applications via the GDI Escape function. The Escape function supports many subfunctions; some are required by the driver, and others are specifically defined to take full advantage of unique technologies. In some cases, these additional escapes provide Windows applications with support not available through GDI, such as a mechanism to take advantage directly of higher-level devices. See Chapter 11, "Device Driver Escapes," for more information on and detailed descriptions of these escapes.

This chapter describes the support you need to provide in your printer driver. Of course, the extent of the support you provide depends on the type of hardware supported. However, we strongly encourage you to implement all the structures and functions defined in this chapter that are applicable to your device. By doing so, Windows applications will be able to take full advantage of your hardware device.

5.1 Basic Information

A Windows printer driver is a Windows dynamic-link library (DLL) that implements a set of standard graphics primitives for a particular printer device. Unlike most device drivers, a printer driver is generally not responsible for hardware communication with the printer, a task normally reserved for the Windows Print Manager and the communications driver. The printer driver essentially translates the device-independent GDI interface into a stream of printer commands and data.

5.1.1 The GDI Interface

GDI's interface to a printer driver is much simpler than an application's interface to GDI, since GDI takes care of all the coordinate transformations, object management, and most clipping. There is, however, an analogy between the two sets of functions.

Applications that call GDI functions draw into GDI device contexts. A *device context* (DC) is GDI's device-independent virtualization of the output device, be it the display, a printer, a memory bitmap, or even a metafile not associated with any specific device.

When GDI creates a device context, the printer driver is asked to supply information about the printer, for example, the resolution of the device and the number of colors. In addition, the printer driver is asked to create its own internal state information about the device. The device-dependent state (referred to as the physical device structure or PDEVICE) corresponds to the DC at the driver level.

The application can request information about the device context, for example, to determine what fonts are available. GDI will call the driver to supply device-specific information. GDI will supply the driver with the PDEVICE structure that corresponds to the DC the application is using.

Another portion of GDI deals with managing drawing tools or objects, such as pens, brushes, and fonts. For the most part, GDI manages these objects itself. However, when the application uses one of the objects to create output, GDI asks the driver to create a device-specific representation of the object.

When an application performs output to a particular DC by using the graphics functions, GDI will first perform any necessary transformations and, then, call the driver to perform that output into the corresponding physical device. For simpler printer drivers, GDI may also simulate complex graphics, such as filled polygons, with simpler driver primitives.

5.1.2 Additional Printer Driver Responsibilities

Printer drivers are responsible for providing a means for applications and the system Control Panel to set up and manage the printer. Each printer driver contains a dialog box for selecting printer options, such as paper size and orientation. In Windows 3.0 printer drivers, this setup may also be directly manipulated by applications without going through the dialog box.

In most cases, printer drivers are not responsible for sending bytes directly to the output port. Instead, printer drivers call special GDI functions to perform output. Depending on the options selected by the user, those functions will route the output to a specific port, to a disk file, across a network connection, or to a temporary file for later output by Print Manager.

5.1.3 Printer-Driver Developer Responsibilities

Since printer drivers are Windows DLLs, the prospective printer-driver developer should understand Windows programming, particularly in the areas of libraries, which are different from applications, and memory management, which is essential for ensuring good performance and cooperation within Windows. An understanding of application programming is less important but useful for writing the driver's Setup dialog box and for understanding the printer driver's role in providing Windows device-independent graphics.

Source code for several printer drivers has been supplied on the disks provided in this kit. These sources can provide a good basis for developing drivers for similar printers. The samples cover a wide range of devices, including the following:

- A simple driver for the Epson ® printer
- A color driver for IBM's PC Color Printer
- Drivers for two different laser printer command languages, PCL (for the HP ® LaserJet ®) and POSTSCRIPT.

The next few sections provide descriptions of the services the driver must provide and the Windows functions intended for driver use.

5.2 Printer Driver Initialization

When Windows creates a device context (DC) for a device, GDI makes two calls to an entry point in the device's driver called **Enable()**.

The first call to **Enable()** is used to create a data structure for GDI called GDIINFO. The purpose of the GDIINFO structure is to describe the device to GDI. The structure of GDIINFO is the same for all device drivers. (See Section 5.2.2, "The GDIINFO Data Structure," in this chapter for a complete description. See also Chapter 2, "Display Drivers," for another description of GDIINFO from a display driver's perspective.)

The second call to **Enable()** causes the driver to initialize a second data structure called PDEVICE. PDEVICE is entirely device dependent and generally used by the device driver to store state information for a printing job.

5.2.1 The Enable() Function and Its Parameters

The **Enable()** function is declared as follows:

```
WORD FAR PASCAL Enable(
        LPSTR       lpDest,
        WORD        style,
        LPSTR       lpDevice,
        LPSTR       lpOutput,
        LPDEVMODE   lpInitData);
```

The *lpDest* parameter may contain a far pointer to either a GDIINFO or a PDEVICE structure, depending on the contents of the *style* parameter. If the low-order bit of *style* contains 0, the device driver initializes the GDIINFO structure. Otherwise, *lpDest* points to the PDEVICE structure the driver must initialize.

In either case, if the high-order bit is set, GDI creates an information context (i.e., a device context used only for querying the device for information and not for output). Some drivers can reduce the amount of memory they allocate by assuming no output.

The *lpDevice* parameter points to a NULL-terminated string that identifies the device (e.g., PCL / HP Laserjet). For printer devices, the name of the device is the string installed in the [*devices*] section of the WIN.INI file.

The *lpOutput* parameter points to a NULL-terminated filename or MS-DOS device name that identifies, for printer devices, the output port to which Print Manager sends the printer data.

Printers often have a number of Setup options that a device driver can represent in a device-dependent initialization data format. This data may appear in a buffer pointed to by *lpInitData*. The role this data plays in controlling the driver will be discussed in Section 5.3, "The Printer Driver Environment."

Enable() returns the size of the structure copied if successful or zero if an error occurred.

5.2.2 The GDIINFO Data Structure

The structure of GDIINFO is as follows:

```
typedef struct _gdiinfo {
        int      dpVersion;
        int      dpTechnology;
        int      dpHorzSize.
        int      dpVertSize;
        int      dpHorzRes;
        int      dpVertRes;
        int      dpBitsPixel;
        int      dpPlanes;
        int      dpNumBrushes;
        int      dpNumPens;
        int      futureuse;
        int      dpNumFonts;
        int      dpNumColors;
        unsigned dpDEVICEsize;
        unsigned dpCurves;
        unsigned dpLines;
        unsigned dpPolygonals;
        unsigned dpText;
        unsigned dpClip;
        unsigned dpRaster;
        int      dpAspectX;
        int      dpAspectY;
        int     dpAspectXY;
        int     dpStyleLen;
        POINT   dpMLoWin;
        POINT   dpMLoVpt;
        POINT   dpMHiWin;
        POINT   dpMHiVpt;
        POINT   dpELoWin;
        POINT   dpELoVpt;
        POINT   dpEHiWin;
        POINT   dpEHiVpt;
        POINT   dpTwpWin;
        POINT   dpTwpVpt;
        int     dpLogPixelsX;
        int     dpLogPixelsY;
        int     dpDCManage;
        int     futureuse3;
        int     futureuse4;
        int     futureuse5;
        int     futureuse6;
        int     futureuse7;
        int     dpPalColors;
        int     dpPalReserved;
        int     dpPalResolut;
} GDIINFO;
```

This rather large structure really has three kinds of fields in it:

- Driver management fields
- Driver capabilities fields
- Device dimension fields

The Driver Management Fields

These fields include the following:

Field	Description
dpVersion	Contains the version of Windows for which the driver was written (not the driver version number). This field should contain 0x300 for drivers written for Windows 3.0.
dpTechnology	Contains an index that broadly classifies the device. Possible classifications are either: Vector Plotter (0),Raster Display (1), Raster Printer (2), Raster Camera (3), Character Stream (4), Metafile or VDM (5), or Display File (6).
dpDEVICEsize	Provides Size of (PDEVICE) or the number of bytes that GDI should allocate for the device's PDEVICE structure, which is initialized by the driver during the second **Enable()** call.

Field	Description
dpDCManage	Specifies how multiple device contexts for the same device are to be treated.

The **dpDCManage** field contains a combination of three bits.

Bit	Description
DC_SPDevice (1)	Uses separate GDIINFO and PDEVICE structures for each DC created for the driver.
DC_1PDevice (2)	For each combination of device name and output port name, only one DC is allowed to exist (i.e., only one PDEVICE is allocated).
DC_IgnoreDFNP (4)	Uses the same PDEVICE and GDIINFO structures for all the DCs.

If none of these bits is set, there will be one GDIINFO and PDEVICE structure allocated per output file, regardless of the number of device contexts created on the port.

When **DC_1PDevice** and **DC_IgnoreDFNP** are combined, only one DC is allowed to exist for a device, regardless of the output filename. Other combinations of bits are invalid.

Printer drivers normally set the **DC_SPDevice** bit, allowing multiple independent jobs to be spooled to the same printer simultaneously.

The Driver Capabilities Fields

The driver capabilities are specified by the following fields:

Field	Description
dpNumBrushes	The number of predefined brushes supported by the device.
dpNumPens	The number of device pens.
dpNumFonts	The number of device fonts in the printer.
dpNumColors	The actual number of physical, non-dithered colors that the device can print.
dpCurves	A bit field describing the driver's ability to output curved graphics (such as ellipses, arcs, or wedges) and what kind of effects can be supported.
dpLines	A bit field describing the driver's ability to draw lines and polylines, and the effects that may be achieved with such figures.

Field	Description
dpPolygonals	The capability of the device to support polygons and scanlines.
dpText	A bit field describing the level of support in the driver for text output, such as support for Windows fonts, transformations and effects, and scaling and output precision.
dpClip	A bit field describing whether or not the device can do clipping. If this field is 0, the device cannot clip output. If it contains 1, the device can clip to an arbitrary rectangle.
dpRaster	Additional capabilities for raster and display devices, such as bit-map and banding support.

Brushes, pens, fonts, and colors are discussed in more detail in Section 5.6, "GDI Graphics Objects."

The bit fields for curves through rasters are discussed in more detail in Section 12.7.1, "Information Data Structures."

There are several new bits for the **dpRaster** field that have been defined for version 3.0 drivers. They are as follows:

Bit	Description
RC_DI_BITMAP	The device supports the conversion of device-independent bit-maps to compatible-memory bitmaps in all the DIB resolutions (1, 4, 8, and 24 bits-per-pixel). However, if the flag is *not* set, GDI will simulate in monochrome.
RC_PALETTE	The device uses a color palette. This bit is primarily for display devices.
RC_DIBTODEV	The device can copy a device-independent bitmap to the page via **SetDIBitsToDevice()**.
RC_BIGFONTS	The device driver supports Windows font files in the new, version 3.0 format, which supports greater than 64K fonts. However, if the flag is *not* set, all the fonts will be in the old, version 2.0 format.
RC_STRETCHBLT	The device supports **StretchBlt()**.
RC_FLOODFILL	The device supports **FloodFill()**.

For more information on these new, version 3.0 output functions, see Section 5.7, "Performing Output."

There are three additional fields that are used only if the driver is marked as version 0x300 or higher. Generally, these entries are used only for display drivers that use a color palette. The fields are as follows:

Field	Description
dpPalColors	The number of palette colors
dpPalReserved	The number of reserved palette registers
dpPalResolut	The color resolution or the number of bits in the palette registers

These fields will be ignored if the **RC_PALETTE** bit is not set in the **dpRaster** field. However, they must be present (and accounted for in the length returned by **Enable()**) if the driver is version 3.0.

The Device Dimension Fields

The last category of GDIINFO entries provides the dimension information. The fields are as follows:

Field	Description
dpHorzSize	The width of the printable area in millimeters.
dpVertSize	The height or length of the printable area in millimeters.
	NOTE The maximum width and length of text that can be printed on a page is determined by choosing either the Portrait or Landscape mode. In Portrait mode, the page is taller than wide, when viewing the text upright. In Landscape mode, the page is wider than tall, when viewing the text upright.
dpHorzRes	The width of the printable area in device units (pixels).
dpVertRes	The height or length of the printable area in device units (raster lines or pixels).
dpBitsPixel	The number of bits required to represent the state of a single pixel in a single graphics plane.
dpPlanes	The number of graphics planes.
	These last two fields are used mostly by bitmapped display drivers. Generally, one value is 1 and the other is the number of bits per pixel, depending on the layout of the display memory. Most printers do not support color, so 1 can be used in both fields. Banding color drivers should use the same values they use for their band bitmaps.

Field	Description
	NOTE Devices choose numbers that match how they output color. To use the color brute functions for dot matrix support, these must match what is used there.
dpAspectX	The horizontal component of the aspect ratio.
dpAspectY	The vertical component of the aspect ratio.
dpAspectXY	The diagonal component of the aspect ratio (along the hypotenuse of a right triangle of sides **dpAspectX** and **dpAspectY**). These last three values represent the device aspect ratio. Since they are used relative to one another, they may be scaled as needed to get accurate integer values. They should be kept under 1000 for numerical stability in GDI calculations. In general, $\mathbf{dpAspectXY}^2 = \mathbf{dpAspectX}^2 + \mathbf{dpAspectY}^2$. For example, a device with a 1:1 aspect ratio (such as a 300 dpi laser printer) can use 100 for **dpAspectX** and **dpAspectY** and 141 (100 * 1.41421...) for **dpAspectXY**.
dpStyleLen	This value specifies the minimum length of a dot in a styled line relative to the **dpAspectXY** value. It is usually 2 * **dpAspectXY.**
dpLogPixelsX	The logical pixels per inch along the horizontal axis.
dpLogPixelsY	The logical pixels per inch along the vertical axis of the page. The adjective "logical" is for displays. For readability reasons, they use a logical inch, which is larger than a real ruler inch. Printer drivers should always use real inches. A 300 dpi laser printer puts 300 in both fields.

The POINT Dimension Fields and Mapping Modes

The remaining dimension fields of the GDIINFO structure (i.e., those with type POINT) are used for scaling coordinates when certain mapping modes are in use. For each mapping mode, there is a *viewport* point and a *window* point.

Place the device resolution in pixels-per-inch in the viewport fields and the number of logical units-per-inch in the window fields. The *y*-coordinate of the viewport is negated to reflect the fact that the *x*-axis ($y = 0$) is along the top of the paper in the default mapping mode (MM_TEXT, which specifies device coordinates) with *y* increasing while going down the page; whereas in the other mapping modes, the *x*-axis ($y = 0$) is along the bottom edge of the page.

There are five mapping modes with which to be concerned:

Mode	Description
MM_LOENGLISH	100 logical points per inch. Specified in the **dpELoWin** and **dpELoVpt** fields.
MM_HIENGLISH	1000 points per inch. Specified in the **dpEHiWin** and **dpEHiVpt** fields.
MM_LOMETRIC	10 points per millimeter or 254 points per inch. Specified in the **dpMLoWin** and **dpMLoVpt** fields.
MM_HIMETRIC	100 points per millimeter or 2540 points per inch. Specified in the **dpMHiWin** and **dpMHiVpt** fields.
MM_TWIPS	1440 points per inch. Specified in the **dpTwpWin** and **dpTwpVpt** fields. (A *twip* is a twentieth of a *point*, which is 1/72nd of an inch.)

For example, on a 300 dpi laser printer, the **MM_TWIPS** mapping mode will require that **dpTwpWin** be set to (1440,1440) and **dpTwpVpt** be set to (300, -300).

After **Enable()** returns GDIINFO to GDI, GDI will use the **dpDEVICEsize** field to allocate a PDEVICE structure and, then, call **Enable()** again to initialize it.

The structure of PDEVICE is entirely up to the device driver writer. This structure will be passed to the driver on all output or information function calls and is the driver's view of the device context. You should store any information about the print job that the driver needs to keep in the PDEVICE structure.

NOTE Static storage in the driver's automatic data segment is undesirable for the following reasons: there may be multiple, independent DCs in existence simultaneously, and it increases the size of the driver even when no DC is in existence.

The only constraint on the PDEVICE structure is that the first WORD contain a non-zero value. A device driver may be called to perform an output operation to a bitmap rather than to the device itself. In this case, a BITMAP structure replaces the PDEVICE structure, with the first WORD = 0. This mechanism is discussed in more detail in Section 5.7, "Performing Output."

Once GDI has successfully loaded the driver and allocated and initialized both the GDIINFO and PDEVICE structures, GDI will return a device context handle to the application. The application may query the DC for metric information or create a print job, by using a mechanism described in Section 5.4, "Print Manager Support."

When the application is finished with the device context that was created by **CreateDC()**, it will call **DeleteDC()** to destroy the device context. GDI will inform the device driver by calling the **Disable()** function, declared as follows:

```
void FAR PASCAL Disable(LPPDEVICE lpPDevice);
```

After this call, the PDEVICE structure will be deallocated. If there are no other DCs in existence using this device driver, the driver DLL (dynamic-link library) will be unloaded from the system.

5.3 The Printer Driver Environment

Printers normally have a large number of options from which the user can select such things as paper size, paper source, or installed font cartridges.

This information can come from any of four sources:

1. The driver's default setup.
2. The driver's WIN.INI section of user options. The WIN.INI should maintain at least one such section so that modified printer setups can be retained from session to session. This information is edited by the driver's Setup dialog box.
3. The driver may call GDI to retain the driver's environment from DC to DC on a port-by-port basis. This allows faster initialization of the driver and avoids the time-consuming process of reading options from the WIN.INI file.
4. The application can pass the environment to the driver in a buffer pointed to by the *lpInitData* parameter of the **Enable()** function.

Upon device initialization (i.e., during the pair of **Enable()** calls), this information is used to set up information in the GDIINFO and PDEVICE structures. For example, the paper size selection will affect the height and width fields. Also, a printer that allows multiple graphics densities will modify the various resolution fields.

5.3.1 The DEVMODE Data Structure

The DEVMODE data structure is used for the environment and the initialization data (which are the same). By convention, all drivers place the device name in the first 32 bytes of DEVMODE as a NULL-terminated string. All the other data is device dependent.

For Windows 3.0, a new convention has been adopted that defines an additional number of fields. These fields allow you to do some device-independent manipulation of the device environment. When **Enable()** is called, the device driver should first check *lpInitData* to see if the application has supplied valid initialization data. If it is valid, then the driver should use that environment, not the default one, to initialize the GDIINFO and PDEVICE structures. The driver should neither use nor modify the default environment information.

5.3.2 The GetEnvironment() Function

If no initialization information is supplied, the driver should check the printer environment maintained by GDI using the **GetEnvironment()** function, which is declared as follows:

```
WORD FAR PASCAL GetEnvironment (
        LPSTR      lpPort,
        LPDEVMODE  lpDevMode,
        WORD       cbDevMode);
```

GetEnvironment() will copy into the DEVMODE buffer (pointed to by *lpDevMode*) the first *cbDevMode* bytes of the environment for the port whose name is specified by the NULL-terminated string pointed to by *lpPort*. The return value is the number of bytes actually copied (which may be less than anticipated), or 0 if there is no environment for the port.

If the environment cannot be found or if the data obtained is invalid or intended for another device, the device driver should extract user settings from the WIN.INI file. To do this, it would use the profile string functions documented in the *Microsoft Windows Software Development Kit*. However, the driver should contain useful defaults for all strings, so that it can create a valid environment even if the WIN.INI file is empty.

The driver should use the device name string at the beginning of the DEVMODE structure to determine whether or not the environment obtained from **GetEnvironment()** is correct.

A driver may also maintain additional information in its DEVMODE structure to determine validity if the device name matches one the driver supports.

5.3.3 The SetEnvironment() Function

If the environment was not found, the driver should set the environment so that future DCs created with the driver can use the environment. This is accomplished with the **SetEnvironment()** call, which is declared as follows:

```
WORD FAR PASCAL SetEnvironment(
LPSTR      lpPort,
LPDEVMODE  lpDevMode,
WORD       cbDevMode);
```

Similar to the **GetEnvironment()** function, the *lpPort* parameter points to the output port for which the environment is being maintained, *lpDevMode* points to the driver data, and *cbDevMode* contains the length. If *cbDevMode* is zero, the environment is deleted entirely. The return value is the number of bytes copied, -1 if the environment is being deleted, or 0 if an error occurs.

The driver should always set up the default environment if it is not present, except when the driver is initialized with a non-default environment (i.e., the *lpInitData* parameter to **Enable()** points to application-supplied data).

5.3.4 The DeviceMode() Function

All printer drivers are required to export a function called **DeviceMode()**, which pops up a dialog box to edit the default environment. This function sets the profile strings in the WIN.INI file for the options chosen by the user. It should also set the environment using the **SetEnvironment()** function.

DeviceMode() is declared as follows:

```
WORD FAR PASCAL DeviceMode(
        HWND    hWnd,
        HANDLE  hInstance,
        LPSTR   lpDevice,
        LPSTR   lpPort);
```

This is the only function ever called directly by an application and the only function that does not involve any GDI or driver data structures such as PDEVICE.

Since the function must pop up a dialog box with which the user can modify printer settings, the application supplies the dialog's parent window with the *hWnd* parameter. The application also supplies the module handle of the driver DLL in the *hInstance* parameter. The *lpDevice* parameter points to the device name (e.g., PCL / HP Laserjet), and *lpPort* points to the output port.

The **DeviceMode()** function needs to use the Windows API for dialog boxes, which is described fully in the *Systems Application Architecture, Common User Access: Advanced Interface Design Guide*. This document describes the suggested appearance and user interface for the Setup dialog box.

The most common way to call **DeviceMode()** is with the Control Panel application. However, other applications that make heavy use of printer output, such as Microsoft Write or Microsoft Excel, may also provide a means for calling the printer driver's **DeviceMode()** function.

5.3.5 The ExtDeviceMode() and DeviceCapabilities() Functions

Newer drivers also export two environment-related functions: **ExtDeviceMode()** and **DeviceCapabilities()**. These two functions are part of the new environment conventions for Windows 3.0, which were designed to allow greater application control over the printer environment.

ExtDeviceMode() allows the application to call the driver to obtain device initialization data either from the user or from the application's modifications to the default environment.

(Craig, don't we need to add something about Device Caps?)

See the *Microsoft Windows Software Development Kit* for complete documentation on the new Windows 3.0 device initialization convention.

5.4 Print Manager Support

A printer driver does not need to manipulate any hardware. Windows handles all the port output for the driver, either directly or through the Print Manager. GDI contains several functions a device driver can call to perform output. The driver does not need to know if output is being queued or written directly to the port.

5.4.1 The OpenJob() Function

To create a print job, a driver calls the **OpenJob()** function, which is defined as follows:

```
HANDLE FAR PASCAL OpenJob(LPSTR lpOutput, LPSTR lpTitle, HDC hdc);
```

The output will be sent to the port or file specified by *lpOutput*. This information is passed to the driver during the **Enable()** call. The *lpTitle* parameter points to an application-supplied title for the document; this title appears in the Print Manager display. The *hdc* parameter is the application's device context, i.e., the GDI handle for the application's print job.

The *lpTitle* and *hdc* parameters are obtained from the application via job control calls made to the driver from the application. This mechanism is described in detail in Section 5.5, "The Control() Function."

The return value from **OpenJob()** is a handle used by the driver to refer to the printer job in other GDI calls.

5.4.2 The StartSpoolPage() and EndSpoolPage() Functions

Printer output is divided into *pages*. Each page is stored in a temporary file on the machine's hard disk when Print Manager is running. Dividing a print job into pages allows Print Manager to begin printing one page while the driver is still generating output on later pages. A Print Manager page does not need to correspond to a physical page of printed output; the division is the driver's decision.

When Print Manager is not running, page division is not very important since temporary files are not involved. However, starting and ending to at least one Print Manager page is still required.

There are two calls for manipulating Print Manager pages:

```
int FAR PASCAL StartSpoolPage(HANDLE hJob);

int FAR PASCAL EndSpoolPage(HANDLE hJob);
```

Calls to **StartSpoolPage()** and **EndSpoolPage()** can occur at any point during the output. Some drivers use one spool page per physical page. Others use one page for the whole job. Notice that the printing of a particular page by the Print Manager application does not begin until it receives the corresponding **EndSpoolPage()** call.

A driver can perform output at any point between these two calls. When **EndSpoolPage()** is called and Print Manager is loaded, the page's temporary file is submitted to the Windows Print Manager.

Both functions return a status code. Positive values indicate success. Negative values indicate that an error has occurred. The defined error conditions are as follows:

Error code	Definition
SP_ERROR	General error condition
SP_APPABORT	The application aborted the job by returning FALSE to GDI from an application-supplied function.
SP_USERABORT	The user deleted the job via the Print Manager application.
SP_OUTOFDISK	There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY	The function failed due to a low memory condition.

These status values are returned by all the Print Manager-supported GDI calls except **OpenJob()**.

5.4.3 The WriteSpool() and WriteDialog() Functions

There are two functions for performing output:

```
int FAR PASCAL WriteSpool(HANDLE hJob, LPSTR lpData, WORD cch);
```

```
int FAR PASCAL WriteDialog(HANDLE hJob, LPSTR lpMsg, WORD cch);
```

WriteSpool() is the function used to output device data to the port or Print Manager temporary file. It must be called after a call to **StartSpoolPage()** and before the corresponding **EndSpoolJob()**.

The *hJob* parameter is a handle returned by **OpenJob()**, *lpData* is a far pointer to the device-dependent data to write, and *cch* is the number of bytes to write.

WriteDialog() is used by a driver to pop up a message box at a certain point in the print job. For example, a driver for a printer using manual paper loading can call **WriteDialog()** to ask the user to place a new sheet in the printer. The print job will not continue printing until the user presses the OK button in the message box. The user may also click a Cancel button to terminate the print job.

5.4.4 The CloseJob() and DeleteJob() Functions

There are two ways to terminate a print job. Normally, a driver calls the **CloseJob()** function,

```
int FAR PASCAL CloseJob(HANDLE hJob);
```

Here, *hJob* is the handle returned by **OpenJob()**. The return value is either positive if no errors occurred or it is one of the error codes defined in Section 5.4.2.

If the driver detects an error condition or is asked to terminate a job by the application (as described in Section 5.5, "The Control() Function"), the driver will call **DeleteJob()**.

```
int FAR PASCAL DeleteJob(HANDLE hJob, WORD wDummy);
```

The *hJob* parameter is the job handle returned from **OpenJob()**. The *wDummy* parameter is currently unused.

The return value is positive if the job is successfully deleted, or is one of the negative status values if an error occurs.

5.5 The Control() Function

The Print Manager functions in GDI are often called when the printing application performs job control actions, such as starting a job or inserting a page break. All of these special functions go through the **Control()** function, which is declared as follows:

short FAR PASCAL Control(LPPDEVICE *lpPDevice*, WORD *nFunction*, LPSTR *lpInData*, LPSTR *lpOutData*);

The *lpPDevice* parameter points to the PDEVICE structure maintained by the device driver, which was initialized during the **Enable()** call.

The *nFunction* parameter contains the index of a device-dependent operation to perform.

The *lpInData* parameter is a far pointer to input data defined by the function specified by *nFunction.*

The *lpOutData* parameter points to a buffer for output data, if required.

The specific subfunctions are often called *escapes*, since the application calls the driver's **Control()** function through the GDI function **Escape()**. Notice, however, that GDI modifies some escapes before calling **Control()**. This chapter will focus on those escapes that are common to most or all printer drivers.

A complete list of all the escapes appears in Chapter 11, "Device Driver Escapes." The driver writer should consult that chapter for details on all the **Control()** subfunctions.

Notice that similar documentation appears in the *Microsoft Windows Software Development Kit* (SDK). However, applications call through the GDI **Escape()** function, which has different parameters, and the behavior of some escapes differs between the driver and the application because of GDI interaction. For this reason, whenever there are any disagreements between the two documents, this document should be used when dealing with the driver's **Control()** function, and the SDK should be used when calling **Escape()** from a Windows application.

The return value from **Control()** depends on the particular escape being called. In general, positive values indicate success, negative values indicate an error, and zero can indicate either an unimplemented escape or a general error condition.

Some escapes that copy data of an indeterminate length to a buffer addressed by *lpOutData* return the number of bytes (or array elements, etc.) copied or, if *lpOutData* is NULL, the size of the data that would be copied if *lpOutData* contained a non-NULL pointer. This would allow the application to query for the size of a buffer to allocate before actually calling the escape.

When an escape uses negative values to indicate an error, the driver should use the predefined SP_* values whenever appropriate, especially when the Print Manager library functions are involved. The **Control()** function should always return zero (i.e., unimplemented) for escapes that are unimplemented or unrecognized.

5.5.1 The QUERYESCSUPPORT Escape

All drivers are required to implement the QUERYESCSUPPORT escape. For this escape, *lpInData* points to a WORD that contains the index of another escape.

The driver must return a positive number if the driver implements that escape, or zero if the escape is unimplemented. The driver always returns non-zero if the escape queried for is QUERYESCSUPPORT.

5.5.2 The SETABORTPROC Escape

SETABORTPROC is the first escape an application calls when actually printing. An application calling the GDI function **Escape()** for the SETABORTPROC escape passes a pointer to a callback function in *lpInData*. This callback function is used to check for user actions such as aborting the print job. The printer driver, however, is not responsible for the callback function; GDI modifies the SETABORTPROC escape so that *lpInData* points to the application's device context handle.

The *hDC* parameter given to the driver by this escape should be used with the **OpenJob()** function to enable the output functions in GDI to call the application's abort procedure. Printer drivers generally save this handle in the PDEVICE structure.

5.5.3 The STARTDOC Escape

Usually, STARTDOC is the next escape an application calls. STARTDOC indicates to GDI and the device driver that the application is actually interested in printing and is not just querying the printer DC for information.

This escape also supplies a Print Manager job title in a NULL-terminated string pointed to by *lpInData*. The *lpOutData* parameter is unused. This supplies the title used by the **OpenJob()** function.

Together with the port name supplied as a parameter to **Enable()** and the *HDC* supplied by the SETABORTPROC escape, the driver now has all the data necessary to call **OpenJob()**.

5.5.4 Raster vs. Vector Devices

From this point, there are two paths the driver can take, depending on the printer technology. Many printers (such as dot matrix and most laser printers) are *raster* printers, i.e., they print out text and graphics as bitmaps or raster lines.

Other devices (such as plotters and POSTSCRIPT-based printers) are *vector* devices, which draw text and graphics as a sequence of vectors or lines. (Although POSTSCRIPT printers are based on raster engines, the language itself is vector oriented except where bitmaps are concerned.)

Raster devices usually have constraints that cause problems for implementing the full GDI device model. Raster devices, for example, do not implement any vector graphics operations. Therefore, all vector graphics must be drawn into a bitmap before they are sent to the printer. Some devices, such as dot matrix printers, do not allow the driver to print anywhere on the page. They require that text and graphics be output in the order of the print direction position on the page.

These bitmaps can be enormous for a device such as a 300 dpi laser printer. In such cases, the driver can break up the page into smaller rectangles that are printed individually. For each of the rectangles, GDI or the application will draw all the graphics that fit in each rectangle into a bitmap and, then, print each individual bitmap.

These rectangles are called *bands*, and the printing process that uses these bands is called *banding*. It is usually necessary to band raster printers; however, banding is not necessary for vector devices.

For vector devices (i.e., non-banding devices), the application calls GDI graphics functions, which are translated into device driver graphics primitives (see Section 5.7, "Performing Output," for further details). After each page, the application uses the NEWFRAME escape to eject the page. NEWFRAME uses neither *lpInData* nor *lpOutData*.

5.5.5 Using Banding Drivers

With banding drivers, there are two possibilities. An application can either treat the driver as if it were a non-banding device by calling the GDI functions and ending each page with the NEWFRAME escape, in which case GDI performs the banding, or it can handle the banding itself.

The NEXTBAND Escape

In either case, the view from the driver is similar. Before any graphics are drawn, the driver is called upon to perform the NEXTBAND escape. When **Control()** is called for the NEXTBAND escape, *lpInData* points to a POINT structure, and *lpOutData* points to a

RECT structure. The driver should initialize its band bitmap and set the RECT structure to the size of the rectangle in device coordinates that the band represents on the page.

The POINT structure is added by GDI to determine the *scaling factor* for graphics output. Some devices support the use of graphics at a lower resolution than text to allow for faster output. The *x*-coordinate of the POINT corresponds to horizontal scaling and the *y*-coordinate to vertical scaling.

The value in the structure corresponds to a shift count. A point of (0,0) specifies graphics at the same density as text, whereas a point of (1,1) specifies half-density graphics in both directions, e.g., a 300 dpi laser printer printing bitmaps at 150 dpi.

The driver's output function is then called to perform output into the band bitmap. When all the output for the band is finished, the driver is called for another NEXTBAND escape. The driver outputs the band in the band bitmap, reinitializes the bitmap, sets a new rectangle, and continues with the next band as it did with the first.

When all the bands on the page are exhausted, and the driver receives a NEXTBAND escape, it should output the last graphics band and, then, set the rectangle pointed to by *lpOutData* to (0,0,0,0) to indicate that there are no more bands on the page. It should also perform all the processing necessary to eject the completed page. The next NEXTBAND escape will correspond to the first band of the next page.

If the application performs banding, it will call **Escape()** to get the band rectangles. If GDI is handling banding on behalf of an application, then GDI collects all the graphics calls on a page into a *metafile*, i.e., a temporary file containing a list of the graphics calls and their parameters. When the application calls **Escape()** to perform the NEWFRAME escape, GDI turns this escape into a sequence of NEXTBAND calls to **Control()**. GDI sets the clip region for the actual printer DC to the band rectangle and, then, plays back the metafile, which recreates all of the application's output in the band bitmap. GDI does this for each band until the band rectangle returned by the driver is empty.

Some devices, such as raster laser printers, allow text to be placed anywhere on the page at any time. Furthermore, these printers do not place text into the band bitmap, since all the device fonts exist in printer or cartridge memory. To optimize text output, their drivers use a single, full-page band for all the text output and a sequence of smaller bands for bitmapped graphics.

As an optimization, some of these drivers maintain a flag to detect whether or not any output, other than text, is attempted during the first, full-page band. If not, the driver skips the graphics bands.

The BANDINFO Escape

Some devices, such as laser printers, can print text and graphics anywhere on the page but still require banding support for vector graphics operations. Since these devices usually use their own internal device fonts, they can greatly improve their text printing performance by using a single, full-page band for text only as the first band. The driver ignores graphics calls during this band and handles only **ExtTextOut()** or **StrBlt()** calls. Graphics are printed on subsequent, smaller bands.

An application that is aware of this process can speed up its printing operation by determining whether text or graphics will be printed on the current band. It may do so using the BANDINFO escape. The application can also use BANDINFO to optimize the banding process.

For the BANDINFO escape, both *lpInData* and *lpOutData* point to the BANDINFOSTRUCT structure:

```
typedef _bandinfostruct {
        BOOL fGraphics;
        BOOL fText;
        RECT rcGraphics;
        } BANDINFOSTRUCT;
```

The application calls BANDINFO immediately after NEXTBAND. If *lpOutData* is non-NULL and graphics will be printed in the current band, the driver will set the **fGraphics** flag in the output structure. If text will be printed, the **fText** flag will be non-zero. The **rcGraphics** flag is not used for output.

Therefore, on the first band, the driver would set the rectangle returned by NEXTBAND to the whole page. If it receives a BANDINFO escape, it will set **fText** and clear **fGraphics.**

On subsequent bands, it will band the page in small rectangles, handle only graphics calls, and, if the application calls BANDINFO, clear **fText** and set **fGraphics.**

The application can also optimize the banding process somewhat by describing the page with the structure passed by *lpInData.* The application sets the **fGraphics** flag, if there are any graphics on the page, and the **fText** flag if there is any text. If there are no graphics, the driver may be able to skip the graphics bands. The application should also set **rcGraphics** to the rectangle bounding all non-text graphics on the page. The driver has the option of banding only the specified graphics rectangle rather than the whole page.

Vector fonts complicate the process somewhat. Since vector devices using banding generally cannot print vector fonts, these fonts are simulated using polylines or scanlines. Therefore, they appear to the driver to be graphics in the text band. Since vector fonts can appear anywhere on the page and require graphics banding support, the driver must band graphics on the whole page even if the BANDINFOSTRUCT passed by the application specifies otherwise.

If the application never calls BANDINFO, the driver can decide whether or not to band graphics by maintaining a flag that is set if any graphics calls are seen during the text band. See Chapter 11, "Device Driver Escapes," for complete documentation on the escapes.

5.5.6 The ENDDOC and ABORTDOC Escapes

When an application has completed all output, it calls the ENDDOC escape. ENDDOC does not use either *lpInData* or *lpOutData.* At this point, the driver may call **CloseJob()**.

Another common escape is ABORTDOC, which is also called ABORTPIC in older documentation or applications and has the same number assigned. This escape allows GDI or the application to abort a print job. Generally, if the job is valid, the driver will clean up and call **DeleteJob()**.

5.5.7 Final Notes on Escapes

Few, if any, applications use QUERYESCSUPPORT to look for SETABORTPROC, STARTDOC, NEWFRAME, ENDDOC, or ABORTDOC. Therefore, a printer driver should handle all of these escapes.

In addition, there are a few applications that perform banding without verifying that banding is required either by using QUERYESCSUPPORT or the **GetDeviceCaps()** function (which examines the GDIINFO structure). A non-banding driver can easily support such an application by returning the full page as the band rectangle on the first NEXTBAND call and returning an empty rectangle for the next NEXTBAND call and ejecting the page.

There are a large number of other escapes that may or may not be appropriate to a specific driver. They are all listed alphabetically and described in detail in Chapter 11, "Device Driver Escapes."

5.6 GDI Graphics Objects

Applications use the GDI graphics objects to perform drawing operations. The objects that a device driver needs to be aware of are the following:

- Pens
- Brushes
- Fonts

5.6.1 Logical and Physical Objects

Applications create *logical* objects that are device-independent forms of graphics tools. *Physical* objects are device-dependent representations of the same tools that are passed to the driver to perform output. The process of converting a logical object into a physical object is called *realizing* the object. A physical object may be used many times for drawing. The realization process is used to eliminate the overhead of converting a logical to a physical object at the time of output.

Applications use the **SelectObject()** function to select logical objects into a device context for use in drawing.

GDI and the driver convert the logical object into a physical object. GDI first queries the driver for the size of a given physical object and, then, allocates storage for it and calls the driver to perform the realization.

The contents and organization of the data structure defining a physical object are completely up to the driver writer. Usually, the data structure for the physical object includes the logical object plus some other information that the driver needs.

5.6.2 Device Objects

Most drivers will also maintain and manipulate *device* objects. Device objects are data structures that represent a graphic primitive that the device supports very well.

The most common type of device object is a *device font.* Most printers are capable of printing some set of built-in fonts. The concept of device fonts enables drivers and applications to take advantage of a device's ability to render fonts. Device fonts are also expected to produce better results (print faster and look better) than GDI fonts.

However, drivers may also wish to support GDI raster and vector fonts. For banding devices, it is usually not difficult to support GDI raster fonts because the brute functions for dot-matrix support may be used to render these fonts into the banding bitmap. GDI raster fonts are only useful for devices (such as lower-resolution dot matrix printers) with resolutions near those of the display.

For non-banding devices, supporting GDI raster fonts is not as easy. In fact, the POSTSCRIPT driver (a non-banding device) does not support GDI raster fonts.

Supporting vector fonts is also optional. If a driver does not support vector fonts, GDI will simulate them by drawing line segments.

5.6.3 The GDI Information Functions

The functions discussed in this section are used to provide GDI and applications with information about the driver and the device. They do the following:

- Realize objects
- Enumerate objects
- Translate logical and physical colors
- Determine character widths

The RealizeObject() Function

Physical objects are created to avoid the overhead of translating logical objects to physical ones at drawing time. A realized object can be used to perform many drawing operations. Therefore, the **RealizeObject()** function should produce physical objects that can be used with minimal overhead by the output functions. **RealizeObject()**, not the output functions, should perform any operations (such as translating or converting) that would slow down the output functions.

The EnumDFonts() Function

EnumDFonts() is used to enumerate all the device fonts. It is passed a face name that refers to the device font to enumerate.

The first call to **EnumDFonts()** passes a NULL face name. This indicates that the driver should enumerate each face name that it supports. Subsequent calls will pass in one of these face names. The driver should then enumerate all the sizes of that font.

The EnumObj() Function

EnumObj() is used to enumerate all the pens and brushes that a device supports. The enumeration process communicates logical descriptions of objects to the application. All these objects must be unique, i.e., when translated into physical objects and used for drawing, they should produce different output.

All the styles and colors of pens and brushes should be enumerated. Since pens are defined to be only pure colors, only logical colors that will translate to pure physical colors should be enumerated. For devices that support many colors (e.g., 8-bit displays), only a subset of all the colors should be enumerated.

The ColorInfo() Function

ColorInfo() is used to translate physical and logical color representations. It will translate in both directions, i.e., from physical to logical and from logical to physical. When given a logical color, the nearest physical color should be returned. When given a physical color, the logical color that best describes that color should be returned. This function is used to support the GDI function **GetNearestColor()**.

The GetCharWidth() Function

GetCharWidth() is used to determine character widths for variable width fonts. It is important that the values returned by this function match and that the actual widths be used when displaying characters on the display surface. Any differences will produce misalignments, and any text formatting or justification will not work as intended.

5.6.4 The GDI Information Brute Functions

Since output must be supported to both the display surface and memory bitmaps, each of the preceding functions should check to see if the PDEVICE passed indicates that a memory bitmap, rather than the actual device, is referenced. If this is the case, the equivalent *brute* function (for dot-matrix support) should be called, passing on all the parameters. The following brute functions take the same parameters as the corresponding driver functions:

- **dmRealizeObject()**
- **dmEnumDFonts()**

- **dmEnumObject()**
- **dmColorInfo()**
- **dmGetCharWidth()**

Notice, however, that **dmEnumDFonts()** always returns 1. Therefore, it need not be called, and you can simply return 1 when the output device is a memory bitmap.

5.7 Performing Output

There is a relatively small set of functions used for performing output. These functions take as parameters a destination device, parameters that control the output, and physical objects that have been realized previously.

How the output functions are implemented depends on whether or not the device uses banding. Banding devices have their output stored in a metafile. This metafile is replayed for every band that is rendered (either by GDI or applications that wish to implement banding). Therefore, output coordinates must be mapped into the current band, and output outside of the band must be clipped.

Non-banding devices perform output to the device in one pass. Therefore, the device must have access to the entire display surface. Drivers must be able to perform all the output functions to both the display surface and to memory bitmaps. This restriction would make it very difficult for devices that supported complex drawing primitives if it were not for the help that GDI and the display driver supply.

5.7.1 The GDI Output Brute Functions

There are also *brute* functions that match all the output functions (and all the object and enumeration calls) except **FastBorder()** and **ExtTextOut()**. For banding devices, this means that the output functions do little more than detect where output is going (to a memory bitmap or to the current band bitmap), translate the output coordinates into band coordinates and, then, call the brute functions. Only **ExtTextOut()** may require a little more code to break character drawing up into **StrBlt()** calls. (See Section 2.6, "The StrBlt/ExtTextOut Functions," for more detailed information on these requirements.)

The following output brute functions take the same parameters as their corresponding driver functions:

- **dmBitBlt()**
- **dmOutput()**
- **dmPixel()**
- **dmStrBlt()**
- **dmScanLR()**

The brute functions are used extensively by banding devices to build their bands' bitmaps. However, non-banding devices must also use the brute functions when the destination is a memory bitmap. Notice that the brute functions only support monochrome devices (1 bit-per-pixel, 1-plane bitmaps). If the driver is not monochrome, it must either use the supplied color brute function library or implement the brute functions itself.

5.7.2 The GDI Color Library

The dot-matrix (brute) library functions in GDI, such as **dmBitBlt()** and **dmOutput()**, are written for monochrome printers and do not support color. To simplify the writing of color drivers, source code for a color version of the required functions is supplied with the DDK.

This color library implements color versions of all the **dm*()** functions except **dmTranspose()**, which does not depend on color format. The arguments and return values of these functions are the same as those for the GDI monochrome versions of these functions.

The library implements color using a 3-plane RGB (Red, Green, Blue) banding bitmap, which is converted to CMY (Cyan, Magenta, Yellow) when the bitmap is sent to the printer. If your printer requires a different format, you will need to modify the supplied sources for your driver.

A sample printer driver is supplied for the IBM Personal Computer Color Printer. This driver is similar in structure to the sample Epson driver and has several files in common with it.

If you are modifying or upgrading a driver written for Windows 2.x to support Windows 3.0, you should be aware of modifications made to these libraries for protected-mode memory-management support. Both **dmBitBlt()** and **dmOutput()** compile short, efficient functions into an automatic variable and, then, call them to perform the actual operation. In protected mode, this requires creating a CS alias for the stack segment. (See Section 5.9, "Updating 2.x Printer Drivers to 3.0," for more information on memory management and code and data segment mixing.) This is performed at the beginning of the two affected functions.

A selector must be allocated for these two functions to operate. It is stored in the global **ScratchSelector**, which is external to the library and which must be supplied by the driver. In the sample IBM PC Color Printer driver, it is allocated and freed in **Enable()** and **Disable()**, respectively. These functions appear in the file RESET.C. The code for the affected library functions appears in BITBLT.ASM and OUTPUT.ASM.

5.7.3 The GDI dmTranspose() Function

The GDI library of dot-matrix supporting functions also provides a function for transposing bits in a bitmap. The **dmTranspose** (*lpSrc*, *lpDst*, *WidthBytes*) function assumes, however, that neither the source nor the destination bitmap exceeds 64K. The source and destination bitmaps must also be disjoint; **dmTranspose()** does not transpose in place.

(CraigC, please review this next paragraph's first sentence carefully. I changed it around from what was in the old DDK and so I'm nervous about its accuracy.)

This function copies the *WidthBytes* number eight times (i.e., it copies eight scanlines) from the source pointer (the string to be copied and transposed) to the *destination* pointer (the buffer that holds the transposed string), transposing bits as it copies. The eight most significant bits, one from each of the eight scanlines, make up the first byte of the transposed line; the eight next most significant bits make up the next byte, and so on.

The *WidthBytes* parameter is a short integer specifying the number of bits in each scanline. Normally, the order in which bits from the eight scanlines are packed into a byte is the same as the order of the scanlines. This order can be reversed by giving a negative value for *WidthBytes*. When *WidthBytes* is negative, the most significant bit from the first scanline becomes the eighth bit in the first byte of the transposed line, the most significant bit from the second scanline becomes the seventh bit, and so on.

5.7.4 The GDI Priority Queue Functions

The GDI library provides the priority queue data type that is used with device-specific fonts to sort output strings into the correct order on the page. Priority queues are accessed through a two-byte value, known as the "key." Each key can also have two bytes of information, called a "tag," associated with it.

The following list provides brief descriptions of each of the priority queue functions.

Function	Description
CreatePQ(*size*):*hPQ*	Creates a priority queue. Size, a short integer value, is the maximum number of items to be inserted into this priority queue. *hPQ* is a handle to the priority queue if the function is successful. Otherwise, *hPQ* is zero.
MinPQ(*hPQ*):*tag*	Returns the *tag* associated with the key having the smallest value in the priority queue, without removing this element from the queue.
ExtractPQ(*hPQ*):*tag*	Returns the *tag* associated with the key having the smallest value in the priority queue and removes the key from the queue.
InsertPQ(*hPQ, tag, key*):*Result*	Inserts the *key* and its associated tag into the priority queue. *Result*, a short integer value, is TRUE if the insertion is successful. Otherwise, it is ERROR(-1).

Function	Description
SizePQ(*hPQ, sizechange*):*Result*	Increases or decreases the size of the priority queue. *Sizechange* **(Craig, the old book showed newsize in the description but size-change in the syntax line?)** is a short integer value specifying the number of entries to be added or removed. *Result*, also a short integer value, is the number of entries that can be accommodated by the resized priority queue. *Result* is ERROR(-1) if the resulting size is smaller than the actual number of elements in the priority queue.
DeletePQ(*hPQ*):*Result*	Deletes a priority queue. *Result*, a short integer value, is TRUE if the queue is deleted. Otherwise, it is ERROR(-1).

5.7.5 Interpreting GDI Coordinates

Drivers must be careful to interpret properly the coordinates that GDI passes to the output functions. GDI uses an upper-left inclusive and lower-right exclusive rectangle. Therefore, the rectangle coordinates (0,0,3,3) imply the type of fill shown in Figure 5.1.

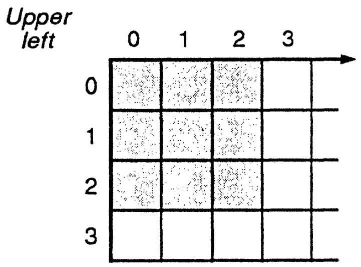


Figure 5.1 GDI Coordinate Mapping (prnt_01)

What this means is that the upper-left coordinates of a rectangle map to a device pixel into which the driver should draw. The lower-right coordinates are not drawn into. This concept is important to understand to avoid the problems created by being "off by one."

5.7.6 Output Functions Summary

Detailed descriptions of the following output functions appear in Chapter 10, "Common Functions ."

- **Output()**

- **BitBlt()**
- **StrBlt()**
- **ExtTextOut()**

The display driver descriptions of these may also be useful for non-banding devices that require a more detailed understanding of the semantics of these functions. **FastBorder()** need not be implemented for printers. It is intended only for displays.

Printers with special bitmap-manipulating abilities should support the new Windows 3.0 DIBs functions. (See Section 3.4, "Device-Independent Bitmaps," for a description of the functions.)

5.8 Stub Functions

Since printer drivers are dynamic-link libraries (DLLs) that GDI loads via the **LoadLibrary()** function, they must also export the termination function called **WEP(***bSystemExit***)**, or Windows Exit Procedure, to accommodate the support of DLLs. This function indicates whether all of Windows is shutting down or just the single DLL. More detailed descriptions are provided in Chapter 10, "Common Functions," and in the SDK's *Guide to Programming*.

Printer drivers must also include the same two stub functions (i.e., **SetAttribute** and **DeviceBitmap**) for which sample code is provided in Chapter 2, "Display Drivers." Simply copy verbatim the code reproduced there.

5.9 Updating 2.x Printer Drivers to 3.0

Windows 3.0 offers several enhancements that affect printer drivers. The following are those with the greatest effect and will be discussed in this section:

- Memory management
- New device initialization conventions
- New driver interface functions

5.9.1 Memory Management

Memory management significantly affects printer drivers that are being updated. Drivers that do not support the new Windows 3.0 printer driver features will still run with Windows 3.0, although they will not be as functional as fully updated drivers. However, drivers that are *unaware* of the new Windows 3.0 protected-mode memory-management requirements will probably fail to operate at all.

Windows 3.0, in both standard and enhanced mode, takes advantage of protected-mode memory management. This memory management is less tolerant of the following kinds of driver bugs or practices:

- References to NULL pointers
- Reads and writes past the allocated length of segments
- Mixing code and writable data in the same segment
- Huge pointer arithmetic based on the overlapping segment architecture of the 8086 processor

NULL Pointer References

The printer driver sample sources shipped in the Windows 2.1 DDK passed, as the default string, NULL pointers into the Kernel function **GetProfileString()**. This is improper behavior that will cause a General Protection (GP) fault in Windows 3.0.

The correct practice is to pass an empty string (i.e., a far pointer to a zero byte). The sample sources for the Windows 3.0 DDK have been updated to reflect this.

Segment Overrun Problems

(Craig: please check the wording here.)

Another problem with the Windows 2.1 DDK sample sources was a segment-overrun bug. Banding raster printer drivers, such as the Epson and IBM graphic printer drivers, use the GDI function **dmTranspose()** when converting the band bitmap to the printer's native command language. The **dmTranspose()** function always converts eight rows at a time. Therefore, drivers that output seven rows at a time generally do the following:

- Transpose eight rows
- Set the most significant bit to 1 or zero
- Output that data
- Then skip seven rows in the bitmap

That way, the next first row will be the last row that the previous pass ignored.

This presents a problem for the last row. If a row has a (hypothetical) width of 100-bytes, which corresponds to 800 pixels, then **dmTranspose()** will access 800 bytes to convert eight rows. However, the last row will leave only 700 bytes in the segment. In protected mode, the first access made by **dmTranspose()** past the 700-byte limit will result in a GP fault. The simplest solution is to add a row of "slop" bytes to the end of the band bitmap, allowing **dmTranspose()** to function as expected.

Code and Data Segment Mixing

There is an additional problem that usually is only an issue for display drivers, but that also affects color raster printer drivers (such as the sample IBMCOLOR driver). Since the dot-matrix library (also referred to as the "brute" functions) in GDI supports only monochrome bitmaps, color drivers generally include their own version of these functions, which are extended to support their particular bitmap format. These functions are generally derived from the "color dot-matrix library" supplied with the DDK.

The **BitBlt()** and **Output()** functions work by compiling a short function that efficiently implements the desired operation. These functions are written into arrays allocated on the stack. To compile the function, it must appear in a writable data segment (which the stack always is). However, to actually call the function, it must be in a code segment (which the stack never is). Thus, **BitBlt()** and **Output()** will cause a GP fault when attempting to call the compiled function.

The solution is to create a *CS alias*. Normally, only one selector is used to refer to a region of memory. However, if the descriptor for another selector contains the same linear address and length as the original, accessing either selector will refer to the same linear address. Then, if the new selector is marked as a code segment, a program can call a function in that segment.

Windows device drivers can call **AllocSelector()** to get an uninitialized selector and **PrestoChangoSelector()** to copy the base and length and the modified attributes to the new selector. This is demonstrated in the color library for the Windows 3.0 DDK.

If a printer driver is allocating read/write variables in a code segment, it will encounter the same problem since the code segment is no longer writable. In this case, the driver is probably better off placing the variables in its default data segment. This is because the Windows KERNEL always adjusts far-call entry prologs to point to the correct segment. It is also generally considered poor programming practice to use a CS alias to access code segment variables.

Huge Pointer Arithmetic and Selector Tiling

None of the printer drivers shipped as DDK sources has ever used segment arithmetic for accessing objects greater than 64K at a time. However, if you are updating a driver that does so, you will need to address this issue.

Windows 3.0 uses selector tiling to allow drivers (and applications) to allocate objects larger than 64K and access them with 16-bit offsets. In short, Windows will allocate multiple selectors (all 64K, if necessary, except possibly the last one) and use selectors separated by some fixed amount, represented by the variable **__ahincr**. In real mode, **__ahincr** is set to 1000H, allowing the same code to execute in any memory configuration.

Selector tiling is explained in more detail in the *Microsoft Windows Software Development Kit.*

5.9.2 Device Initialization Conventions

A new device initialization convention adopted for Windows 3.0 printer drivers allows applications much greater control over printer drivers. All the sample printer drivers included with the DDK implement the new device initialization convention. This convention is explained in detail in the *Microsoft Windows Software Development Kit.*

The convention adds to a printer driver the following two new functions:

- **ExtDeviceMode()**
- **DeviceCapabilities()**

ExtDeviceMode() enables applications to obtain device initialization data, by optionally prompting the user or making device-independent modifications to the initialization data, which can then be passed to the **CreateDC()** function. GDI then calls the **Enable()** function with a pointer to this information, allowing the driver to preset its GDIINFO and PDEVICE structures according to the application's options, rather than the defaults. That way, the application can store different printer settings for itself and its documents or even request specific setup properties, such as orientation.

(CraigC, don't we need to explain DeviceCapabilities() here?!)

5.9.3 Driver Interface Functions

There are a number of new functions that a driver can export for GDI's benefit. They are used to support the new Device-Independent Bitmap (DIB) format or to improve performance on certain GDI operations, such as **StretchBlt()**. The functions include the following:

- **StretchBlt()**
- **SetDIBitsToDevice()**

(Chrisg, can you give me a list of the others?)

5.10 Checklist for Printer Drivers

The following checklist is a summary of the points made in this chapter along with some additional general information for updating all device drivers.

❑ All Windows 3.0 drivers have to be bimodal, i.e., they have to run in protected mode as well as 8086 real address mode. To do so, you may need to import some of these functions from KERNEL:

  ☐ **AllocSelector()** (@175) — to get a selector

  ☐ **FreeSelector(***wSel***)** (@176) — to free a selector

- ☐ **PrestoChangoSelector**(*wSrcSel, wDestSel*) (@177) — for code<—>data selector conversion
- ☐ **AllocCSToDSAlias**(*wSel*) (@170) — to get a data alias of the code selector
- ☐ **AllocDSToCSAlias**(*wSel*) (@171)— to get a code alias of the data selector
- ☐ **__AHIncr** (@114) — to do selector huge increments
- ☐ **LongPtrAdd** (@180) — to do "segment" arithmetic

❑ To avoid general protection faults, *do not* do any of the following:

- ☐ Access (read or write) an array beyond its limits.
- ☐ Have an offset wrap-around (going from 0FFFFH to 0 using a string instruction).
- ☐ Load an invalid selector into a segment register.
- ☐ Update code segment variables.
- ☐ Do segment arithmetic (except as described) for selector registers.
- ☐ Compare segment (selector) registers to see which is lower in memory.
- ☐ Do CLIs and STIs.
- ☐ Use undocumented MS-DOS calls. You should use Windows calls whenever possible.

Chapter 6 Network Support

This chapter provides an overview of network support in Microsoft Windows version 3.0 and contains descriptions of the following areas:

- New benefits provided to network users
- Issues that may cause incompatibilities between Windows and networks
- How network vendors can make their software work well with Windows 3.0
- Which networks are supported in Windows 3.0 and how such pieces are distributed

6.1 New Features

The following subsections detail the new support that Windows 3.0 offers to network users.

Some basic restrictions still apply:

- The user must start the network before starting Windows.
- There is no support for workstations running Windows while also acting as servers.

6.1.1 Alleviating the Memory Crunch

The most pressing problem for Windows 2.x has been memory. Windows has always been hampered by the 640K barrier, and loading network software (typically between 50 and 150 KB) often reduces available memory drastically. Often, users find they need to choose between running large applications and running their network.

Windows 3.0 solves this in the most dramatic way possible. Instead of gradually decreasing memory requirements, the 640K barrier was removed altogether. Both the standard and enhanced modes for Windows now run in protected mode on the appropriate hardware, allowing Windows and Windows applications to directly access all the memory on the system.

This by itself should solve 80 per cent of our customers' problems. However, protected mode also brings along some new compatibility issues, which are discussed in Section 6.3, "Compatibility Issues and Solutions."

6.1.2 Adding and Deleting Network Connections

Previous versions of Windows made it inconvenient for the user to change network connections. Under Windows/286, users had to start up a non-Windows application (such as USE.EXE) to make the connection. Under Windows/386, we recommended that the user not even attempt to change connections.

Under Windows 3.0, the user can:

- Connect and disconnect network drives from File Manager.
- Connect and disconnect network printers from Control Panel.

Both processes are integrated into the standard Windows user interface and do not require spawning off any network utilities. They are actually performed using calls to the network driver (described in Section 6.2.1, "Windows User Interface: The Windows Network Driver"). See Figure 6.1 for an example of the Connect Network Drive dialog box.

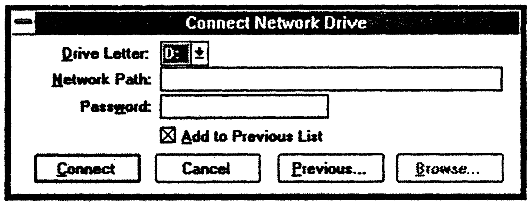


Figure 6.1 Creating a net connection from File Manager

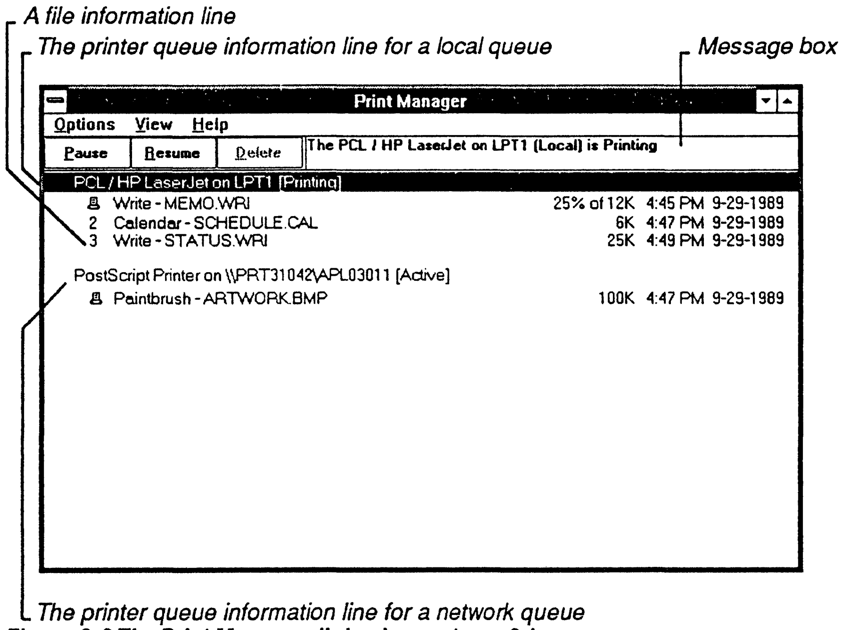
On some networks, the user can browse through a list of the available network resources and select ones to connect to. On all networks, the user can save a list of commonly used connections and reconnect to them with a minimum number of keystrokes.

6.1.3 Network Printing

The Windows Print Manager (formerly known as the Spooler) can recognize networks. It handles local printers much as it did for version 2.x, but it now recognizes network printers and manipulates them using the network driver. (See Figure 6.2 for an example of a Print Manager dialog box.) This allows the following advantages.

- Faster printing to network printers
- The ability to view the contents and status of network print queues, including detailed information on each job
- The ability to pause, resume, and delete your own jobs waiting in network print queues

- Accurate and detailed network error messages


A file information line
The printer queue information line for a local queue
Message box
Print Manager
Options View Help
Pause Resume Delete
The printer queue information line for a network queue

```
The PCL / HP LaserJet on LPT1 (Local) is Printing

PCL / HP LaserJet on LPT1 [Printing]
  Write - MEMO.WRI                      25% of 12K  4:45 PM 9-29-1989
2 Calendar - SCHEDULE.CAL                       6K  4:47 PM 9-29-1989
3 Write - STATUS.WRI                           25K  4:49 PM 9-29-1989

PostScript Printer on \\PRT31042\APL03011 [Active]
  Paintbrush - ARTWORK.BMP                    100K  4:47 PM 9-29-1989
```

Figure 6.2 The Print Manager dialog box-netsup_2.img.

6.1.4 Network Error Messages

In many cases, Windows can now give more accurate error messages when there are network errors. This is most noticeable in Print Manager, but is also seen under conditions such as network access violations and severe network errors.

6.1.5 Network-Specific Dialog Functions

Control Panel allows the user to invoke a network-specific dialog box, which can provide access to any additional features the network supports. For example, the dialog box could allow the user to log in under a new name or send messages to another user, if the network supports these operations.

The dialog box is provided by the network driver, and its design is completely up to the developer. However, the design should follow the guidelines described in the *Systems Application Architecture, Common User Access: Advanced Interface Design Guide*, and Microsoft should be allowed to review these designs before the drivers are distributed. If this dialog box offers the user any options, some mechanism should be included to enable the user to access on-line help. How to use the Windows Help Engine is described in the *Microsoft Windows Software Development Kit.* **(Greg, Still True?)**

It is also the place where the driver version, copyright, and other information can be displayed. We strongly recommend that this dialog box provide some way for the user to learn the name of the network and the version numbers of both the network software and the Windows network driver.

6.1.6 Running Windows From a Network Drive

Many corporations are choosing to put a single copy of Windows on the network, rather than putting separate copies on every workstation. This is much easier for MIS departments to maintain, upgrade, and control.

Under Windows 2.x, this was not feasible, since the Windows files were linked for particular hardware during Setup. Drivers and fonts were bound together into a few large binary files (i.e., WIN200.BIN and WIN200.OVL), and the entire Setup process had to be repeated to change the configuration.

In Windows 3.0, we have attempted to remove these barriers. All the necessary drivers and fonts are left as separate files on the user's disk (or the network Windows directory). The ones to be used are specified in the user's SYSTEM.INI file, and changing configurations consists of nothing more than copying the proper files to the user's disk (if not already there) and modifying entries in their .INI files. **(Greg, still true?)**

The advantage for networks is that a single Windows directory can contain files for more than one configuration. Each file is left unmodified and with its original name. A network administrator can install all the files their users require, and individual users need only have their own personal directory containing .INI and temporary files. The users can modify their own .INI file (manually or using Setup) without affecting others using the same software. This is possible because Setup has a special option to create a personal directory that references the common Windows directory on a network.

6.1.7 Supporting Large Numbers of Outstanding NCBs

Previous versions of Windows/386 imposed a limit on the number of Network Control Blocks (NCBs) that could be outstanding. However, this limit was too small for some network-intensive applications. The NETBIOS device for enhanced Windows 3.0 now supports up to 4K of NCBs, which should be adequate for most situations.

6.2 Attaining Compatibility

With previous versions of Windows/386, many network vendors had to modify their network software to be "Windows/386 aware" for the two products to coexist at all.

For the most part, that is still true with version 3.0. However, we now have new alternatives and new problems that need to be confronted. Windows running in enhanced mode has become much easier to support, and Windows running in standard or real mode has become much harder.

6.2.1 Windows User Interface: The Windows Network Driver

As described in Section 6.1, "New Features," the Windows Control Panel, File Manager, and Print Manager provide the user with access to network functions. To enable these to work under many different networks (each with its own capabilities and API), we have designed a Windows network driver. Just as a printer driver takes high-level calls designed for a mythical "super printer" and maps them to calls understood by one specific printer, the network driver takes a standard set of calls and translates them to the API of one specific network.

The standard Windows package contains several drivers to support popular networks. Microsoft provides support for Microsoft LAN Manager and a generic driver for NETBIOS and Microsoft Network-based networks. Most networks based on these products can use the standard drivers provided by Microsoft. However, network vendors can also write custom drivers to take advantage of their own features and extensions.

For each network, a single driver should be used for Windows in both real and protected modes. The driver may also serve as a convenient place to put other code unrelated to the functions it provides to Windows. Common examples include the following:

- Code that uses INT 31H to map APIs between protected and real modes (as described in Section 6.3.4, "The Problem of Protected Mode API").
- Entry points for high-level functions provided for network-specific Windows applications.
- Code to hook incoming network messages and display then in a pop-up dialog box for the user.

See Chapter 7, "Network Drivers," for more information on developing Windows network drivers.

6.2.2 Enhanced Windows 3.0: Virtual Device Architecture

In previous versions of Windows/386, it was difficult to add support for new configurations. It required the developer to modify code entwined throughout the Windows/386 core software and build a customized version of the product. The changes made by one developer were incompatible with additions made by other third parties.

For version 3.0, the entire system has been redesigned. Support for new hardware or software configurations can be added in modular format, as self-contained "virtual devices," which can be developed independently. Each virtual device is a self-contained program module that can be distributed and installed separately, can be loaded at runtime as needed, and can cooperate with the other virtual devices in a harmonious way.

Windows 3.0 also includes two network-related virtual devices: VNETBIOS (for NET-BIOS support) and DOSNET (for Microsoft Networks). Either may be modified, extended, or replaced to fit your network's needs. Notice, however, that the DOSNET device provides services that are also used by other virtual devices. Therefore, it cannot be completely removed; you must leave at least a stub to provide those services. See the DOSNET source code for more details.

Volume 2 of the *Microsoft Windows Device Development Kit* contains more detailed information on creating enhanced Windows virtual devices.

6.2.3 Standard Windows3.0: The DOS Extender's Domain

For Windows 3.0 running in standard mode, the protected-mode operation not only offers great benefits but also causes new compatibility problems.

Windows running in standard or real mode does not provide the same sophisticated, extensible environment that enhanced Windows does. Luckily, its needs are also not as great because it does not multitask non-Windows applications. Most hardware devices will not require changes to the DOS Extender; for example, displays are handled by the Windows "grabber," and support for all major mice is already built into the standard DOS Extender.

However, some problems still exist. Just as with Windows 2.x, the problem of asynchronous events still arises when switching between non-Windows applications, and the problem of the protected-mode API occurs for Windows applications calling the network. (Both of these problems are discussed in more detail in Section 6.3, "Compatibility Issues and Solutions.") The latter can be handled by placing code in the network driver, but the former may require changes to the DOS Extender itself.

The DOS Extender is an independent executable file distributed with Windows.

It takes care of changing the processor state between real and protected modes, mapping APIs, and virtualizing system components that require global supervision.

Just as with Windows/386 2.x, support for these types of devices must be integrated into the body of the DOS Extender. That is, a new version of the DOS Extender must be created to support a new configuration. The standard version supplied by Microsoft will support the most common configurations, but specialized adaptations will have to be done by third-party vendors. Some network vendors may have to do this work. To decide whether or not it is needed for specific cases will require reading the detailed descriptions in this chapter and in Chapter 7, "Network Drivers," as well as thorough testing.

One unfortunate limitation of this design should be noted: it will be difficult to combine the adaptations done by two third-party vendors. That is, if a network vendor creates a DOS Extender to support their software, and another company creates a DOS Extender to support their hardware, there will be no way to run the two at the same time. Someone will have to build a merged version of the two products, or any users running with both needs will be unable to run protected-mode Windows. Fortunately, few configurations will actually require modifying the DOS Extender, so conflicts of this sort should be rare.

6.2.4 Testing Compatibility

The DDK's *Installation and Update Notes* includes guidelines for third parties to follow to make sure their products work with Windows 3.0. These include detailed testing procedures to help track down the problems discussed in this document. These procedures constitute the acceptance test used by Microsoft to verify compatibility.

6.3 Compatibility Issues and Solutions

The following subsections detail the major compatibility issues that existed with Windows 2.x and now with version 3.0, along with recommended solutions to these problems.

6.3.1 The Problem of Space

The biggest problem in running Windows with various networks has been lack of space. In Windows version 3.0, this problem is eliminated by moving Windows and Windows applications into protected mode, where they have access to all the memory in the system, not just the first 640K.

However, while this solution is good for Windows applications, it does not help non-Windows applications. The network software is global. It will continue to take the same amount of memory from both Windows and non-Windows applications. The difference is that, in the latter case, it is taking up *n* kilobytes out of 640K, whereas from Windows applications it is taking *n* from the much larger memory space available in protected mode.

6.3.2 The Problem of Global EMS

Many networks are attempting to free up space for other applications by loading portions of themselves into expanded memory (EMS).

Windows can coexist with most EMS systems found on 80286 machines, allowing this process to continue. However, on 80386 machines, we run into two possible problems:

- While running, enhanced Windows provides its own EMS emulation, which takes precedence over existing expanded memory managers. However, unless a cooperating EMS emulator is used, there can be no EMS continuity before, during, and after enhanced Windows is run.
- Windows cannot run in protected mode at the same time as other protected-mode software, including EMS emulators for the 80386 processor.

In many cases, these problems will prevent global software (such as MS-DOS or a network) from using EMS in situations where you need to run Windows.

However, Microsoft provides a solution to these problems. Windows 3.0 includes a new version of EMM386.SYS (the MS-DOS EMS emulator for the 80386 processor) that cooperates with enhanced Windows.

If this EMS emulator is installed in the system and is being used when enhanced Windows is started, enhanced Windows extracts information about its current handles and continues to support these while it is running. EMM386 is turned off during this period. When enhanced Windows exits, it turns EMM386 back on.

There are certain restrictions that apply to any global software that will make use of this feature:

- The software must not allocate or deallocate handles while enhanced Windows is running. That is, EMS handles must be the same size when enhanced Windows exits as they were when it started.
- Any time the software needs to map a handle while enhanced Windows is running, it must save the current context, perform the mapping, use the data, and then restore the original context. It must declare itself to be in a critical section during this entire process. Even though the Expanded Memory Manager (EMM) handles are global, any mapping is local to the current virtual machine (VM). Therefore, if enhanced Windows were to switch tasks when the software was using EMM, the mapping would suddenly change. Entering a critical section prevents this VM switching from occurring. Notice that MS-DOS device drivers are already in a critical section when called from MS-DOS.

The following table and explanatory notes describe the ways in which Windows reacts to other expanded memory managers:

	Enhanced Windows Open handles		**Standard Windows Open handles**	
Expanded Memory Manager	**Yes**	**No**	**Yes**	**No**
Unknown 386 Emulator	Error (1)	Error (1)	Error (1)	Error (1)
Physical Expanded Memory	Error (1)	OK (2)	OK (3)	OK (3)
EMM386 4.0 or Compaq CEMM	Error (1)	OK (4)	Error (1)	OK (5)
EMM386 for Windows 3.0	OK (6)	OK (6)	Error (1)	OK (5)

1. Windows will print an error message and abort.
2. Enhanced Windows will bypass and ignore an installed physical EMS board that has no open handles. Enhanced Windows will then provide simulated expanded memory to Windows applications and non-Windows applications. When enhanced Windows exits, the physical EMS will once again be available.
3. Standard Windows will ignore the installed physical EMS, which will continue to function and may be used by Windows applications and non-Windows applications. There

is no break in continuity across standard Windows execution, so memory-resident applications can continue uninterrupted.

4. Enhanced Windows can turn off certain expanded memory emulators (including Microsoft EMM386 4.0 and Compaq CEMM), provided they have no open handles. Enhanced Windows will then provide simulated EMS to Windows applications and non-Windows applications. The EMS emulator will be reenabled when enhanced Windows exits.
5. Standard Windows can turn off certain expanded memory emulators (including all the versions of Microsoft EMM386 and Compaq CEMM), provided they have no open handles. There will be no EMS available while standard Windows is running. The EMS emulator will be reenabled when standard Windows exits.
6. Enhanced Windows will take over any open handles and disable the emulator until enhanced Windows exits. As noted above, EMS-using software must obey certain guidelines in this configuration.

6.3.3 The Problem of Asynchronous Events

Handling asynchronous events is a problem in all multitasking situations. The problem occurs when network software does not realize that it is being called by several different applications. It tries to communicate with an application that has moved or been replaced, and the system usually crashes.

Figures 6.3 and 6.4 show a typical scenario in which enhanced Windows, MS-DOS, MS-DOS device drivers, and the network TSR are all loaded before enhanced Windows and are, therefore, shared between all the virtual machines. While the two applications are running concurrently in separate virtual machines, the following happens:

1. App2 makes an asynchronous network call, passing in the addresses of a buffer and a callback function.
2. The network software stores the addresses, sends off the asynchronous request to the network, and immediately returns to the caller.
3. Time passes, and enhanced Windows gives control to App3.
4. The network sends back the requested information. The network software gets control (either through the network card generating an interrupt or by polling on a timer interrupt) and reads the information.
5. The network software copies the information to the buffer address that it stored. However, in the current VM's context, this address does not point to App2's buffer but to the middle of App3's code. If App3 were to run after this, it would probably crash.
6. The network software calls the procedure address that it stored earlier. In VM2 this was App2's post routine, but in VM3 it is a random piece of App3's data.
7. Therefore, random things result.

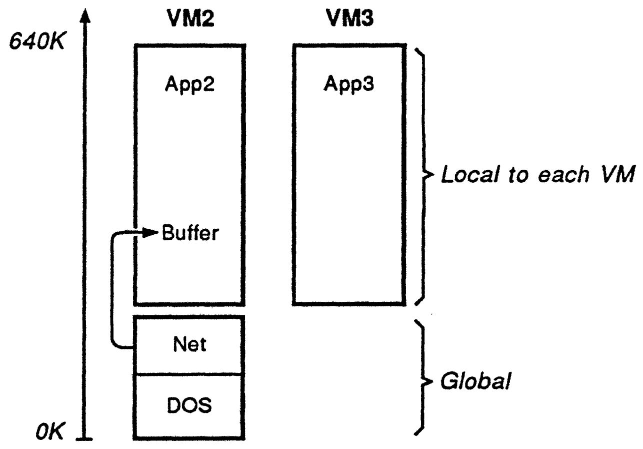


Figure 6.3 Address recorded for App2's buffer NETSP_03.EPS

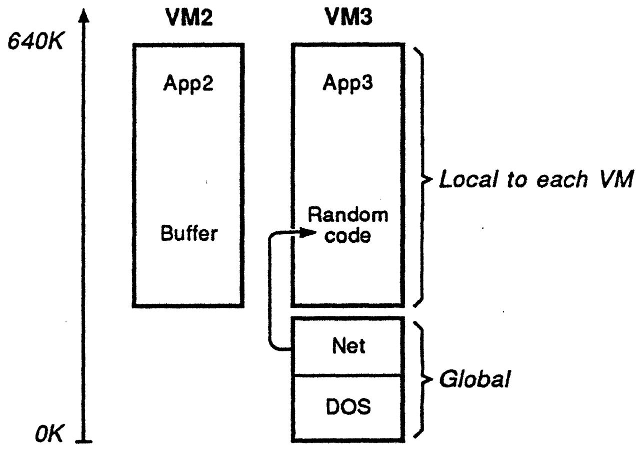


Figure 6.4 Writes to right address but in the wrong VM NETSP_04.EPS

As indicated earlier, this is not a problem unique to enhanced Windows. It is also true in real-mode Windows with EMS, as well as other applications that bank code or data in expanded memory, and even normal applications that use overlays. In each case, the environment or the application has to make a special effort to work with the moving targets.

When is this not a problem? If the application is doing only synchronous network operations, and those operations occur entirely in a critical section (such as an MS-DOS call), you are safe since neither mode of Windows will switch contexts until the operation has completed. This means that no work is usually required to support an application accessing a redirected drive.

The problem is less severe under standard or real-mode Windows, since it does not multitask non-Windows applications. In this case, the user must explicitly choose to switch tasks. However, in enhanced Windows, tasks are constantly being switched without the user's knowledge.

The following subsections discuss various solutions to problems resulting from asynchronous calls.

Modifying the Network Software to be Enhanced Windows Aware

The first solution is relatively easy. It requires changes to the network software, but no customizing of enhanced Windows.

Every time the network saves an application's buffer or callback address, it must ask enhanced Windows for the current virtual machine's ID number (VM ID) and save that with the addresses. Later, when it needs to access these locations, it can again ask for the current VM ID. If the two numbers are not the same, then the network software knows its stored addresses are invalid for this VM.

This solution worked fine under enhanced Windows 2.x. The network could get the current VM's ID using an INT 2FH call. If it found itself in the wrong context, it could simply go back to sleep and try again at the next timer interrupt, repeating the process until it happened to occur in the right virtual machine.

Version 3.0 provides the same solution, but with an improvement. When the network wakes up in the wrong virtual machine, it can tell enhanced Windows to schedule a callback for itself the next time the proper VM is run. This way it does not have to keep waking up unnecessarily with every timer tick. **(Where refer? INT 2FH?)**

This method still leaves the possibility of overrunning the network's buffer if too many packets come in before enhanced Windows is able to schedule a switch to the target VM. However, overrun problems can happen occasionally even in real mode so the software should be able to handle it correctly.

Creating A Virtual Device to Route Interrupts

With enhanced Windows version 3.0, it is possible to write a virtual device that understands the network well enough to route events to their proper virtual machines.

For example, when the network card generates an interrupt telling the network software that its information packet has arrived, the virtual device could read in the information and determine which VM it was supposed to reach. If that VM is running right then, the virtual device can immediately simulate the interrupt down into virtual mode and pass the information off to the network software. Otherwise, it can schedule an event for the target VM, including boosting its priority to get it to run as soon as possible, and, at that time, simulate the interrupt.

Using this method, the network software never has to worry about seeing an interrupt in the wrong context. Thus, the network software should not need to be modified at all.

Notice also that the virtual device could take on some part of the network functionality, thus bypassing the normal network software in some or all cases. This would avoid the bottleneck of having to call the normal network software in virtual mode; the virtual device could copy the data directly to the target application's buffers, and so forth. This would also avoid data-overrun problems when enhanced Windows is unable to switch to the proper virtual machine in a reasonable amount of time. And, finally, it could free up memory for non-Windows applications by, in effect, moving the entire network into protected mode.

Treating Asynchronous Calls as Critical Sections

A filter or the network software could declare a critical section around the entire duration of an asynchronous call. It would start the critical section when it first sent the request over the wire and end it only when the response came back. During that time, Windows would not be able to switch tasks, so the return event would always occur in the proper application's context.

The problem with this is that some asynchronous requests can be outstanding for a very long time. It is common for a network or an application to keep an asynchronous read always outstanding, to make sure that it never loses any data coming in. Of course, this would never allow Windows to switch away; the user would usually have to exit the application to reach other tasks.

If this method were only being used by a specific application, it might be worth it for the user to run it exclusively like this. If it were being done constantly by the network software, it would be unusable.

Advising Against Switching Away

In this scenario, you do not modify enhanced Windows or the network. You simply tell the user that all the applications that use asynchronous network requests must be run in exclusive mode (with multitasking disabled), and the user must not switch away manually while the application is active.

This achieves basically the same results as the previous example, except that it requires no code changes and allows the user to accidentally crash their system if they carelessly disobey the advice.

Configuring the PIF File to Prevent Switching

By setting the appropriate bits in the application's PIF file, you can prevent Windows from ever switching away when the application is running. This is not the most convenient way to use the application, but it does prevent the user from switching away while asynchronous events are outstanding.

The main weakness is that the user may not have proper PIF files for all their network applications.

The *Microsoft Windows User's Guide* contains more information on configuring PIF options.

6.3.4 The Problem of Protected-mode API

This is a problem with Windows 3.0, but only for Windows applications that call non-Windows APIs.

In Windows 3.0, Windows applications run in protected mode, while MS-DOS, the BIOS, and all terminate-and-stay-resident (TSR) programs (such as networks) run in virtual mode. The former use selector:offset addresses, the latter use segment:offset addresses. Because the two types are incompatible, it presents problems when a protected-mode application (e.g., any Windows application) needs to pass an address to MS-DOS, the BIOS, the network, or any other software loaded before Windows. The problem can occur as follows:

1. The Windows application places the address of its buffer into a pair of registers or on the stack. They are in the form of a selector:offset pair, pointing to some data in the application's data segment.
2. The application executes an interrupt to talk to the network.
3. Enhanced Windows or the DOS Extender reflect the interrupt down into virtual mode, so that the network software (which cannot run in protected mode) can handle it.
4. The network extracts the selector:offset pair passed in and misinterprets it as a segment:offset address.
5. The network reads from or writes to a random piece of conventional memory, causing unpredictable but usually fatal results.

In practice, the process may vary slightly, but it usually ends with memory being trashed.

The solution is to intelligently translate the application's selector:offset into a valid segment:offset before handing the address off to the network software. This is often referred to as "mapping" the API between modes. The mapper must hook the interrupt in question and determine the exact API being used. In every case where an address is being passed, the mapper must copy the passed data or buffer space down into virtual-mode memory, replace the selector:offset register values with a segment:offset pair pointing to the data's new location, and, finally, simulate the original interrupt to let the recipient software handle it. (See Figure 6.4 for an illustration of this process.)

When the network software returns from the interrupt, the API mapper must go through the same process but in reverse, copying data from the real-mode address to a protected-mode location, and readjusting the pointer to a selector: offset again so that the original caller can use it correctly.

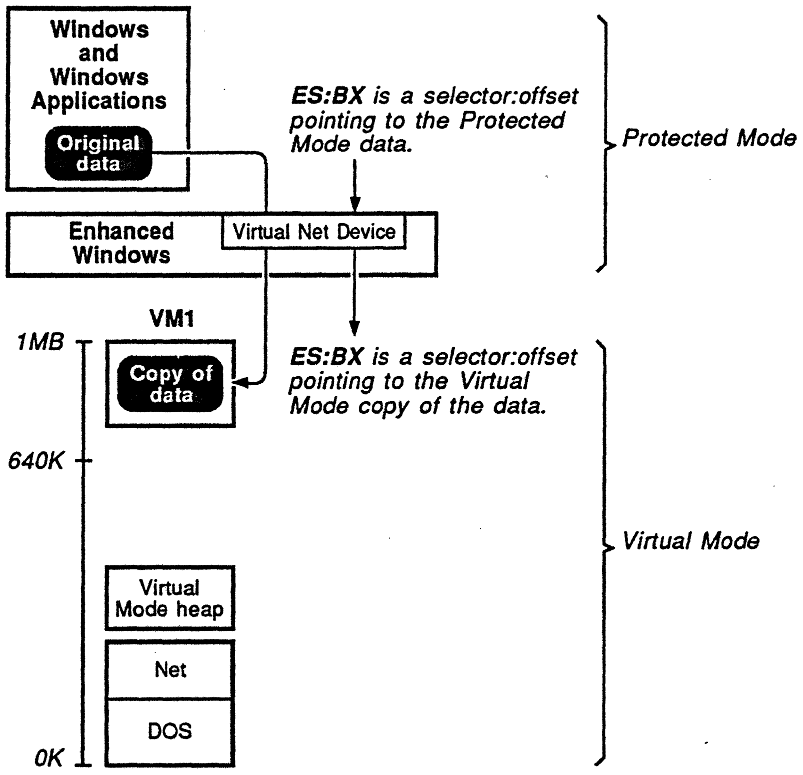

Windows and Windows Applications
Original data
ES:BX is a selector:offset pointing to the Protected Mode data.
Protected Mode
Enhanced Windows
Virtual Net Device
VM1
1MB
Copy of data
ES:BX is a selector:offset pointing to the Virtual Mode copy of the data.
640K
Virtual Mode
Virtual Mode heap
Net
DOS
0K

Figure 6.5 Mapping an address from PM to virtual mode

If the API involves passing buffers that contain segment addresses, the API mapper must also translate all of these, as well as copy the data they point to into V86 address space.

Notice that this is not usually a problem. Windows already takes care of translating the following most commonly used APIs:

- Standard MS-DOS calls (including network-related MS-DOS functions)
- Standard BIOS calls
- Standard NETBIOS calls

In addition, no work is required in the following cases:

- An application doing I/O directly to the hardware.
- An API going only between applications running in protected mode.

- An API going only between software running in real mode.
- An API that does not involve passing addresses or any data in segment registers or on the stack.

Usually, it is a problem only when a protected-mode Windows application needs to pass addresses to a real-mode TSR or device driver, using an API that Windows does not understand. Unfortunately, many networks fit all these requirements.

The following subsections describe several ways to implement API mapping.

Enhanced Windows Virtual Devices and the DOS Extender

In enhanced Windows, a virtual device can be written to handle API mapping; in standard Windows the code must be integrated into the standard Windows DOS Extender. This is the standard method used for all APIs supported by the packaged product.

When the API is relatively simple, the enhanced Windows code to do the mapping can be almost entirely table driven, requiring only simple modifications to the basic examples provided with this kit.

The main disadvantage of this method is that separate code needs to be written for the Windows, real, standard, and enhanced mode environments.

The INT 31H Translation Services

Enhanced Windows and the DOS Extender both provide a set of services to help protected-mode applications do their own API mapping. These INT 31H calls include services for LDT selector-management, DOS memory-management, interrupts, and translations. They can allow a single piece of code to map an API under either the real, standard, or enhanced modes of Windows 3.0. Therefore, this is the recommended method for adding such support.

These services do not provide intelligence; they only provide the mechanical means for such activities as copying to and from real mode. It is still the responsibility of the application to analyze the arguments being passed and translate them one by one between protected and real modes.

So who will actually call the translation services? They have to be called by code running in protected mode, and that means Windows code. It can be done in several ways:

- A dynamic-link library (DLL) could be written that traps all network interrupts generated in protected mode. It would then use INT 31H services to map the arguments down into real mode and simulate the interrupt for the real-mode software to handle. This method has the advantage of supporting existing applications that use interrupt-based APIs.
- A similar DLL could be written but, instead of trapping interrupts, it could provide the appropriate high-level network API to Windows applications. The DLL would look at the current mode: if operating in real mode, it would simply call the standard network

interrupts, but if running in protected mode, it would use INT 31H. This method is more efficient than the preceding one because it avoids the considerable overhead of generating and trapping interrupts.

- A combination DLL could provide both a high-level procedural API for newer applications and transparent interrupt-mapping services for existing applications.

Once the DLL is written, it can be installed in various ways:

- The code can be built into the Windows network driver.
- The code can reside in a standalone DLL, which is installed by listing it on the WIN.INI file's LOAD= line. This would have no user interface.
- The DLL can be built into a small application that the user starts manually or by using the LOAD= entry in WIN.INI. This application could then provide other functions or a user interface as well.
- If the API is going to be called by only a single application, the DLL could be distributed as part of the application. We recommend that the code remain in a DLL rather than be embedded in the application to maintain portability. The DLL can be dynamically linked at load time or explicitly when the application is running in protected mode.

A Shortcut for Simple NETBIOS Extensions

The basic Windows installations come with NETBIOS support, because this is the most common network interface across platforms.

Since many networks add simple extensions to this interface, the standard NETBIOS device for enhanced Windows and the DOS Extender both support a special INT 2FH call for communicating with NETBIOS software. The network (or a simple TSR or device driver) should watch for this interrupt and respond by passing the address of a table describing the complete API to be mapped. The mapper can then use this table to handle both standard NETBIOS and the particular network's extensions.

The exact calling sequences will be described in Appendix E, "Enhanced Windows INT 2FH API."

In cases where the changes would have been trivial, this method will let the network vendor write a single piece of code and avoid modifying the Windows NETBIOS mappers. If a TSR is used, this method also has the advantage that no changes need be made to the real-mode network software.

Handling the API Entirely in Protected–mode

As mentioned in Section 6.3.3, "The Problem of Asynchronous Events," a virtual device could be written that would emulate all the functionality of the real-mode network software, watching for interrupts from protected or real mode and talking directly to the network card.

The advantages of this are enormous: not only would it remove the necessity of copying arguments between real and protected mode, but it could also allow the user to remove the real-mode software altogether, freeing up space for non-Windows applications.

However, this is obviously a lot of work; in effect it means rewriting the network software for a new platform. It is also not a feasible solution for Windows in standard mode.

6.3.5 The Problem of Virtualizing Connections

Virtualizing connections is a problem in enhanced Windows (for both versions 2.x and 3.0) and to a lesser extent in other multitasking systems.

When dealing with multiple application contexts but global network software, we run into problems if the user is able to add and delete network connections. Some of these are real issues of compatibility and system integrity, while others are matters of maintaining a reasonable user interface. For example:

- If App1 is reading files from a network drive and the user goes into another virtual machine and deletes the connection to that server, what happens to App1?
- If the user adds a new connection while in one virtual machine, should an application running in another VM suddenly see another valid drive letter appear?
- If the user makes a connection in Windows and, then, starts another virtual machine, should the connection just made be inherited by the second VM?
- If the user makes a connection in one task and, then, exits that task without deleting the connection, the network may not be able to clean up its data structures. This can leave us with an "orphan" connection that can never be accessed or deleted. If you accumulate too many orphan connections, the network becomes clogged and useless.

In a perfect world, Windows should either maintain completely separate virtual machines, each with its own set of network connections, or keep a single global state. Unfortunately, these are not achievable without being intimate with the specific version of MS-DOS and the network installed.

However, some basic virtualization is necessary; problems might occur if one application could change another's current directory.

Windows in standard mode takes a simple approach. It saves and restores the current drive:directory context whenever it switches between non-Windows applications. This is done using the standard DOS calls.

Windows in enhanced mode is more complex. It provides a separate set of drive structures for each task. We supplement this with some support for MS-Net and NETBIOS based networks, trying to maintain a balance between user expectations and system integrity.

On MS-Net networks, the enhanced Windows DOSNET device provides the following support:

- If a connection exists before Windows is started, it is global and cannot be deleted from within Windows or any VMs.
- If the user makes a connection in Windows and, then, spawns off another virtual machine, that VM inherits Windows' connections as of that time.
- If a VM has inherited a connection from Windows, it cannot delete it.
- If any VM exists that inherited a particular connection from Windows, Windows cannot delete that connection.

In addition, the enhanced Windows NETBIOS device provides the following support:

- When a VM terminates, enhanced Windows will cancel any outstanding asynchronous network requests it had and cancel all its local connections.
- When Windows exits, enhanced Windows will cancel any outstanding asynchronous network requests it had and cancel all its local connections.

Notice that this will not work on other networks. In fact, since enhanced Windows is virtualizing the drive data structures, some networks might get terribly confused to see their drives changing as tasks are switched.

Some networks may choose to handle the virtualization themselves. By keeping track of the current virtual machine ID, the network can maintain a separate state for each VM and manage these however it sees fit.

This can also be improved by providing an enhanced Windows virtual device that can inform the network software when virtual machines are created and destroyed. The virtual device can even manage instance data within the network software by transparently inserting the appropriate data for the current virtual machine context.

6.4 Support and Distribution

Various networks may require one or more of the following:

- A Windows network driver
- One or more enhanced Windows virtual devices
- A modified DOS Extender
- A modified version of the network software

Microsoft provides the components necessary to support Microsoft Network and Microsoft LAN Manager. Most networks that are compatible with these can continue to use the standard versions.

In addition, the source to the Microsoft Network and NETBIOS components is made available as examples and can be modified when necessary.

In cases where the network software needs to be modified, those changes need to be distributed by the network vendor. If a customer calls Microsoft with problems, they will be informed of the necessary upgrade and told to contact their vendor.

Network vendors who develop other components should also make these components available to their customers. In some cases, though, Microsoft may be able to distribute these components. See Appendix C, "Creating Distribution Disks for Drivers," in the *Microsoft Windows Virtual Device Adaptation Guide* for more information on distribution possibilities.

Chapter 7 Network Drivers

The Windows network driver provides the Windows Shell and system utilities with a generic network interface. This interface provides a subset of functions that are generally available across many networks. The main functions specified in this chapter, and which should be supported in the driver, include:

- Making and breaking network connections
- Printing over a network connection
- Tracking the progress of print jobs on the network spooler

The Windows Shell, Control Panel, and Print Manager use these functions. Therefore, the driver should be fully functional to support each of these features, if applicable to your network.

In some cases, there will also be a unique network driver for each version of a particular network product. The functions they export will be called via entry points in the USER module of the Windows core.

The driver should export calls with their export ordinals. (See Section 7.11, "Function Summary," for a complete list of the functions and their export ordinals.) It is not necessary to have stub routines for unimplemented functions.

7.1 Initializing, Enabling, and Disabling

The driver's initialization routine (the DLL's entry point) should check to make sure that the appropriate network is actually installed and running. If it is not, the initialization routine should put up an appropriate error dialog box to inform the user and return a value of FALSE, which will prevent the driver from being loaded and taking up space unnecessarily. The system then behaves exactly as if no network driver were installed.

The driver also has the option of implementing the **Enable()** and **Disable()** functions. If it does implement them, they must be exported with the proper ordinal values of 21 for **Enable()** and 22 for **Disable()**. These functions are then called when Windows is enabling and disabling the OEM layer. **(Greg, please check the Enable and Disable functions in the Common Functions chapter. Is it appropriate for me to refer to them here as they are now or do I need to add some pertinent comments to them?)**

Enable() is called when Windows is loading the driver for the first time and when resuming Windows after it has been suspended. That is, when running in real or standard modes and returning to Windows after switching away from a non-Windows application.

Disable() is called before the driver's **WEP()** function when Windows is closing down and when Windows is about to be suspended. That is, when running in real or standard modes and Windows is about to be suspended to switch to a non-Windows application.

For example, if a network driver needs to hook an interrupt, it should make such a hook during the **Enable()** function and remove it during the **Disable()** function. This would prevent the driver's hook routine from being called when Windows was swapped out to disk.

7.2 Passing Buffers

The following calls take the address and size of a buffer into which the function will place a variable-sized data structure.

- **WNetGetConnection()**
- **WNetGetErrorText()**
- **WNetGetUser()**

In each case, the mechanism used is the same. First, the caller allocates a buffer. It then passes its address to the function in *lpBuffer* and the address of a WORD containing the buffer size in *lpBufferSize*. The function then copies as much of the requested data structure as it can into the buffer.

If it all fits, the function returns successfully. However, if it does not, the data may be left incomplete, and the function returns the WN_MORE_DATA status. In both cases, *lpBufferSize* is filled with the number of bytes actually required by the data structure. This way, if the buffer passed in was too small and the function failed, the caller may allocate a new buffer of the required size and call the function again.

All the data structures used are of a fixed size. They are allocated in a contiguous array starting at the beginning of the buffer. When they need to refer to variable-length strings, the data structures will contain pointers in the form of WORD offsets into the buffer. The driver should allocate space in the buffer to store these strings, starting at the end of the buffer and growing downward. When there is insufficient room in the middle for one more structure and its attendant strings, the driver stops adding data and returns the WN_MORE_DATA value.

7.3 Determining Network Capabilities

WORD FAR PASCAL WNetGetCaps(WORD *nIndex***);**
Export Ordinal @ 13

This function is used for determining which network calls are supported by the current driver.

Unlike all the other calls, this function does not return an error status. Instead, the *nIndex* parameter specifies a query, and that defines the possible values returned.

A few of the *nIndex* values cause a constant to be returned. However, in most cases, the *nIndex* parameter specifies which set of services is being queried, and the return value is a bitmask indicating which services in that set are supported. A zero value would indicate that none of the services in that set is supported.

Each value for *nIndex* is listed below, along with the constants defining the bits in the returned mask.

```
#define WNNC_SPEC_VERSION                   0x0001
```

The high and low bytes of the return value contain the major and minor version numbers of the network driver specification to which the driver conforms.

For this version, it would return 0x0300.

```
#define WNNC_NET_TYPE                       0x0002
```

Returns a WORD value; the high byte contains the network type, and the low byte may contain a subtype. The following Net Type values are defined:

```
#define WNNC_NET_None                       0x0000
#define WNNC_NET_MSNet                      0x0100
#define WNNC_NET_LanMan                     0x0200
#define WNNC_NET_NetWare                    0x0300
#define WNNC_NET_Vines                      0x0400
#define WNNC_NET_10NET                      0x0500
```

Developers working on new drivers should register their new type and subtype values with Microsoft.

```
#define WNNC_DRIVER_VERSION                 0x0003
```

Returns the driver version number.

```
#define WNNC_USER                           0x0004
```

Returns a mask of:

```
#define WNNC_USR_GetUser                    0x0001
```

```
#define WNNC_CONNECTION                     0x0006
```

Returns a mask of:

```
#define WNNC_CON_AddConnection              0x0001
#define WNNC_CON_CancelConnection           0x0002
```

```
#define WNNC_CON_GetConnections          0x0004
#define WNNC_CON_AutoConnect             0x0008
#define WNNC_CON_BrowseDialog            0x0010
```

Notice that the WNNC_CON_AutoConnect bit does not represent a network driver function. Instead, it means that the network will support the standard MS-DOS **Open** function for automatically connecting to network resources.

```
#define WNNC_PRINTING                    0x0007
```

Returns a mask of:

```
#define WNNC_PRT_OpenJob                 0x0002
#define WNNC_PRT_CloseJob                0x0004
#define WNNC_PRT_HoldJob                 0x0010
#define WNNC_PRT_ReleaseJob              0x0020
#define WNNC_PRT_CancelJob               0x0040
#define WNNC_PRT_SetJobCopies            0x0080
#define WNNC_PRT_WatchQueue              0x0100
#define WNNC_PRT_UnwatchQueue            0x0200
#define WNNC_PRT_LockQueueData           0x0400
#define WNNC_PRT_UnlockQueueData         0x0800
#define WNNC_PRT_ChangeMsg               0x1000
#define WNNC_PRT_AbortJob                0x2000
```

Notice that the WNNC_PRT_ChangeMsg bit does not represent a network driver function, but rather means that the driver will send SP_QUEUECHANGED messages to Print Manager. See Section 7.8.1, "Watching a Network Print Queue," for details.

```
#define WNNC_DEVICEMODE                  0x0008
```

Returns a mask of:

```
#define WNNC_DEVM_DeviceMode             0x0001
```

```
#define WNNC_ERROR                       0x000A
```

Returns a mask of:

```
#define WNNC_ERR_GetError                0x0001
#define WNNC_ERR_GetErrorInfo            0x0002
```

7.4 Displaying the Driver-Specific Dialog Box

WORD FAR PASCAL WNetDeviceMode(HWND *hParent*);
Export Ordinal @ 14

As with printer drivers, the network software may have many options that cannot be predicted or incorporated into a generic interface. Therefore, we provide a method for bringing up a dialog box that is provided by the network driver and is tailored to the specific network's needs. (See the *Systems Application Architecture, Common User Access: Advanced Interface Design Guide* for a description of the style to which the user interface should conform.)

This dialog box is the proper place for the authors to display information such as the driver's name, version numbers of both the driver and the installed network software, and copyright notice.

This is also the place from which the user can access functionality that is specific to the network. For example, if the network supports sending messages to another user, the driver can supply an interface in its device-mode dialog box. The user will be able to invoke this dialog box from the Windows Control Panel.

The *hParent* parameter specifies the handle to a window that should be used as the dialog box's parent.

7.5 Displaying the Browse Dialog Box

WORD FAR PASCAL WNetBrowseDialog(HWND *hParent*, **WORD** *nType*, **LPSTR** *szPath*);
Export Ordinal @ 15

This function displays one or more dialog boxes that enable the user to select a network resource. The *nType* parameter specifies what kind of resource you are looking for; possible values include WNBD_CONN_DISKTREE and WNBD_CONN_PRINTQ. The *szPath* string is used to pass back the network path of the selected resource. The *hParent* parameter is the handle to a window that should be specified as the parent of the new dialog box.

The function returns a string containing the complete network path selected. It should be formatted so that it can be passed unaltered to **WNetAddConnection()**.

If the caller allocates a buffer that is too small to hold the path, it cannot call again because that would cause the driver to prompt the user a second time. Therefore, the caller will allocate a buffer of at least 128 characters in length.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_VALUE	Invalid *nType* value
WN_CANCELLED	Cancelled at user's request
WN_BAD_POINTER	Invalid pointer
WN_OUT_OF_MEMORY	Out of memory

7.6 Getting the Current Username

WORD FAR PASCAL WNetGetUser(LPSTR *szUser*, **LPWORD** *nBufferSize*);
Export Ordinal @ 16

This function places the current username into the *szUser* string. It uses the standard mechanism of returning the buffer size required in the *nBufferSize* parameter. (See Section 7.2, "Passing Buffers," for more detailed information.)

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_USER	Not logged in; no current username
WN_MORE_DATA	The buffer was too small
WN_OUT_OF_MEMORY	Out of memory

7.7 Device Redirecting Functions

These are calls that redirect standard MS-DOS devices, such as drive letters and LPT ports, so that standard applications may use them and access the network in a totally transparent manner.

7.7.1 Adding Network Connections

WORD FAR PASCAL WNetAddConnection(LPSTR *szNetPath*, **LPSTR** *szPassword*, **LPSTR** *szLocalName*);
Export Ordinal @ 17

This function redirects a local device (either a disk drive or a printer port) to a shared device on a remote server.

All *szLocalName* strings (such as "LPT1:") are case independent. The following illustrate the most common names:

```
"A:"  -  "Z:"
"LPT1:"  -  "LPT4:"
""
```

The empty string ("") indicates a non-redirected connection. Such connections are not accessible by accessing a redirected port or drive letter. They must be opened explicitly by using the MS-DOS **Open** function.

In these cases, **WNetAddConnection()** serves only to specify a password for this network path. Later, when the same network path is specified in an MS-DOS **Open** call, the network software may use this password.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_NETNAME	Invalid network resource name
WN_BAD_LOCALNAME	Invalid local device name
WN_BAD_PASSWORD	Invalid password
WN_ACCESS_DENIED	Security violation
WN_ALREADY_CONNECTED	Local device already connected to a remote resource
WN_OUT_OF_MEMORY	Out of memory

7.7.2 Removing Network Connections

WORD FAR PASCAL WNetCancelConnection(LPSTR *szName*, **BOOL** *fForce*);
Export Ordinal @ 18

This function cancels a redirection. The *szName* string specifies the name of the redirected local device (such as "LPT1:"). If a fully-qualified network path is specified, the driver will cancel all the connections to that resource.

The *fForce* parameter indicates whether or not any open files or open print jobs on the device should be closed before the connection is cancelled. If *fForce* is FALSE and there are open files or jobs, the connection will not be cancelled and the function will return an error value.

Return Values

WN_SUCCESS	Success

WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_VALUE	*szName* is not a valid local device or network name
WN_NOT_CONNECTED	*szName* is not a redirected local device or currently accessed network resource
WN_OPEN_FILES	Files are open and *fForce* was FALSE. Connection was not cancelled.
WN_OUT_OF_MEMORY	Out of memory

7.7.3 Listing Network Connections

WORD FAR PASCAL WNetGetConnection(LPSTR *lpLocalName*, **LPSTR** *lpRemoteName*, **LPWORD** *nBufferSize*);
Export Ordinal @ 12

This function returns the name of the network resource associated with a redirected local device. (See Section 7.2, "Passing Buffers," for more detailed information.)

The *szLocalName* string specifies the name of the redirected local device. The name of the remote network resource is returned in the *lpRemoteName* parameter.

This function uses the standard mechanism of taking the maximum length of *lpRemoteName* in *nBufferSize* and returning the actual required length in the same location.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_VALUE	*szLocalName* is not a valid local device
WN_NOT_CONNECTED	*szLocalName* is not a redirected local device
WN_MORE_DATA	Buffer was too small
WN_OUT_OF_MEMORY	Out of memory

7.8 Net Printing Functions

These are calls for printing over the network as well as for tracking network print jobs.

The driver may assume that it will only be called by Print Manager. To get maximum support from Print Manager, you should implement all the functions that your network software can handle.

7.8.1 Watching a Network Print Queue

WORD FAR PASCAL WNetWatchQueue(HWND *hWnd*, **LPSTR** *szLocal*, **LPSTR** *szUser*, **WORD** *nQueue*);
Export Ordinal @ 8

This function instructs the driver that Print Manager is interested in the specified remote queue.

The *hWnd* parameter is the handle to a window that should be notified when the status of a job in the queue changes. See Section 7.8.5, "Notification of Queue Status Changes," for details.

The *szLocal* string should specify the name of a local device (such as "LPT1:") that has been redirected to a network print queue.

The caller may also specify an *szUser* string that says that it is only concerned with jobs belonging to a particular user. This parameter is advisory.

Print Manager will watch each redirected device appearing in the Print Manager client area.

The caller may specify an *nQueue* value, which is an arbitrary WORD. When the queue-change notification occurs, the network driver will pass the *nQueue* WORD specified for this queue as the *wParam* value. This will allow Print Manager to determine which queue to refresh.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_VALUE	Invalid window handle
WN_BAD_LOCALNAME	Invalid local device name or local device not redirected

WN_ALREADY_LOCKED	Local device already being watched
WN_OUT_OF_MEMORY	Out of memory

7.8.2 Stop Watching a Network Print Queue

WORD FAR PASCAL WNetUnwatchQueue(LPSTR *szLocal*);
Export Ordinal @ 9

This function informs the driver that Print Manager is no longer interested in a queue.

The *szLocal* string specifies the name of a local device that Print Manager no longer wants to watch.

The driver may assume that watched queues will be cancelled before Print Manager terminates.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_QUEUE	*szDestination* is not a valid net queue or redirected device
WN_BAD_POINTER	Invalid pointer
WN_OUT_OF_MEMORY	Out of memory

7.8.3 Locking Network Queue Data

WORD FAR PASCAL WNetLockQueueData(LPSTR *szQueue*, LPSTR *szUser*, LPQUEUESTRUCT FAR * *lplpQueueStruct*);
Export Ordinal @ 10

Print Manager uses this function to lock a buffer, maintained by the driver, that describes the state of the queue. (The structure can be created at this time if it was not already available.) The buffer will contain a single QUEUESTRUCT followed by zero or more JOBSTRUCTs and the variable-length strings to which they refer. The number of JOBSTRUCT structures is specified by the *pqJobCount* field in QUEUESTRUCT. The strings referenced by JOBSTRUCT and QUEUESTRUCT are stored by the mechanism described in Section 7.2, "Passing Buffers."For example, assuming that *lpQS* is the LPQUEUESTRUCT returned by **WNetLockQueueData()**, the expression

```
(LPSTR) lpQS + lpQS -> pqComment
```

is a FAR pointer to the queue comment. All offsets are relative to this pointer, not the beginning of individual JOBSTRUCTs.

The *szQueue* string is the name of a local or remote queue.

The driver places a FAR pointer to QUEUESTRUCT at the location pointed to by the *lplpQueueStruct* parameter.

If the *szUser* string is not NULL, it indicates that Print Manager is only interested in jobs belonging to that user. However, it is advisory, and Print Manager will not assume that QUEUESTRUCT will only contain information on those jobs.

While Print Manager's lock is in effect, the queue data must not be modified or moved.

Queue Structure

```
typedef  struct _queuestruct {
        WORD   pqName;          /* queue name (offset)            */
        WORD   pqComment;       /* queue comment string (offset)  */
        WORD   pqStatus;        /* queue status                   */
        WORD   pqJobcount;      /* number of JOBSTRUCTs following */
        WORD   pqPrinters;      /* number of printers for queue   */
                                /*   (zero if not available)      */
        } QUEUESTRUCT;
```

Queue Statuses

```
#define WNPRQ_ACTIVE    0x0000  /* ok                            */
#define WNPRQ_PAUSE     0x0001  /* this queue is paused          */
#define WNPRQ_ERROR     0x0002  /* network error                 */
#define WNPRQ_PENDING   0x0003  /* queue has been deleted        */
                                /*   it will last until current  */
                                /*   jobs are done; no new jobs  */
                                /*   may be started              */
#define WNPRQ_PROBLEM   0x0004  /* all printers are stopped      */
```

Job Structure

```
typedef struct _jobstruct {
        WORD  pjId;             /* job ID                          */
        WORD  pjUsername;       /* submitting user name (offset)   */
        WORD  pjParms;          /* implementation string (offset)  */
        WORD  pjPosition;       /* position of job in the queue    */
        WORD  pjStatus;         /* job status                      */
        DWORD pjSubmitted;      /* time when job was submitted     */
                                /*   (from 1970-1-1 in seconds)    */
        DWORD pjSize;           /* job size in bytes               */
                                /*   (-1L means not available)     */
        WORD  pjCopies;         /* number of copies                */
                                /*   (0 means not available)       */
        WORD  pjComment;                 /* comment for this job (offset)*/
        } JOBSTRUCT;
```

Job Statuses

```
#define WNPRJ_QSTATUS                   0x0007  /* Bits 0-2                */

        #define WNPRJ_QS_QUEUED         0x0000  /* job queued              */
        #define WNPRJ_QS_PAUSED         0x0001  /* job paused              */
        #define WNPRJ_QS_SPOOLING       0x0002  /* job spooling            */
        #define WNPRJ_QS_PRINTING       0x0003  /* job printing            */

#define WNPRJ_DEVSTATUS                 0x1FF8 /* Bit 3-12                 */

        #define WNPRJ_DS_COMPLETE       0x0008 /* complete                 */
        #define WNPRJ_DS_INTERV         0x0010 /* intervention required    */
        #define WNPRJ_DS_ERROR          0x0020 /* printer error            */
        #define WNPRJ_DS_DESTOFFLIN     0x0040 /* printer offline          */
        #define WNPRJ_DS_DESTPAUSED     0x0080 /* printer paused           */
        #define WNPRJ_DS_NOTIFY         0x0100 /* notify owner             */
        #define WNPRJ_DS_DESTNOPAPER    0x0200 /* printer out of paper     */
        #define WNPRJ_DS_DESTFORMCHG    0x0400 /* form change required     */
        #define WNPRJ_DS_DESTCRTCHG     0x0800 /* cartridge change req     */
        #define WNPRJ_DS_DESTPENCHG     0x1000 /* pen change required      */
```

Notice that if the queue status is WNPRQ_PROBLEM, which means that all the printers on this queue have stopped, examining the job status for the first job in the queue will usually reveal the exact nature of the problem. However, this may not be true if the queue is serviced by multiple printers that could have different problems, or if the first job is paused and the first non-paused job is being held up by an error.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_BAD_QUEUE	*szDestination* is not a valid net queue or redirected device
WN_BAD_POINTER	Invalid pointer
WN_OUT_OF_MEMORY	Out of memory

7.8.4 Unlocking Network Queue Data

WORD FAR PASCAL WNetUnlockQueueData(LPSTR ***szQueue*****);**
Export Ordinal @ 11

This function informs the driver that Print Manager is no longer examining a block of queue data. The driver is free to deallocate, reallocate, move, or modify the queue information at this point.

The *szQueue* string is the name of a network printer queue and is either a redirected local name or a remote name. This device should have been previously specified in a call to **WNetWatchQueue()**.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_BAD_QUEUE	*szDestination* is not a valid net queue or redirected device
WN_BAD_POINTER	Invalid pointer
WN_OUT_OF_MEMORY	Out of memory

7.8.5 Notification of Queue Status Changes

A driver indicates that it can support notification by setting the WNNC_PRT_ChangeMsg bit returned by **WNetGetCaps()**. See Section 7.3, "Determining Network Capabilities," for more detailed information.

When Print Manager is watching a queue, the driver is free to notify it of queue changes by posting SP_QUEUECHANGED messages to the window handle specified by the *hWnd* parameter. The driver must use the **PostMessage()** function and not the **SendMessage()** function. The *wParam* parameter should contain the value passed by Print Manager in its call to **WNetWatchQueue()**. The *lParam* parameter should be NULL.

The driver should only send messages relating to the queue that was specified in *lpszLocal*, and only when jobs belonging to the current username change status. Moreover, the message is purely advisory; the driver should not assume that a Lock/Unlock pair will result from the posting of this message, or that Print Manager will only lock a queue in response to one of these messages. Conversely, Print Manager does not assume that any job has actually changed status simply because this message was received.

As you can see, polling is easy to support with this mechanism; all that is required is for the driver to post the SP_QUEUECHANGED messages periodically.

The Queue Changed Message

```
#define SP_QUEUECHANGED 0x0500  /* advises Print Manager that  */
                                /* a queue has changed         */
```

7.8.6 Opening a Network Print Job

WORD FAR PASCAL WNetOpenJob(LPSTR *szQueue*, **LPSTR** *szJobTitle*, **WORD** *nCopies*, **LPWORD** *pfh*);
Export Ordinal @ 1

This function initializes a print job for transmission across the net. It returns a file handle in the buffer indicated by the *pfh* parameter, which Print Manager can use to write to the spoolfile.

The *szQueue* string may specify either a redirected local device or a fully qualified network name of a print queue.

If the *szJobTitle* string is omitted, the driver may use any appropriate default.

On some networks, open print jobs will automatically be closed and printed after some timeout period. It is the responsibility of the driver to make sure that the network does not timeout on open print jobs, even if the application is inactive for long periods of time.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_QUEUE	*szDestination* is not a valid net queue or redirected device
WN_CANT_SET_COPIES	Warning, printing one copy
WN_OUT_OF_MEMORY	Out of memory

7.8.7 Closing a Network Print Job

WORD FAR PASCAL WNetCloseJob(WORD *fh*, **LPWORD** *pidJob*, **LPSTR** *szQueue*);
Export Ordinal @ 2

This function finishes the processing of a spooled print job. It returns a unique Job ID in the buffer indicated by the *pidJob* parameter and the normalized name of the network queue in the *szQueue* string. Together, these can be used to reference the job at any later time.

If the driver cannot return the correct Job ID, it should return the constant value of WN_NULL_JOBID, as shown below:

```
#define WN_NULL_JOBID 0x0000 /* Job ID not available */
```

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_HANDLE	File handle is invalid
WN_OUT_OF_MEMORY	Out of memory

7.8.8 Putting a Print Job on Hold

WORD FAR PASCAL WNetHoldJob(LPSTR *szQueue*, **WORD** *wJobID*);
Export Ordinal @ 4

This function attempts to hold a previously spooled network job.

The exact effect this has will depend upon the network. In most cases, the held job will continue to move up in the queue until it is ready to print. However, it will then not be allowed to print and will sit with the queue number of 1, while other jobs move around it and are printed. This will last until the job is released using the **WNetReleaseJob()** function.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_JOBID	Job ID is invalid
WN_JOB_NOT_FOUND	No job with this ID found on this queue
WN_BAD_QUEUE	*szDestination* is not a valid net queue or redirected device
WN_ACCESS_DENIED	Security violation
WN_OUT_OF_MEMORY	Out of memory

7.8.9 Releasing a Held Print Job

WORD FAR PASCAL WNetReleaseJob(LPSTR *szQueue*, **WORD** *wJobID*);
Export Ordinal @ 5

This function attempts to release a previously held network job.

Return Values

WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_QUEUE	*szDestination* is not a valid net queue or re-directed device
WN_BAD_JOBID	Job ID is invalid
WN_JOB_NOT_FOUND	No job with this ID found on this queue
WN_JOB_NOT_HELD	The job is not held
WN_ACCESS_DENIED	Security violation
WN_OUT_OF_MEMORY	Out of memory

7.8.10 Cancelling a Print Job

WORD FAR PASCAL WNetCancelJob(LPSTR *szQueue*, **WORD** *wJobID*);
Export Ordinal @ 6

This function attempts to cancel a previously spooled network job.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_QUEUE	*szDestination* is not a valid net queue or redirected device
WN_BAD_JOBID	Job ID is invalid
WN_JOB_NOT_FOUND	No job with this ID found on this queue

WN_ACCESS_DENIED	Security violation
WN_OUT_OF_MEMORY	Out of memory

7.8.11 Changing the Number of Copies

WORD FAR PASCAL WNetSetJobCopies(LPSTR *szQueue*, **WORD** *wJobID*, **WORD** *nCopies*);
Export Ordinal @ 7

This function attempts to change the number of copies of a previously spooled job. If the number is 0, the function will return an error status and will have no effect. If the number is too great for the queue to support, the number should be changed to the largest one possible.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_VALUE	Invalid number of copies
WN_BAD_QUEUE	*szDestination* is not a valid net queue or redirected device
WN_BAD_JOBID	Job ID is invalid
WN_JOB_NOT_FOUND	No job with this ID found on this queue
WN_ACCESS_DENIED	Security violation
WN_OUT_OF_MEMORY	Out of memory

7.8.12 Aborting a Print Job

WORD FAR PASCAL WNetAbortJob (LPSTR *szQueue*, **WORD** *fh*);
Export Ordinal @ 3

This function cancels a print job while the file handle is still open.

The *szQueue* string is the name of the local device, and the *fh* parameter is the file handle returned by **WNetOpenJob()**.

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function not supported
WN_NET_ERROR	Network error
WN_BAD_POINTER	Invalid pointer
WN_BAD_QUEUE	*szDestination* is not a valid net queue or redirected device
WN_BAD_HANDLE	File handle is invalid
WN_ACCESS_DENIED	Security violation
WN_OUT_OF_MEMORY	Out of memory

7.9 Extended Error Functions

These functions are designed to get more detailed information about network-dependent errors.

7.9.1 Getting the Current Network Error

WORD FAR PASCAL WNetGetError(LPWORD *nError*)
Export Ordinal @ 19

This function returns the network status code from the last network operation.

Notice that this is only guaranteed to be correct if the application calls immediately after receiving the network driver error status. If another application is given a chance to run, it may call the driver and the original extended error condition would be overwritten.

Return Values

WN_SUCCESS	Success
WN_BAD_POINTER	Invalid pointer
WN_NOT_SUPPORTED	Function is not supported
WN_NO_ERROR	No error status available
WN_OUT_OF_MEMORY	Out of memory

7.9.2 Getting Extended Error Information

WORD FAR PASCAL WNetGetErrorText(WORD *nError*, **LPSTR** *lpBuffer*, **LPWORD** *nBufferSize*)
Export Ordinal @ 20

This function returns a text description associated with a network error code. This is called with the *nError* parameter containing a network error code (as returned by **WNetGetError()**). All the parameters are handled as described in Section 7.2, "Passing Buffers."

Return Values

WN_SUCCESS	Success
WN_NOT_SUPPORTED	Function is not supported
WN_NET_ERROR	A network error occurred when attempting to get the text
WN_MORE_DATA	The buffer was too small
WN_NO_ERROR	No error status available
WN_BAD_POINTER	Invalid pointer
WN_OUT_OF_MEMORY	Out of memory

7.10 Return Values

Most network driver functions return zero if successful and unique non-zero values for various error conditions.

Currently, there is one exception: **WNetGetCaps()**, which returns a mask of bits.

The following is a list of the normal return values:

```
#define WN_SUCCESS          0x0000 /* success                     */
#define WN_NOT_SUPPORTED    0x0001 /* function not supported      */
#define WN_NET_ERROR        0x0002 /* misc network error          */
#define WN_MORE_DATA        0x0003 /* warning: buffer too small   */
#define WN_BAD_POINTER      0x0004 /* invalid pointer specified   */
#define WN_BAD_VALUE        0x0005 /* invalid value specified     */
#define WN_BAD_PASSWORD     0x0006 /* incorrect password specified */
#define WN_ACCESS_DENIED    0x0007 /* security violation          */
#define WN_FUNCTION_BUSY    0x0008 /* function can't be reentered */
                                   /*  and is currently being used */
#define WN_BAD_USER         0x000A /* invalid username specified  */
#define WN_OUT_OF_MEMORY    0x000B /* out of memory               */
#define WN_CANCELLED        0x000C /* operation cancelled at user's */
                                   /*   request                   */
```

```
#define WN_NOT_CONNECTED        0x0030 /* device is not redirected      */
#define WN_OPEN_FILES           0x0031 /* connection couldn't be can-   */
                                       /*  celled, files are still open */
#define WN_BAD_NETNAME          0x0032 /* network name is invalid       */
#define WN_BAD_LOCALNAME        0x0033 /* invalid local device name     */
#define WN_ALREADY_CONNECTED    0x0034 /* local device already con-     */
                                       /*   nected to a remote resource */
#define WN_BAD_JOBID            0x0040 /* invalid job ID                */
#define WN_JOB_NOT_FOUND        0x0041 /* no job found with this ID     */
#define WN_JOB_NOT_HELD         0x0042 /* job cannot be released because*/
                                       /*   it is not currently held    */
#define WN_BAD_QUEUE            0x0043 /* name given does not correspond*/
                                       /*   to a network queue name or a*/
                                       /*   redirected local device     */
#define WN_BAD_HANDLE           0x0044 /* not a valid file handle, or   */
                                       /*   handle not to network print */
                                       /*   file opened by driver       */
#define WN_CANT_SET_COPIES      0x0045 /* warning: cannot set number of */
                                       /*   copies, printing just one   */
#define WN_ALREADY_LOCKED       0x0046 /* queue specified is already    */
                                       /*   locked; from LockQueue func.*/
#define WN_NO_ERROR             0x0050 /* No error status available     */
```

7.11 Function Summary

The following is a complete alphabetical list of the network driver functions and their export ordinals. Notice that if the driver exports functions that are not on this list, then those functions should have export ordinals greater than 500. The numbers below 500 are reserved for functions with prescribed ordinal values. For example, the **DeviceMode()** dialog box callback function should have an ordinal value above 500 so as not to conflict with any required ordinals defined in future versions.

Function	Export Ordinal
WNetAbortJob()	@3
WNetAddConnection()	@17
WNetBrowseDialog()	@15
WNetCancelConnection()	@18
WNetCancelJob()	@6
WNetCloseJob()	@2
WNetDeviceMode()	@14
WNetGetCaps ()	@13
WNetGetConnection()	@12

Function	Export Ordinal
WNetGetError()	@19
WNetGetErrorText()	@20
WNetGetUser()	@16
WNetHoldJob()	@4
WNetLockQueueData()	@10
WNetOpenJob()	@1
WNetReleaseJob()	@5
WNetSetJobCopies()	@7
WNetUnlockQueueData()	@11
WNetUnwatchQueue()	@9
WNetWatchQueue()	@8

Chapter 8 Keyboard Drivers

Among the disks provided with your Device Development Kit, you will find ones containing source code for our Windows 3.0 keyboard driver. We recommend that you use our driver either in its entirety or with any necessary customizing rather than try to write your own. Section 8.9, "A Checklist for Modifying a 3.0 Keyboard Driver," provides steps for you to follow after reviewing the information provided in this chapter.

When Windows is running, the Windows keyboard driver entirely replaces the DOS keyboard driver. It is designed, as much as possible, to provide a hardware-independent interface between the keyboard driver and Windows. This was done to allow the Windows code that deals with user interaction (e.g., menus and dialogs) to be as hardware-independent as possible.

The Windows keyboard driver contains two major functions: the hardware interrupt function, and the **ToAscii()** function. The first translates hardware scan codes to Windows virtual key codes, which are passed to Windows for action and queuing (as keyboard events), and the second is called to translate the virtual keycodes in the queue (along with the current state of shift-lock, etc.) to ANSI characters.

For Windows 3.0, the keyboard driver has been split up into two modules, the driver and a library of keyboard tables. The basic driver is somewhat hardware-specific and contains translation tables only for the enhanced USA keyboard. Tables for other keyboards (i.e., for other countries and other physical keyboards) are contained in an optional dynamic-linked library (DLL), which is copied by Setup or another application such as Control Panel to the Windows directory.

Manufacturers should put hardware-specific code into their own version of the keyboard driver and avoid changing the DLLs, unless their keyboard layout differs from an IBM-compatible system.

The following sections provide a description of the driver's sample source code.

8.1 Initialization Code

Initialization code is code that is executed only when the keyboard driver is *first* loaded. This code is mainly used for identifying the hardware type.

The code will set a special flag if Windows is running in the OS/2 compatibility box. This flag is used to determine whether or not certain keys (the SYSRQ key for SYMDEB and screen switching) need to be handled in a special manner for OS/2.

8.2 Keyboard Entries: Exported Functions

Once Windows finishes loading everything, it then calls the following exported functions. (See Section 8.10, "Functions Reference," for descriptions of these functions.)

- **AnsiToOem()**
- **AnsiToOemBuff()**
- **Disable()**
- **Enable()**
- **EnableKBSysReq()**
- **GetKBCodePage()**
- **GetKeyboardType()**
- **Inquire()**
- **MapVirtualKey()**
- **OEMKeyScan()**
- **OemToAnsi()**
- **OemToAnsiBuff()**
- **ScreenSwitchEnable()**
- **SetSpeed()**
- **ToAscii()**
- **VkKeyScan()**

8.3 Internal Functions

The following functions are used internally by the keyboard driver and the keyboard DLL. Control Panel is the only application that calls them. (See Section 8.10, "Functions Reference," for descriptions of these functions.) **(Greg/Peter, what else can we say about internal functions?)**

- **GetTableSeg()**
- **NewTable()**

8.4 The Keyboard Interrupt Handler and Event Procedure Call

This function primarily traps the keyboard hardware interrupt (INT 09H on most systems), reads the hardware scan codes generated by the keyboard, and translates hardware scan codes to Windows virtual key codes. These virtual key codes are passed to the Windows USER module by calling the keyboard event procedure. This address was passed to the keyboard driver in the **Enable()** function. (See Section 8.10, "Function Reference," for a description of the **Enable()** function.)

8.4.1 Parameter Details

The keyboard event procedure is called with the following parameters:

AH = 0 for down stroke, 80H for up stroke
AL = Windows virtual key code
BH = 0 if no prefix byte, 1 if E0 prefix byte preceded this scan code
(for IBM-compatible enhanced keyboard. For more information, see Section 8.4.2, "Extended Keyboards.").
BL = "Hardware" scan code

Notice that while the hardware scan code (which is OEM-dependent) is passed to Windows, it is not used internally in Windows. It is subsequently passed to **ToAscii()**, which may use it for special purposes.

This function detects the System Request function (normally CTRL+ALT+PRINT SCREEN) and generates an NMI interrupt, if SYMDEB appears to be trapping the NMI interrupt.

Under certain circumstances, control is passed to the MS-DOS keyboard interrupt handler, which was installed when Windows was started. This is the case for CTRL+ALT+DELETE (soft reboot) and for the PAUSE key.

In the Windows 3.0 keyboard driver, the PRINTSCREEN key is used for making a screen snapshot, which is saved in the Windows Clipboard. This is handled in the interrupt routine by calling the event procedure with VK_SNAPSHOT in **AL**, with the value equal to 0 (for full-screen snapshot) or 1 (for current-window snapshot) in **BL**.

8.4.2 Extended Keyboards

Some keyboards, such as the IBM enhanced keyboard, have an extra ENTER key on the numeric keypad, as well as separate keys for such functions as CURSOR CTRL, INSERT, and DELETE. On the IBM enhanced keyboard, these keys have the same scan codes as the main ENTER key, certain numeric-pad keys (with NUM LOCK OFF), etc., but the keyboard sends a special prefix byte to indicate that these keys are pressed. In this case, the Windows 3.0 keyboard driver sets bit 0 of **BH** when calling the keyboard event procedure. If this bit is set, bit 24 is set in the long parameter of the resulting WM_KEYDOWN, WM_CHAR, etc. messages.

The following are the keys on the IBM enhanced keyboard for which this extended bit is set:

- INSERT, HOME, PAGE UP, DELETE, END, PAGE DOWN
- Cursor Keys
- Numeric-pad DIVIDE and ENTER keys.

For keyboards that have extended keys with special scan codes instead of prefixes, we recommend that the scan codes be translated to the IBM scan codes (to allow Windows to translate these keys properly) and that **BH** be set to 1 when the event procedure is called for a second ENTER key or for cursor keys, etc.

8.4.3 The OS/2 Compatibility Box

If Windows is running in the OS/2 compatibility box, two things are done in the interrupt routine:

1. The CTRL+ALT+SYSRQ handling for SYMDEB is disabled.
2. The CTRL and ESC combinations are passed on to the OS/2 keyboard interrupt handler, so that OS/2 screen switching will work.

8.5 Keyboard Driver Internal Tables

The following subsections deal with various internal structures and tables used with this driver.

8.5.1 Keyboard State Vector

This is a 256-byte vector, maintained by Windows, that contains an entry for each possible virtual key code. For each entry, bit 7 is set, if the key is down, and cleared when the key is released. Likewise, bit 0 is toggled each time a key is pressed if it was not already down (i.e., repeats do not toggle this bit).

The address of this vector is passed to **ToAscii()**, so that this function may examine the current state of the SHIFT KEYS, CAPS LOCK, etc. It is also passed to the **Enable()** function, so that the entries for various shift keys are synchronized with the shift state bits that MS-DOS maintains at 40H:17H.

Mouse buttons also get translated into their corresponding virtual key codes (i.e., VK_LBUTTON, VK_RBUTTON, VK_CANCEL, and VK_MBUTTON) in the key state vector. See Section 8.8, "Windows Virtual Key Codes," for a complete list of the virtual key codes supported in Windows. This information is also available in the WINDOWS.H file.

8.5.2 Keyboard Information (KBINFO) Data Structure

The following is the data structure for keyboard information (KBINFO).

```
typedef struct tagKBINFO
  {
    BYTE  Begin_First_range;           /* used for KANJI */
    BYTE  End_First_range;             /* used for KANJI */
    BYTE  Begin_Second_range;          /* used for KANJI */
    BYTE  End_Second_range;            /* used for KANJI */
    int   StateSize;                   /* size of ToAscii state block*/
    int   NumFuncKeys                  /* number of function keys */
  } KBINFO;
```

8.5.3 Key Translation Tables

The following description is specific to the Microsoft keyboard drivers. Several of the translation tables are associative tables that map a virtual key code or a combination of a virtual key code and some other value (e.g., shift state) into an ASCII character. All such tables in these drivers are assembled using special macros, defined in TRANS.INC, that arrange the tables so that the first column of the table is a single vector, which may be searched with a single-string scan instruction.

Table	Description
keyTrTab	This table maps hardware scan codes to Windows virtual key codes. Some entries in this table in TABS.ASM may be overlaid when a keyboard DLL is loaded.
USTransPatch	This is a copy of part of **keyTrTab**, which is used to overlay **keyTrTab** if an error occurs while changing keyboard DLLs.

Most of the remaining tables are either in a discardable segment in the driver (if no DLL is loaded) or in a DLL. If they are in the driver, functions such as **ToAscii()** that are using them make sure that the table segment is loaded by calling **GetTableSeg()**.

The tables in the driver are for the US keyboard. Several of the following tables are blank for the US keyboard since it has no dead keys or CTRL+ALT keys.

In the DLL versions of these tables, there is often padding (i.e., zeros) at the end of a table to allow for overlaying the tables.

The USA tables are available in the TAB4.INC file, which is included with the sample source code. These tables for the USA enhanced keyboard are duplicated in the KBDUS.ASM file.

Table	Description
AscTranVK, AscTran	This associative table translates virtual key codes to ANSI for unshifted and shifted key combinations. **AscTranVK** is a byte array of virtual key codes; **AscTran** is a WORD array of their translations. In each WORD, the low byte is the unshifted translation, and the high byte is the shifted translation. If the virtual key code is one for a letter (A..Z), this table is bypassed.
AscControlVK, AscControl	This associative table translates virtual key code + CTRL to a control character, when the virtual key code is not one for a letter. For letters, the table is not used.
AscCtlAltVK, AscCtlAlt	This associative table translates virtual key code + CTRL + ALT to an ANSI character. This is empty for the US keyboard.
AscShCtlAltVK, AscShCtlAlt	This associative table translates virtual key code + CTRL + ALT + SHIFT to an ANSI character. This is empty for the US keyboard.
CapitalTable	This is a list of the virtual keys for which the CAPS LOCK key is effective, in addition to the letter keys (VK_A .. VK_Z). This is empty for the US keyboard.
SGCapsVK, SGTrans	This table has entries only in the DLL for Swiss-German keyboards. It handles a special case where keys with SHIFT LOCK are translated differently from shifted keys.
Morto, MortoCode	This table is searched to check if a particular virtual key code and shift combination is for a dead key. If it is, the dead key value is returned by **ToAscii()**, with a negative character count. This is empty for the US keyboard.
DeadKeyCode, DeadChar	This associative table translates a combination of a dead key (usually an accent character) and a letter into an accented letter. This table is accessed when the key *after* a dead key is struck. If a translation is not found in this table, **ToAscii()** will return two characters in its output buffer: the dead key plus the second character. This table is empty for the US keyboard.

8.6 Keyboard DLL

The KBDxx.ASM tables file contains data in two segments: the CODE segment, which is load-on-call and disposable, and the DATA segment, which is fixed. Data from the CODE segment of the DLL is used to overlay the tables in the DATA segment of the driver. Data in the DATA segment of the DLL is not copied; instead, a pointer in the driver is set to the DLL's DATA segment address.

The initial values in the tables in the DLL's DATA segment are for Enhanced (type 4) keyboards. For other keyboard types, the tables in the DLL's DATA segment must be patched or overlaid from tables in the CODE segment.

The copying or overlaying is done by the **GetKbdTable()** DLL function, which is called from the driver after the driver loads the library. Once **GetKbdTable()** is called, the driver accesses the DLL's fixed DATA segment directly. After initialization, the tables in the CODE segment of the DLL are no longer used.

A header, containing offsets and sizes of the various tables in the DLL's DATA segment, is always copied to the driver no matter what type of keyboard is installed.

Some DLLs contain the function **GetKeyString()**. This function is called by the keyboard's **GetKeyNameText()** function (see the *Microsoft Windows Software Development Kit* for a description of this function) to obtain key name strings in the language appropriate to the keyboard. (See Section 8.10, "Functions Reference," for more details on **GetKbdTable()** and **GetKeyString()**.)

8.7 SYSTEM.INI Keyboard Information

With Windows 3.0, some keyboard information is now set in the SYSTEM.INI file by the Setup program when Windows is first installed. Some of the values, though, may be changed by the Windows part of Setup during runtime. They appear in the following order in the SYSTEM.INI file:

```
[keyboard]

type = 4                     ;Enhanced keyboard. This value overrides any
                             ;type the keyboard driver can determine. This
                             ;number generally defines a keyboard layout and
                             ;is passed to the DLL initialization code. If
                             ;you want the driver to select this value(IBM-
                             ;compatibles), leave this blank. (1 is XT,
                             ;3 is AT, 4 is Enhanced)

subtype = 0                  ;Used only for Olivetti/AT&T. Only affects
                             ;anything now for one AT&T 6300+ keyboard, [but
                             ;may be useful for Hewlett-Packard for
                             ;distinguishing Vectra/non-Vectra keyboards.

keyboard.dll = KBDDA.DLL     ;Keyboard layout (Danish keyboard in this
                             ;example). May be omitted if you have a USA
                             ;XT (83-key), AT (84-key), or Enhanced
                             ;(101-key) keyboard. [This means you no
                             ;longer copy the keyboard DLL to KBDDLL.MOD.]

oemansi.bin = XLATNO.BIN     ;Danish/Norwegian code page translation. This
```

```
;setting is only for NO/DA, PO, ES, CA code
;pages now. Must be blank for code page 437
;(normal IBM).
;(Note:  Another SYSTEM.INI or WIN.INI setting
;must be made to select the corresponding OEM
;font for real-mode Windows or Windows/386.)
```

8.8 Windows Virtual Key Codes

This section describes the Windows virtual key codes that are generated by the hardware interrupt function in the keyboard driver. The translation of OEM scan codes to virtual key codes is intended to provide a hardware-independent interface to Windows.

The virtual key code set consists of 256 byte values in the range 0 to 255. Most, but not all, of the values used by standard drivers are in the range 0 to 127. There is a loose distinction between "standard" keys, which do not vary much, and "OEM" keys, which do vary from keyboard to keyboard.

Every OEM keyboard driver must provide the following virtual keys. Shift keys must be available in the combinations Unshifted, SHIFT, CTRL, and CTRL+ALT.

Type	Virtual Key Code	Description
Shift keys	VK_SHIFT	Either SHIFT key
	VK_CONTROL	The CTRL key
	VK_MENU	The ALT key
Alphabetic keys	VK_A..VK_Z	"A".."Z"
Numeric keys	VK_0..VK_9	The numeric keys at the top of the keyboard.
Cursor and editing keys	VK_UP	
	VK_DOWN	
	VK_LEFT	
	VK_RIGHT	
	On most keyboards, the following group of virtual key codes is generated on the numeric keys *only* if NUMLOCK is OFF:	
	VK_INSERT	
	VK_DELETE	
	VK_HOME	
	VK_END,	
	VK_PRIOR	The PAGE UP key
	VK_NEXT	The PAGE DOWN key
Required function keys	VK_1..VK_10	F1..F10

Type	Virtual Key Code	Description
Other keys	VK_ESCAPE	The ESC key
	VK_TAB	The TAB key
	VK_SPACE	The SPACE key
	VK_SNAPSHOT	The PRINTSCREEN key
	VK_BACK	The BACKSPACE key

All the above virtual key codes must be generated for full Windows functionality.

Keyboards commonly contain various "lock" keys, such as VK_CAPITAL and VK_NUMLOCK. If a keyboard driver generates ANSI characters on the numeric key pad using ALT + numeric-pad keys, it must do this translation only if NUMLOCK is ON. Also, care must be taken (on IBM-compatible keyboards) that the cursor and editing keys on extended keyboards do *not* produce this translation.

If a keyboard has a numeric pad, the numeric keys will frequently be used as cursor-control and editing keys if NUMLOCK is OFF. If NUMLOCK is ON, the virtual keycodes VK_NUMPAD0..VK_NUMPAD9 are used for the digits. Keyboards with a DELETE key that also generates the decimal point (period or comma) use VK_DELETE and VK_DECIMAL to distinguish the two uses of the key.

Other keys may vary from keyboard to keyboard. The following set of virtual key codes is generally used for punctuation keys, accented letter keys, and dead keys in the main section of a keyboard:

```
VK_OEM_1 .. VK_OEM_8
VK_OEM_102
VK_OEM_PLUS, VK_OEM_MINUS, VK_OEM_COMMA, VK_OEM_PERIOD
```

Applications that send characters to other Windows applications by sending WM_KEYDOWN and WM_KEYUP messages are expected to get the appropriate virtual key codes to send by using the **VkKeyScan()** function. They must not assume any fixed translation of the VK_OEM_* keys.

If a keyboard has more than 16 function keys, we recommend you use the virtual key codes in the range VK_OEM_17..VK_OEM_24.

The mouse button virtual key codes (VK_LBUTTON, VK_RBUTTON, VK_MBUTTON) are generated internally by Windows and are never generated by keyboard or mouse drivers.

Keyboard drivers should *not* generate VK_EXECUTE or VK_SEPARATER.

The following list includes the virtual key codes that are defined for Windows. VK_SNAPSHOT is new for Windows 3.0. This information is also in the VKWIN.INC and VKOEM.INC files in the keyboard driver sources.

```
; values <80H. 0, 0ffh can't be used.

VK_LBUTTON      = 01H   ; left mouse button
VK_RBUTTON      = 02H   ; right mouse button
```

```
VK_CANCEL       = Ø3H   ; used for control-break processing
VK_MBUTTON      = Ø4H   ; middle mouse button (3-button mouse).

; 4..7 undefined

VK_BACK         = Ø8H
VK_TAB          = Ø9H

; Øah .. Øbh undefined

VK_CLEAR        = ØcH
VK_RETURN       = ØdH

VK_SHIFT        = 1ØH
VK_CONTROL      = 11H
VK_MENU         = 12H
VK_PAUSE        = 13H
VK_CAPITAL      = 14H
                ; 15h..1ah undefined
VK_ESCAPE       = 1bH
                ; 1ch..1fh undefined
VK_SPACE        = 2ØH
VK_PRIOR        = 21H   ; page up
VK_NEXT         = 22H   ; page down
VK_END          = 23H
VK_HOME         = 24H
VK_LEFT         = 25H
VK_UP           = 26H
VK_RIGHT        = 27H
VK_DOWN         = 28H
VK_SELECT       = 29H
VK_PRINT        = 2aH   ; only used by Nokia..
VK_EXECUTE      = 2bH   ; never used
VK_SNAPSHOT     = 2ch   ; PRINTSCREEN key starting with 3.Ø Windows..
VK_INSERT       = 2dH
VK_DELETE       = 2eH
VK_HELP         = 2fH
VK_Ø            = 3ØH
VK_1            = 31H
VK_2            = 32H
VK_3            = 33H
VK_4            = 34H
VK_5            = 35H
VK_6            = 36H
VK_7            = 37H
VK_8            = 38H
VK_9            = 39H
                ; 4Øh
VK_A            = 41H
VK_B            = 42H
VK_C            = 43H
VK_D            = 44H
VK_E            = 45H
```

```
VK_F            = 46H
VK_G            = 47H
VK_H            = 48H
VK_I            = 49H
VK_J            = 4AH
VK_K            = 4BH
VK_L            = 4CH
VK_M            = 4DH
VK_N            = 4EH
VK_O            = 4FH
VK_P            = 50H
VK_Q            = 51H
VK_R            = 52H
VK_S            = 53H
VK_T            = 54H
VK_U            = 55H
VK_V            = 56H
VK_W            = 57H
VK_X            = 58H
VK_Y            = 59H
VK_Z            = 5AH
                ; 5bh..5fh undefined
VK_NUMPAD0      = 60H
VK_NUMPAD1      = 61H
VK_NUMPAD2      = 62H
VK_NUMPAD3      = 63H
VK_NUMPAD4      = 64H
VK_NUMPAD5      = 65H
VK_NUMPAD6      = 66H
VK_NUMPAD7      = 67H
VK_NUMPAD8      = 68H
VK_NUMPAD9      = 69H
VK_MULTIPLY     = 6AH
VK_ADD          = 6BH
VK_SEPARATER    = 6CH   ; never generated by keyboard driver
VK_SUBTRACT     = 6DH
VK_DECIMAL      = 6EH
VK_DIVIDE       = 6FH

VK_F1           = 70H
VK_F2           = 71H
VK_F3           = 72H
VK_F4           = 73H
VK_F5           = 74H
VK_F6           = 75H
VK_F7           = 76H
VK_F8           = 77H
VK_F9           = 78H
VK_F10          = 79H
VK_F11          = 7aH
VK_F12          = 7bH
VK_F13          = 7cH
VK_F14          = 7dH
```

```
VK_F15          = 7eH
VK_F16          = 7fH

; values >= 80H
; The codes VK_OEM_1 through VK_OEM_8 apply to all keyboards.
; VK_OEM_102 applys to non-USA Enhanced keyboards.
; Other VK codes in this range may only apply to OEM special keyboards.

; This group is used for Nokia (Ericsson) keyboards.
VK_OEM_F17      = 80H   ; Nokia
VK_OEM_F18      = 81H   ; Nokia
VK_OEM_F19      = 82H   ; Nokia
VK_OEM_F20      = 83H   ; Nokia
VK_OEM_F21      = 84H   ; Nokia
VK_OEM_F22      = 85H   ; Nokia
VK_OEM_F23      = 86H   ; Nokia
VK_OEM_F24      = 87H   ; Nokia

; 88h..8Fh unassigned

; These apply to ALL keyboards.
VK_NUMLOCK      = 090H  ; NumLock on ALL keyboards
VK_OEM_SCROLL   = 091H  ; Scroll Lock on ALL keyboards

; 92h..B9h unassigned

; This group is used for punctuation keys on ALL keyboards.
VK_OEM_1        = 0BAH
VK_OEM_PLUS     = 0BBH
VK_OEM_COMMA    = 0BCH
VK_OEM_MINUS    = 0BDH
VK_OEM_PERIOD   = 0BEH
VK_OEM_2        = 0BFH
VK_OEM_3        = 0C0H

; C1h..DAh unassigned

; Punctuation continued.. (ALL keyboards)
VK_OEM_4        = 0DBH
VK_OEM_5        = 0DCH
VK_OEM_6        = 0DDH
VK_OEM_7        = 0DEH
VK_OEM_8        = 0DFH

; keycodes for Olivetti 'ICO' extended keyboard
; used internally, not seen by applications
VK_F17          = 0E0H  ; F17 key on ICO
VK_F18          = 0E1H  ; F18 key on ICO

; IBM-compatible 102 Enhanced keyboard (non-USA).
VK_OEM_102      = 0E2H  ; "<>" or "\|" on Enhanced 102-key

; More Olivetti ICO keyboard codes.
```

```
VK_ICO_HELP      = 0E3H  ; Help key on ICO.
VK_ICO_00        = 0E4H  ; 00 key on ICO. Appears internally in driver
                         ; tables, never appears in Windows messages.

; E5h unassigned

; More Olivetti 'ICO'
VK_ICO_CLEAR     = 0E6H  ; ICO keyboard only.

; E7h .. E8h unassigned

; More Nokia/Ericsson definitions
VK_OEM_RESET     = 0E9H  ; Nokia
VK_OEM_JUMP      = 0EAH  ; Nokia
VK_OEM_PA1       = 0EBH  ; Nokia
VK_OEM_PA2       = 0ECH  ; Nokia
VK_OEM_PA3       = 0EDH  ; Nokia
VK_OEM_WSCTRL    = 0EEH  ; Nokia
VK_OEM_CUSEL     = 0EFH  ; Nokia
VK_OEM_ATTN      = 0F0H  ; Nokia
VK_OEM_FINNISH   = 0F1H  ; Nokia
VK_OEM_COPY      = 0F2H  ; Nokia
VK_OEM_AUTO      = 0F3H  ; Nokia
VK_OEM_ENLW      = 0F4H  ; Nokia
VK_OEM_BACKTAB   = 0F5H  ; Nokia

; F6h..FEh unassigned.

; FF can't be used.
```

8.9 A Checklist for Modifying a 3.0 Keyboard Driver

If you already have a Windows 2.x keyboard driver, we recommend that you follow the steps in this checklist to modify the sample Windows 3.0 keyboard driver, rather than your old Windows 2.x keyboard driver. The result will be more compatible with Windows 3.0.

Also, you should always use the latest available version of the DDK sources.

- ❑ Use IFDEFs for the parts that you modify.

  If you can assemble the driver with your IFDEFs OFF and generate the "normal" KBD.DRV driver, then it will be easier to integrate support for your hardware into Microsoft's sources in the future.

- ❑ Identify or select the keyboard type.
  - ☐ You should avoid doing any I/O during initialization to determine the keyboard type. This should be done in Setup, not in the keyboard driver.

- ☐ There are three things that are defined in SYSTEM.INI on the basis of the keyboard hardware and read with **GetPrivateProfileString()** or **GetPrivateProfileInt()** in TABS.ASM:
- ■ The name of the keyboard driver in the "KEYBOARD.DRV =" statement.
- ■ The keyboard type in the "TYPE =" statement.

  This is used to select among several different keyboard layouts for a particular country, along with some other attributes. This value is also used by the keyboard DLL to determine what patches to make in the keyboard translation tables.
- ■ The keyboard subtype in the "SUBTYPE =" statement.

  This may be used to differentiate among keyboards that have the same layout (i.e., the same basic translation table) but that may have some special attributes. The meaning of the subtype will depend on the particular OEM driver.
- ☐ The automatic detection of the keyboard type is mainly done for fairly generic systems. However, it is possible for a driver developer to modify the SETUP.INF file to provide manual selection of a keyboard driver (and its type and subtype).
- ☐ If you read any data from the ROM BIOS, you *must* use the special segment selector __ROMBIOS, which corresponds to real address 0F0000H. See the code in INIT.ASM.

❏ Compare your old interrupt function (i.e., DATACOM.ASM in Windows 2.x and TRAP.ASM in Windows 3.0) with the one in the Windows 3.0 sample keyboard driver.

You will find that the basic flow of the Windows 3.0 function is quite similar to that of the one in the Windows 2.x keyboard driver. Simply make similar modifications in the Windows 3.0 TRAP.ASM. The table **keyTrTab[]** in TABS.ASM is used to translate scan codes to virtual key codes. You will see some existing IFDEFs for special keyboards in this file.

❏ Integrate special code for your keyboard in TOASCII.ASM by moving it from the Windows 2.x to the Windows 3.0 version of the keyboard driver.

Notice that the **Inquire()** function (in ENABLE.ASM) tells Windows how much space to allocate for **ToAscii()**'s state vector, which maintains state between successive calls to **ToAscii()**.

❏ Examine the INIT code and the **Enable()** and **Disable()** functions for any special actions that need to be taken when Windows is loaded, when the keyboard driver is enabled, and when the keyboard driver is disabled.

Notice that the Windows keyboard and mouse drivers are disabled when switching from Windows to a full-screen MS-DOS application. The calls to the keyboard and mouse **Enable()** functions are always done in the reverse order from the calls to the respective **Disable()** functions. This is done to handle systems in which the keyboard and mouse must hook or unhook the same interrupts.

❏ If the repeat speed of your keyboard cannot be set by software, the **SetSpeed()** function may be omitted (but then the definition of the function *must* be omitted from the .DEF file).

8.10 Functions Reference

The following is an alphabetically organized reference section that includes descriptions of each of the functions listed in this chapter.

AnsiToOem(lpSrc, lpDst)

Description This function translates a string in the ANSI character set to a string in the OEM character set (which is code page 437 for most systems).

Parameter	Description
lpSrc	A long pointer to the input ANSI string.
lpDst	A long pointer to the output OEM string.

NOTE *lpSrc* and *lpDst* may be the same.

Return Value Undefined

Comments The default translation is for Code Page 437 (the standard USA IBM-PC character set).

Calling the **Enable()** function will cause the tables for this translation to be initialized. If the desired code page is not 437, they will be read from a file.

AnsiToOemBuff(lpSrc, lpDst, nCount)

Description This function performs the same translations as **AnsiToOem()**, but is used for translating fixed-length byte arrays, such as database records, which may contain NULL bytes.

Parameter	Description
lpSrc	A long pointer to the input ANSI string.
lpDst	A long pointer to the output OEM string.
nCount	The byte count.

NOTE *lpSrc* and *lpDst* may be the same.

Return Value Undefined

Disable()

Description This function restores the MS-DOS keyboard interrupt vector when exiting Windows.

Parameters None

Return Value None

Comments The **Disable()** function restores the keyboard hardware interrupt vector to the keyboard handler that was installed previously.

Disable() is called when exiting Windows and before starting or switching to a full-screen MS-DOS application running under real or standard-mode Windows.

When **Disable()** is called, the MS-DOS keyboard flags (at 40H:17H in most IBM-compatible systems) must reflect the state of the SHIFT LOCK, NUMLOCK, and SCROLL LOCK keys. (Normally, the Windows hardware interrupt function will handle this.)

The keyboard and mouse **Enable()** functions are always called in the reverse order of the keyboard and mouse **Disable()** functions since some mouse drivers may hook into the keyboard interrupt.

Enable(eventProc, lpKeyState)

Description This function installs the hardware interrupt for Windows and performs the necessary initialization of the driver.

Parameter	Description
eventProc	The address (long) of the keyboard event procedure in USER. (See Section 8.4, "The Keyboard Interrupt Handler and Event Procedure Call," for more information on this function.)
lpKeyState	The address (long) of the 256-byte key state vector in USER. It is accessed here primarily to synchronize the entries for SHIFT LOCK and NUMLOCK with the current state maintained by the MS-DOS keyboard driver.

Return Value None

Comments The **Enable()** function must save the original hardware interrupt address in static memory. The driver should maintain such things as the shift and numeric lock state, and the state of indicator lights. Linkage with a keyboard DLL should be established (if one exists).

Enable() is called once when Windows is started up. It is also called when returning to Windows from a full-screen application for which the keyboard was disabled.

EnableKBSysReq(fWord)

Description This function is used for enabling and disabling the CTRL+ALT+SYSREQ trap to the debugger.

Parameter	Description
fWord	If nonzero, then an internal byte counter is incremented. If zero and the internal byte counter is nonzero, then the byte counter is decremented.

Return Value The value of the byte counter *after* any change.

Comments The initial value of the byte counter is zero, which means that the debugger trap is disabled. If the byte counter is nonzero, the trap to the debugger is enabled.

GetKBCodePage()

Description This function is used to determine which OEM/ANSI tables are loaded in the keyboard driver.

Parameters None

Return Value A code page (integer). This indicates which OEM/ANSI translation tables are loaded.

Comments If the OEMANSI.BIN file is in the Windows directory and the code page is not 437, the file will be read when Windows is booted and will overwrite the CP 437 OEM/ANSI translation tables in the keyboard driver. Common values are as follows:

Code Page	Description
437	Default (USA, most countries. No OEMANSI.BIN file.)
860	Portugal (OEMANSI.BIN = XLATPO.BIN)
863	French Canada (OEMANSI.BIN = XLATCA.BIN)
865	Norway/Denmark (OEMANSI.BIN = XLATNO.BIN)
850	International code page (OEMANSI.BIN = XLAT850.BIN)

If the code page is not 437, then the Setup program must copy one of the XLAT*.BIN files to OEMANSI.BIN. If the code page is 437, Setup must delete OEMANSI.BIN if it exists.

Each one of these .BIN files contains the following:

(word)	Number of bytes that follow (must be 258)
(word)	Code page number
(256 bytes)	OEM/ANSI translation tables

GetKbdTable(iType, lpKeyTrTab, lpHeader)

Description This DLL function is called from the driver to do copying or overlaying.

Parameter	Description
iType	Keyboard type 1:XT, M24 83-key 2:Olivetti M24 102-key "ICO" 3:AT 84- or 86-key 4:RT Enhanced 101- or 102-key 5:Nokia (Ericsson) 1050, 1051 6:Nokia 9140 keyboard
lpKeyTrTab	A FAR pointer to the **KeyTrTab()** in the driver.
lpHeader	A FAR pointer to the header for the translation tables.

Return Value This returns a FAR pointer to this DLL's DATA segment in **DX:AX** (where **AX** is 0).

Comments The tables are patched, and various data are copied to the driver.

This function is in a load-on-call disposable segment, which means its memory may be reclaimed. The DATA segment is fixed.

GetKeyboardType(wWhich)

Description This function enables an application to determine the type of keyboard that is attached.

Parameter	Description
wWhich	Selects whether the basic type or the OEM subtype is returned.

Return Value

If *Which* = 0, it returns keyboard type (1..6)

1: IBM PC, XT or compatible (83-key)
2: Olivetti M24 "ICO" (102-key)
3: IBM AT (84 keys) or similar
4: IBM Enhanced (101 or 102 keys)
5: Nokia 1050, etc.
6: Nokia 9140, etc.

If *Which* = 1, it may return OEM-dependent subtype information, which should be nonzero.

GetKeyString(int nString, LPSTR lpStringOut)

Description

This function is called by the keyboard's **GetKeyNameText()** function to obtain key name strings in the language appropriate to the keyboard.

Parameter	Description
nString	The index to a list of strings.
lpStringOut	The selected string is copied to this address.

Return Value

The size of the string (exclusive of NULL termination) is returned in AX.

Comments

GetKeyNameText() checks for this function in the DLL to obtain the localized version of the key name.

GetTableSeg()

Description

This function is for internal use in the keyboard driver only. It is used when a DLL does not exist.

Parameters

None

Return Value

Paragraph address of a discardable keyboard table segment.

Comments

This function is called, only when a keyboard DLL does not exist, to load the segment containing the default keyboard translation tables and return its address.

This function is to be called *only* within the keyboard driver. It is not part of the Windows API.

Inquire(lpKBInfo)

Description This function fills in the data structure with information about the keyboard hardware.

Parameter	Description
lpKBInfo	A long pointer to the KBINFO data structure.

Return Value The number of bytes transferred is returned in AX.

MapVirtualKey(wCode, wMapType)

Description This function is intended to be used by the PIF Editor and Windows to get information about keyboard mapping.

Parameter	Description
wCode	The input scan code or VK code
wMapType	Selects mapping as follows: *wMapType* = 0: Map VK to scan code *wMapType* = 1: Map scan code to VK *wMapType* = 2: Map VK to ASCII

Return Value AX = Mapped value

Returns 0 if mapping cannot be performed (scan codes and VK codes are always > 0).

Comments For the first 2 mappings, the scan code to VK code translation table is examined to determine the translations. Some valid codes may not be translated in this table.

For the 3rd type of mapping, uppercase letters A..Z are returned for VK_A..VK_Z. ASCII digits are returned for the top-row numeric keys VK_0..VK_9. For punctuation and dead keys in the main part of the keyboard, the unshifted character will be returned.

NewTable()

Description This function determines the keyboard type and tries to load a keyboard DLL.

Parameters None

Return Value None

Comments The following entries in WIN.INI are accessed:

```
[keyboard]
type = 4; 1..6.
OliType = 0     ; 0 for all but Olivetti systems
```

Internal keyboard type variables are set. If a keyboard DLL is found and loaded, its handle is saved; this is subsequently used by the driver to determine whether to use the tables in the movable TABS segment or the DLL tables.

This function determines the type of keyboard that is installed by reading WIN.INI values. If they are not found, the default type is determined by the type of system for which the particular driver is designed.

OEMKeyScan(wOemChar)

Description This function maps OEM ASCII codes (0..0FFH) into OEM scan codes and shift states. This function provides information that enables a program to send OEM text to another program by simulating keyboard input. Windows, when running in 386 Enhanced mode, uses it specifically for this purpose.

Parameters It is passed the OEM ASCII code as *wOemChar*.

Return Value AX = scan code
DX = shift state
  bit 2 = CTRL + SHIFT depressed
  bit 1 = either SHIFT depressed

If the character is not defined in the tables, -1 is returned in both DX and AX.

Comments This function does not provide translations for characters that require CTRL + ALT or dead keys. Characters that are not translated by this function must be copied by simulating ALT + numeric-pad input.

This function calls **VkKeyScan()** in the Windows 3.0 drivers.

OemToAnsi(lpSrc, lpDst)

Description This function translates a string in the current OEM character set to ANSI.

Parameter	Description

lpSrc	A long pointer to the input ANSI string.
lpDst	A long pointer to the output OEM string.

NOTE *lpSrc* and *lpDst* may be the same.

Return Value Undefined

Comments The default translation is for code page 437 (the standard USA IBM-PC character set).

Calling the **Enable()** function will cause the tables for this translation to be initialized. If the desired code page is not 437, they will be read from a file.

OemToAnsiBuff(lpSRC, lpDst, nCount)

Description This function performs the same translations as **OemToAnsi()**, but is used for translating fixed-length byte arrays, such as database records, which may contain NULL bytes.

Parameter	Description
lpSrc	A long pointer to the input ANSI string.
lpDst	A long pointer to the output OEM string.
nCount	The byte count.

NOTE *lpSrc* and *lpDst* may be the same.

Return Value Undefined

ScreenSwitchEnable(wEnable)

Description This function enables or disables screen switching under OS/2. It does not apply to running under MS-DOS.

Parameter	Description
wEnable	Zero for disable, non-zero for enable.

Return Value Undefined

Comments

This function is called by the display driver to inform the keyboard driver that the display driver is in a critical section and, therefore, it should ignore all OS/2 screen switches until the display driver leaves its critical section.

The parameter is saved as a flag that is tested in the interrupt function; if it is zero, the OS/2 key combination for screen switching is ignored.

On entry, screen switches are enabled.

SetSpeed(Rate)

Description

This function is used by Control Panel and by Windows initialization to set the repeat rate of the keyboard.

Parameter	Description
Rate	An integer value. The lowest 5 bits define the desired repeat rate. If *Rate* is -1, this function returns a value indicating speed-setting capability.

Return Value

If rate_of_speed = -1

AX = -1 if not speed capable
AX = 0 if yes capable

If rate_of_speed does not = -1

AX = speed actually set on keyboard
AX = -1 if unsuccessful

Comments

This function is used in conjunction with a Control Panel dialog for setting the keyboard repeat rate.

If the keyboard speed cannot be set from software, this function may be reduced to a stub which always returns -1. In that case, the Control Panel program will not display a menu selection for setting keyboard speed.

ToAscii(VirtKey, Scancode, lpKeyState, lpState, KeyFlags)

Description

ToAscii() translates the virtual key code passed to it, along with the current keyboard state, to an ANSI character.

Parameter	Description
VirtKey	The Microsoft Windows virtual key code. (WORD value)
Scancode	The "hardware" raw scan code originally passed to Windows when the hardware interrupt function called the keyboard event procedure in USER. The sign bit of this parameter is set if this is an UP key transition. (WORD value)
lpKeyState	The vector of key state flags maintained in USER. (See the following *Comments* section and Section 8.5.1, "Keyboard State Vector," for a description.) (long value)
lpState	The vector of data words. This is used mainly for the output of ANSI characters. (long value)
KeyFlags	The bit 0 flag's menu display. (WORD value)

Return Value

The value returned in AX indicates the number of characters returned (1 byte per WORD) in the state block (pointed to by *lpState*). Negative values indicate dead keys. Normally, the following values occur:

Parameter	Description
2	Two characters are returned (mainly an accent and a dead key character, when a dead key cannot be translated otherwise).
1	One ANSI character is returned.
0	This virtual key code has no translation (for the current state of shift keys, etc).
–1	This key is a dead key. The character returned is normally an ANSI accent character representing the dead key.

Comments

ToAscii() is called mainly whenever **TranslateMessage()** is called to translate a virtual key code (e.g., for WM_KEYDOWN messages).

The given parameters to **ToAscii()** are not necessarily sufficient to translate the virtual key code. This is because a previous dead key is stored internally in the driver. Also, the MS-DOS shift state byte is accessed by **ToAscii()**.

ToAscii() is responsible for maintaining the state of the keyboard LED indicator lights. For most AT-compatible systems, this is done by making a ROM BIOS interrupt 16H call; for others, I/O must be done directly to the keyboard.

ToAscii() also has a special case which, if it is called with the virtual key code = 0, will only set the keyboard lights according to the state of the appropriate entries in the keyboard state vector. *This function is intended to be called from the USER function* ***SetKey-***

boardState() and not used directly by applications. The *Scancode* parameter is ignored. The vector pointed to by *lpState* should be different from the one used by normal ToAscii() calls in USER and at least 4 bytes long.

Most translations are made on the basis of the Windows virtual key code. However, the *Scancode* parameter's sign bit is used to distinguish key depressions (sign cleared) from key releases (sign set). Also, the scan code is used in the translation of ALT + number key translations.

lpKeyState points to a 256-byte vector indexed by the virtual key code. In each byte, the high-order bit indicates the state of the key and the low-order bit is toggled each time the key is pressed. The CAPSLOCK key is handled in a special manner for some European keyboards.

VkKeyScan(wChar)

Description This function translates an ANSI character code into a virtual key code and a shift state. It is intended for applications that send text to other applications by simulating keyboard input.

Parameter	Description
wChar	An ANSI character

Return Value AL = Windows virtual key code

AH = Shift state

0	No shift
1	Character is shifted
2	Character is CTRL character
6	Character is CTRL + ALT
7	Character is SHIFT + CTRL + ALT
3,4,5	These combinations never occur.

If no key with this translation is found, -1 is returned in AX.

Comments Virtual key codes applying to the numeric pad (VK_NUMPAD0..VK_DIVIDE) are not returned. This is done to force a translation for the main keyboard, if it exists.

Chapter 9 Miscellaneous Drivers

This chapter provides some basic information on several device drivers for which source code is included with the *Microsoft Windows Device Development Kit* (DDK). We recommend that, if possible, you simply modify these drivers instead of writing your own.

9.1 Updating 2.x Drivers to 3.0

None of the new feature enhancements in Windows 3.0 affects communications or sound drivers. **(What about being bi-modal for real and protected modes?)**

However, we recommend that you rebuild your 2.x driver with the new Windows 3.0 building tools provided in the *Software Development Kit* (SDK). You should also thoroughly test your driver under Windows 3.0, and especially while running in protected mode. This will ensure full compatibility with Windows 3.0.

The supplied communications driver source code has also been enhanced from earlier versions of Windows. It now supports the COMM3 and COMM4 ports.

9.2 Communications and Sound Drivers

The Windows communications driver has entries that assign and deassign serial device instances to ports, enable and disable interrupt handlers for input from assigned serial devices, and send and receive characters to assigned serial output devices. The data transmission protocol is communicated in a Device Control Block (DCB) when the serial device is assigned.

The Windows sound driver generates the sounds specified by a parameter block. The block is a series of notes whose first word specifies the number of notes. Each subsequent pair of words defines the duration of the note (in milliseconds) and the frequency of the note.

In most cases, there will be no need to modify the drivers shipped with Windows. However, if you need to customize them, the source for the standard versions is provided on one of the disks included with the DDK. We recommend that you modify these instead of writing your own.

The APIs that these drivers support are fully defined in the SDK.

The Device Control Block (DCB) structure, however, is provided in this section for your convenience.

9.2.1 DCB - Device Control Block Structure

RS-232 configuration parameters are communicated in a Device Control Block (DCB). The Device Control Block structure is defined below; the C-structure definition is given.

```
typedef struct {
        char   Id;                     /* Internal device ID                */
        ushort Baudrate;               /* Operating speed                   */
        char   ByteSize;               /* Transmit/receive byte size        */
        char   Parity;                 /* Ø,1,2,3, or 4                     */
        char   StopBits;               /* Number of stop bits               */
        ushort RlsTimeout;             /* Timeout for RLSD to be set        */
        ushort CtsTimeout;             /* Timeout for CTS to be set         */
        ushort Dsrtimeout;             /* Timeout for DSR to be set         */
        ushort fBinary: 1;             /* Binary mode flag                  */
        ushort fRtsDisable: 1;         /* Disable RTS                       */
        ushort fParity: 1;             /* Enable parity checking            */
        ushort fDummy: 5;
        ushort fOutX: 1;               /* Enable output X-ON/X-OFF          */
        ushort fInX: 1;                /* Enable input X-ON/X-OFF           */
        ushort fPeChar: 1;             /* Enable parity error replacement   */
        ushort fNull: 1;               /* Enable null stripping             */
        ushort fChEvt: 1;              /* Enable Rx character event         */
        ushort fDtrflow: 1;            /* Enable DTR flow control           */
        ushort fRtsflow: 1;            /* Enable RTS flow control           */
        ushort fDummy2: 1;
        char   XonChar;                /* Transmit/receive X-ON character   */
        char   XoffChar;               /* Transmit/receive X-OFF character  */
        ushort XonLim;                 /* Transmit X-ON threshold           */
        ushort XoffLim;                /* Transmit X-OFF threshold          */
        char   PeChar;                 /* Parity error replacement character */
        char   EofChar;                /* End-of-input character            */
        char   EvtChar;                /* Event-generating character        */
        ushort TxDelay;                /* Amount of time between characters */
        } DCB;
```

The fields in the DCB data structure have the following meanings:

Field	Description
Id	Device ID byte (COM1 = 0, COM2 = 1, etc.) This is also the value returned by **cOpen**, when successful.
Baudrate	Operating speed; any baud rate supported by the hardware.
Bytesize	Transmitting and receiving byte size; normally in the range 4-8.
Parity	Parity setting, as follows:
	0 None
	1 Odd

Field	Description
	2 Even
	3 Mark
	4 Space
Stopbits	Number of stop bits, as follows:
	0 1 stop bit
	1 1.5 stop bits
	2 2 stop bits
RlsTimeout	Amount of time, in milliseconds, to wait for RLSD (receiving line signal detect) to become high. RLSD flow control can be achieved by specifying infinite timeout. (0xFFFF)
CtsTimeout	Amount of time, in milliseconds, to wait for CTS (clear to send) to become high. CTS flow control can be achieved by specifying infinite timeout. (0xFFFF)
DsrTimeout	Amount of time, in milliseconds, to wait for DSR (data set ready) to become high. DSR flow control can be achieved by specifying infinite timeout. (0xFFFF)
fBinary	Binary mode flag (0 is ASCII mode, 1 is binary). In ASCII mode, EOFCHAR is recognized and remembered as end of received data.
fRtsDisable	If set, disables RTS line for as long as this device is open. Normally, RTS is enabled when the device is opened and disabled when closed.
fParity	If set, enables parity checking.
fOutX	If set, indicates that X-ON/X-OFF flow control is to be used during transmission. The transmitter will halt if an X-OFF character is received and start again when an X-ON character is received.
fInX	If set, indicates that X-ON/X-OFF flow control is to be used during reception. An X-OFF character will be transmitted when the receive queue comes within 10 characters of being full, after which an X-ON character will be transmitted when the queue comes within 10 characters of being empty.
fPeChar	If set, indicates that characters received with parity errors are to be replaced with the specified PECHAR.

Field	Description
fNull	If set, received NULL characters are to be discarded.
fChEvt	If set, indicates that the reception of EVTCHAR is to be flagged as an event.
fDtrFlow	If set, indicates that the DTR signal is to be used for receive flow control.
fRtsFlow	If set, indicates that the RTS signal is to be used for receive flow control.
XonChar	X-ON character for both transmit and receive.
XoffChar	X-OFF character for both transmit and receive.
XonLim	Threshold value for receive queue. If the number of characters in the receive queue drops below XONLIM and an X-OFF character has been sent, an X-ON character is sent (if X-ON flow control is enabled) and DTR is set (if enabled).
XoffLim	Threshold value for send queue. When the number of characters in the receive queue exceeds this value, an X-OFF character is sent (if X-OFF flow control is enabled) and DTR is dropped (if enabled).
PeChar	Character to be used as replacement when a parity error occurs.
EofChar	Character that signals the end of the input.
EvtChar	Character that triggers an event flag.
TxDelay	Minimum amount of time that must pass between transmission of characters.

9.3 Mouse Drivers

The Windows mouse driver provides the following:

- Initialization and termination functions for the mouse
- Information about whether or not a mouse is connected to the system
- The number of buttons on the mouse
- The rate at which it generates interrupts
- The threshold for acceleration of horizontal and vertical motion
- The resolution of the screen

The mouse hardware interrupt handler is called whenever the mouse generates an interrupt. The interrupt handler must place state flags in AX. These flags include information about whether or not the mouse has moved and about button transitions.

The low-order bit is set if there was movement since the last interrupt; otherwise, it is cleared. Bits 2-15 in AX specify the state of the buttons, which are numbered 1-N for this purpose. Bit 2N is set if button N is depressed, bit 2N+1 is set if button N is up. **DX** is set to the number of buttons.

If there has been mouse motion (low-order bit of AX is set), **BX** and CX hold the integer values of motion since the last mouse interrupt was generated for X and Y, respectively. When this data is in place, the interrupt handler calls an event-handling procedure supplied by Microsoft Windows.

9.3.1 Mouse Functions

The following three functions are specific to the mouse driver.

Inquire(lpMOUSEINFO)

This primitive returns information about the mouse hardware.

The *lpMOUSEINFO* parameter is a long pointer to a data structure of type MOUSEINFO. The structure contains information about the mouse hardware that is present, the number of buttons on the mouse, and the rate at which the mouse can issue interrupts.

On return, AX holds the number of bytes actually written into the data structure.

Enable (lpEventProc)

This primitive sets up the mouse to call the procedure whose address was passed for each mouse interrupt.

The *lpEventProc* parameter is a long pointer to the procedure that is to be called for each mouse interrupt.

Disable( )

This primitive removes the existing interrupt procedure from the mouse driver. After a call to this procedure, there will be no support for interrupts from the mouse driver.

9.3.2 Addition to MOUSE.DEF

A call has been added to MOUSE.DEF for enhanced Windows initialization support. This call,

```
MouseGetIntVect @4
```

returns the mouse's vector number under DOS. If there is no physical mouse, it returns -1.

NOTE Under OS/2, some interrupt vectors are not shareable, including the mouse interrupt. To determine whether or not a mouse driver is already installed, you can do the following:

```
xor     AX,AX
int     33h
```

If the interrupt returns != 0, then a mouse driver is, indeed, installed, and you must use it. If it returns 0, you may install your driver.

If you are not operating under OS/2, you can still check for an installed mouse driver. However, you are not obliged to use it; you may replace it with your own.

9.3.3 MOUSEINFO - Mouse Hardware Characteristics Structure

The values of the fields in this structure should be set so that they correctly reflect the relationship between quadrature changes and pixel movement for the system's mouse and the usual display.

```
typedef struct {
        char   msExist;
        char   msRelative;
        short  msNumButtons;
        short  msRate;
        short  msXThreshold;
        short  msYThreshold;
        short  msXRes;
        short  msYRes;
        } MOUSEINFO;
```

The following is a description of the fields in this structure:

Field	Description
msExist	Nonzero if the device initialization code was able to find and initialize a mouse device.
msRelative	Nonzero if the mouse device is set to return coordinates relative to the previous position. It is zero if the mouse returns absolute coordinates.
msNumButtons	Identifies how many buttons are on the installed mouse. For the IBM PC with a Microsoft Mouse, this field is set to 2.
msRate	Specifies the maximum number of hardware interrupts per second that the mouse can generate. For the IBM PC with the bus version of the Microsoft Mouse, this field is 34.
msXThreshold	Specifies the mouse acceleration threshold for horizontal motion. The threshold specifies the mickey per second rate at which the mouse must travel before the pixel per mickey rate of the mouse cursor is accelerated.

Field	**Description**
msYThreshold	Specifies the mouse acceleration threshold for vertical motion. The threshold specifies the mickey per second rate at which the mouse must travel before the pixel per mickey rate of the mouse cursor is accelerated.
msXRes	Reserved.
msYRes	Reserved.

9.3.4 CURSORINFO - Cursor Information Data Structure

This data structure contains information about the system display's cursor module.

```
typedef struct {
        short  dpXRate;
        short  dpYRate;
        } CURSORINFO;
```

The fields in this data structure have the following meanings:

Field	**Description**
dpXRate	The horizontal mickey-to-pixel ratio for this display. For the IBM PC with a Microsoft Mouse, this is 1.
dpYRate	The vertical mickey-to-pixel ratio for this display. For the IBM PC with a Microsoft Mouse, this is 2.

Chapter 10 Common Functions

This chapter describes the common functions used by Microsoft Windows. The functions performed by a GDI device driver fall into the following groups:

- Control
- Environment
- Information
- Output
- Attribute
- Cursor

The following are the control functions (with their ordinal reference numbers) that Windows uses to initialize and disable the physical display and to control the various output operations. They are required for all device drivers. **(Lisa, true? Check the other lists also and give me required vs. optional for which drivers.)**

- **Control()** @ 3
- **Disable()** @
- **Enable()** @
- **WEP()** @

The following two environment functions (with their ordinal reference numbers) are part of GDI and can be used by device drivers to manage the printing environment for a given port. **DeviceMode** is called by applications and is required to be in the printer driver.

- **DeviceMode()** @ 13
- **GetEnvironment()** @
- **SetEnvironment()** @

The following information functions (with their ordinal reference numbers) pass information about the graphics peripheral to which this device driver is attached. This includes the physical characteristics (technology, size of output surface, resolution, colors, etc.) of the graphics peripheral, and information about the capabilities of the device driver. They also pass information about fonts or other data structures. These functions are required for **(Lisa, which?)** drivers.

- **ColorInfo()** @
- **DeviceBitmap()** @
- **EnumDFonts()** @ 6
- **EnumObj()** @
- **GetCharWidth()** @ 15

The following output functions (with their ordinal reference numbers) perform all the actual graphics operations on the display surface. Output is made to the device output surface or to a bitmap in memory. Most of these functions are required for display and printer drivers. Some are also used by **(Lisa, which?)** drivers. **FastBorder()** and **StretchBlt()** are optional display drivers.

NOTE The output functions must not overwrite the display cursor while it is still on the display screen. Each function must check for the location of the cursor and remove it from the screen if it is in any portion of the screen to be updated or read. This ensures that the cursor, if visible at all, is on the top level of the screen. i.e., "in front of" all the other items. The origin is the upper-left corner of the display surface. Notice that X increases when going to the right, and Y increases when going downward.

- **BitBlt()** @
- **ExtTextOut()** @
- **FastBorder()** @
- **Output()** @
- **Pixel()** @
- **SaveScreenBitmap()** @
- **ScanLR()** @ 12
- **StretchBlt()** @ 27
- **StrBlt()** @

The following display attribute functions (with their ordinal reference numbers) handle the creation of physical representations of attribute bundles suitable for the device. Those physical representations are the actual parameters to output primitive calls. These functions are required for **(Lisa, which?)** drivers.

- **RealizeObject()** @
- **SetAttribute()** @

The following cursor functions (with their ordinal reference numbers) allow the Original Equipment Manufacturer (OEM) to take advantage of any special cursor display hardware. The OEM is responsible for hiding the cursor, if necessary, when the screen display changes. These functions provide position and visibility control of the cursor, and the ability to specify the bitmap to be displayed as the cursor. *They are used for dedicated display modules only.* These functions are required for (**Lisa, which**) drivers.

NOTE A cursor consists of two monochrome bitmaps: an AND mask (applied first at the current screen position) and an XOR mask (applied after the AND mask).

- **CheckCursor()** @
- **Inquire()** @
- **MoveCursor()** @
- **SetCursor()** @ 102

Detailed descriptions of all the functions mentioned above follow this introductory section and are presented in alphabetical order.

BitBlt()

Syntax **BitBlt** (*lpDestDev*, *DestXOrg*, *DestYOrg*, *lpSrcDev*, *SrcXOrg*, *SrcYOrg*, *Xext*, *Yest*, *Rop3*, *lpPBrush*, *lpDrawMode*)

Transfers bits delimited by a source rectangle from the source bitmap to the area delimited by a destination rectangle on the destination bitmap. The type of transfer is controlled by the raster operation that allows the specification of all the possible Boolean operations on three variables (source, destination, and the pattern in the brush). Notice that the source and destination may overlap, so the implementation must be careful about the direction in which bits are copied. (See Section 2.5 "The BitBlt Function," for a more detailed discussion of the function and its parameters.)

Parameter	**Description**
lpDestDev	A long pointer to a data structure of type PDEVICE or BITMAP.
DestXOrg and *DestYOrg*	Two short integers specifying the coordinate origin of the destination rectangle on the destination device in device units.
lpSrcDev	A long pointer to a data structure of type PDEVICE or BITMAP.

Parameter	Description
SrcXOrg and *SrcYOrg*	Two short integers specifying the coordinate origin of the source rectangle on the source device in device units.
Xext	A short integer that specifies the horizontal extent of the rectangle on both the source and destination devices in device units.
Yext	A short integer that specifies the vertical extent of the rectangle on both the source and destination devices in device units.
Rop3	A long integer specifying a ternary raster operation code that defines the combining function to be used on the source, destination, and pattern information to produce the color that is placed at the destination for each pixel being rewritten. (See Chapter 14, "Raster Operation Codes and Definitions.")
lpPBrush	A long pointer to a structure of type PBRUSH that was previously realized by this device. This brush is used as the current pattern.
lpDrawMode	A long pointer to the DRAWMODE data structure. The color information in the structure is only used to carry out color conversions in bitmaps.

Return Value None

Comments The source and destination rectangles, defined by their origin and extent on each PDEVICE (in bitmap units), are the same size and may overlap. When this function is used for filling the destination rectangle with a brush, the source device is ignored.

Refer to the GDIINFO data structure for a description of how **BitBlt** registers its output capabilities.

When the source, brush pattern, and destination are not in the same color format, **BitBlt** must convert the source and brush pattern to the same format before copying to the destination.

To convert a monochrome bitmap to a color bitmap, **BitBlt** must do the following:

1. Convert white bits (1) to the background color given in DRAWMODE.
2. Convert black bits (0) to the text (foreground) color given in DRAWMODE.

To convert a color bitmap to a monochrome bitmap, **BitBlt** must do the following:

1. Convert all pixels that match the background color to white (1).
2. Convert all pixels that do not match the background color to black (0).

The current background and foreground colors are defined by the current drawing mode pointed to by *lpDrawMode*.

CheckCursor()

Syntax **CheckCursor** ()

This function is called on every timer interrupt. It allows the cursor to be displayed if it is no longer excluded.

Return Value None

ColorInfo()

Syntax **ColorInfo** (*lpDestDev*, *Colorin*, *lpPCOLOR*) : *rgbColor*

This function converts RGB color values to physical colors and vice versa. The operation to be performed depends on the value of *lpPCOLOR*.

If *lpPCOLOR* is a nonzero value, *Colorin* is assumed to be an RGB color value. The function should choose the best possible physical color to match this color and, then, copy this physical color to the location pointed to by *lpPCOLOR* and return the RGB color value that corresponds to this physical color.

If *lpPCOLOR* is NULL, *Colorin* is assumed to be a physical color and the function should return the corresponding RGB color value.

Physical colors returned by this function are only used by GDI to set text colors, background colors, and pixel colors using Pixel.

Parameter	Description
lpDestDev	A long pointer to a data structure of type PDEVICE. See the PDEVICE description in Chapter 12, "Data Structures and File Formats."
Colorin	A long integer that holds the desired intensities of red, green, and blue (each of 8 bits). The color definition occupies 3 bytes of the long integer, with red in the low byte, green in the second byte, blue in the third byte, and the fourth byte reserved.

Parameter	Description
lpPCOLOR	A long pointer to a variable of type PCOLOR. See the PCOLOR description in Chapter 12, "Data Structures and File Formats."

Return Value *rgbColor* is a long integer that holds the intensities of red, green, and blue (each of 8 bits) of the actual color that the device would use if asked to perform the *Colorin* color.

For palette-capable devices only, the high-WORD can be either 0 or 0FFH. If the high byte equals 0FFH, then an index (not an RGB) is in the low-word. Just return the index that was passed in. No color conversion needs to be performed.

Control()

Syntax **Control** (*lpDestDev*, *Function*, *lpInData*, *lpOutData*) : *wReturnVal*

Parameter	Description
lpDestDev	A long pointer to the destination device bitmap.
Function	The predefined subfunctions for **Control** are described in Chapter 11, "Device Driver Escapes."
lpInData	A long pointer to function-specific input data.
lpOutData	A long pointer to function-specific output data.

Return Value This depends on the subfunction.

DeviceBitmap()

Syntax **DeviceBitmap** (*lpDestDev*, *Command*, *lpBitmap*, *lpBits*) : *wSuccess*

The call to this function is not yet implemented in GDI. It must be implemented as a stub function.

Parameter	Description
lpDestDev	A long pointer to a data structure of type PDEVICE.
Command	An integer containing the number of the command.
lpBitmap	A long pointer to a data structure of type BITMAP containing a description of the device bitmap.

Parameter	Description
lpBits	A long pointer to the contents of the device bitmap.

Return Value This call is only a stub at this time. It may be used in future versions of Windows. It currently returns **AX** = 0.

Comments You must set up the stack frame correctly to ensure correct returns to GDI should the stub ever be called.

DeviceMode()

Syntax **DeviceMode** (*hWnd*, *hInstance*, *lpDestDevType*, *lpOutputfile*)

The **DeviceMode** function sets the current printing modes for the device by prompting for those modes using a dialog box. An application calls **DeviceMode** directly when it wants the user to change the printing modes of the corresponding device. The function copies the mode information to the environment block associated with the device and kept by GDI. GDI initializes this environment block when the application calls **CreateDC** and gives access to it through the **SetEnvironment** and **GetEnvironment** functions.

Parameter	Description
hWnd	A handle to the application's window.
hInstance	The instance handle of the application.
lpDestDevType	A long pointer to a NULL-terminated string containing the device name. The application passes the same device type name as given in the **CreateDC** function.
lpOutputfile	A long pointer to a NULL-terminated string containing the name of an MS-DOS file or device port. The application passes the same device type name as given in the **CreateDC** function.

Return Value None

Comments **DeviceMode** should be included in and exported from any device driver that permits the user to change modes.

(Lisa, please review this carefully.)

The driver should allow all dialog boxes created with **DeviceMode** to be dismissed at any time by pressing ESC. This is because the Help application, when setting up printers, must be able to respond to a request for help from another application at any time. It will try to do so by sending a WM_COMMAND (IDCANCEL) message (equivalent to pressing the ESC key) to all its task windows, including those brought up by the device driver.

To determine if your code does the right thing, do the following test:

1. From Program Manager, choose the File and Run commands and, then, run WINHELP.EXE.
2. From WINHELP, choose the File and Printer Setup commands.
3. Select your printer and press the Setup button.
4. Then, from every state reachable from that point, press ALT + ESC and F1. **(Lisa, all 3 together or do you get an action between ALT + ESC and F1?)**

Help should respond by removing all its dialog boxes (including all the device driver dialog boxes) and displaying help for Program Manager.

Disable()

Syntax **Disable** (*lpDestDev*)

If the Windows session is ending or if a non-Windows application (e.g., Microsoft Word) is being run, the display needs to be disabled. In the case of a non-Windows application, the display will be re-enabled later by a call to **Enable**.

This call disables the display for either purpose.

Parameter	Description
lpDestDev	A long pointer to a data structure of type PDEVICE.

Return Value None

Enable()

Syntax **Enable** (*lpDestDev*, *Style*, *lpDestDevType*, *lpOutputFile*, *lpData*) : *wSize*

This function initializes a device driver or returns information about the driver as defined by the value of *Style*.

This function should save the current hardware state in static storage (usually within the PDEVICE data structure passed) so that it can be restored when Microsoft Windows terminates.

Parameter	Description
lpDestDev	A long pointer to a data structure of type PDEVICE (if the D0 bit of *Style* is 0) or GDIINFO (if the D0 bit of *Style* is 1).
Style	An integer value specifying the type of action to take. If the high bit is set, only an information context is requested. (This is done if no hardware is actually connected.) ■ 0x0000 Initialize the support module (PDEVICE) and peripheral hardware. ■ 0x0001 Fill the GDIINFO data structure with module information. ■ 0x8000 Initialize the PDEVICE structure. ■ 0x8001 Fill the GDIINFO data structure with module information.
lpDestDevType	A long pointer to a null-terminated ASCII string giving the name of the type of physical device to be initialized. This string only applies to support modules that can drive more than one type of device. The parameter can be NULL if only one type of device is supported.
lpOutputFile	A long pointer to a null-terminated ASCII string giving the MS-DOS filename of the physical device. For example, "COM1" for a plotter or a null string for the dedicated system display. This string will be the same string passed into the **CreateDC** function as the desired physical device. It can be NULL.
lpData	A long pointer to device-specific information that is to be used by the support module to initialize the environment of the given physical device. It can be NULL if no such information is needed.

Return Value

On return, **AX** holds zero if it is unsuccessful (e.g., hardware is not initialized). Otherwise, **AX** returns nonzero.

Comments

This function will only be called once for most physical devices. It can be called more than once for a display, (e.g., if a non-Windows application is run; this requires that the display be Disabled and, when the application is terminated, re-Enabled.)

In some cases, GDI may request a raster device to write on a memory bitmap without enabling the device first. This only occurs with raster devices that can write to memory bitmaps.

NOTE Because of the interaction between a dedicated display driver and the keyboard driver, every dedicated display driver should use the following procedure to ensure that the keyboard works correctly under Windows:

EnumDFonts()

Syntax

EnumDFonts (*lpDestDev*, *lpFaceName*, *lpCallbackFunc*, *lpClientData*) : *wLastCallback*

This function is used to enumerate the fonts available on the device. For each appropriate font, the callback function is called with the information for that font. The callback function is called until there are no more fonts or the callback function returns zero.

Parameter	Description
lpDestDev	A long pointer to a data structure of type PDEVICE.
lpFaceName	A long pointer that determines the method of enumeration. If *lpFaceName* points to a string containing the name of a font face, all the fonts of that typeface are enumerated. If there are no fonts of that face, none is enumerated and **EnumDFonts** returns 1. If *lpFaceName* is NULL, one font of each face available is selected at random and enumerated. Again, if there are no fonts, none is enumerated and **EnumDFonts** will return 1.
lpCallbackFunc	A long pointer to the user-supplied callback function. See the following *Comments* section.
lpClientData	A long pointer to the user-supplied data.

Return Value

This function returns the last value returned by the callback function.

Comments

The callback function has the following form:

CallbackFunction (*lpLogFont*, *lpTextMetrics*, *wFontType*, *lpClientData*)

where *lpLogFont* is a long pointer to a data structure of type LOGFONT defined such that it maps to the enumerated font; *lpTextMetrics* is a long pointer to a data structure of type TEXTMETRIC defined with the values that would be returned by a **GetTextMetrics** call; *FontType* is an integer value indicating the type of the font; and *lpClientData* is a long pointer to the user-supplied data passed to **EnumDFonts**. Sizes are in device units.

The label "RASTER_FONTTYPE" is to be ORed into *wFontType* to indicate that the font is composed of a raster bitmap rather than vector strokes. If the device is capable of text transformations such as scaling and italicizing, only the base font will be enumerated. The user is responsible for inquiring the device's text transformation abilities to determine which additional fonts are available directly from the device.

EnumObj()

Syntax

EnumObj (*lpDestDev, Style, lpCallbackFunc, lpClientData*) : *wLastCallback*

This function is used to enumerate the pens and brushes available on the device. For each object belonging to the given style, the callback function is called with the information for that object. The callback function is called until there are no more objects or the callback function returns zero.

Parameter	Description
lpDestDev	A long pointer to a data structure of type PDEVICE.
Style	An integer value specifying the type of object to be enumerated. It can be any one of the following values: 1: Enumerate pens. 2: Enumerate brushes. All objects of the given type are enumerated. If there are no objects of that type, none is enumerated and **EnumObj** returns 1.
lpCallbackFunc	A long pointer to the user-supplied callback function.
lpClientData	A long pointer to the user-supplied data.

Return Value

This function returns the last value returned by the callback function.

Comments

The callback function has the following form:

CallbackFunction(*lpLogObj, lpClientData*)

where *lpLogObj* is a long pointer to a data structure of type LOGPEN or LOGBRUSH, depending on the style selected. **EnumObj** must map each physical object to a logical object before passing to the callback function. *lpClientData* is a long pointer to the user-supplied data passed to **EnumObj**.

When initializing, some older applications (such as Microsoft Excel for versions earlier than 2.1) mistakenly expected the first 8 pens returned to them through the **EnumObj** function to be the 8 default EGA colors. Therefore, even though their pens are not normally enumerated in that EGA order, driver writers should first return the 8 default EGA pens with these RGB values and, then, enumerate as many others as they want to enumerate. They should do this for both brushes and pens.

EGA Pen	RGB Value
Black	0,0,0

EGA Pen	RGB Value
White	FF,FF,FF
Red	FF,0,0
Green	0,FF,0
Blue	0,0,FF
Yellow	FF,FF,0
Magenta	FF,0,FF
Cyan	0,FF,FF

(Ron G. please review this carefully.)

This standard order for enumerating pens and brushes is also useful when you have a device with full 24-bit resolution. First, you enumerate the eight standard colors. Then, you can enumerate your own colors. However, you should also consider here the color applications, such as Microsoft Excel, that enumerate 16 colors. Therefore, we recommend you pick the eight most desirable colors. Then, after the first 16 are enumerated, if the program asks for more colors, you can select another set of colors up to a total of 256 colors.

We also recommend that you enumerate your solid pens first and, then, if you want to do so, the patterned ones.

For brushes, only enumerate the solid ones. And if they can be dithered, only enumerate the "true" solid ones. GDI already knows about the hatched ones. Since there can be a very large number of patterned ones, it is better not even to start enumerating them.

When enumerating brushes, the background color for hatched brushes is not returned.

ExtTextOut()

Syntax

ExtTextOut (*lpDestDev, DestXOrg, DestYOrg, lpClipRect, lpString, Count, lpFontInfo, lpDrawMode, lpTextXForm, lpCharWidths, lpOpaqueRect, Options*) : *dwSuccess*

This function transfers the pattern for each character in the string from the font bitmap to the destination device, starting at the origin passed. In each character pattern, a one bit specifies the character foreground, and a zero bit specifies the character background. It is effectively the Windows 2.0 and later **StrBlt** function. (See Section 2.6,"The StrBlt/ExtTextOut Functions," for a more detailed discussion of the function and its parameters.)

Parameter	Description
lpDestDev	A long pointer to the destination device bitmap.
DestXOrg	The left origin of the string.

Parameter	Description
DestYOrg	The top origin of the string.
lpClipRect	A long pointer to the clipping rectangle. Only pixels within the rectangle are to be drawn. The upper-left corner of the rectangle is assumed to be located at the upper-left corner of a pixel (not the center). Thus, no pixels are drawn if the clipping rectangle is empty (zero width and height), and only one pixel is drawn if it has a width and height of 1. See also the description of the RECT data structure in Chapter 12, "Data Structures and File Formats."
lpString	A long pointer to the string itself.
Count	*Count* has one of three meanings: If *Count* is greater than zero, it is the number of characters to display from the string. The placement of characters is determined by the state of the Differential Data Analyzer (DDA) in the DRAWMODE structure. On exit, the DDA must be reset to its original state. (**BreakErr** is left as it was upon entry to **StrBlt** or **ExtTextOut**.) If *Count* is less than zero, no output is produced, and the extent of the string is returned as a long integer. The X and Y values of the extent are 16-bit quantities packed with Y in the high word and X in the low word. The extent is defined as the bounding box in pixels that the string would occupy if the clipping rectangle were infinite. The size of the string is determined by the state of the DDA in the DRAWMODE structure. On exit, the DDA must be set to its new state as advanced by the contents of the string. (**BreakErr** is modified.) If *Count* is zero, then check the *Options* flag. If the 2s bit is set in the *Options* flag, then it infers that you have to draw an opaque rectangle. See Section 2.6.2, "The ExtTextOut Parameters,," for a more detailed discussion.
lpFontInfo	A long pointer to a data structure of type FONTINFO that represents the physical font in use.
lpDrawMode	A long pointer to a data structure of type DRAWMODE that includes the current text color, background mode, background color, text justification, and character spacing. Refer to the DRAWMODE data structure description in Chapter 12, "Data Structures and File Formats," for a description of text justification and character spacing.

Parameter	Description
lpTextXForm	A long pointer to a data structure of type TEXTXFORM that describes text appearance that may differ from the actual values specified by *lpFontInfo*. This parameter allows more capable devices to make changes to the standard font. For example, if **ExtTextOut** (or **StrBlt**) registers itself as capable of sizing characters, *lpTextXForm* may specify a different point size from the one specified by *lpFontInfo*. If a 16-point font replaces an 8-point font, **ExtTextOut** must do bit doubling (or vector doubling) to produce the desired font size. If **ExtTextOut** has no transform capabilities and registers itself as such, the *lpTextXForm* parameter may be ignored.
lpCharWidths	The user may exercise explicit control over the spacing of each character by passing in a vector of *x* movements. If *lpCharWidths* is non-NULL, then *lpCharWidths*[*n*] is the adjustment from the start of the *n*th character to character *n*+1. This number may be larger or smaller than the actual width of the *n*th character.
lpOpaqueRect	If non-NULL, a long pointer to the opaquing rectangle.
Options	An integer with bits set to indicate **ExtTextOut** options. If bit D2 (0x0004) is set in the *Options* flag, then the rectangle pointed to by *lpOpaqueRect* is to be intersected with the rectangle pointed to by *lpClipRect*, with the resulting area being used to clip the string. If bit D1 (0x0002) is set in the *Options* flag, then the rectangle pointed to by *lpOpaqueRect* is to be intersected with the rectangle pointed to by *lpClipRect*, and the resulting area filled with the background color given in DRAWMODE. The area is to be filled regardless of opaque/transparent mode. Notice that the text string bounding box and the opaquing rectangle are allowed to be disjoint rectangles.

Return Value

Under certain circumstances, (e.g., if the specified font is not supported), **ExtTextOut** returns **DX:AX** = 8000:0000H to signify an error.

Otherwise, it returns **DX:AX** = 0000:0000H to signify success.

However, if *Count* =< 0, it returns **DX:AX** = *Yext*:*Xext*

Comments

Refer to the GDIINFO data structure for a description of how **ExtTextOut** registers its output capabilities and their meanings.

The upper-left corner of the string is placed starting at the point defined by *DestYOrg*. This means that the characters in the string appear below and to the right of the starting point.

ExtTextOut uses the current drawing mode to determine the current text color, the background mode (or Transparent/Opaque flag), and the background color. The background mode determines whether or not **ExtTextOut** must draw an opaque bounding box before drawing the characters. The background color determines what color that box must be. **ExtTextOut** does not use the current binary raster operation mode (ROP2).

For further information on TEXTXFORM, see Chapter 13, "The Font File Format."

FastBorder()

Syntax

FastBorder (*lpRect*, *wHorizBorderThick*, *wVertBorderThick*, *dwRasterOp*, *lpDestDev*, *lpPBrush*, *lpDrawMode*, *lpClipRect*) : *wSuccess*

This function draws a rectangle with a border on the screen. However, the size is subject to the limits imposed by the specified clipping rectangle. The border is drawn within the boundaries of the specified rectangle.

Parameter	Description
lpRect	A long pointer to the rectangle to be framed.
wHorizBorder-Thick	The width in pixels of the left and right borders.
wVertBorderThick	The width in pixels of the top and bottom borders.
dwRasterOp	The raster operation to be used.
lpDestDev	A long pointer to a data structure of type PDEVICE, i.e., the device to receive the output.
lpPBrush	A long pointer to a data structure of type PBRUSH.
lpDrawMode	A long pointer to a data structure of type DRAWMODE that includes the current text color, background mode, background color, text justification, and character spacing. See the DRAWMODE data structure description in Chapter 12, "Data Structures and File Formats," for a description of text justification and character spacing.
lpClipRect	A long pointer to the clipping rectangle.

Return Value

FastBorder returns AX = 0 on error, AX = 1 on success.

Comments

The specified rectangle should be given as (UpperLeftCorner, LowerRightCorner). If it is specified incorrectly, the sample function will draw the borders outside of the specified rectangle, instead of correctly drawing them inside.

The function is optional for display drivers. It is required at the GDI level but not at the display level.

The raster operation to be used will never have a source operand within it.

The *lpDrawMode* parameter is simply a long pointer to the DRAWMODE data structure. It is included only for compatibility with earlier versions and is not crucial. The only field that you may use from there is **BackgroundColor**.

GetCharWidth()

Syntax

GetCharWidth (*lpDestDev*, *lpBuffer*, *FirstChar*, *LastChar*, *lpFontInfo*, *lpDrawMode*, *lpFontTrans*) : *wSuccess*

This function returns, for the specified font, the widths of the characters within the given range. Characters outside of the font's range are given the width of the default character.

Parameter	Description
lpDestDev	A long pointer to a data structure of type PDEVICE. (Not currently used by display drivers.)
lpBuffer	A long pointer to the character width data, an array of 16-bit values.
FirstChar	The first character of the range.
LastChar	The last character of the range.
lpFontInfo	A long pointer to a data structure of type FONTINFO.
lpDrawMode	A long pointer to a data structure of type DRAWMODE that includes the current text color, background mode, background color, text justification, and character spacing. Refer to the DRAWMODE data structure description in Chapter 12, "Data Structures and File Formats," for a description of text justification and character spacing. (Not currently used by display drivers.)
lpFontTrans	A long pointer to a data structure of type TEXTXFORM. (Only needed if the device can do font transformations, such as scaling and italicizing.)

Return Value

This function returns its information in a buffer to which *lpBuffer* points. In the event of an error, it returns AX = 0.

GetEnvironment()

Syntax

GetEnvironment (*lpPort, lpDevMode, cbDevMode*) : *Bytes*

This function copies the current environment associated with the device attached to the system port specified by *lpPort* into the buffer specified by *lpDevMode*. The environment, maintained by GDI, contains binary data used by GDI whenever a display context (DC) is created for the device on the given port.

The function fails if there is no environment for the given port.

Parameter	Description
lpPort	A long pointer to a NULL-terminated string specifying the name of the desired port.
lpDevMode	A long pointer to the buffer that receives the environment.
cbDevMode	An integer value specifying the maximum number of bytes to copy.

Return Value

Bytes is an integer value specifying the number of bytes copied to *lpData*. It is zero if the environment cannot be found.

Inquire()

Syntax

Inquire (*lpCURSORINFO*)

This function returns the mouse's mickey-to-pixel ratio for your screen.

Parameter	Description
lpCURSORINFO	A long pointer to a device information block (data type CURSORINFO) that is filled in by the support module (**device driver?**). The first word is the X mickey-to-pixel ratio, and the second word is the Y mickey-to-pixel ratio.

Return Value

On return, AX holds the number of bytes (4) actually written into the data structure.

Comments

This function is called once per initialization before the **Enable** function.

MoveCursor()

Syntax

MoveCursor (*absX, absY*)

This function moves the cursor to the given screen coordinates. If the cursor is a composite of screen and cursor bitmaps (i.e., not a hardware cursor), this function must ensure that screen bits under the current cursor position are restored and the bits under the new position are saved. The function must move the cursor, even if the cursor is not currently displayed.

Parameter	Description
absX and *absY*	Absolute X and Y screen coordinates of the new cursor position.

Return Value None.

Comments Microsoft Windows may specify a position at which the cursor shape would lie partially outside of the display bitmap. The OEM function is responsible for clipping the cursor shape to the display boundary.

The **MoveCursor** function is called at mouse interrupt time, outside of the main thread of Windows processing. Since **MoveCursor** may even interrupt its own processing, the device driver should disable interrupts while reading the *absX* and *absY* coordinates by using the **EnterCrit** and **LeaveCrit** macros. Do *not* use STI and CLI instructions in the driver.

Output()

Syntax **Output** (*lpDestDev*, *Style*, *Count*, *lpPoints*, *lpPPen*, *lpPBrush*, *lpDrawMode*, *lpClipRect*) : *wReturnVal*

This function consists of the output primitive group, which includes all the shape-drawing functions registered in GDIINFO.

Parameter	Description
lpDestDev	A long pointer to the destination device.
Style	A short integer that defines the type of geometric primitive to be drawn. The interpretation of the remaining parameters depends on the style. The available style primitives are described following the parameter definitions.
Count	A 16-bit integer specifying the number of points in the points list.
lpPoints	A long pointer to an array of short integers. The array has *Count* elements, and each element contains two short integers. For most of the output primitives, these are simply the device coordinates for each point on the figure.

Parameter	Description
lpPPen	A long pointer to a data structure of type PPEN.
lpPBrush	A long pointer to a data structure of type PBRUSH.
lpDrawMode	A long pointer to a data structure of type DRAWMODE that includes information required to decide how to alter the pixels (ROP2). It includes a specification of a mode in which to draw the line (a logical function combining source and destination), a background mode, and a physical foreground and background color.
lpClipRect	A long pointer to the clipping rectangle to be used to clip output. For polygons and lines, *lpClipRect* contains the bounding box of all the lines to be drawn. This parameter is ignored if the device is unable to clip (see the **dpClip** field of the GDIINFO structure). See also the RECT data structure. If *lpClipRect* is zero, then the clipping rectangle is the entire display surface.

Return Value

Output returns the following:

- AX= 1: success
- AX= 0: unrecoverable failure
- AX= -1: device driver could not support the passed style

The -1 return represents support for "smart" devices in the Windows GDI.

Refer to the GDIINFO data structure for a description of how **Output** registers its output capabilities and their meanings.

Comments

lpClipRect should be intersected with the bitmap as well. The display driver should do it and not rely on GDI to do it.

lpClipRect is only in effect in the GDIINFO data structure if the **dpClip** field says that you can clip. Otherwise, the device can ignore it because the device cannot clip itself.

The only **Output** styles required by Windows 2.0 and later GDI are OS_SCANLINES and OS_POLYLINES. With certain "smart" devices, you may want to use the device's capability to draw complex figures. However, in many cases the device is either limited by such things as the number of vertices in a polygon or is not able to draw these complex figures into a main memory bitmap, although it can draw them to the screen. In these cases, the device driver is now allowed to return a -1 failure code. When GDI receives this return code, it breaks the complex figure into component scanlines and polylines and draws the figure with them.

The defined styles are listed in this *Comments* section, with a brief description of what the **Output** primitive does when each style is specified. The style type determines which of the parameters in the **Output** primitive contain meaningful information. The *lpDestDev* (device pointer) and *lpDrawMode* (raster op) parameters are common to all style types and are not described further here.

For styles that define a closed area, the current interior pattern specified by the *lpPBrush* parameter is used. For styles that define lines or borders, the current line pattern specified by the *lpPPen* parameter is used.

If **Output** passed both a pen and a brush (i.e., if neither *lpPBrush* nor *lpPPen* is NULL), then the interior should be drawn first with the brush and followed by the border drawn with the pen.

The *lpPoints* parameter points to an array of pairs of short integers that designate the points used to produce the style specified. When the number of points required to produce a style can vary (e.g., the polygon), the *Count* parameter specifies the exact number of points. For arc styles, *Count* is always five, specifying two points for the upper left and lower right corners of the bounding rectangle, a start point, a stop point, and a special point structure that actually contains the pair of angles used to sweep out the arc. The circle and ellipse styles use only the first two of these points. The other styles use the *Count* and *lpPoints* parameters to produce the appropriate output.

All the styles are briefly described below:

Style	**Description**
OS_ARC (3)	Causes an arc to be drawn on the device. If the points are all the same, a point is drawn. If they are collinear, a line is drawn. Otherwise, a circular arc is defined that passes from the start to the stop point (counterclockwise, as defined by the upper and lower points and the size of the specified angles). This style does not define a closed figure even if the start and stop points are identical.
OS_PIE (23)	Causes a Pie-type closed arc to be drawn on the device. The arc is drawn as described above, then two additional lines are drawn (one from each endpoint) to the implicit center of the circular arc, defining a wedge-shaped enclosed area which is filled.

Style	Description
OS_CHORD (39)	Causes a Chord-type closed arc to be drawn on the device. The arc is drawn as described above, then the two endpoints are connected with a straight line and the enclosed area is filled.
OS_CIRCLE (55)	Causes a circle to be drawn on the device. The circle is centered at the implicit center, determined by the upper leftmost and lower rightmost points passed, and has a radius equal to half the width of the rectangle. A zero radius colors a pixel at the center of the circle.
OS_ELLIPSE (7)	Causes an ellipse to be drawn on the device. The ellipse is centered at the implicit center, determined by the upper leftmost and lower rightmost points passed. It also has the width and height implied by these points.
OS_ALTERNATE_FILL_POLYGON (22)	Specifies a polygonal area that is to be drawn on the device and filled using the alternate filling method (i.e., every other enclosed region within a complex polygon is filled). The *Count* parameter contains the number of points to be passed. The polygon is drawn from the first point, passed through subsequent points, and closed back to the first point, if necessary.
OS_TRAPEZOID (20)	Formerly called Winding Number Fill Polygon, it specifies a polygonal area that is to be drawn on the device and filled using the winding number filling method (i.e., only enclosed regions within a complex polygon are filled). The *Count* parameter contains the number of points to be passed. The polygon is drawn from the first point, passed through subsequent points, and closed back to the first point, if necessary.

Style	Description
OS_RECTANGLE (6)	Causes a rectangle to be drawn on the device, using two points passed as the two corners. If the corner points are called (X1,Y1),(X2,Y2), then the driver should draw a rectangle defined by (X1,Y1) and (X2-1,Y2-1) and *exclude* the bottom right corner. For more information on the RECT data structure that is passed to the driver, see Chapter 12, "Data Structures and File Formats."
OS_POLYLINE (18)	Causes a set of line segments to be drawn on the device. The value of *Count* must be at least two (2). Each line segment is drawn from its starting point up to, but not including, its end point. If more than one line segment is drawn, each new segment starts at the end point of the previous segment. Polylines do not define filled areas and are not implicitly closed. This style is required for both raster and vector devices. Polylines do not use brushes. The line style is determined from *lpPPen*.
OS_SCANLINES (4)	Provides a means to rapidly fill a set of intervals with the pattern for a particular raster line. There are always an even number of X coordinates. Lines are drawn from the starting point up to, but not including, the end point. Thus x1-x2, x3-x4, are all lines drawn with the brush pointed to by *lpPBrush*, or by the pen pointed to by *lpPPen* if *lpPBrush* is NULL. This style can be used with memory bitmaps as well as on devices. The first point gives the Y coordinate of the scan. Each successive point is a pair of X values that determine the position and length of the scanline.

Style	Description

OS_BEGINNSCAN()
OS_ENDNSCAN()
	(Ron G, please review this carefully.) Two new styles, for Windows 3.0 and later versions, that will always come in pairs. The first one indicates that a series of **Output** calls of the OS_SCANLINE style will follow. The pen, brush, and drawmode parameters will correspond to the ones used by the series of scanlines. The following OS_SCANLINE calls will also have information about the scanlines, such as the *Count* and *lpPoints*. The end of the series is marked by the new **Output** call of the OS_ENDNSCAN style.
	
	These styles will speed up the filling of polygons, widelines, and floodfills. The bracketing of the scanlines will enable devices to set up the pen, brush, and drawmode information only once per figure instead of once per scanline. Devices that do not understand these styles will ignore them, and the scanline call will remain unchanged.

Output uses the current binary raster operation mode (ROP2) when drawing lines and scanlines. It also uses the current background mode and color, but not the text color.

When drawing solid lines, Output replaces the destination pixel with a combination of the destination and the line color. The binary raster operation defines how the colors are combined. When drawing styled lines (e.g., lines with gaps), the function replaces the destination pixels under the solid part of the line with a combination of destination and line colors, the same as for a solid line. If the current background mode is OPAQUE, **Output** replaces destination pixels under the gaps with a combination of the destination and the current background color. Again, the raster operation mode defines how to combine these colors. If the current background mode is TRANSPARENT, **Output** leaves the destination pixels unchanged. For example, to draw a line that inverts the destination color, use the XOR binary raster operation code and a white pen, or use the NOT binary raster operation code and a black pen.

Output uses a brush pattern to draw scanlines. When drawing a scanline, **Output** replaces the destination pixel color with a combination of the destination color and the color of an individual pixel in the brush. The binary raster operation code defines how the colors are to be combined. **Output** leaves the destination pixel color unchanged if the current background mode is TRANSPARENT and the brush pixel and background colors are equal. If *lpBrush* is NULL, the pen is used for the scanlines. A pen is considered to be the same as a solid brush for this purpose. However, unlike a brush, it cannot be dithered (except in the case of the LS_INSIDEFRAME pen style), and should not be considered totally interchangeable with a brush.

If a device registers in the GDIINFO data structure that it can do styled polylines, then it must do them for all devices, no matter whether it is the bitmap or the screen. It cannot fail them.

We do not recommend that you implement the OS_CIRCLE style (55). GDI will always call the OS_ELLIPSE style (7) to draw a circle. Since there are no efficiencies gained, there is no reason to support the circle style.

We do not recommend that you support the OS_RECTANGLE style (6). You can support it, though, and make it work. However, there is some special casing that you have to do with little gain in efficiency or speed. If you do not support it, GDI will simply call **BitBlt** and the OS_POLYLINE style (18) to draw the rectangle.

However, if there is no pen passed in the *lpPPen* field, then you use the brush and draw the filling from the starting coordinate up to, but not including, the last pixel. If there is a pen passed, then the border starts on that first pixel, and you do not start drawing the fill until the second pixel. You stop two pixels from the end because the border is the last one.

You should always stop drawing scanlines one pixel before the ending coordinate. You do not draw through the last coordinate.

Pixel()

Syntax

Pixel (*lpDestDev*, *wX*, *wY*, *dwPhysColor*, *lpDrawMode*) : *PhysColor*

This function sets or retrieves the color of the specified pixel. If *lpDrawMode* is not NULL, this function sets the given pixel to the color given by *dwPhysColor*, using the binary raster operation given by *lpDrawMode*. If *lpDrawMode* is NULL, the function returns the physical color of the pixel given by *wX* and *wY*. (See the following *Return Value* section.)

Parameter	**Description**
lpDestDev	A long pointer to a data structure of type PDEVICE. See the PDEVICE description in Chapter 12, "Data Structures and File Formats."
wX and *wY*	Integer values that specify the device coordinates of the pixel to be acted on.
dwPhysColor	A physical color value of type PCOLOR. See the PCOLOR description in Chapter 12, "Data Structures and File Formats."
lpDrawMode	A long pointer to a data structure of type DRAWMODE that includes the binary raster operation to carry out on the given pixel.

Return Value

If *lpDrawMode* is NULL, the function returns the physical color of the pixel given by *wX* and *wY*.

If *lpDrawMode* is non-NULL, the function returns **DX:AX** = 0000:0001.

On error, in either the **SetPixel** or **GetPixel** mode, the function returns **DX:AX** = 8000:0000.

RealizeObject()

Syntax

RealizeObject (*lpDestDev*, *Style*, *lpInObj*, *lpOutObj*, *lpTextXForm*) : *wSize*

This primitive directs the device driver to fill an attribute structure created by GDI that will be used when drawing output primitives. It may also direct the driver to return the size of such a structure.

If *lpOutObj* is a nonzero value, it is assumed to be a long pointer to a data structure to be filled with the physical attributes of an object. *Style* specifies the type of object to be realized and *lpInObj* is a long pointer to a structure defining the logical attributes of the object. The function must translate the logical attributes into sufficient information to accurately describe a physical object for use by output functions when drawing. Only the device driver uses the PPEN and PBRUSH objects (except that GDI also uses the device fonts). Therefore, the format of these structures is up to the device driver writer.

If *lpOutObj* is NULL, the function is expected to return the size (in bytes) of the physical data structure. After receiving the object size, GDI allocates space for the realized object and calls **RealizeObject** again, passing a pointer to the allocated space in *lpOutObj*.

Parameter	**Description**
lpDestDev	A long pointer to a data structure of type PDEVICE.
Style	An integer that specifies the type of object to be realized. The predefined objects are as follows: OBJ_PEN (=1) Pen - used to stroke out borders OBJ_BRUSH (=2) Brush - used to cover the interior of figures OBJ_FONT (=3) Fonts - used to specify the appearance of characters If a negative *Style* is passed, the specified object is to be deleted.

Parameter	Description
lpInObj	A long pointer to a data structure of type LOGPEN, LOGBRUSH, or LOGFONT, depending on the given *Style*. This parameter describes the logical attributes of the object.
lpOutObj	A long pointer to a data structure to receive the realized object. For pens and brushes, the structure types are PPEN and PBRUSH, respectively. For fonts, the structure must contain fields identical to the fields **dfType** through **dfFace**, with valid pointers to device (if any) and facename strings, in the FONTINFO data structure. Additional information is copied to a data structure of type TEXTXFORM pointed to by *lpTextXForm*.
lpTextXForm	A doubleword length value. It can serve one of two purposes, depending on the value of *Style*. If *Style* is OBJ_BRUSH, *lpTextXForm* is not a pointer. Rather, it is a data structure of type POINT, which contains the screen coordinates of the window's origin. The bits in the realized brush should be rotated so the upper-left corner aligns with some OEM-defined point relative to the new origin. If *Style* is OBJ_FONT, *lpTextXForm* is a long pointer to a data structure of type TEXTXFORM, which contains additional information about the appearance of a realized font. Both the realized font and the contents of the TEXTXFORM structure are later passed to the **ExtTextOut** function, allowing more capable devices to make changes to the standard font.

Return Value If a device cannot realize an object, **RealizeObject** returns zero.

Comments For further information on TEXTXFORM data structures, see Chapter 13, "The Font File Format."

SaveScreenBitmap()

Syntax **SaveScreenBitmap** (*lpRect*, *wCommand*) : *wSuccess*

This function saves a single bitmap from the display or restores a single (previously stored) bitmap to the display. It is used, for example when a menu is pulled down, to store the part of the screen that is "behind" the menu until the menu is closed.

Parameter	Description

lpRect	A long pointer to the rectangle to use.
wCommand	0: Save the rectangle.
	1: Restore it.
	2: Discard previous save, if there was one.

Return Value

This function returns AX = 1 if successful, AX = 0 for any of the following error conditions:

- "Shadow memory" does not exist (save, restore, ignore).
- "Shadow memory" is already in use (save).
- "Shadow memory" is not in use (restore).
- "Shadow memory" has been stolen or trashed (restore).

Comments

Because **SaveScreenBitmap** can save only one bitmap at a time, the device driver must maintain a record of whether or not the save area is currently in use.

The bitmap is stored in "shadow memory" (i.e., memory for which the device has control of allocation). Therefore, the device can save the bitmap in whatever form is most convenient for it, without the rest of Windows worrying about where it goes.

ScanLR()

Syntax

ScanLR (*lpDestDev*, *wX*, *wY*, *wPhysColor*, *Style*) : *wReturnVal*

This function scans the device surface in a left or right direction from the given pixel looking for the first pixel having (or not having) the given color. **ScanLR** is used by Microsoft Windows Paintbrush to perform flood fills.

Parameter	**Description**
lpDestDev	A long pointer to a data structure of type PDEVICE. See the PDEVICE description in Chapter 12, "Data Structures and File Formats."
wX and *wY*	Integer values specifying the X and Y coordinates of the pixel from which to start the scan.
dwPhysColor	A physical color value of type PCOLOR. See the PCOLOR description in Chapter 12, "Data Structures and File Formats."

Parameter	Description
Style	An integer value specifying the scan style and direction. Bits 1 and 2 of this integer are active and can be set as follows: ■ If bit 1 is set, scan for a pixel with color matching *dwPhysColor.* ■ If bit 1 is cleared, scan for a pixel with color that does not match. ■ If bit 2 is set, scan to the left. ■ If bit 2 is cleared, scan to the right.

Return Value

On return, AX holds the X coordinate of the first pixel satisfying the given scan condition.

If either the given X or Y is not in the range of coordinates of the display surface or the bitmap, then AX = 8000H.

If no pixel is found that satisfies the given scan condition, then AX = -1.

SetAttribute()

Syntax

SetAttribute (*lpDestDev*, *StateNum*, *Index*, *Attribute*) : *wReturnVal*

The code that calls this function is not yet implemented in GDI.

Parameter	Description
lpDestDev	A long pointer to a data structure of type PDEVICE.
StateNum	An integer that specifies the state number.
Index	An integer.
Attribute	An integer.

Return Value

At this time, this call is just a stub function and returns AX = 0.

Comments

You must set up the stack frame correctly to ensure correct returns to GDI should the stub function ever be called.

SetCursor()

Syntax

SetCursor (*lpCURSORSHAPE*)

This function sets the cursor bitmap that defines the cursor shape. Each call replaces the previous bitmap with that pointed to by *lpCURSORSHAPE.* If *lpCURSORSHAPE* is NULL, the cursor has no shape and its image is removed from the display screen.

Parameter	Description
lpCURSORSHAPE	A long pointer to a data structure of type CURSORSHAPE that specifies the appearance of the cursor for the specified device.

Return Value

None.

Comments

The cursor bitmap is actually two bitmaps. The first bitmap is ANDed with the contents of the screen, and the second is XORed with the result. This helps to preserve the appearance of the screen as the cursor is replaced and ensures that at least some of the cursor is visible on all the potential backgrounds.

SetEnvironment()

Syntax

SetEnvironment (*lpPort, lpDevMode, cbDevMode*) : *Bytes*

This function copies the contents of the buffer specified by *lpDevMode* into the environment associated with the device attached to the system port specified by *lpPort.* **SetEnvironment** overwrites any existing environment. If there is no environment for the given port, **SetEnvironment** creates it. If *cbDevMode* is zero, the existing environment is deleted and not replaced.

Parameter	Description
lpPort	A long pointer to a NULL-terminated string specifying the name of the desired port.
lpDevMode	A long pointer to the buffer containing the new environment.
cbDevMode	An integer value specifying the number of bytes to copy.

Return Value

Bytes is an integer value specifying the number of bytes copied to the environment. It is zero if there is an error. It is -1 if the environment is deleted.

StrBlt()

Syntax

StrBlt (*lpDestDev*, *DestXOrg*, *DestYOrg*, *lpClipRect*, *lpString*, *Count*, *lpFontInfo*, *lpDrawMode*, *lpTextXForm*)

This is an alternate entry point to **ExtTextOut**, which is provided for compatibility with Windows 1.XX.

Parameters

For Windows 2.0 and later versions, **StrBlt** should be implemented as a call to **ExtTextOut**, in the following manner:

```
cProc    StrBlt, <FAR,PUBLIC>,<si, di>
         parmd   lpDestDev
         parmw   DestXOrg
         parmw   DestYOrg
         parmd   lpClipRect
         parmd   lpString
         parmw   Count
         parmd   lpFontInfo
         parmd   lpDrawMode
         parmd   lpTextXForm
cBegin   <nogen>;don't fool with the stack frame
                        ;until we get to ExtTextOut
;
;First, we must save our caller's far return address.
;Use CX & BX for this.
;
pop             cx      ;
pop             bx      ;
;
;Now dummy up NULL parameters for the extra
:parameters needed by ExtTextOut:
;
xor             ax,ax   ;
push            ax      ;push a dword for lpCharWidths
push            ax      ;
push            ax      ;push a dword for lpOpaqueRect
push            ax      ;
push            ax      ;push a word for Options
push            bx      ;push the caller's return address
push            cx      ;
jmp             ExtTextOut   ;now go do the StrBlt using
                        ;ExtTextOut!
;
;
cProc    ExtTextOut,<FAR,PUBLIC,WIN,PASCAL>,<si, di>
         parmd   lpDestDev
         parmw   DestXOrg
         parmw   DestYOrg
         parmd   lpClipRect
         parmd   lpString
         parmw   Count
```

```
                parmd   lpFontInfo
                parmd   lpDrawMode
                parmd   lpTextXForm
                parmd   lpCharWidths
                parmd   lpOpaqueRect
                parmw   Options
        cBegin
```

Return Value None

StretchBlt()

Syntax **StretchBlt** (*lpPDevice*, *DestX*, *DestY*, *DestXE*, *DestYE*, *lpSrcPDevice*, *SrcX*, *SrcY*, *SrcXE*, *SrcYE*, *Rop*, *lpPBrush*, *lpdm*, *lpClip*) : **(What do I put here for the Return Value?)**

This function allows devices that support the scaling of bitmaps to use this capability under Windows.

Parameter	Description
lpPDevice	A long pointer to the destination PDevice.
DestX	X on the destination rectangle.
DestY	Y on the destination rectangle.
DestXE	X extent on the destination rectangle.
DestYE	Y extent on the destination rectangle.
lpSrcPDevice	A long pointer to the source PDevice.
SrcX	X on the source rectangle.
SrcY	Y on the source rectangle.
SrcXE	XE on the source rectangle.
SrcYE	YE on the source rectangle.
Rop	The raster operation to be used.
lpPBrush	A long pointer to a data structure of type PBRUSH.
lpdm	A long pointer to a data structure of type DRAWMODE that includes the current text color, background mode, background color, text justification, and character spacing. Refer to the DRAWMODE data structure description in Chapter 12, "Data Structures and File Formats," for a description of text justification and character spacing.

Parameter	Description
lpClip	A long pointer to the clipping rectangle given in destination coordinates.

Return Value

StretchBlt() returns a -1 upon failure and, then, GDI will simulate.

Comments

If the device cannot support a given call, it may fail (i.e., return a -1) and GDI will perform the stretching. This allows devices to perform stretching on cases that are supported but have GDI do the work on those that are not. For example, if a device can stretch by integer factors, or powers of two, it can use this capability.

If a device wants to support **StretchBlt()**, it must set the RC_STRETCHBLT bit in the raster capabilities field in the GDIINFO structure.

The driver must export the **StretchBlt()** function with the ordinal number 27.

WEP()

Syntax

WEP(*bSystemExit*)

This is a termination function, called Windows Exit Procedure, that is required to accommodate the support of dynamic-link libraries (DLLs). This function indicates whether all of Windows is shutting down or just the single DLL.

Parameter	Description
bSystemExit	**(Gunter, can you provide?)**

Return Value

None.

Chapter 11 Device Driver Escapes

The device driver escapes enable applications and device drivers to add support that is otherwise not available through GDI's Application Program Interface (API). Although predefined escapes are useful for this purpose, they are not necessarily the perfect solution. The more escapes we define and implement, the more code you tend to include, which burdens the driver and application.

We recommend that all applications support all the escapes for changing printer settings and any others for specialized support that are really necessary.

Each device driver should support all the escapes that are possible for that particular device. The device driver developer should recognize that an application will assume that the printer setting escapes are available and that it may request the more specialized escapes on high-end devices.

11.1 Introduction to Driver Escapes

The Graphics Device Interface (GDI) includes an entry point called **Escape()** that is used by applications to perform a device-dependent operation that may or may not be supported on a given device or that may be related to the job control of a printing operation. GDI translates **Escape()** calls into calls to the device driver's **Control()** function. GDI may perform its own interpretation of the escape if it needs to. Therefore, there is not necessarily a one-to-one correspondence between **Escape()** and **Control()** calls. However, the capabilities of these calls are referred to on both sides as *driver escapes.*

The **Control()** function is required for all device drivers. However, you may choose to support only a few of the escapes documented here or only the minimal functionality required. For example, you can use QUERYESCSUPPORT to tell the calling application that you support a subset or none of the **Control** functions. Notice, though, that most of these escapes are applicable only to printers. Most applications will not have to support calls to these functions for display devices. In fact, the only escapes that are recommended for support by display drivers are the following three:

- QUERYESCSUPPORT
- GETCOLORTABLE
- SETCOLORTABLE (*not* for palette-capable devices)

The most common escapes used by all printing applications and printer drivers are as follows:

- QUERYESCSUPPORT
- SETABORTPROC
- STARTDOC
- NEWFRAME
- ENDDOC
- ABORTDOC
- NEXTBAND

The **Control()** function is declared for printers in C as follows:

```
Int FAR PASCAL Control(
PDEVICE FAR * lpPDevice,
WORD  iFunction,
LPSTR lpInData,
LPSTR lpOutData
);
```

Where:

lpPDevice	Points to a structure describing the physical device in use and is defined by the device driver itself.
iFunction	Selects the specific escape function to be performed.
lpInData	Points to input data.
lpOutData	Points to a buffer for output data.

Although *lpInData* and *lpOutData* are declared as pointing to characters, they generally point to some structure type. The precise type will depend on the escape function selected.

The generalized stack frame to expect on a call to **Control()**, when using CMACROS in assembly language for display drivers, is as follows:

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

11.2 Generalized Error Return Codes

All the return codes are returned as signed integers in the AX register. The following are the generalized return codes used by the printer driver **Control** subfunctions. They are also referred to in this document by their symbolic names. For all the **Control** subfunctions (i.e., escapes), a positive number indicates success and a zero or negative return code indicates a failure. It is best, however, to use the specific generalized error return codes documented here whenever they are indicated as appropriate.

Name (Integer)	Description
SP_ERROR (-1)	A general error in banding.
SP_APPABORT (-2)	The job was aborted because the application's callback returned false (0).
SP_USERABORT (-3)	The user aborted the job through the Print Manager's "abort job" function.
SP_OUTOFDISK (-4)	A lack of disk space caused the job to abort.
SP_OUTOFMEMORY (-5)	A lack of memory caused the job to abort.

11.3 Driver Escape Descriptions

The following is an alphabetical list of the escape functions, with their corresponding numbers, that are included in this chapter. Following the list, you will find the detailed function descriptions for each escape.

Additional escapes that are appropriate only to applications are provided in the *Microsoft Windows Software Development Kit.*

Escape Name (Number)

- ABORTDOC (2)
- BANDINFO (24)
- BEGIN_PATH (4096)
- CLIP_TO_PATH (4097)
- DRAFTMODE (7)
- DRAWPATTERNRECT (25)
- ENABLEDUPLEX (28)

- ENABLEPAIRKERNING (769)
- ENBLERELATIVEWIDTHS (768)
- ENDDOC (11)
- END_PATH (4098)
- ENUMPAPERBINS (31)
- ENUMPAPERMETRICS (34)
- EPSPRINTING (33)
- EXT_DEVICE_CAPS (4099)
- EXTTEXTOUT (512)
- FLUSHOUTPUT (6)
- GETCOLORTABLE (5)
- GETEXTENDEDTEXTMETRICS (256)
- GETEXTENTTABLE (257)
- GETFACENAME (513)
- GETPAIRKERNTABLE (258)
- GETPHYSPAGESIZE (12)
- GETPRINTINGOFFSET (13)
- GETSCALINGFACTOR (14)
- GETSETPAPERBINS (29)
- GETSETPAPERMETRICS (35)
- GETSETPRINTORIENT (30)
- GETTECHNOLOGY (20)
- GETTRACKKERNTABLE (259)
- GETVECTORBRUSHSIZE (27)
- GETVECTORPENSIZE (26)
- NEWFRAME (1)
- NEXTBAND (3)
- PASSTHROUGH (19)

- QUERYESCSUPPORT (8)
- RESTORE_CTM (4100)
- SAVE_CTM (4101)
- SELECTPAPERSOURCE (18) (superceded by GETSETPAPERBINS)
- SETABORTPROC (9)
- SETALLJUSTVALUES (771)
- SET_ARC_DIRECTION (4102)
- SET_BACKGROUND_COLOR (4103)
- SET_BOUNDS (4109)
- SETCOLORTABLE (4)
- SETCOPYCOUNT (17)
- SETDIBSCALING (32)
- SETKERNTRACK (770)
- SETLINECAP (21)
- SETLINEJOIN (22)
- SETMITERLIMIT (23)
- SET_POLY_MODE (4104)
- SET_SCREEN_ANGLE (4105)
- SET_SPREAD (4106)
- STARTDOC (10)
- TRANSFORM_CTM (4107)

"Japanese Version of Windows" Escape Name (Number)

- GAIJIAREASIZE (2577)
- GAIJIFONTSIZE (2576)
- GAIJIITTOCODE (2580)
- GAIJILOCALCLOSE (2582)
- GAIJILOCALOPEN (2581)
- GAIJILOCALRESTORE (2585)

- GAIJILOCALSAVE (2584)
- GAIJILOCALSETFONT (2583)
- GAIJISYSTEMGETFONT (2578)
- GAIJISYSTEMSETFONT (2579)
- TTYMODE (2560)

ABORTDOC (Escape # 2)

Syntax

short Control (*lpDevice*, **ABORTDOC**, *lpInData*, *lpOutData*)

This escape aborts the current job, erasing everything the application has written to the device since the last ENDDOC escape.

The ABORTDOC escape should be used for printing operations that do not specify an abort function (with the SETABORTPROC escape) and to terminate printing operations that have not yet reached their first NEWFRAME or NEXTBAND call.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, the destination device bitmap.
lpInData	Not used.
lpOutData	Not used.

Return Value

The return value is positive if the function is successful; otherwise, it is negative.

Comments

This escape is called by GDI when a banding error occurs. It is also called by an application when an error occurs or when the application wants to cancel the print job. The driver can delete the Print Manager job.

In some earlier printer drivers, this escape was called ABORTPIC.

BANDINFO (Escape # 24)

Syntax

short Control (*lpDevice*, **BANDINFO**, *lpInData*, *lpOutData*)

This escape copies information about a device with banding capabilities to a structure pointed at by *lpIndata*.

Banding is a property of an output device that allows a page of output to be stored in a metafile and divided into bands, each of which is sent to the device to create a complete page. Devices with banding capabilities avoid problems associated with devices that cannot scroll backwards.

The information copied to the structure pointed at by *lpIndata* includes a flag indicating whether or not there is graphics in the next band, a flag indicating whether or not there is text on the page, and a rectangle structure that contains a bounding rectangle for all graphics on the page.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a BANDINFOSTRUCT data structure containing the following items:

Field	Type/Definition
fGraphicsFlag	BOOL Is non-zero if there are graphics on the page; otherwise, it is zero.
fTextFlag	BOOL Is non-zero if there is text on the page; otherwise, it is zero.
GraphicsRect	RECT Is a rectangle structure that contains the coordinates for a rectangle that bounds the graphics on the page.

This data structure provides the primary communication between the driver and the application as to what (graphics and/or text) is actually on the page. See Section 5.5.5, "Using Banding Drivers," for a description of its use.

lpOutData	A long pointer, which may be NULL, to a data structure that is filled in by the driver and that contains the same items as *lpInData*.

Field	Type/Definition
fGraphicsFlag	BOOL Is non-zero if this is a graphics band; otherwise, it is zero.
fTextFlag	BOOL Is non-zero if this is a text band; otherwise, it is zero.
GraphicsRect	No valid return data.

Return Value The return value is one if the escape function is successful; otherwise, it is zero.

Comments This escape should only be implemented for devices that use banding. It should be called immediately after each call to the NEXTBAND escape.

BEGIN_PATH (Escape # 4096)

Syntax **short Control** (*lpDevice*, **BEGIN_PATH**, *lpInData*, *lpOutData*)

This escape opens a path. A path is a connected sequence of primitives drawn in succession to form a single polyline or polygon. Paths enable applications to draw complex borders, filled shapes, and clipping areas by supplying a collection of other primitives defining the desired shape.

Printer escapes that support paths enable applications to render images on sophisticated devices such as PostScript printers without generating huge polygons to simulate them.

To draw a path, an application first issues the BEGIN_PATH escape. It then draws the primitives defining the border of the desired shape and issues an END_PATH escape. The END_PATH escape includes a parameter specifying how the path is to be rendered.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpOutData	Not used and can be set to NULL.

Return Value This escape returns a short integer value specifying the current path nesting level. If the escape is successful, the number of BEGIN_PATH calls without a corresponding END_PATH call is the result. Otherwise, zero is the result.

Comments You may open a path within another path. A path drawn within another path is treated exactly like a polygon (if the subpath is closed) or a polyline (if the subpath is open).

You may use the CLIP_TO_PATH escape to define a clipping area corresponding to the interior or exterior of the currently open path.

Device drivers that implement this escape must also implement the END_PATH escape and should also implement the SET_ARC_DIRECTION escape.

CLIP_TO_PATH (Escape # 4097)

Syntax **short Control** (*lpDevice*, **CLIP_TO_PATH**, *lpClipMode*, *lpOutData*)

This escape defines a clipping area bounded by the currently open path. It enables the application to save and restore the current clipping area and to set up an inclusive or exclusive clipping area bounded by the currently open path.

To clip a set of primitives against a path, an application should follow these steps:

1. Save the current clipping area using the CLIP_TO_PATH escape.
2. Begin a path using the BEGIN_PATH escape.
3. Draw the primitives bounding the clipping area.
4. Set the clipping area using the CLIP_TO_PATH escape.
5. Close the path using the END_PATH escape.
6. Draw the primitives to be clipped.
7. Restore the original clipping area using the CLIP_TO_PATH escape.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpClipMode	A long pointer to a short integer specifying the clipping mode. It may be one of the following:
	CLIP_SAVE(0). Saves the current clipping area.
	CLIP_RESTORE(1). Restores the previous clipping area.
	CLIP_INCLUSIVE(2). Sets a clipping area such that portions of primitives falling outside the interior bounded by the current path are clipped.
	CLIP_EXCLUSIVE(3). Sets a clipping area such that portions of primitives falling inside the interior bounded by the current path should be clipped.
lpOutData	Not used and can be set to NULL.

Return Value

A nonzero value is returned if the call is successful. Otherwise, zero is the result.

Comments

Device drivers implementing the CLIP_TO_PATH escape must also implement the BEGIN_PATH and END_PATH escapes and should also implement the SET_ARC_DIRECTION escape.

DRAFTMODE (Escape # 7)

Syntax **short Control** (*lpDevice*, **DRAFTMODE**, *lpDraftMode*, *lpOutData*)

This escape turns draft mode off or on.

Turning draft mode on instructs the device driver to print faster and with lower quality (if necessary). The draft mode can only be changed at page boundaries, for example, after a NEWFRAME escape.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpDraftMode	The return value is a 32-bit address to a short integer value specifying the draft mode. It is one for draft mode on and zero for draft mode off.
lpOutData	Not used and can be set to NULL.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

Comments The default draft mode is off.

DRAWPATTERNRECT (Escape # 25)

Syntax **short Control** (*lpDevice*, **DRAWPATTERNRECT**, *lpInData*, *lpOutData*)

This escape creates a pattern, gray scale, or solid black rectangle using the pattern/rule capabilities of PCL printers. With the HP LaserJet IIP, this escape can also create a solid white rectangle. A *gray scale* is a gray pattern that contains a specific mixture of black and white pixels. A PCL printer is an HP LaserJet or LaserJet-compatible printer.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a data structure containing the following items:

Field	Type/Definition
prPosition	POINT A point structure identifying the upper-left corner of the rectangle.
prSize	POINT A point structure identifying the lower-right corner of the rectangle.

	prStyle	WORD Specifies the type of pattern. It can be one of the following:
		Black Rule 0 White Rule 1 Gray Scale 2 HP-Defined 3
	prPattern	WORD Ignored for a black rule. It represents the percent of gray for a gray scale pattern. It represents one of six patterns for HP-defined patterns.
lpOutData	Not used and can be set to NULL.	

Return Value The return value is non-zero if the escape function is successful; otherwise, it is zero.

Comments An application should use QUERYESCSUPPORT to determine whether or not a device is capable of drawing patterns and rules before implementing this escape. If a printer is capable of outputting a white rule, the return value for QUERYESCSUPPORT is two.

The effect of a white rule is to erase any text or other pattern rules already written in the specified area. For example, it is possible to draw a large shaded area using **prStyle** = 1 and, then, print text in the erased area.

The driver sends all text and rules in band1 before any GDI bitmap graphics are sent. Therefore, it is not possible to erase bitmap graphics with white rules.

If an application uses the BANDINFO escape, all patterns and rectangles sent using DRAWPATTERNRECT should be enumerated as text and sent on a text band.

Patterns and rules created with this escape may not be erased by placing opaque objects over them unless you have white rule capability. An application should use the function calls provided in GDI to obtain this effect.

ENABLEDUPLEX (Escape # 28)

Syntax **short Control** (*lpDevice*, **ENABLEDUPLEX**, *lpInData*, *lpOutData*)

This escape enables the duplex printing capability of a printer. A device that has duplex printing capability is able to print on both sides of the output medium.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.

Parameter	Description
lpInData	A long pointer to a WORD that contains one of the following values: 0 = Simplex 1 = Duplex with vertical binding 2 = Duplex with horizontal binding
lpOutData	Not used and can be set to NULL.

Return Value The return value is one if the escape function is successful; otherwise, it is zero.

Comments An application should use the QUERYESCSUPPORT escape to determine whether or not an output device is capable of creating duplex output. If QUERYESCSUPPORT returns a nonzero value, the application should send the ENABLEDUPLEX escape even if simplex printing is desired. This guarantees the overriding of any values set in the driver-specific dialog. If duplex printing is enabled and an uneven number of NEXTFRAME escapes is sent to the driver prior to the ENDDOC escape, the driver will add one page eject before ending the print job.

ENABLEPAIRKERNING (Escape # 769)

Syntax **short Control** (*lpDevice*, **ENABLEPAIRKERNING**, *lpInData*, *lpOutData*)

This escape enables or disables the driver's ability to kern character pairs automatically. When it is enabled, the driver automatically kerns those pairs of characters that are listed in the font's character-pair kerning table. The driver reflects this kerning both on the printer and in **GetTextExtent** calls.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a short integer value that specifies whether or not automatic pair kerning is to be enabled (1) or disabled (zero).
lpOutData	A long pointer to a short integer variable that will receive the previous automatic pair-kerning flag.

Return Value The return value is one if the function is successful; zero if not or if the escape is not implemented.

Comments The default state of this capability is zero, that is, automatic character-pair kerning is disabled.

A driver does not have to support this escape just because it supplies the character-pair kerning table to the application via the GETPAIRKERNTABLE escape. When the GETPAIRKERNTABLE escape is supported but the ENABLEPAIRKERNING escape is not, it is the application's responsibility to space the kerned characters properly on the output device.

ENABLERELATIVEWIDTHS (Escape # 768)

Syntax

short Control (*lpDevice*, **ENABLERELATIVEWIDTHS**, *lpInData*, *lpOutData*)

This escape enables or disables relative character widths. When it is disabled (the default setting), each character's width can be expressed as an integer number of device units. This guarantees that the extent of a string will equal the sum of the extents of the characters in the string. Such behavior enables applications to build an extent table manually using one-character **GetTextExtent** calls. When it is enabled, the sum of a string may or may not equal the sum of the widths of the characters. Applications that enable this feature are expected to retrieve the font's extent table and compute relatively-scaled string widths themselves.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a short integer value that specifies whether or not relative widths are to be enabled (1) or disabled (zero).
lpOutData	A long pointer to a short integer variable that will receive the previous relative character-width flag.

Return Value

The return value is one if the function is successful; zero if not or if the escape is not implemented.

Comments

The default state of this capability is zero, that is, relative character widths are disabled.

Enabling this feature causes values that are specified as "font units" and accepted and returned by the escapes described in this chapter to be returned in the relative units of the font.

NOTE It is assumed that only linear scaling devices will be dealt with in a relative mode. Non-linear scaling devices should not implement this escape.

ENDDOC (Escape # 11)

Syntax **short Control** (*lpDevice*, **ENDDOC**, *lpInData*, *lpOutData*)

This escape ends a print job that is started by a STARTDOC escape and that is to be ended normally (i.e., not by an abort).

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpOutData	Not used and can be set to NULL.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

Comments On a printing error, the ENDDOC escape should not be used to terminate the printing operation.

END_PATH (Escape # 4098)

Syntax **short Control** (*lpDevice*, **END_PATH**, *lpInfo*, *lpOutData*)

This escape ends a path. A path is a connected sequence of primitives drawn in succession to form a single polyline or polygon. Paths enable applications to draw complex borders, filled shapes, and clipping areas by supplying a collection of other primitives defining the desired shape.

Printer escapes supporting paths enable applications to render images on sophisticated devices such as PostScript printers without generating huge polygons to simulate them.

To draw a path, an application first issues the BEGIN_PATH escape. It then draws the primitives defining the border of the desired shape and issues an END_PATH escape.

The END_PATH escape takes as a parameter a pointer to a structure specifying the manner in which the path is to be rendered. The structure specifies whether or not the path is to be drawn and whether or not it is open or closed. Open paths define polylines, and closed paths define fillable polygons.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInfo	A long pointer to a structure of type PATH_INFO, which is defined as follows:

```
typedef struct {
        short     RenderMode;
        BYTE      FillMode;
        BYTE      BkMode;
        LOGPEN    Pen;
        LOGBRUSH  Brush;
        DWORD     BkColor;
        }PATH_INFO;
```

The **RenderMode** field of the structure referenced by *lpInfo* specifies how the path is to be rendered. If **RenderMode** is one of these, then the path is as follows:

NO_DISPLAY(0).	Not drawn.
OPEN(1).	Drawn as an open polygon.
CLOSED(2).	Drawn as a closed polygon.

The **FillMode** field of the structure referenced by *lpInfo* specifies how the path is to be filled. If **FillMode** is one of these, then the fill is as follows:

ALTERNATE(1).	Done using the alternate-fill algorithm.
WINDING(2).	Done using the winding-fill algorithm.

The **BkMode** field of the structure referenced by *lpInfo* also specifies how the path is to be filled. This field is the equivalent of the **BkMode** field found in the DRAWMODE structure passed to **Output()**. Drivers that encounter a **BkMode** of zero should assume *Transparent* and ignore **BkColor**.

The **Pen** field of the structure referenced by *lpInfo* specifies the pen with which the path is to be drawn. If **RenderMode** is NO_DISPLAY, the pen is ignored.

The **Brush** field of the structure referenced by *lpInfo* specifies the brush with which the path is to be filled. If **RenderMode** is NO_DISPLAY or OPEN, the pen is ignored.

Parameter	Description
	The **BkColor** field of the structure referenced by *lpInfo* also specifies how the path is to be filled. This field is the equivalent of the **BkColor** field in the DRAWMODE structure passed to **Output()**.
lpOutData	Not used and can be set to NULL.

Return Value This escape returns a short integer value specifying the current path nesting level. If the escape is successful, the number of BEGIN_PATH calls without a corresponding END_PATH call is the result. Otherwise, -1 is the result.

Comments You may draw a path within another path. A path drawn within another path is treated exactly like a polygon (if the subpath is closed) or a polyline (if the subpath is open).

You may use the CLIP_TO_PATH escape to define a clipping area corresponding to the interior or exterior of the currently open path.

Device drivers that implement this escape must also implement the BEGIN_PATH escape and should also implement the SET_ARC_DIRECTION escape.

ENUMPAPERBINS (Escape # 31)

Syntax **short Control** (*lpDevice*, **ENUMPAPERBINS**, *lpInData*, *lpOutData*)

This escape retrieves attribute information about a specified number of paper bins. The GETSETPAPERBINS escape retrieves the number of bins available on a printer.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to an integer that specifies the number of bins for which information is to be retrieved.
lpOutData	A long pointer to a data structure to which information about the paper bins is copied. The size of the structure depends on the number of bins for which information was requested. See the following *Comments* section for a description of this data structure.

Return Value The return value specifies the outcome of the escape. It is one if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments

The value pointed to by *lpInData* is an integer that specifies the number of paper bins for which information is to be retrieved.

The *lpOutData* data structure consists of two arrays. The first is an array of short integers containing the paper-bin identifier numbers in the following format:

short BinList⟦*cBinMax*⟧

The number of integers in the array *cBinMax* is equal to the value pointed to by the *lpIn-Data* parameter.

The second array in the *lpOutData* structure is a ragged array of characters in the following format:

char PaperNames⟦*cBinMax*⟧⟦*cchBinName*⟧

The *cBinMax* value is equal to the value pointed to by the *lpInData* parameter; the *cchBin-Name* value is the length of each string (currently 24).

ENUMPAPERMETRICS (Escape # 34)

Syntax

short Control (*lpDevice*, **ENUMPAPERMETRICS**, *lpInData*, *lpOutData*)

This escape performs one of two jobs according to the mode. The first is to determine the number of paper types supported and return this value, which can then be used to allocate an array of RECTs.

These RECTs get filled in with the coordinates of the imageable area during a second call. That is, for example:

```
top    = cyMargin
left   = cxMargin
right  = cxPage + cxMargin
bottom = cyPage + cyMargin
```

Where:
The units are device coordinates.
The orientation returned is always portrait.
For every supported paper type, the imageable area for each margin state is returned.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	If the long pointer points to zero, then the return value indicates how many RECTs to allocate for subsequent calls.
lpOutData	If *lpInData* points to a one, then *lpOutData* is filled with RECTs that describe all the imageable areas supported, as shown in the code example given in the following *Comments* section.

Return Value It is positive if successful, zero if the escape is not implemented, and negative if an error occurs.

Comments The following is an example of code:

```
#define ENUMPAPERMETRICS        (34)
#define INFORM                  (0)
#define PERFORM                 (1)

int result;
HANDLE  hDCPrinter;
HANDLE  hPapers;
LPRECT  lpPapers;
int nPapers, j;

j = INFORM;
result = Escape (hDCPrinter,ENUMPAPERMETRICS,sizeof(int),(LPSTR)&j,(LPSTR)&nPapers);

if (hPapers = GlobalAlloc(GPTR,(long int) (nPapers * sizeof(RECT))))

{
lpPapers = (LPRECT)GlobalLock(hPapers);

j = PERFORM;
result = Escape (hDCPrinter,ENUMPAPERMETRICS,sizeof(int),(LPSTR)&j,(LPSTR)lpPapers);

...perform operations
GlobalUnlock(hPapers);
GlobalFree(hPapers);
}
```

EPSPRINTING (Escape # 33)

Syntax **short Control (*lpDevice*, EPSPRINTING, *lpBool*, *lpOutData*)**

This escape only controls the downloading of the control portions of the PostScript prolog. Its functionality includes CYM (Cyan, Yellow, Magenta) setup and automatic font downloading.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpBool	A long pointer to a flag indicating that this should be turned on (TRUE or 1) or off (FALSE or 0).

Return Value

It is positive if successful, zero if the escape is not implemented, and negative if an error occurs.

Comments

Coding example:

```
#define EPSPRINTING     (33)
#define ON              (1)
#define OFF             (0)

int result;
HANDLE hDCPrinter;
bool fEps;
fEps = ON

result = Escape (hDCPrint,EPSPRINTING,sizeof(bool),(LPSTR)&fEps,NULL);

...setup
...print
...cleanup
fEps = OFF;

result = Escape (hDCPrint,EPSPRINTING,sizeof(bool),(LPSTR)&fEps,NULL);
```

This escape is used to suppress the output of the Windows PostScript header control section, which is about 10K. If it is used, *no* GDI calls are allowed.

EXT_DEVICE_CAPS (Escape # 4099)

Syntax

short Control (*lpDevice*, **EXT_DEVICE_CAPS**, *lpIndex*, *lpCaps*)

This escape retrieves information about device-specific capabilities. It serves as a supplement to the **GetDeviceCaps** function supplied by Windows.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpIndex	A long pointer to a short integer specifying the index of the capability to be retrieved.
lpCaps	A long pointer to a 32-bit integer into which the capabilities will be copied. The following capabilities are supported:

Parameter	Description

R2_CAPS(1). Specifies which of the 16 binary raster operations (see the SetROP2 function, which is documented in the *Microsoft Windows Software Development Kit*) are supported by the device driver. A bit will be set for each supported raster operation. For example, the following code fragment tests for support of the R2_XORPEN raster operation.

```
((1<<R2_XORPEN)&&Caps)!=0
```

PATTERN_CAPS(2). Specifies the maximum dimensions of a pattern brush bitmap. The low-order WORD of the capability value contains the maximum width of a pattern brush bitmap; the high-order WORD contains the maximum height.

PATH_CAPS(3). Specifies whether or not the device is capable of creating paths using alternate and winding interiors, and whether or not the device can do exclusive or inclusive clipping to path interiors. The path capabilities are obtained by OR'ing together the following values:

PATH_ALTERNATE	1
PATH_WINDING	2
PATH_INCLUSIVE	4
PATH_EXCLUSIVE	8

POLYGON_CAPS(4). Specifies the maximum number of polygon points supported by the device. The capability value is an unsigned value specifying the maximum number of points.

PATTERN_COLOR_CAPS(5). Specifies whether or not the device can convert monochrome pattern bitmaps to color. The capability value is one if the device can do pattern bitmap color conversions and zero if it cannot.

R2_TEXT_CAPS(6). Specifies whether or not the device is capable of performing binary raster operations on text. The low-order WORD of the capability value specifies which raster operations are supported on text. A bit is set for each supported raster operation, as in the R2_CAPS escape. The high-order WORD specifies to which type of text the raster capabilities apply. It is obtained by OR'ing together the following values:

RASTER_TEXT	1
DEVICE_TEXT	2
VECTOR_TEXT	4

Return Value This escape returns a nonzero value if the specified extended capability is supported and a zero if it is not.

Comments A device driver implementing this escape must not modify the value of the 32-bit integer described by *lpCaps* unless it returns a valid value for the capability.

EXTTEXTOUT (Escape # 512)

This is an older escape that has been replaced by the **ExtTextOut** driver function.

FLUSHOUTPUT (Escape # 6)

Syntax **short Control** (*lpDevice*, **FLUSHOUTPUT**, *lpInData*, *lpOutData*)

This escape flushes output in the device's buffer.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpOutData	Not used and can be set to NULL.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

Comments This escape is intended for banding printer drivers. When called, they should reinitialize the banding bitmap (i.e., eliminate anything in the bitmap that is only partially drawn).

GETCOLORTABLE (Escape # 5)

Syntax **short Control** (*lpDevice*, **GETCOLORTABLE**, *lpIndex*, *lpColor*)

This escape retrieves an RGB color-table entry and copies it to the location specified by *lpColor*.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.

lpIndex	A 32-bit address to a short integer value specifying the index of a color-table entry. Color-table indices start at zero for the first table entry.
lpColor	A 32-bit address to the long integer to receive the RGB color value for the given entry.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

Comments Display drivers will probably want to support this escape. It returns RGB colors that are mapped to the physical-color index passed in.

GETEXTENDEDTEXTMETRICS (Escape # 256)

Syntax **short Control** (*lpDevice*, **GETEXTENDEDTEXTMETRICS**, *lpInData*, *lpOutData*)

This escape fills the buffer pointed to by *lpOutData* with the extended text metrics for the currently selected font.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a data structure of type EXTTEXTDATA described in the following *Comments* section.
lpOutData	A long pointer to a data structure of type EXTTEXTMETRIC. See the following *Comments* section for a description of this data structure.

Return Value The return value is the number of bytes copied to the buffer pointed to by *lpOutData*. This value will never exceed *nSize* and will be zero if the function fails or the escape is not implemented.

Comments The EXTTEXTDATA structure pointed to by *lpIndata* contains the following items:

```
typedef struct {
        short               nSize;
        LPAPPEXTTEXTDATA   lpInData;
        LPFONTINFO         lpFont;
        LPTEXTXFORM        lpXForm;
        LPDRAWMODE         lpDrawMode;
        }EXTTEXTDATA;
```

The *lpInData* field points to a WORD that contains the number of bytes pointed to by *lpOutData.*

The values returned in many of the fields of the EXTTEXTMETRIC structure are affected by whether relative character widths are enabled or disabled. See also the ENABLE-RELATIVEWIDTHS escape.

The EXTTEXTMETRIC data structure has the following format:

```
typedef struc{
        short  etmSize;
        short  etmPointSize;
        short  etmOrientation;
        short  etmMasterHeight;
        short  etmMinScale;
        short  etmMaxScale;
        short  etmMasterUnits;
        short  etmCapHeight;
        short  etmXHeight;
        short  etmLowerCaseAscent;
        short  etmUpperCaseDescent;
        short  etmSlant;
        short  etmSuperScript;
        short  etmSubScript;
        short  etmSuperScriptSize;
        short  etmSubScriptSize;
        short  etmUnderlineOffset;
        short  etmUnderlineWidth;
        short  etmDoubleUpperUnderlineOffset;
        short  etmDoubleLowerUnderlineOffset;
        short  etmDoubleUpperUnderlineWidth;
        short  etmDoubleLowerUnderlineWidth;
        short  etmStrikeOutOffset;
        short  etmStrikeOutWidth;
        WORD   etmKernPairs;
        WORD   etmKernTracks;
        }EXTTEXTMETRIC;
```

GETEXTENTTABLE (Escape # 257)

Syntax **short Control** (*lpDevice*, **GETEXTENTTABLE**, *lpInData*, *lpOutData*)

This escape returns the width (extent) of individual characters from a group of consecutive characters in the selected font's character set. The first and last character (from the group of consecutive characters) are function arguments.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.

lpInData	A long pointer to a data structure described in the following Comments section.
lpOutData	A long pointer to an array of type **short**. The size of the array must be at least (*chLast*—*chFirst* +1).

Return Value The return value is one if the function is successful and zero if it is not or if the escape is not implemented.

Comments The structure pointed to by *lpInData* contains the following items:

```
BYTE chFirst;
BYTE chLast;
```

The *chFirst* argument contains the character code of the first character.

The *chLast* argument contains the character code of the last character.

The values returned are affected by whether relative character widths are enabled or disabled. See also the ENABLERELATIVEWIDTHS escape.

GETFACENAME (Escape # 513)

Syntax **short Control** (*lpDevice*, **GETFACENAME**, *lpInData*, *lpFaceName*)

This escape gets the facename of the current physical font.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpFaceName	A long pointer to the buffer for the facename.

Return Value It is positive if successful, zero if the escape is not implemented, and negative if an error occurs.

Comments The following is an example of code:

```
#define GETFACENAME  (513)
int result;
HANDLE hDCPrinter;
char szFacename [60];   /* must be at least 60 bytes */
```

```
if (!(result =  Escape (hDCPrint, GETFACENAME, 0,
                (LPSTR)NULL, szFacename))
       {
       /* Handle error condition */
       }
```

GETPAIRKERNTABLE (Escape # 258)

Syntax **short Control** (*lpDevice*, **GETPAIRKERNTABLE**, *lpInData*, *lpOutData*)

This escape fills the buffer pointed to by *lpOutData* with the character pair-kerning table for the currently selected font.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpOutData	A long pointer to an array of KERNPAIR structures. (See the following Comments section for a description of this data structure.) This array must be large enough to accommodate the font's entire character pair-kerning table. The number of character kerning pairs in the font can be obtained from the EXTTEXTMETRIC structure returned by the GETEXTENDEDTEXTMETRICS escape.

Return Value The return value is the number of KERNPAIR structures copied to the buffer. This value is zero if the font does not have kerning pairs defined, the function fails, or the escape is not implemented.

Comments The values returned in the KERNPAIR structures are affected by whether relative character widths are enabled or disabled. See also the ENABLERELATIVEWIDTHS escape.

The KERNPAIR data structure has the following format:

```
typedef struc {
       union {
            BYTE each [2];    /* UNION: 'each' and 'both'
                                   share the same memory */
            WORD both;
            } kpPair;
       short  kpKernAmount;
       } KERNPAIR;
```

GETPHYSPAGESIZE (Escape # 12)

Syntax **short Control** (*lpDevice*, **GETPHYSPAGESIZE**, *lpInData*, *lpDimensions*)

This escape retrieves the physical page size in device units (i.e., how many pixels wide by how many scanlines high) and copies it to the location pointed to by *lpOutData*.

Parameter	**Description**
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpDimensions	A 32-bit address to the POINT data structure to receive the physical page dimensions. The fields in the structure are filled as follows:

Field	**Contents**
int x	Horizontal size in device units.
int y	Vertical size in device units.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

GETPRINTINGOFFSET (Escape # 13)

Syntax **short Control** (*lpDevice*, **GETPRINTINGOFFSET**, *lpInData*, *lpOffset*)

This escape retrieves the offset from location 0,0 (the upper left-hand corner of the physical page), the point at which the actual printing or drawing begins.

This escape function is not generally useful for devices that allow the user to set the printable origin by hand.

Parameter	**Description**
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpOffset	A 32-bit address to the POINT structure to receive the printing offset. The fields of the structure are filled as follows:

Field	**Contents**

Parameter	Description	
	int x	Horizontal coordinate in device units of the printing offset.
	int y	Vertical coordinate in device units of the printing offset.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

GETSCALINGFACTOR (Escape # 14)

Syntax **short Control** (*lpDevice*, **GETSCALINGFACTOR**, *lpInData*, *lpFactors*)

This escape retrieves the scaling factors for the *x* and *y* axes of a printing device. For each scaling factor, the escape copies an exponent of two to the location pointed to by *lpFactors*.

For example, the value 3 is copied to *lpFactors* for a scaling factor of 8.

Scaling factors are used by printing devices that cannot support graphics at the same resolution as the device resolution. This escape communicates to GDI the factor by which it needs to stretch bitmaps when drawing them to the printer.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpFactors	A 32-bit address to the POINT data structure to receive the scaling factor. The fields of the structure are filled as follows:

Field	Contents
int x	Scaling factor for *x* axis.
int y	Scaling factor for *y* axis.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

GETSETPAPERBINS (Escape # 29)

Syntax **short Control** (*lpDevice*, **GETSETPAPERBINS**, *lpInData*, *lpOutData*)

This escape retrieves the number of paper bins available on a printer and sets the current paper bin. See the following *Comments* section for more information on the actions performed by this escape.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a BININFO data structure that specifies the new paper bin. See the following *Comments* section for a description of this data structure, which may be set to NULL.
lpOutData	A long pointer to a BININFO data structure that contains information about the current or previous paper bin and the number of bins available. This data structure may be set to NULL.

Return Value

The return value specifies whether or not the requested operation was successful.

Comments

There are three possible actions for this escape, depending on the value of the values passed in the *lpInData* and *lpOutData* parameters:

lpInData	*lpOutData*	Action
NULL	BININFO	Retrieves the number of bins and the number of the current bin.
BININFO	BININFO	Sets the current bin to the number specified in the **BinNumber** field of the *lpInData* data structure and retrieves the number of the previous bin.
BININFO	NULL	Sets the current bin to the number specified in the **BinNumber** field of the *lpInData* data structure.

The BININFO data structure has the following format:

```
short   BinNumber;
short   NbrofBins;
short   Reserved;
short   Reserved;
short   Reserved;
short   Reserved;
```

The BININFO data structure has the following fields:

Field	Description
BinNumber	Identifies the current or previous paper bin.

NbrofBins	Specifies the number of paper bins available.

When setting the paper bin with GETSETPAPERBINS, the bin selected is set for the current job by setting the MSB (0x8000) of the bin index. If this bit is not set, the selected paper bin is made the default for later print jobs, and the current job's selection is unchanged. Setting bit 15 enables an application to change bins in mid-job.

This escape was supported in version 3.1 of the PCL and PostScript drivers. However, it did not allow the setting of the current bin, only the default. GETSETPAPERBINS will return an error if the high bit is set on this version of the drivers, which allows the application to determine whether or not the older driver version is in use.

This escape was inconsistently numbered as 30 in the Windows 2.1 PCL driver and as 29 in the PostScript driver. This has now been corrected, and the numbering is consistent with this document.

GETSETPAPERMETRICS (Escape # 35)

Syntax **short Control** (*lpDevice*, **GETSETPAPERMETRICS**, *lpNewPaper, lpOrigPaper)*

This escape sets the paper type according to the given paper-metrics information. It also gets the current printer's paper-metrics information.

This escape expects a RECT data structure, representing the imageable area of the physical page, and assumes the origin is in the upper-left corner. That is, for example,

```
top     =  cyMargin
left    =  cxMargin
right   =  cxPage + cxMargin
bottom  =  cyPage + cyMargin
```

Where:
The units expected and returned are coordinates in device units.
The orientation is set to match that of the input RECT.
The margin state is set.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpNewPaper	A long pointer to a RECT data structure that points to the new imageable area.
lpOrigPaper	A long pointer to a RECT data structure that points to the original value.

Return Value It is positive if successful, zero if the escape is not implemented, and negative if an error occurs.

Comments The following is an example of code:

```
#define  GETSETPAPERMETRICS (35)
int result;
HANDLE  hDCPrinter;
RECT newpaper, origpaper;

/* assuming newpaper's fields have been properly assigned */
result = Escape (hDCPrint,GETSETPAPERMETRICS,sizeof(struct RECT),
         (LPSTR)&newpaper,(LPSTR)&origpaper);

/* or if you do not care about restoring the original paper type */
result = Escape (hDCPrint,GETSETPAPERMETRICS,sizeof(struct RECT),
         (LPSTR)&newpaper,NULL);

/* check to ensure current printer supports the requested paper size */
if (!result)
       {
         ...error dialog - unable to set paper size...
       }

/* or if you only want the original paper type */
result = Escape (hDCPrint,GETSETPAPERMETRICS,0,NULL,(LPSTR)
         &origpaper);
```

GETSETPRINTORIENT (Escape # 30)

Syntax **short Control** (*lpDevice*, **GETSETPRINTORIENT**, *lpInData*, *lpOutData*)

This escape returns or sets the current paper orientation.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to an ORIENT data structure that specifies the new paper orientation. See the following *Comments* section for a description of this data structure. When it is set to NULL, the GETSETPRINTORIENT escape returns the current paper orientation.
lpOutData	Not used and can be set to NULL.

Return Value The return value specifies the current orientation if *lpInData* is NULL; otherwise, it is the previous orientation or -1 if the escape failed.

Comments

The ORIENT data structure has the following format:

```
short  Orientation;
short  Reserved;
short  Reserved;
short  Reserved;
short  Reserved;
```

If Orientation is 1, the new orientation is portrait; if 2, the new orientation is landscape.

The new orientation will take effect for the *next* device context created for the device on this port.

This escape is also known as GETSETPAPERORIENTATION.

This escape was inconsistently numbered as 29 in the Windows 2.1 PCL driver and as 30 in the PostScript driver. This has now been corrected, and the numbering is consistent with this document.

GETTECHNOLOGY (Escape # 20)

Syntax

short Control (*lpDevice*, GETTECHNOLOGY, *lpInData*, *lpTechnology*)

This escape retrieves the general technology type for a printer. This allows an application to perform technology-specific actions.

Parameter	**Description**
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpTechnology	A long pointer to a buffer to which the driver copies a NULL-terminated string containing the printer technology type, such as "PostScript."

Return Value

The return value specifies the outcome of the escape. It is one if the escape is successful and zero if the escape is not successful or not implemented.

GETTRACKKERNTABLE (Escape # 259)

Syntax

short Control (*lpDevice*, GETTRACKKERNTABLE, *lpInData*, *lpOutData*)

This escape fills the buffer pointed to by *lpOutData* with the track-kerning table for the currently selected font.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpOutdata	A long pointer to an array of KERNTRACK structures. (See the following *Comments* section for a description of this data structure.) This array must be large enough to accomodate all the font's kerning tracks. The number of tracks in the font can be obtained from the EXTTEXTMETRIC data structure returned by the GETEXTENDEDTEXTMETRICS escape. If *lpOutData* is NULL, GETTRACKKERNTABLE returns the number of table entries.

Return Value

The return value is the number of KERNTRACK structures copied to the buffer. This value is zero if the font does not have kerning tracks defined, if the function fails, or if the escape is not implemented.

Comments

The values returned in the KERNTRACK structures are affected by whether relative character widths are enabled or disabled. See also the ENABLERELATIVEWIDTHS escape.

The KERNTRACK data structure has the following format:

```
typedef struc {
        short  ktDegree;
        short  ktMinSize;
        short  ktMinAmount;
        short  ktMaxSize;
        short  ktMaxAmount;
        } KERNTRACK;
```

GETVECTORBRUSHSIZE (Escape # 27)

Syntax

short Control (*lpDevice*, **GETVECTORBRUSHSIZE**, *lpInData*, *lpOutData*)

This escape retrieves the size in device units of a plotter pen used to fill closed figures. GDI uses this information to prevent the filling of closed figures (e.g., rectangles and ellipses) from overwriting the borders of the figure.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which contains the device context for the device on which the metafile appears.

Parameter	Description
lpInData	A long pointer to a logical brush data structure that specifies the brush for which data is to be returned.
lpOutData	A long pointer to a POINT data structure that contains in its second WORD the width of the pen in device units. See the following *Comments* section for a description of this data structure.

Return Value The return value specifies the outcome of the escape. It is one if the escape is successful and zero if the escape is not successful or not implemented.

Comments The POINT data structure has the following format:

```
short  x;
short  y;
```

GETVECTORPENSIZE (Escape # 26)

Syntax **short Control** (*lpDevice*, **GETVECTORPENSIZE**, *lpInData*, *lpOutData*)

This escape retrieves the size in device units of a plotter pen. GDI uses this information to prevent hatched brush patterns from overwriting the border of a closed figure.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which contains the device context for the device on which the metafile appears.
lpInData	A long pointer to a logical pen data structure that specifies the pen for which the width is to be retrieved.
lpOutData	A long pointer to a POINT data structure that contains in its second WORD the width of the pen in device units. See the following Comments section for a description of this data structure.

Return Value The return value specifies the outcome of the escape. It is one if the escape is successful and zero if the escape is not successful or not implemented.

Comments The POINT data structure has the following format:

```
short  x;
short  y;
```

NEWFRAME (Escape # 1)

Syntax **short Control** (*lpDevice*, **NEWFRAME**, *lpInData*, *lpOutData*)

This escape informs the device that the application has finished writing to a page.

This escape is used typically with a printer to direct the device driver to advance to a new page by performing a page-break algorithm or form feed.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpOutData	Not used and can be set to NULL.

Return Value The return value is positive if the escape is successful. Otherwise, it is one of the values documented in Section 11.2, "Generalized Error Return Codes."

Comments If you select a font into your printer device context (DC), print some text, and then issue a NEWFRAME, the following text is printed using the *default* font. What happens is that NEWFRAME saves the DC and, then, restores it with the default values. Therefore, you will have to reselect your font after each new page on banding devices.

NEXTBAND (Escape # 3)

Syntax **short Control** (*lpDevice*, **NEXTBAND**, *lpInData*, *lpBandRect*)

This escape informs the device driver that the application has finished writing to a band. This causes the device driver to send the band to the printer and return the coordinates of the next band.

This escape is used by applications that handle banding themselves.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a PTTYPE structure that specifies the shift count for scaling graphics coordinates.
lpBandRect	The return value is a 32-bit address to the RECT data structure to receive the next band coordinates. The device driver copies the device coordinates of the next band into this structure.

Return Value The return value is positive if the escape is successful. Otherwise, it is one of the values documented in Section 11.2, "Generalized Error Return Codes."

Comments This escape is intended mainly for banding printers.

The shift count mentioned in *lpInData* is used for devices such as laser printers that support graphics at a lower resolution than text. The *x* coordinate specifies the X scale count and the *y* coordinate the Y scale count.

PASSTHROUGH (Escape #19)

Syntax **short Control** (*lpDevice*, **PASSTHROUGH**, *lpInData*, *lpOutData*)

This escape enables the application to send data directly to the printer, bypassing the standard printer-driver code.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Points to a structure, the first WORD of which contains the number of bytes of input data. The remaining bytes of the structure contain the data itself.
lpOutData	Not used and can be set to NULL.

Return Value This is the number of bytes transferred to the printer if the function is successful; it is zero if the function is not successful or if the escape is not implemented. If the returned value is nonzero but less than the size of the data, an error prohibited transmission of the entire data block.

Comments

There may be restrictions on the kinds of device data an application may send to the device without interfering with the operation of the driver. In general, applications must avoid resetting the printer or causing the page to be printed. Additionally, applications are strongly discouraged from performing functions that consume printer memory, such as downloading a font or a macro.

The driver should invalidate its internal state variables such as "current position" and "current line style" when executing this escape. The driver may issue a printer "save" command prior to transmitting the first of a sequence of PASSTHROUGH escapes and issue a "restore" command prior to executing the first command after the last PASSTHROUGH escape. In other words, the application must be able to issue multiple, sequential PASSTHROUGH escapes without intervening "saves" and "restores" being inserted by the driver.

This escape is also known as DEVICEDATA.

QUERYESCSUPPORT (Escape # 8)

Syntax

short Control (*lpDevice*, **QUERYESCSUPPORT**, *lpEscNum*, *lpOutData*)

This escape finds out whether or not a particular escape function is implemented by the device driver.

The return value is non-zero for implemented escape functions and zero for unimplemented escape functions. All device drivers must return success if queried about whether they support the QUERYESCSUPPORT escape.

Parameter	**Description**
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpEscNum	A 32-bit address to a short integer value specifying the escape function to be checked.
lpOutData	Not used and can be set to NULL.

Return Value

The return value is non-zero for implemented escape functions; otherwise, it is zero.

RESTORE_CTM (Escape # 4100)

Syntax

short Control (*lpDevice*, **RESTORE_CTM**, *lpInData*, *lpOutData*)

This escape restores the previously saved current transformation matrix (CTM). The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, you can combine these operations in any order to produce the desired mapping for a particular picture.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.
lpOutData	Not used and can be set to NULL.

Return Value This escape returns a short integer specifying the number of SAVE_CTM calls without a corresponding RESTORE_CTM call. If the escape is unsuccessful, -1 is returned as a result.

Comments Applications should not make any assumptions about the initial contents of the current transformation matrix.

Drivers supporting this escape must also implement the SAVE_CTM and TRANSFORM_CTM escapes.

The matrix specification used for this escape is based on the OS/2 GPI (Graphics Programming Interface) model, which is an integer coordinate system similar to the one used by GDI.

SAVE_CTM (Escape # 4101)

Syntax **short Control** (*lpDevice*, SAVE_CTM, *lpInData*, *lpOutData*)

This escape saves the current transformation matrix (CTM). The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, you can combine these operations in any order to produce the desired mapping for a particular picture.

You can restore the matrix by using the RESTORE_CTM escape.

An application typically saves the current transformation matrix before changing it. This enables the application to restore the previous state upon completion of a particular operation.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	Not used and can be set to NULL.

Parameter	Description
lpOutData	Not used and can be set to NULL.

Return Value This escape returns a short integer specifying the number of SAVE_CTM calls without a corresponding RESTORE_CTM call. If the escape is unsuccessful, zero is returned as a result.

Comments Applications should not make any assumptions about the initial contents of the current transformation matrix.

Applications are expected to restore the contents of the current transformation matrix.

Drivers supporting this escape must also implement the RESTORE_CTM and TRANSFORM_CTM escapes.

The matrix specification used for this escape is based on the OS/2 GPI (Graphics Programming Interface) model, which is an integer coordinate system similar to the one used by GDI.

SELECTPAPERSOURCE (Escape # 18)

This is an older escape and is no longer used. It has been superceded by GETSETPAPERBINS.

It enabled the application to determine the available paper sources, select among them, and pass the desired paper source to the device driver.

SETABORTPROC (Escape # 9)

Syntax **short Control** (*lpDevice*, **SETABORTPROC**, *lphDC*, *lpOutData*)

This escape sets the abort function for the print job.

If an application wants to allow the print job to be cancelled during spooling, it must set the abort function before the print job is started with the STARTDOC escape. Print Manager calls the abort function during spooling to allow the application to cancel the print job or to handle out-of-disk-space conditions. If no abort function is set, the print job will fail if there is not enough disk space for spooling.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.

Parameter	Description
lphDC	A long pointer to a handle to the application's device context for the print job.
lpOutData	Not used and can be set to NULL.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

Comments *lphDC* points to the application hDC which is to be passed to the **OpenJob** function to allow GDI to call the application's callback function.

SETALLJUSTVALUES (Escape # 771)

Syntax **short Control** (*lpDevice*, **SETALLJUSTVALUES**, *lpInData*, *lpOutData*)

This escape sets all the text justification values that are used for text output. Text justification is the process of inserting extra pixels among break-characters in a line of text. The blank character is normally used as a break character.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a data structure containing the following items:
lpOutData	Not used and can be set to NULL.

Field	Type/Definition
nCharExtra	**short** Specifies in font units the total extra space that must be distributed over **nCharCount** characters.
nCharCount	**WORD** Specifies the number of characters over which **nCharExtra** is distributed.
nBreakExtra	**short** Specifies in font units the total extra space that is distributed over **nBreakCount** break characters.
nBreakCount	**WORD** Specifies the number of break characters over which **nBreakExtra** units are distributed.

Return Value

The return value is one if the escape function is successful; otherwise, it is zero.

Comments

The units used for **nCharExtra** and **nBreakExtra** are the font units of the device (see the EXTTEXTMETRIC escape) and are dependent on whether or not relative character widths were enabled with the ENABLERELATIVEWIDTHS escape.

The values set with this escape will apply to subsequent calls to **ExtTextOut**. The driver will stop distributing the **nCharExtra** amount when it has output **nCharCount** characters. It will also stop distributing the space specified by **nBreakExtra** when it has output **nBreakCount** characters. A call on the same string to **GetTextExtent** made immediately after the **ExtTextOut** call is processed in the same manner.

To reenable justification with the **SetTextJustification** and **SetTextCharacterExtra** functions, an application should call SETALLJUSTVALUES and set the arguments **nCharExtra** and **nBreakExtra** to zero.

SET_ARC_DIRECTION (Escape # 4102)

Syntax

short Control (*lpDevice*, SET_ARC_DIRECTION, *lpDirection*, *lpOutData*)

This escape specifies the direction in which elliptical arcs are drawn using the GDI arc function.

By convention, elliptical arcs are drawn counterclockwise by GDI. This escape enables an application to draw paths containing arcs drawn clockwise.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpDirection	A long pointer to a short integer specifying the arc direction. It may be either COUNTERCLOCKWISE(0) or CLOCKWISE(1).
lpOutData	Not used and can be set to NULL.

Return Value

This escape returns the previous arc direction.

Comments

The default arc direction is COUNTERCLOCKWISE.

Device drivers that implement the BEGIN_PATH and END_PATH escapes should also implement this escape.

This escape maps to PostScript language elements and is intended for PostScript line devices.

SET_BACKGROUND_COLOR (Escape # 4103)

Syntax

short Control (*lpDevice*, SET_BACKGROUND_COLOR, *lpNewColor*, *lpOldColor*)

This escape enables an application to set and retrieve the current background color for the device. The background color is the color of the display surface before an application draws anything on the device. This escape is particularly useful for color printers and film recorders.

This escape should be sent before the application draws anything on the current page.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpNewColor	A long pointer to a 32-bit integer specifying the desired background color. This parameter can be NULL if the application merely wants to retrieve the current background color.
lpOldColor	A long pointer to a 32-bit integer into which the previous background color will be copied. This parameter can be NULL if the application wants to ignore the previous background color.

Return Value

This escape returns TRUE if the escape is successful and FALSE if it is unsuccessful.

Comments

The default background color is white.

The background color is reset to the default when the device driver receives an ENDDOC or ABORTDOC escape.

SET_BOUNDS (Escape # 4109)

Syntax

short Control (*lpDevice*, SET_BOUNDS, *lpBounds*, *lpOutData*)

This escape sets the bounding rectangle for the picture being output by the device driver that supports the given device context. It is used when creating images in a file format such as Encapsulated PostScript (EPS) and Hewlett Packard Graphics Language (HPGL) for which there is a device driver.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.

Parameter	Description
lpBounds	A long pointer to a rectangle, specified in device coordinates, that bounds the image to be output.
lpOutData	Not used and can be set to NULL.

Return Value This escape returns TRUE if successful; otherwise, it returns FALSE.

Comments The application should issue this escape before each page in the image. For single-page images, this escape should be issued immediately before the STARTDOC escape.

When using coordinate transformation escapes, device drivers may not perform bounding-box calculations correctly. Using this escape saves the driver from the task of calculating the bounding box.

Applications that want to support the EPS printing capabilities that will be built into future PostScript drivers should always issue this escape.

SETCOLORTABLE (Escape # 4)

Syntax **short Control** (*lpDevice*, **SETCOLORTABLE**, *lpColorEntry*, *lpColor*)

This escape sets an RGB color table entry.

If the device cannot supply the exact color, the function sets the entry to the closest possible approximation of the color.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpColorEntry	A 32-bit address to a color table entry data structure. The structure has the following fields:

Field	Type/Definition
Index	**WORD** The color table index. Color table entries start at zero for the first entry.
rgb	**LONG** The desired RGB color value.

Parameter	Description
lpColor	A 32-bit address to the long integer to receive the RGB color value selected by the device driver to represent the requested color value.

Return Value The return value is positive if the function is successful; otherwise, it is negative.

Comments A device's color table is a shared resource; changing the system display color for one window changes it for all the windows.

The SETCOLORTABLE escape has no effect on devices with fixed color tables.

This escape is intended for use by both printer and display drivers. However, the EGA and VGA color drivers do not support it. It should *not* be used with palette-capable display devices.

It is used by applications that want to change the palette used by the display driver. 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 necessarily know the proper color-mapping algorithm and take responsibility for passing the correctly mapped *physical* color to the driver (instead of the *logical* RGB color) in functions 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 wants to reactivate the driver's color-mapping algorithm (i.e., if it restores the original palette when switching from its window context), then the color index pointed to by *lpColorEntry* should be FFFEH.

SETCOPYCOUNT (Escape # 17)

Syntax **short Control** (*lpDevice*, SETCOPYCOUNT, *lpInData*, *lpOutData*)

This escape specifies the number of uncollated copies of each page that the printer is to print.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a short integer value containing the number of uncollated copies to be printed.

Parameter	Description
lpOutData	A long pointer to a short integer variable that will receive the number of copies to be printed. This may be less than the number requested if the requested number is greater than the device's maximum copy count.

Return Value The return value is one if the function is successful and zero if it is not or if the escape is not implemented.

SETDIBSCALING (Escape # 32)

Syntax **short Control** (*lpDevice*, **SETDIBSCALING**, *lpInData*, *lpOutData*)

This escape sets the scaling parameters for subsequent **SetDIBitsToDevice()** calls. This allows devices that can scale device independent bitmaps (DIBs) to do so.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	An LPSTR that points to a DIBSCALE structure that specifies the scaling mode.
lpOutData	Not used and can be set to NULL.

Return Value Returns the previous **ScaleMode** or -1 if an invalid scale mode was requested.

Comments *lpInData* points to a DIBSCALE structure that looks like the following one:

```
typedef struc{
        short  ScaleMode;
        short  dx;
        short  dy;
        }DIBSCALE;
```

ScaleMode can be one of the following values:

Value	Meaning
0	No scaling should be performed. DIBs are mapped directly to device coordinates, which are the default mode values.

Value	Meaning
1	The DIB is scaled to match its true physical size as indicated by the following fields in the BITMAPINFOHEADER data structure: biSize biWidth biHeight biPlanes biBitCount biCompression biSizeImage biXPelsPerMeter biYPelsPerMeter biClrUsed biClrImportant
2	The DIB is scaled to be **dx** by **dy** device units in size. This gives the application complete control over the size of the DIB.

SETKERNTRACK (Escape # 770)

Syntax

short Control (*lpDevice*, **SETKERNTRACK**, *lpInData*, *lpOutData*)

For drivers that support automatic track kerning, this escape specifies which kerning track will be used. A kerning track of zero disables automatic track kerning. When this escape is enabled, the driver automatically kerns all characters according to the specified track. The driver reflects this kerning both on the printer and in **GetTextExtent** calls.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a short integer value that specifies the kerning track to use. A value of zero disables this feature. Values one to **nKernTracks** (see the EXTTEXTMETRIC data structure provided under the description of the GETEXTENDEDTEXTMETRICS escape) correspond to positions in the track-kerning table (using one as the first item in the table).
lpOutData	A long pointer to a short integer variable that will receive the previous kerning track.

Return Value

The return value is one if the escape is successful and zero if it is not or if the escape is not implemented.

Comments

The default state of this capability is zero, which means that automatic track kerning is disabled.

A driver does not have to support this escape just because it supplies the track-kerning table to the application using the GETTRACKKERNTABLE escape. In the case where GETTRACKKERNTABLE is supported but SETKERNTRACK is not, it is the application's responsibility to properly space the characters on the output device.

SETLINECAP (Escape # 21)

Syntax

short Control (*lpDevice*, **SETLINECAP**, *lpInData*, *lpOutData*)

This escape sets the line end cap. An end cap is that portion of a line segment that appears on either end of the segment. The cap may be square or circular; it can extend past, or remain flush with, the specified end points.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to a short integer value that specifies the end-cap type. The possible values and their meanings are given in the following list:

Value	Meaning
-1	Line segments are drawn by using the default GDI end cap.
0	Line segments are drawn with a squared end point that does not project past the specified segment length.
1	Line segments are drawn with a rounded end point; the diameter of this semicircular arc is equal to the line width.
2	Line segments are drawn with a squared end point that projects past the specified segment length. The projection is equal to half the line width.

lpOutData	A long pointer to a short integer value that specifies the previous end-cap setting.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful; otherwise, it is negative.

Comments

The interpretation of this escape varies with page-description languages (PDLs). Consult your PDL documentation for its exact meaning.

This escape is also known as SETENDCAP.

SETLINEJOIN (Escape # 22)

Syntax

short Control (*lpDevice*, SETLINEJOIN, *lpInData*, *lpOutData*)

This escape is used to tell the driver how an application wants to join two lines at an angle. It sets the current line join parameter in the graphics state to *int*, which must be one of the following integers: 0 (for Miter Join), 1 (for Round Join), or 2 (for Bevel Join).

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to the WORD-length, line join value to be used. If not NULL, the driver should use this value. If NULL, then the line join value is not changed.
lpOutData	If not NULL, a long pointer to a WORD-length value in which the previous (or current if *lpInData* is NULL) line join value should be returned.

Return Value

The return value is TRUE (1) if the escape is successful.

Comments

Miter Join (0): The outer edges of the strokes for the two segments are extended until they meet at an angle, as in a picture frame. (If the segments meet at too sharp of an angle, a Bevel Join is used instead; this is controlled by the miter limit parameter established by SETMITERLIMIT.)

Round Join (1): A circular arc with a diameter equal to the line width is drawn around the point where the segments meet and is filled in, producing a rounded corner. (Stroke actually draws a full circle at this point. If path segments shorter than one-half the line width meet at sharp angles, an unintentional "wrong side" of this circle may appear.)

Bevel Join (2): The meeting path segments are finished with butt end caps (see SETLINECAP); then, the resulting notch beyond the ends of the segments is filled with a triangle.

Join styles are significant only at points at which consecutive segments of a path connect at an angle; segments that meet or intersect fortuitously receive no special treatment. Curved lines are actually rendered as sequences of straight line segments, and the current line join is applied to the "corners" between those segments. However, for typical values of the flatness parameter, the corners are so shallow that the difference between join styles is not visible.

SETMITERLIMIT (Escape # 23)

Syntax **short Control** (*lpDevice*, **SETMITERLIMIT**, *lpInData*, *lpOutData*)

This escape is used to tell the driver how an application wants to clip off miter-type line joins when they become too long. It sets the current miter-limit parameter in the graphics state to *num*, which must be a number greater than or equal to one.

Parameter	**Description**
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpInData	A long pointer to the WORD-length, miter-limit value to be used. If not NULL, the driver should use this value. If NULL, then the miter-limit value is not changed.
lpOutData	If not NULL, a long pointer to a WORD-length value in which the previous (or current if *lpInData* is NULL) miter-limit value should be returned.

Return Value The return value is TRUE (1) if the escape is successful.

Comments The miter limit controls the stroke operator's treatment of corners when miter joins have been specified (see the SETLINEJOIN escape). When path segments connect at a sharp angle, a miter join results in a spike that extends well beyond the connection point. The purpose of the miter limit is to cut off such spikes when they become objectionably long.

At any given corner, the miter length is the distance from the point at which the inner edges of the strokes intersect to the point at which the outside edges of the strokes intersect (i.e., the diagonal length of the miter). This distance increases as the angle between the segments decreases. If the ratio of the miter length to the line width exceeds the miter-limit parameter, the corner is treated with a Bevel Join instead of a Miter Join.

The ratio of miter length to line width is directly related to the angle (*x?*) between the segments by the formula:

miter length / line width = 1 / sin(*x?*/2)

The following are examples of miter-limit values:

- 1.415 cuts off miters (converts them to bevels) at angles less than 90 degrees.
- 2.0 cuts off miters at angles less than 60 degrees.
- 10.0 cuts off miters at angles less than 11 degrees.

The default value of the miter limit is 10. Setting the miter limit to 1 cuts off miters at all angles so that bevels are always produced even when miters are specified.

SET_POLY_MODE (Escape # 4104)

Syntax

short Control (*lpDevice*, SET_POLY_MODE, *lpMode*, *lpOutData*)

This escape enables a device driver to draw shapes (such as Bezier curves) that are not supported directly by GDI. This permits applications that draw complex curves to send the curve description directly to a device without having to simulate the curve as a polygon with a large number of points.

The SET_POLY_MODE escape sets the poly mode for the device driver. The poly mode is a state variable indicating how to interpret calls to the GDI functions Polygon and Polyline.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpMode	A long pointer to a short integer specifying the desired poly mode, the state variable indicating how points in Polygon or Polyline calls should be interpreted. The device driver need not support all the possible modes. It is expected to return zero if it does not support the specified mode. The mode parameter may be one of the following: PM_BEZIER. The points define a sequence of 4-point Bezier spline curves. The first curve passes through the first four points, with the first and fourth points as end points, and the second and third points as control points. Each subsequent curve in the sequence has the end point of the previous curve as its start point, the next two points as control points, and the third as its end point. The last curve in the sequence is permitted to have fewer than four points. If the curve has only one point, it is considered a point. If it has two points, it is a line segment. If it has three points, it is a parabola defined by drawing a Bezier curve with the end points equal to the first and third points and the two control points equal to the second point. PM_POLYLINE. The points define a conventional polygon or polyline. PM_POLYLINESEGMENT. The points specify a list of coordinate pairs. Line segments are drawn connecting each successive pair of points.
lpOutData	Not used and can be set to NULL.

Return Value

This escape returns the previous poly mode. If the return value is zero, the device driver is assumed not to have handled the request.

Comments

An application should issue the SET_POLY_MODE escape before it draws a complex curve. It should then call Polyline or Polygon with the desired control points defining the curve. After drawing the curve, the application should reset the driver to its previous state by reissuing the SET_POLY_MODE escape.

Polyline calls are drawn using the currently selected pen.

Polygon calls are drawn using the currently selected pen and brush. If the start and end points are not equal, a line is drawn from the start point to the end point before filling the polygon (or curve).

Polygon calls using PM_POLYLINESEGMENT mode are treated exactly the same as Polyline calls.

A Bezier curve is defined by four points. The curve is generated by connecting the first and second, second and third, and third and fourth points. The midpoints of these consecutive line segments are then connected. Then the midpoints of the lines connecting the midpoints are connected, and so forth as shown in Figure 11.1.

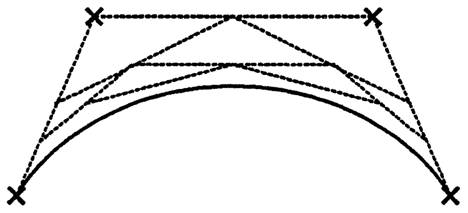


Figure 11.1 An Example of a Bezier Curve

The line segments drawn in this way converge to a curve defined by the following parametric equations, expressed as a function of an independent variable *t*.

$X(t) = (1-t)^3x_1 + 3(1-t)^2tx_2 + 3(1-t)t^2x_3 + t^3x_4$

$Y(t) = (1-t)^3y_1 + 3(1-t)^2ty_2 + 3(1-t)t^2y_3 + t^3y_4$

The points (x_1,y_1), (x_2,y_2), (x_3,y_3), and (x_4,y_4) are the control points defining the curve. The independent variable t varies from 0 to 1.

The supported poly modes are defined as follows:

PM_POLYLINE	1
PM_BEZIER	2
PM_POLYLINESEGMENT	3

Additional primitive types (other than PM_BEZIER and PM_POLYLINESEGMENT) may be added to this escape in the future. Applications are expected to check the return value from this escape to determine whether or not the driver supports the specified poly mode.

SET_SCREEN_ANGLE (Escape # 4105)

Syntax

short Control (*lpDevice*, SET_SCREEN_ANGLE, *lpAngle*, *lpOutData*)

Four-color process separation is the process of separating the colors comprising an image into four component primaries: cyan, magenta, yellow, and black. The image is then reproduced by overprinting each primary.

In traditional four-color process printing, half-tone images for each of the four primaries are photographed against a mask tilted to a particular angle. Tilting the mask in this manner minimizes unwanted moire patterns caused by overprinting two or more colors. The SET_SCREEN_ANGLE escape sets the current screen angle to the desired angle and enables an application to simulate the tilting of a photographic mask in producing a separation for a particular primary.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpAngle	A long pointer to a short integer specifying the desired screen angle in tenths of a degree. The angle is measured counterclockwise.
lpOutData	Not used and can be set to NULL.

Return Value

This escape returns the previous screen angle.

Comments

The default screen angle is defined by the device driver.

SET_SPREAD (Escape # 4106)

Syntax

short Control (*lpDevice*, SET_SPREAD, *lpSpread*, *lpOutData*)

Spot color separation is the process of separating an image into each distinct color used in the image. You then reproduce the image by overprinting each successive color in the image.

When reproducing a spot-separated image, the printing equipment must be calibrated to align each page exactly on each pass. However, differences between passes in such factors as temperature and humidity often cause images to align imperfectly on subsequent passes. For this reason, lines in spot separations are often widened (spread) slightly to make up for problems in registering subsequent passes through the printer.

This process, called *trapping*, is implemented by the SET_SPREAD escape, which sets the spread for the given device. The spread is the amount by which all the non-white primitives are expanded to provide a slight overlap between primitives to compensate for imperfections in the reproduction process.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpSpread	A long pointer to a short integer specifying the amount, in device pixels, by which all the non-white primitives are to be expanded.
lpOutData	Not used and can be set to NULL.

Return Value This escape returns the previous spread.

Comments The default spread is zero.

The current spread applies to all the bordered primitives (whether or not the border is visible) and text.

STARTDOC (Escape # 10)

Syntax **short Control** (*lpDevice*, **STARTDOC**, *lpDocName*, *lpOutData*)

This escape informs the device driver that a new print job is starting and that all subsequent NEWFRAME calls should be spooled under the same job, until an ENDDOC call occurs.

This ensures that documents longer than one page will not be interspersed with other jobs.

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpDocName	A 32-bit address to a NULL-terminated string specifying the name of the document. The document name is displayed in the Print Manager window.
lpOutData	Not used and can be set to NULL.

Return Value The return value is -1 if an error occurs, such as insufficient memory or an invalid port specification. Otherwise, it is positive.

Comments The correct sequence of events in a printing operation is as follows:

1. Create printer device context.

2. Set the abort function to keep out-of-disk-space errors from aborting a printing operation. An abort procedure that handles these errors must be set using the SETABORTPROC escape.
3. Begin the printing operation with STARTDOC.
4. End each new page with NEWFRAME or begin each new band with NEXTBAND.
5. End the printing operation with ENDDOC.

TRANSFORM_CTM (Escape # 4107)

Syntax

short Control (*lpDevice*, TRANSFORM_CTM, *lpMatrix*, *lpOutData*)

This escape modifies the current transformation matrix (CTM). The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, you can combine these operations in any order to produce the desired mapping for a particular picture.

The new current transformation matrix will contain the product of the matrix referenced by the parameter *lpMatrix* and the previous current transformation matrix (CTM = M * CTM).

Parameter	Description
lpDevice	A long pointer to a data structure of type PDEVICE, which is the destination device bitmap.
lpMatrix	A long pointer to a three-by-three array of 32-bit integer values specifying the new transformation matrix. Entries in the matrix are scaled to represent fixed-point real numbers. Each matrix entry is scaled by 65536. Thus, the high-order WORD of the entry contains the whole integer portion, and the low-order WORD contains the fractional portion.
lpOutData	Not used and can be set to NULL.

Return Value

This escape returns TRUE if the escape is successful and FALSE if it is unsuccessful.

Comments

Applications should not make any assumptions about the initial value of the current transformation matrix.

Drivers supporting this escape must also implement the SAVE_CTM and RESTORE_CTM escapes.

The matrix specification used for this escape is based on the OS/2 GPI (Graphics Programming Interface) model, which is an integer coordinate system similar to the one used by GDI.

Japanese Escapes

The following is a list of the escapes reserved by Microsoft Japan and a short description of each. Please contact Microsoft Japan to reserve other escape functions.

Japanese Escape (Number)	**Description/Parameters**
GAIJIFONTSIZE (Escape # 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 (Escape # 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 (Escape # 2578)	The driver returns the Gaiji font pattern in a monochrome bitmap format. The low WORD of *lpInData* contains (*not* points to) the handle to the bitmap. *lpOutData* points to a buffer that contains the Shift JIS (Japanese Industrial Standard) code of the Gaiji font pattern to be retrieved. If successful, the driver should return AX = TRUE.
GAIJISYSTEMSETFONT (Escape # 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 that contains the Shift JIS code of the Gaiji font. If successful, the driver should return AX = TRUE.
GAIJITTOCODE (Escape # 2580)	The driver returns the Shift JIS code from the index to the Gaiji area.The high WORD of *lpInData* contains (*not* points to) an index to the system reserve area. *lpOutData* points to a buffer that will contain the Shift JIS code of the Gaiji upon return. If successful, the driver should return AX = TRUE.

Japanese Escape (Number)	Description/Parameters
GAIJILOCALOPEN (Escape # 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 (Escape # 2582)	The driver releases ownership of the Gaiji area. The high WORD of *lpInData* contains (*not* points to) the handle that was returned to the caller by GAIJILO-CALOPEN. *lpOutData* is undefined and can be ignored. If successful, the driver should return AX = TRUE.
GAIJILOCALSETFONT (Escape # 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 that will contain the Shift JIS code of the Gaiji upon return.
GAIJILOCALSAVE (Escape # 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 (Escape # 2585)	Restores the Gaiji area with the global object saved in GAIJILOCALSAVE. Upon entry, the low WORD of *lpInData* contains the global memory handle that 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.

Japanese Escape (Number)	Description/Parameters
TTYMODE (Escape # 2560)	If called by an application, this call signals Kanji Windows printer drivers to disregard the passed FONTINFO data and print using the printer's default hardware font. You should not attempt 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. ***NOTE*** In Windows 1.x, this was escape number 15. Since this conflicts with the standard Microsoft escape number 15 (MFCOMMENT), it was relocated to the current escape number. Kanji's GDI, for Windows 2.x and later versions, checks for escape number 15 whenever called by an application and remaps it to the correct number.

Chapter 12 Data Structures and File Formats

The virtual machine (VM) that supports Microsoft Windows is comprised of several sets of predefined functions. This chapter describes the data structures and file formats used by those functions. The data structures are presented here in three sections:

1. Data structures returned by the information calls in each device driver used by Windows.
2. Data structures used as parameters to calls on device drivers.
3. Data structures that are device dependent; these structures contain information about physical devices.

The data structure descriptions in this chapter are intended as a guide. The following words, when used in the C-structure declarations discussed in this chapter, have the meanings shown here:

- *char* stands for an unsigned 8-bit integer.
- *short* stands for a signed 16-bit integer.
- *long* stands for a signed 32-bit integer or a long pointer stored as a 16-bit segment address and a 16-bit offset within that segment.

Notice also that rectangles and points within a bitmap are described using a coordinate system with its origin (X=0, Y=0) at the top-left corner of the rectangle or point. The X coordinate increases to the right, and the Y coordinate increases downward.

12.1 Information Data Structures

Device drivers use information data structures to tell others about themselves. This section includes detailed information on the GDIINFO data structure and its fields, with a separate subsection dedicated to the **dpText** field's precision levels. The CURSORINFO data structure is included both in this section and the Mouse driver chapter, which also includes the MOUSEINFO structure. KBINFO can be found in Chapter 8, "Keyboard Drivers," and the Device Control Block (DCB) structure is in Chapter 9, "Miscellaneous Drivers".

12.1.1 The GDIINFO Structure

The GDIINFO data structure describes the characteristics of the attached graphics device in sufficient detail so that GDI can allocate space for the required data structures. This data structure is used to describe the system screen to Microsoft Windows. Additional descriptions of GDIINFO are also included in Chapter 2, "Display Drivers," and in Chapter 5, "Printer Drivers."

The currently defined fields are as follows:

```
typedef  struct  {
        short  int                  dpVersion;
        short  int                  dpTechnology;
        short  int                  dpHorzSize;
        short  int                  dpVertSize;
        short  int                  dpHorzRes;
        short  int                  dpVertRes;
        short  int                  dpBitsPixel;
        short  int                  dpPlanes;
        short  int                  dpNumBrushes;
        short  int                  dpNumPens;
        short  int                  futureuse;
        short  int                  dpNumFonts;
        short  int                  dpNumColors;
        unsigned  short  int        dpDEVICEsize
        unsigned  short  int        dpCurves;
        unsigned  short  int        dpLines;
        unsigned  short  int        dpPolygonals
        unsigned  short  int        dpText;
        unsigned  short  int        dpClip;
        unsigned  short  int        dpRaster;
        short  int                  dpAspectX;
        short  int                  dpAspectY;
        short  int                  dpAspectXY;
        short  int                  dpStyleLen;
        POINT                       dpMLoWin;
        POINT                       dpMLoVpt;
        POINT                       dpMHiWin;
        POINT                       dpMHiVpt;
        POINT                       dpELoWin;
        POINT                       dpELoVpt;
        POINT                       dpEHiWin;
        POINT                       dpEHiVpt;
        POINT                       dpTwpWin;
        POINT                       dpTwpVpt;
        short  int                  dpLogPixelsX;
        short  int                  dpLogPixelsY;
        short  int                  dpDCManage;
        short  int                  futureuse3;
        short  int                  futureuse4;
        short  int                  futureuse5;
        short  int                  futureuse6;
short  int                futureuse7;
short  int                dpPalColors;
short  int                dpPalReserved;
short  int                dpPalResolut;
} GDIINFO;
```

12.1.2 The GDIINFO Field Descriptions

When the term *styled lines* appears here, it actually means patterned polylines.

The fields in the GDIINFO data structure have the following meanings:

Field	Description
dpVersion	Specifies the version number. This must be 300H.
dpTechnology	Specifies the device technology from the following list: Vector plotter(0) Raster display(1) Raster printer(2) Raster camera(3) Character-stream, PLP(4) Metafile, VDM(5) Display file(6)
dpHorzSize	The width of the physical display in millimeters.
dpVertSize	The height of the physical display in millimeters.
dpHorzRes	The width of the display in pixels.
dpVertRes	The height of the display in raster lines.
dpBitsPixel	Specifies the number of adjacent bits on each plane involved in making up a pixel. For a 256-color, 1-plane, high resolution display, this would hold the value 8, while the **dpPlanes** field would hold the value 1.
dpPlanes	Specifies the number of planes in frame-buffer memory. For a typical frame buffer with red, green, and blue bit planes (such as on a 3-plane EGA), this field would be 3.
dpNumBrushes	Specifies the number of device-specific brushes supported by this device.
dpNumPens	Specifies the number of device-specific pens supported by this device.
dpNumFonts	Specifies the number of device-specific fonts supported by this device.

Field	Description
dpNumColors	Specifies the number of entries in the color table for this device or the numbers of reserved colors for palette-capable devices.
dpDEVICEsize	Specifies the size of the data structure of type PDEVICE that must be allocated for this device. It must be at least two bytes.
dpCurves	Specifies to GDI whether or not the device driver can perform circles, pie wedges, chord arcs, and ellipses; whether or not the interior of those figures that can be handled can be brushed in; and whether the borders of those figures that can be handled can be drawn with wide lines, styled lines, or lines that are both wide and styled. The field required is created by setting the appropriate bits, as follows: bit 0set means can do circles bit 1 set means can do pie wedges bit 2 set means can do chord arcs bit 3 set means can do ellipses bit 4 set means can do wide lines bit 5 set means can do styled lines bit 6 set means can do lines that are wide and styled bit 7 set means can do interiors The high byte must be 0.
dpLines	Specifies whether or not the support module can perform polylines and lines; whether or not the interior of those figures that can be handled can be brushed in; and whether the borders of those figures that can be handled can be drawn with wide lines, styled lines, or lines that are both wide and styled. The field required is created by setting the appropriate bits, as follows: bit 0 set means cannot do lines bit 1 set means can do polylines bit 2 reserved bit 3 reserved bit 4 set means can do wide lines bit 5 set means can do styled lines bit 6 set means can do lines that are wide and styled bit 7 set means can do interiors The high byte must be 0.

Field	Description
dpPolygonals	Specifies whether or not the device driver can perform polygons, rectangles, and scanlines; whether or not the interior of those figures that can be handled can be brushed in; and whether the borders of those figures that can be handled can be drawn with wide lines, styled lines, or lines that are both wide and styled. The field required is created by setting the appropriate bits, as follows:

bit 0	set means can do alternate fill polygons
bit 1	set means can do rectangles
bit 2	set means can do winding number fill polygons
bit 3	set means can do scanlines
bit 4	set means can do wide borders
bit 5	set means can do styled borders
bit 6	set means can do borders that are wide and styled
bit 7	set means can do interiors

The high byte must be 0.

dpText Specifies what level of text support is provided by the device driver. The levels of text support are listed below in terms of ability (precision levels):

OutputPrecision	(STRING, CHARACTER, STROKE)
ClipPrecision	(CHARACTER, STROKE)
CharRotAbility	(NONE, 90, ANY)
ScaleFreedom	(X_YIDENTICAL, X_YINDEPENDENT)
ScaleAbility	(NONE, DOUBLE, INTEGER, CONTINUOUS)
EmboldenAbility	(NONE, DOUBLE)
ItalicizeAbility	(UNABLE, ABLE)
UnderlineAbility	(UNABLE, ABLE)
StrikeOutAbility	(UNABLE, ABLE)
RasterFontAble	(UNABLE, ABLE)
VectorFontAble	(UNABLE, ABLE)

Field	Description
	Each precision level (STRING, CHARACTER, NONE, 90, etc.) has a corresponding bit in **dpText** that is set if the device is capable of that precision level. Each precision level within an ability is a superset of the precision levels below it. For example, in ScaleAbility, DOUBLE implies NONE, INTEGER implies DOUBLE and NONE, and CONTINUOUS implies all three. Since it is required that the lowest precision level of each ability be supported, no bit is provided in **dpText** for the lowest level of each ability. Therefore, if INTEGER is set for ScaleAbility, then DOUBLE must also be set, and NONE is implied.
	If a device claims to have an ability, it must have it for all fonts, whether realized by the device or provided by GDI.
	The bits of **dpText** are defined as follows:

bit 0	set means can do OutputPrecision CHARACTER
bit 1	set means can do OutputPrecision STROKE
bit 2	set means can do ClipPrecision STROKE
bit 3	set means can do CharRotAbility 90
bit 4	set means can do CharRotAbility ANY
bit 5	set means can do ScaleFreedom X_YINDEPENDENT
bit 6	set means can do ScaleAbility DOUBLE
bit 7	set means can do ScaleAbility INTEGER
bit 8	set means can do ScaleAbility CONTINUOUS
bit 9	set means can do EmboldenAbility DOUBLE
bit 10	set means can do ItalicizeAbility ABLE
bit 11	set means can do UnderlineAbility ABLE
bit 12	set means can do StrikeOutAbility ABLE
bit 13	set means can do RasterFontAble ABLE
bit 14	set means can do VectorFontAble ABLE
bit 15	reserved. Must be returned zero.

Field	Description
	All the available abilities are described in Section 12.1.3, "GDIINFO_dpText Field Precision Levels," following the descriptions of the remaining GDIINFO fields.
dpClip	Indicates that clipping is available to the device. If the field is 1, the device can clip to a rectangle in the **Output** function. If the field is 0, it cannot clip.
dpRaster	Specifies raster abilities.

Field	Description
	bit 0 set means device has BitBlt capabilities
	bit 1 set means device requires banding support
	bit 2 set means device requires scaling support
	bit 3 set means device supports bitmaps larger than 64K
	bit 4 set means device supports the new (Windows 2.0) output functions (**ExtTextOut**, **FastBorder**, and **GetCharWidth**)
	bit 5 set means device context has state block
	bit 6 set means device can save bitmaps locally in "shadow" memory
	bit 7 set means can do **Get** and **Set** DIBs and RLE to and from memory in all the existing DIB resolutions (1, 4, 8, and 24 bits-per-pixel). However, if the flag is not set, GDI will simulate in monochrome.
	bit 8 set means can do color palette management
	bit 9 set means can do **SetDIBitsToDevice**
	bit 10 set means can do >64K fonts (set only in protected mode). However, if the flag is not set, all the fonts will be version 2.0.
	bit 11 set means can do **StretchBlt**
	bit 12 set means can do **FloodFill**
dpAspectX, dpAspectY, dpAspectXY	Specify the relative width, height, and diagonal width of a device pixel and correspond directly to the device's aspect ratio. For the IBM PC CGA (640 x 200 pixels) screen, these fields are 5, 12, and 13, respectively (that is, every pixel is a 5 by 12 rectangle). This corresponds to an aspect ratio of 5 vertical pixels to every 12 horizontal. For devices whose pixels do not have integral diagonal widths, the field values can be multiplied by a convenient factor to preserve information. For example, pixels on a device with a 1 to 1 aspect ratio have a diagonal width of 1.414. For good results, the aspect fields should be set to 100, 100, and 141, respectively. For numerical stability, the field values should be kept under 1000.
dpStyleLen	Specifies the minimum length of a dot generated by a styled pen. The length is relative to the width of a device pixel and should be given in the same units as **dpAspectX**. For example, if **dpAspectX** is 5 and the minimum length required is 3 pixels, **dpStyleLen** should be 15.
dpMLoWin	A constant specifying the width and height of the metric (low resolution) window. Width is *HorzSize**10; height is *VertSize**10.

Field	Description
dpMLoVpt	A constant specifying the horizontal and vertical resolutions of the metric (low resolution) viewport. Horizontal is *HorzRes*; vertical is -*VertRes*.
dpMHiWin	A constant specifying the width and height of the metric (high resolution) window. Width is *HorzSize**100; height is *VertSize**100.
dpMHiVpt	A constant specifying the horizontal and vertical resolutions of the metric (high resolution) viewport. Horizontal is *HorzRes*; vertical is -*VertRes*.
dpELoWin	A constant specifying the width and height of the English (low resolution) window. Width is *HorzSize**1000; height is *VertSize**1000.
dpELoVpt	A constant specifying the horizontal and vertical resolutions of the English (low resolution) viewport. Horizontal is *HorzRes**254; vertical is -*VertRes**254.
dpEHiWin	A constant specifying the width and height of the English (high resolution) window. Width is *HorzSize**10,000; height is *VertSize**10,000.
dpEHiVpt	A constant specifying the horizontal and vertical resolutions of the English (high resolution) viewport. Horizontal is *HorzRes**254; vertical is -*VertRes**254.
dpTwpWin	A constant specifying the width and height of the twip window. There are 20 twips per 1 printer's point and 72 printer's points per inch. Width is *HorzSize**14400; height is *VertSize**14400.
dpTwpVpt	A constant specifying the horizontal and vertical resolutions of the twip viewport. Horizontal is *HorzRes**254; vertical is -*VertRes**254.
dpLogPixelsX	This 2-byte field specifies the number of pixels per logical inch along a horizontal line on the display surface. This is used to match fonts.
dpLogPixelsY	This 2-byte field specifies the number of pixels per logical inch along a vertical line on the display surface. This is used to match fonts.
dpDCManage	This 2-byte field contains the following bits:

Field	Description
	DC_SPDevice (001) DC_1PDevice (010) DC_IgnoreDFNP (100)

Value	Description
000	Action as previously existed. Multiple DCs are allowed to exist for every device/filename pair (DFNP), and they will share the same PDevice. Multiple DFNPs can exist, each having its own PDevice.
001	Each attempt to create a DC with the same DFNP will cause a new PDevice to be allocated and initialized. A new DFNP will cause a new PDevice to be allocated and initialized.
010	There will only be one DC per DFNP. An attempt to create a second DC with the same DFNP will return an error. A new DFNP will cause a new PDevice to be allocated and initialized.
011	Invalid.
100	Multiple DCs are allowed to exist, and they will share the same PDevice, regardless of the DFNP.
101	Invalid.
110	Only one DC can exist. An attempt to create a second DC will return an error.
111	Invalid.

Field	Description
dpPalColors	Specifies the total number of simultaneous colors available in Windows 3.0 for palette-capable devices. Other devices ignore this field but must put something here.
dpPalReserved	Specifies the even number of reserved system colors available in Windows 3.0 for palette-capable devices. Other devices ignore this field but must put something here.

Field	Description
dpPalResolut	Specifies the palette resolution, which equals the number of bits going into video DACS. Nonpalette-capable devices ignore this field but must put something here. See Chapter 3, "Display Drivers: New Features," for more information on color palette management and these three new fields for Windows 3.0.

NOTE The window/viewport pair fields are the numerator and denominator of the scale fraction used to correct for the device aspect ratio and to set to a fixed unit of measurement, either metric or English. These numbers should be integers in the range of -32768 to 32767. When calculating these constants, out-of-range values can be divided by some number to bring them back into range as long as the corresponding window or viewport constant is divided by the same number. See Chapter 2, "Display Drivers," for an example of these calculations.

The **dpRaster** field is also used to indicate a scaling device. If the RC_SCALING bit (bit 2) is set, the device does graphics scaling. Certain devices perform graphics at one resolution and text at another. Some applications require that character cells be an integral number of pixels. If a device reported that its graphics resolution was 75 dpi but its text resolution was 300 dpi, then its character cells would not be an integral number of pixels (since they were digitized at 300 dpi). To get around this problem, GDI uses scaling devices. The device driver registers itself as a 300 dpi device and all the graphics at 300 dpi are scaled to 75 dpi. Any device that scales must have the RC_SCALING bit set. Scaling always reduces the resolution, never increases it. GDI calls the control procedure with GETSCALINGFACTOR before graphics is done to a device. The scaling factor is a SHIFT count that is a power of two. Therefore, a scale factor of 2 means reduce by 4, and a scale factor of 1 means reduce by 2.

12.1.3 GDIINFO — dpText Field Precision Levels

The following is a detailed description of each of the text-support precision levels for the GDIINFO **dpText** field.

OutputPrecision (STRING, CHARACTER, STROKE)

OutputPrecision specifies which font attributes the output function may ignore. The device is not required to ignore any given attribute; it is merely allowed to do so if that will facilitate output. OutputPrecision has no effect on emboldening, italicizing, underlining, or strikeout. If a device registers these abilities, it must perform them when requested.

Whenever either the character orientation or the difference between the character orientation and the escapement angle is a multiple of 90 degrees, the intercharacter and interword spacing will be the standard intercharacter spacing used for bounding boxes plus the **CharacterExtra** and **BreakExtra** spacing. (Refer to the DRAWMODE data structure for a description of intercharacter and interword spacing.)

The standard intercharacter spacing at a given escapement angle and character orientation is defined as the minimum spacing along the escapement vector, such that the character origins are on the escapement vector, and the character bounding boxes touch. Variable pitch fonts are achieved by using variable width bounding boxes. This model applies at all attribute values. When the sides of the bounding boxes touch, extra space is added in X and, when the tops touch, it is added in Y.

In all other escapement and orientation cases, the standard intercharacter spacing is device dependent. The preferred implementation is as for the 90-degree cases. In all cases, it is required that all character origins lie on the escapement vector. The precision levels for output are described in detail as follows:

Level	Description
STRING	This level of precision is used where simplicity and efficiency are more important than geometric precision of the text. The goal of STRING precision is to make the most use of hardware character generation as possible. The effects of the text attributes Height, Width, Escapement, and Orientation on appearance are device dependent. Height and Width are used to determine a "best-fit" character size. For STRING and CHARACTER precision, the largest font that does not exceed the requested size will be used. If no such font exists, the smallest available font will be used. Intercharacter and interword spacing must adhere to their current settings. The device has the option of ignoring escapement and character orientation. The starting point of the STRING is subject to any transforms in effect.
CHARACTER	This level of precision is used when it is important that the string occupy a given region, such as when labeling the axes of charts and graphs. CHARACTER precision makes use of the hardware character generation on a character-by-character basis. The effects of the text attributes Height, Width, and Orientation on the appearance are device dependent. Size is determined as for STRING precision. CHARACTER precision must adhere to escapement. Only character orientation may be ignored. The starting point of the string is subject to any transforms in effect.
STROKE	This level of precision treats the characters as if they were generated by being stroked out as vectors. STROKE precision must adhere to all current attributes, including size. The starting point of the string is subject to any transforms in effect.

ClipPrecision (CHARACTER, STROKE)

ClipPrecision specifies how accurately **StrBlt** can clip. At CHARACTER precision, a character in the string is entirely invisible if, and only if, any portion of the character is outside the clip region. With STROKE precision, only those portions of each character that are outside the clip region are invisible. The rest of the character is visible.

CharRotAbility (NONE, 90, ANY)

CharRotAbility refers to the ability to rotate individual character cells. NONE implies that characters cannot be rotated. 90 means that characters can only be rotated in 90 degree increments. ANY implies arbitrary rotation angles.

It is assumed that arbitrary escapement angles can be achieved, if by no other means than, by placing each character as a separate entity. Many devices are able to do arbitrary character rotation only if the character orientation matches the escapement angle. For such devices, it is assumed that the driver will place each character individually at the proper orientation and escapement, when escapement and character orientation do not match.

ScaleFreedom (X_YIDENTICAL, X_YINDEPENDENT)

ScaleFreedom specifies how the characters in a font may be scaled. X_YIDENTICAL means that the characters must be scaled by the same amount in each direction. X_YINDEPENDENT implies that the characters may be scaled independently in each direction.

ScaleAbility (NONE, DOUBLE, INTEGER, CONTINUOUS)

ScaleAbility specifies by what amount the characters can be scaled. NONE implies no scaling. DOUBLE means the characters can be doubled. INTEGER allows any integer multiple. CONTINUOUS gives exact scaling. Whenever a device cannot match a requested size exactly, because of X_YIDENTICAL or noncontinuous scaling, it is required that the device use the largest size available that will not exceed the requested size in either direction.

EmboldenAbility (NONE, DOUBLE)

EmboldenAbility indicates whether or not **StrBlt** can alter the weight of a font. NONE implies nothing can be done. DOUBLE can double the weight, usually by shifting one pixel and overstriking. However, this method will *not* double the weight of bigger fonts. This ability is not affected by OutputPrecision.

ItalicizeAbility (UNABLE, ABLE)

ItalicizeAbility is set ABLE if **StrBlt** can take a nonitalic font and skew it to make it italic. This ability is not affected by OutputPrecision.

UnderlineAbility (UNABLE, ABLE)

UnderlineAbility is set ABLE if **StrBlt** can underline a font. This ability is not affected by OutputPrecision.

StrikeOutAbility (UNABLE, ABLE)

StrikeOutAbility is set ABLE if **StrBlt** can strike out a font by drawing a line through it. This ability is not affected by OutputPrecision.

RasterFontAble (UNABLE, ABLE)

RasterFontAble indicates whether or not the device is capable of using raster format fonts.

VectorFontAble (UNABLE, ABLE)

VectorFontAble indicates whether or not the device is capable of using vector format fonts.

NOTE If the device driver returns the abilities listed below, it need never implement or adhere to any of the font attributes. The only parameters affecting output will be the font face (physical font), fixed or variable pitch, and text justification as specified in the DRAWMODE data structure.

OutputPrecision:	STRING
ClipPrecision:	CHARACTER
CharRotAbility:	NONE
ScaleFreedom:	X_YIDENTICAL
ScaleAbility:	NONE
EmboldenAbility:	NONE
ItalicizeAbility:	UNABLE
UnderlineAbility:	UNABLE
StrikeOutAbility:	UNABLE
RasterFontAble:	UNABLE or ABLE
VectorFontAble:	NOT RasterFontAble

The ClipPrecision ability must be implemented with STROKE precision on the console device for Microsoft Windows to operate properly.

12.1.4 CURSORINFO — Cursor Information Data Structure

This data structure contains information about the system display's cursor module.

```
typedef struct {
      short   dpXRate;
```

```
        short   dpYRate;
        } CURSORINFO;
```

The fields in this data structure have the following meanings:

Field	Description
dpXRate	The horizontal mickey-to-pixel ratio for this display. For the IBM PC and the Microsoft Mouse, this is 1.
dpYRate	The vertical mickey-to-pixel ratio for this display. For the IBM PC and the Microsoft Mouse, this is 2.

12.2 Parameter Data Structures

The following are the data structures that are used as parameters to the functions described in this document.

12.2.1 POINT — Point Data Structure

This data structure describes the format of a point as used by other data structures.

```
typedef struct {
        short  x;
        short  y;
        } POINT;
```

The fields in this data structure have the following meanings:

Field	Description
x	The X-coordinate value of a point.
y	The Y-coordinate value of a point.

12.2.2 RECT — Rectangle Data Structure

A rectangle is characterized by two points.

```
typedef struct {
        short   left,   top;
        short   right,  bottom;
        } RECT;
```

The fields in this data structure have the following meanings:

Field	Description
left, top	The coordinates that specify the upper-left corner of the rectangle (points inclusive).
right, bottom	The coordinates that specify the lower-right corner of the rectangle (points exclusive).

NOTE The "**right, bottom**" coordinates are actually one greater than the actual size the rectangle would appear to require. Therefore, if passed a "**right, bottom**" coordinate pair of (640,480), the device driver should use a rectangle with its lower right corner at (639,479).

12.2.3 RGB — Logical Color Specification

A logical color specifies the color desired by an application.

```
typedef long RGB;
```

The long integer is divided into four 1-byte fields, three of which specify the intensity of the primary colors. The intensities of the color values are on a scale of 0-255. The values are packed in the low three bytes of the long integer in the following format:

r + 256*g + 64K*b.

That is, the lowest-order byte contains Red information, the next byte contains Green information, and the third byte contains Blue information. For palette-capable devices, if the high byte is 00H, then this is an RGB color. However, if the high byte is 0FFH, then the low WORD is color index, not an RGB.

Each byte represents an intensity level for the specified color (i.e., red, green, and blue); 0 is the minimum intensity, 255 the maximum. When the colors are combined, they form a new color. For example, when the colors are at minimum intensity (0,0,0), the result is black. When the colors are at maximum intensity (255, 255, 255), the result is white. Gray is half intensity in all colors (127,127,127); solid green is (0, 255, 0), and so on.

If a display device is not capable of all the possible RGB color combinations, the OEM must decide which colors to display for the given RGB color values. For example, in a black and white display with only one bit per pixel, the OEM must choose a cutoff intensity at which all the RGB values above the intensity are white and all below are black. One method used to compute the cutoff intensity is to add the individual color intensities and divide by two:

```
(R+G+B)/2
```

In this case, the cutoff intensity is 382, or (255+255+255)/2.

12.2.4 DRAWMODE — Drawing Mode Specification

A drawing mode includes the information required to draw lines on a display.

```
typedef struct {
        short   Rop2;
        short   BackgroundMode;
        pColor  BackgroundColor;
        pColor  TextColor;
        short   TBreakExtra;
        short   BreakExtra;
        short   BreakErr;
        short   BreakRem;
        short   BreakCount;
        short   CharacterExtra;
        pColor  LogicalBGColor;
        pColor  LogicalTextColor;
        } DRAWMODE;
```

The fields within the DRAWMODE structure have the following meanings:

Field	Description
Rop2	A short integer value between 1 and 16 specifying the Boolean combining function (of source and destination colors) to be used. All the possible Boolean functions of two variables, using the binary operations *AND*, *OR*, and *XOR*, and the unary operation *NOT*, are defined using the binary raster operations table below:

Binary Raster Op Table	Rop Code	Function of Dest and Pen (or Pattern)
R2_BLACK	1	/* 0 */
R2_NOTMERGEPEN	2	/* DPon */
R2_MASKNOTPEN	3	/* DPna */
R2_NOTCOPYPEN	4	/* Pn */
R2_MASKPENNOT	5	/* PDna */
R2_NOT	6	/* Dn */
R2_XORPEN	7	/* DPx */
R2_NOTMASKPEN	8	/* DPan */
R2_MASKPEN	9	/* DPa */

R2_NOTXORPEN	10	/* DPxn */
R2_NOP	11	/* D */
R2_MERGENOTPEN	12	/* DPno */
R2_COPYPEN	13	/* P */
R2_MERGEPENNOT	14	/* PDno */
R2_MERGEPEN	15	/* DPo */
R2_WHITE	16	/* 1 */

BackgroundMode	A short integer specifying whether parts of lines not being drawn in foreground color should be drawn in the background color (opaque background) or left transparent (transparent background). This mode also applies when a brush is used for interiors, scanlines, and text.
BackgroundColor	A physical color specifying the background color to be used. See Section 12.3.3, "PCOLOR–Physical Color Definition," for more information about the PCOLOR data structure.
TextColor	A physical color specifying the text color to be used. See Section 12.3.3, "PCOLOR–Physical Color Definition," for more information about the PCOLOR data structure.

The remaining fields in DRAWMODE specify text justification as follows:

Field	Description
TBreakExtra	A short integer specifying the total number of pixels that must be shared by, and inserted into, all the character breaks in the string(s) output. Also known as the Proportional String flag.
BreakExtra	A short integer specifying the number of pixels to insert at every character break: div (**TBreakExtra, BreakCount**).
BreakErr	A short integer that maintains a running error term to be used by **StrBlt** to track the number of **BreakRem** pixels that have been consumed. This error term allows an application to do justification across a line composed of several different output strings using different fonts.
BreakRem	A short integer specifying the remaining pixels to be scattered among the character breaks: mod (**TBreakExtra, BreakCount**).

Field	Description
BreakCount	A short integer specifying the number of character breaks into which the extra pixels specified by TBreakExtra must be inserted.
CharacterExtra	A short integer specifying in pixels (or device units) the amount of extra space to put between characters output by **StrBlt**.
LogicalBGColor	Specifies the logical background color. See Section 12.3.3, "PCOLOR–Physical Color Definition," for more information about the PCOLOR data structure.
LogicalTextColor	Specifies the logical text color. See Section 12.3.3, "PCOLOR–Physical Color Definition," for more information about the PCOLOR data structure.

NOTE If no justification is required, **TBreakExtra** will be set to zero. To enable justification, an application must set **TBreakExtra** and **BreakCount** to the desired values. The other justification fields are evaluated using these values and **BreakErr** is set to **BreakCount**/2+1.

It is expected that **StrBlt** will be implemented as described below, but any implementation that spreads the excess pixels across the character breaks satisfies the requirements of text justification.

```
width = width of char
if TBreakExtra <> 0 and char = BreakChar then
        width = width + BreakExtra
        BreakErr = BreakErr - BreakRem
        if BreakErr <= 0 then
                width = width + 1
                BreakErr = BreakErr + BreakCount
        endif
endif
width = width + CharacterExtra move over by width
```

12.2.5 RASTEROP — Raster Operations

A raster operation specifies how to combine the source, pattern, and destination during a **BitBlt**.

GDI RASTEROP includes the complete set of Boolean functions, using the binary operators *AND*, *OR*, and *XOR*, and the unary operator *NOT*, on three variables. Those combining functions are listed in Chapter 14, "Raster Operation Codes and Definitions," where the actual name and reverse Polish notation for each value are given. Notice that the actual values used to denote the 256 functions are 32-bit unsigned integers. Therefore, the table of values is sparse, and one must be careful when specifying the RASTEROP.

Chapter 14, "Raster Operation Codes and Definitions," also includes a description of the process used to generate the 32-bit numbers.

Some of the more commonly used raster operation codes are listed below:

```
#define SRCCOPY        0x00CC0020 /*dest=source */
#define SRCPAINT       0x00EE0086 /*dest=source OR dest */
#define SRCAND         0x008800C6 /*dest=source AND dest */
#define SRCINVERT      0x00660046 /*dest=source XOR dest */
#define SRCERASE       0x00440328 /*dest=source AND (NOT dest ) */
#define NOTSRCCOPY     0x00330008 /*dest=(NOT source) */
#define NOTSRCERASE    0x001100A6 /*dest=(NOT source) AND (NOT dest) */
#define MERGECOPY      0x00C000CA /*dest=(source AND pattern) */
#define MERGEPAINT     0x00BB0226 /*dest=(source AND pattern) OR dest */
#define PATCOPY        0x00F00021 /*dest=pattern */
#define PATPAINT       0x00FB0A09 /*DPSnoo */
#define PATINVERT      0x005A0049 /*dest=pattern XOR dest */
#define DSTINVERT      0x00550009 /*dest=(NOT dest) */
#define BLACKNESS      0x00000042 /*dest=BLACK */
#define WHITENESS      0x00FF0062 /*dest=WHITE */
```

12.2.6 CURSORSHAPE — Cursor Data Structure

The CURSORSHAPE data structure is used by Microsoft Windows to generate a cursor on a physical display at the current cursor position. A cursor contains a hotspot within the cursor shape that is aligned with the cursor position. It also contains two bitmaps of equal size that are used to determine the appearance of the cursor as a function of the display contents under the cursor. The first bitmap is ANDed with the contents of the display and the second bitmap is XORed with the result to generate the final appearance of the cursor as opaque white, opaque black, transparent, or invert.

```
typedef struct {
        short csHotX;
        short csHotY;
        short csWidth;
        short csHeight;
        short csWidthBytes;
        short csColor;
        char  csBits;
        } CURSORSHAPE;
```

The fields within the CURSORSHAPE structure have the following meanings:

Field	Description
csHotX, csHotY	A point within the cursor shape that should be aligned with the cursor position when displaying the cursor. Negative coordinates are allowed, so that the hotspot can lie outside the cursor shape.
csWidth	Width of the cursor shape in pixels.

Field	Description
csHeight	Height of the cursor shape in raster lines.
csWidthBytes	Width of the cursor shape in bytes. This is currently 4.
csColor	Format of color information in following pixel array. This field should contain zero.
csBits	An array of bits containing the two masks that define the cursor shape. The first **csHeight*csWidthBytes** bytes define the AND mask. The second **csHeight*csWidthBytes** bytes define the XOR mask.

12.2.7 LOGPEN - Logical Pen Attribute Information

The logical pen attribute structure is used by the device driver while drawing lines or perimeters.

(Chris G, please correct the structure and definitions.)

```
typedef struct {
        long  lopnStyle;
        POINT lopnWidth;
        long  lreserved;
        long  lopnColor;
        } LOGPEN;
```

The fields within the LOGPEN structure have the following meanings:

Field	Description
lopnStyle	A long integer value specifying the type of interruptions to be used in generating the pen. Predefined pen styles (with indexes 0-5, respectively) include: LS_SOLID LS_DASHED LS_DOTTED LS_DOTDASHED LS_DASHDOTDOT LS_NOLINE Any other style should be ignored.
lopnWidth	A data structure of POINT type whose fields specify the width and height of dots created by the pen (in device units). A zero-width pen is drawn with the system's smallest width. Negative-width pens have no width and are NULL pens.

Field	Description
lreserved	A long integer reserved for future use.
lopnColor	A long integer specifying the RGB color with which the pen is to be drawn. However, for palette-capable devices, if the high byte is 0FFH, then the low WORD is a color index. Use that index for the physical color in the corresponding PPEN structure. If the high byte is 00H, then you have an RGB.

12.2.8 LOGBRUSH — Logical Brush Attribute Information

The logical brush attribute structure is used while filling interiors.

```
typedef struct {
        short  lbStyle;
        long   lbColor;
        short  lbHatch;
        long   lbBkColor;
        } LOGBRUSH;
```

The fields within the LOGBRUSH structure have the following meanings:

Field	Description
lbStyle	Selects the type of brush. Predefined brush types (with indexes 0-3, respectively) include the following: BS_SOLID BS_HOLLOW BS_HATCHED BS_PATTERN The information in the remaining fields varies depending on the brush type selected.
lbColor	If the brush style is BS_HOLLOW, the **lbColor** field is not used. If it is BS_PATTERN, the field is a long pointer to the physical bitmap (type BITMAP) defining the pattern. If the brush style is BS_SOLID or BS_HATCHED, the **lbColor** field specifies the color in which the brush is to be drawn. However, for palette-capable devices, if the high byte is 0FFH, then the low WORD is a color index. Use that index for the physical color in the corresponding PPEN structure. If the high byte is 00H, then you have an RGB.

Field	Description
lbHatch	If the brush style is BS_SOLID or BS_HOLLOW, the **lbHatch** field is not used. If the brush style is BS_HATCHED, the **lbHatch** field specifies the orientation of the lines used to create the hatch. The possible hatch values are as follows:

HS_HORIZONTAL	horizontal hatch
HS_VERTICAL	vertical hatch
HS_FDIAGONAL	45-degree upward hatch from left to right
HS_BDIAGONAL	45-degree downward hatch from left to right
HS_CROSS	horizontal and vertical cross-hatch
HS_DIAGCROSS	45-degree cross-hatch

lbBkColor	If the brush style is BS_HATCHED, the **lbBkColor** field specifies the background color of the hatched brush. However, for palette-capable devices, if the high byte is 0FFH, then the low WORD is a color index. Use that index for the physical color in the corresponding PPEN structure. If the high byte is 00H, then you have an RGB.

12.3 Physical Data Structures

This section describes data structures that contain information about physical devices. These data structures vary across devices, and their definition depends on the actual device.

12.3.1 BITMAP — Physical Bitmap Data Structure

A physical bitmap describes a rectangle of bits in main memory (private bitmap).

```
typedef struct {
        short bmType;
        short bmWidth;
        short bmHeight;
        short bmWidthBytes;
        byte  bmPlanes;
        byte  bmBitsPixel;
        long  bmBits;
        long  bmWidthPlanes;
        long  bmlpPDevice;
        short bmSegmentIndex;
        short bmScanSegment;
        short bmFillBytes;
        short reserved1;
```

```
    short reserved2;
    } BITMAP;
```

The fields within the BITMAP structure have the following meanings:

Field	Description
bmType	If the memory backing the bitmap is allocated in main memory, this field is zero, and the remaining fields in the data structure have the meanings defined here. If this is a display bitmap, **bmType** is a unique number describing that physical display, and the remaining information in the data structure is as defined by the OEM (see the preceding PDEVICE structure description). This field permits hardware architectures to treat display memory differently than main memory, which may be of importance if the display bitmap is organized differently than bitmaps in main memory. For physical display bitmaps, this field and all the remaining fields are initialized by the OEM-supplied Enable function for the dedicated display.
bmWidth	Width of the bitmap in pixels.
bmHeight	Height of the bitmap in raster lines.
bmWidthBytes	Number of bytes in each raster line of this bitmap. This value is precomputed for easy calculation of the address of the next raster line. **bmWidthBytes** must be an even number so that all the scanlines will be aligned on a WORD boundary. Notice that **bmWidthBytes*8** can exceed **bmWidth** in the case of a bitmap whose width in bits is not a multiple of 16.
bmPlanes	Specifies the number of planes in frame-buffer memory.
bmBitsPixel	Specifies the number of adjacent bits on each plane that are involved in making up a pixel.
bmBits	A long pointer to the array of pixels for this bitmap. For main memory bitmaps, this is an actual memory address. This memory address is guaranteed to be aligned on a WORD boundary.
bmWidthPlanes	Specifies the width in bytes of each plane involved in making up the bitmap. It is equal to **bmWidthBytes*bmHeight.**
bmlpPDevice	A long pointer to the PDEVICE structure of the device for which this bitmap is compatible.

Field	Description
bmSegmentIndex	For bitmaps that are greater than 64K in length, this field is nonzero. Otherwise, it is zero. It is used as a flag to tell you whether or not you have a "huge" bitmap. To compute the segment address for segment *n*, add *n****bmSegmentIndex** to the starting segment address of the bitmap.
bmScanSegment	Specifies the number of raster (scan) lines contained in each (64K) segment of a "huge" bitmap. (It is not used for small bitmaps.) The total number of segments is equal to ceilling of (**bmHeight/bmScanSegment**). A raster line is equal to **bmWidthBytes** bytes. No segment may contain more than 64K.
bmFillBytes	Specifies the number of extra bytes in each segment. Bitmaps are allocated on 16-byte boundaries.

A Huge Bitmap Example

This example is for a display, such as the 3-plane EGA, that registers itself (in the GDIINFO data structure) as having more than one bitplane. The following is the bitmap in RAM:

```
+-------------------------------------+
|         Bitmap Header
|     as described above
+-------------------------------------+
             *
             *
             *
+-------------------------------------+ beginning of first segment
|Plane 0, scan line 0
|Plane 1, scan line 0
|Plane 2, scan line 0
|Plane 0, scan line 1
|Plane 1, scan line 1
|Plane 2, scan line 1
|Plane 0, scan line 2
|(etc.)      *
|            *
|            *
|Plane 0, line j-1
|Plane 1, line j-1    ________________
|Plane 2, line j-1  |  bmFillBytes    | <-- number of bytes of fill
+-------------------------------------+ end of first/beginning of second
|Plane 0, scan line j
|Plane 1, scan line j
|Plane 2, scan line j
|Plane 0, scan line j+1
|Plane 1, scan line j+1
```

```
|Plane 2, scan line j+1
|(etc.)     *
|           *
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ (gap for rest of 2nd and further)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ continuation of n-1st
|           *                               (this is the last complete
segment)
|           *
|           *
|Plane 0, scan line m-2
|Plane 1, scan line m-2
|Plane 2, scan line m-2
|Plane 0, scan line m-1
|Plane 1, line m-1   ________________
|Plane 2, line m-1  |  bmFillBytes    | <-- number of bytes of fill
+-------------------------------------+ end of n-1st/beginning of nth
|Plane 0, scan line m
|Plane 1, scan line m
|Plane 2, scan line m
|(etc.)     *
|           *                                 (this segment is not
completely
|           *                                 filled, and does NOT have any
|Plane 0, scan line n-1                       FillBytes.)
|Plane 1, scan line n-1
|Plane 2, scan line n-1                       end of last scan line.
+-------------------------------------+
      Diagram 1: Huge Bitmap, in RAM
```

All the planes of a particular scanline must fit within the segment. Otherwise, go to the next segment and leave **bmFillBytes** worth of empty bytes at the end of the current segment.

The "outside world" (in this case, the EGA) does not see things this way, and some translation must occur. It is performed by GDI. Diagram 2 shows the format expected by the EGA:

```
+-------------------------------------+ beginning of first plane
|scan line 0
|scan line 1
|scan line 2
|scan line 3
|           *
|           *
|           *
|scan line n-1
+-------------------------------------+ end of first/beginning of second
|scan line 0
|scan line 1
|scan line 2
|scan line 3
|           *
```

```
|          *
|          *
|scan line n-1
+------------------------------------+ end of second/beginning of third
|scan line 0
|scan line 1
|scan line 2
|scan line 3
|          *
|          *
|          *
|scan line n-1
+------------------------------------+ end of third; also end of screen
Diagram 2: 3 Plane EGA Format, not a Huge Bitmap
```

12.3.2 PDEVICE — Private Device Data Structure

This data structure varies across devices. Device-dependent information can be stored in this data structure to indicate the current state of a given device. The type of information stored may include the current pen, the current position, and the communication port of a particular device.

The PDEVICE data structure is allocated by GDI before any call to enable a device driver. The size of this data structure may vary and must be specified in the **dpDEVICEsize** field of the GDIINFO data structure. A single field is present in all cases, which allows GDI to determine whether this is a system display or some other device.

```
typedef struct {
        short magic;
        } PDEVICE;
```

The first WORD is the one constant field in this data structure that is required for all devices. This field, when zero, specifies that the device is a memory bitmap. When the field is nonzero, it specifies that the device is a physical system display bitmap. The Enable function sets this field to a nonzero value.

12.3.3 PCOLOR - Physical Color Definition

A physical color specifies the color bits to be activated to achieve a given color on the device.

```
typedef long PCOLOR;
```

The definition of a physical color differs, depending on whether the colors are generated by contiguous bits or by multiple color planes. This specification is entirely dependent on the device.

GDI does not use physical colors directly. Instead, it passes them to the appropriate output functions to be realized by the device driver.

12.3.4 PPEN - Physical Pen Data Structure

The PPEN structure is filled by the **RealizeObject** function and passed to the **Output** function to specify the physical pen to be used for drawing lines. The exact size and content of a physical pen depend on the device for which it is formed. The **RealizeObject** function must fill this structure with appropriate values by translating the logical pen definition passed to it into physical pen specifications for the device. Before calling **RealizeObject**, GDI allocates sufficient space for the structure by calling **RealizeObject** with a NULL pointer to the **Output** object to get the size of the structure for which it needs to allocate space.

12.3.5 PBRUSH — Physical Brush Data Structure

The PBRUSH structure is filled by the **RealizeObject** function and passed to the **Output** function to specify the physical brush to be used for painting regions. The exact size and content of a physical brush depends on the device for which it is formed. The **RealizeObject** function must fill this structure with appropriate values by translating the logical brush definition passed to it into physical brush specifications for the device. Before calling **RealizeObject**, GDI allocates sufficient space for the structure by calling **RealizeObject** with a NULL pointer to the **Output** object to get the size of the structure for which it needs to allocate space.

12.4 Raster and Vector Font File Formats

In addition to the information in the header of the file, a raster font file contains a string of bytes, the actual bitmap, just as it will be loaded into contiguous memory by GDI. That string begins in the file at the offset specified in the **dfBitsPointer** field described in the following section.

The header information for a vector font file is also described in the following section. This section describes some additional information for vector font files.

Each character is composed of a series of vectors consisting of a pair of signed relative coordinate pairs starting from the character cell origin. Each pair may be preceded by a special value indicating that the next coordinate is to be a pen-up move. The special pen-up value depends on how the coordinates are stored. For 1-byte quantities, it is -128 (080H) and for 2-byte quantities, it is -32768 (08000H).

The character cell origin must be at the upper-left corner of the cell so that the character hangs down and to the right of where it is placed.

The storage format for the coordinates depends on the size of the font. If either **dfPixHeight** or **dfMaxWidth** is greater than 128, the coordinates are stored as 2-byte quantities; otherwise, they are stored as 1-byte quantities.

12.4.1 FONTINFO — The Physical Font Descriptor

A font descriptor contains all the information about a physical font needed by the low-level character draw primitives. This data structure is identical to the font file format described in Chapter 13, "The Font File Format," but with two exceptions. First, this FONTINFO does not include the **dfVersion**, **dfSize**, and **dfCopyright** fields. Second, the **dfDevice**, **dfFace**, **dfBitsPointer**, and **dfBitsOffset** fields are offset from the beginning of the segment containing the FONTINFO data structure rather than from the beginning of the file.

For Windows 3.0, these two versions of FONTINFO now also include the glyph table in **dfCharTable**, which consists of structures that describe the bits for the characters in the font file, and six new fields: **dfFlags**, **dfAspace**, **dfBspace**, **dfCspace**, **dfColorPointer**, and **dfReserved1**. The Windows 2.x version of FONTINFO, however, is still supported.

```
typedef struct {
        short dfType;
        short dfPoints;
        short dfVertRes;
        short dfHorizRes;
        short dfAscent;
        short dfInternalLeading;
        short dfExternalLeading;
        char  dfItalic;
        char  dfUnderline;
        char  dfStrikeOut;
        short dfWeight;
        char  dfCharSet;
        short dfPixWidth;
        short dfPixHeight;
        char  dfPitchAndFamily;
        short dfAvgWidth;
        short dfMaxWidth;
        char  dfFirstChar;
        char  dfLastChar;
        char  dfDefaultChar;
        char  dfBreakChar;
        short dfWidthBytes;
        long  dfDevice;
        long  dfFace;
        long  dfBitsPointer;
        long  dfBitsOffset;
        char  dfReserved;
        long  dfFlags;
        short dfAspace;
        short dfBspace;
        short dfCspace;
        long  dfColorPointer;
        long  dfReserved1[4];
        short dfCharTable;
        char  Facename[n];
char  Devicename[n];
char  BitMaps[n];
} FONTINFO;
```

The fields within the FONTINFO structure have the following meanings:

Field	Description
dfType	Two bytes specifying the type of font file. The low-order byte is for exclusive GDI use. If the low-order bit of the WORD is zero, it is a bitmap (raster) font file. If the low-order bit is 1, it is a vector font file. The second bit is reserved and must be zero. If no bits follow in the file and the bits are located in memory at a fixed address specified in **dfBitsOffset**, the third bit is set to 1; otherwise, this bit is set to zero. The high-order bit of the low byte is set if the font was realized by a device. The remaining bits in the low byte are reserved and set to zero. The high byte is reserved for device use and will always be set to zero for GDI-realized standard fonts. Physical fonts with the high-order bit of the low byte set may use this byte to describe themselves. GDI will never inspect the high byte.
dfPoints	Two bytes specifying the nominal point size at which this character set looks best.
dfVertRes	Two bytes specifying the nominal vertical resolution (dots per inch) at which this character set was digitized.
dfHorizRes	Two bytes specifying the nominal horizontal resolution (dots per inch) at which this character set was digitized.
dfAscent	Two bytes specifying the distance from the top of a character definition cell to the baseline of the typographical font. It is useful for aligning the baseline of fonts of different heights.
dfInternalLeading	Specifies the amount of leading inside the bounds set by **dfPixHeight**. Accent marks may occur in this area. This may be zero at the designer's option.
dfExternalLeading	Specifies the amount of extra leading that the designer requests the application add between rows. Since this area is outside of the font proper, it contains no marks and will not be altered by text output calls in either the OPAQUE or TRANSPARENT mode. This may be zero at the designer's option.
dfItalic	One byte specifying whether or not the character definition data represent an italic font. The low-order bit is 1 if the flag is set. All other bits are zero.

Field	Description
dfUnderline	One byte specifying whether or not the character definition data represent an underlined font. The low-order bit is 1 if the flag is set. All other bits are zero.
dfStrikeOut	One byte specifying whether or not the character definition data represent a struck out font. The low-order bit is 1 if the flag is set. All other bits are zero.
dfWeight	Two bytes specifying the weight of the characters in the character definition data, on a scale from 1-1000. A value of 400 specifies regular weight type; 700 is bold; and so on.
dfCharSet	One byte specifying the character set defined by this font. The IBM PC hardware font has been assigned the designation 255 (FF Hex) and the ANSI character set has been assigned the designation zero.
dfPixWidth	Two bytes. For vector fonts, it specifies the width of the grid on which the font was digitized. For raster fonts, if **dfPixWidth** is non-zero, it represents the width for all characters in the bitmap; if it is zero, the font has variable width characters whose widths are specified in the **dfCharWidth** array.
dfPixHeight	Two bytes specifying the height of the character bitmap (raster fonts) or the height of the grid on which a vector font was digitized.
dfPitchAndFamily	Specifies the pitch and font family. The low bit is set if the font is variable pitch. The high 4 bits give the family name of the font. Font families describe, in a general way, the look of a font. They are intended for specifying fonts when the exact facename desired is not available. The families are as follows:

FF_DONTCARE(00H)	Don't care or don't know.
FF_ROMAN(10H)	Proportionally spaced fonts with serifs. Times Roman, Century Schoolbook, Bodoni, etc.
FF_SWISS(20H)	Proportionally spaced fonts without serifs. Helvetica, Univers, Swiss, etc.
FF_MODERN(30H)	Fixed-pitch fonts. Pica, Elite, Courier, etc.
FF_SCRIPT(40H)	Cursive or script fonts.

Field	Description
	FF_DECORATIVE (50H) — Novelty fonts. Old English, etc.
dfAvgWidth	Two bytes specifying the width of characters in the font. For fixed-pitch fonts, this is the same as **dfPixWidth.** For variable-pitched fonts, this is the width of the character "X."
dfMaxWidth	Two bytes specifying the maximum pixel width of any character in the font. For fixed-pitch fonts, this is simply **dfPix-Width.**
dfFirstChar	One byte specifying the first character code defined by this font. Character definitions are stored only for the characters actually present in a font, so this field should be used when calculating indexes into either **dfBits** or **dfCharWidth.**
dfLastChar	One byte specifying the last character code defined by this font. Notice that all the characters with codes between **dfFirst-Char** and **dfLastChar** must be present in the font character definitions.
dfDefaultChar	One byte specifying the character to be substituted whenever a string contains a character out of the range **dfFirstChar** through **dfLastChar.** The character is given relative to **dfFirst-Char** so that **dfDefaultChar** is the actual value of the character less **dfFirstChar.** Ideally, **dfDefaultChar** should be a visible character in the current font, e.g., a period (.).
dfBreakChar	One byte specifying the character that will define word breaks. This character defines word breaks for word wrapping and wordspacing justification. The character is given relative to **dfFirstChar** so that **dfBreakChar** is the actual value of the character less **dfFirstChar. dfBreakChar** is normally (32 - **dfFirstChar**), which is an ASCII space.
dfWidthBytes	Two bytes specifying the number of bytes in each row of the bitmap (raster fonts). No meaning for vector fonts. **dfWid-thBytes** is always an even quantity so that rows of the bitmap start on WORD boundaries.
dfDevice	Four bytes specifying the offset from the beginning of the segment containing the FONTINFO data structure to the string giving the device name. For a generic device, this value will be zero (0).
dfFace	Four bytes specifying the offset from the beginning of the segment containing the FONTINFO data structure to the NULL-terminated string that names the face.

Field	Description
dfBitsPointer	Four bytes specifying the absolute machine address of the bitmap. This is set by GDI at load time. **dfBitsPointer** is guaranteed to be even.
dfBitsOffset	Four bytes specifying the offset from the beginning of the segment containing the FONTINFO structure to the beginning of the bitmap information. If the 04H bit in **dfType** is set, then **dfBitsOffset** is an absolute address of the bitmap (probably in ROM). For raster fonts, it points to a sequence of bytes that make up the bitmap of the font, whose height is the height of the font, and whose width is the sum of the widths of the characters in the font rounded up to the next WORD boundary. For vector fonts, it points to a string of bytes or WORDS (depending on the size of the grid on which the font was digitized) that specifies the strokes for each character of the font. **dfBitsOffset** must be even.
dfFlags	Four bytes specifying the bits flags, which are additional flags that define the format of the Glyph bitmap, as follows:

`DFF_FIXED equØØØ1h;`	`font is fixed pitch`
`DFF_PROPORTIONAL equØØØ2h;`	`font is proportional pitch`
`DFF_ABCFIXED equØØØ4h;`	`font is an ABC fixed font`
`DFF_ABCPROPORTIONAL equØØØ8h;`	`font is an ABC proportional font`
`DFF_1COLOR equØØ1Øh;`	`font is one color`
`DFF_16COLOR equØØ2Øh;`	`font is 16 color`
`DFF_256COLOR equØØ4Øh;`	`font is 256 color`
`DFF_RGBCOLOR equØØ8Øh;`	`font is RGB color`

dfAspace	Two bytes specifying the global A space, if any. **dfAspace** is the distance from the current position to the left edge of the bitmap.
dfBspace	Two bytes specifying the global B space, if any. **dfBspace** is the width of the character.

Field	Description
dfCspace	Two bytes specifying the global C space, if any. **dfCspace** is the distance from the right edge of the bitmap to the new current position. The increment of a character is the sum of the three spaces. These apply to all glyphs and is the case of DFF_ABCFIXED.
dfColorPointer	Two bytes specifying the offset to the color table for color fonts, if any. The format of the bits is like a DIB, but without the header. That is, the characters are not split up into disjoint bytes. Instead, they are left intact. If no color table is needed, this entry is NULL.
dfCharTable	For raster fonts, this field contains four bytes for each character in the font. The first two bytes give the width of the character in pixels, and the second two bytes give the offset to the beginning of the character from the beginning of the segment that contains the FONTINFO data structure. For fixed pitch vector fonts, each 2-byte entry in this array specifies the offset from the start of the bitmap to the beginning of the string of stroke-specification units for the character. The number of bytes or WORDs to be used for a particular character is calculated by subtracting its entry from the next one. For proportionally spaced vector fonts, each 4-byte entry is divided into two 2-byte fields. The first field gives the starting offset from the start of the bitmap of the character strokes as for fixed-pitch fonts. The second field gives the pixel width of the character. ***NOTE*** In each font, there is an extra entry at the end of the **dfCharTable** table. This is to allow you to calculate the width or number of bytes of definition of the last character. Though this applies only to vector fonts, the entry is present for all fonts.
Facename	An ASCII character string specifying the name of the font face. The size of this field is the length of the string plus a NULL terminator.
Devicename	An ASCII character string specifying the name of the device if this font file is for a specific device. The size of this field is the length of the string plus a NULL terminator.
Bitmaps	This field contains the bitmap definitions. The size of this field is whatever length the total bitmaps occupy. Each row of a raster bitmap must start on a WORD boundary. This implies that the end of each row must be padded to an even length.

NOTE When a device realizes a font using the **RealizeObject** function, the **dfFace** and **dfDevice** fields must point to valid character strings containing the face and device names.

The glyph entries become dependent on the format of the Glyph bitmap.

```
DFF_FIXED
DFF_PROPORTIONAL

GlyphEntry      struc
geWidth         dw      ?       ; width of character bitmap in pixels
geOffset        dd      ?       ; pointer to the bits
GlyphEntry      ends

DFF_ABCFIXED
DFF_ABCPROPORTIONAL

GlyphEntry      struc
geWidth         dw      ?       ; width of character bitmap in pixels
geOffset        dd      ?       ; pointer to the bits
geAspace        dd      ?       ; A space in fractional pixels (16.16)
geBspace        dd      ?       ; B space in fractional pixels (16.16)
geCspace        dd      ?       ; C space in fractional pixels (16.16)
GlyphEntry      ends
```

The fractional pixels are expressed as a 32-bit signed number with an implicit binary point between bits 15 and 16. This is referred to as a 16.16 ("sixteen dot sixteen") fixed-point number.

The ABC spacing here is the same as defined above. However, here there are specific sets for each character.

```
DFF_1COLOR
DFF_16COLOR
DFF_256COLOR
DFF_RGBCOLOR

GlyphEntry      struc
geWidth         dw      ?       ; width of character bitmap in pixels
geOffset        dd      ?       ; pointer to the bits
geHeight        dw      ?       ; height of character bitmap in pixels
geAspace        dd      ?       ; A space in fractional pixels (16.16)
geBspace        dd      ?       ; B space in fractional pixels (16.16)
geCspace        dd      ?       ; C space in fractional pixels (16.16)
GlyphEntry      ends

DFF_1COLOR means 8 pixels per byte

DFF_16COLOR means 2 pixels per byte

DFF_256COLOR means 1 pixel per byte
```

```
DFF_RGBCOLOR means RGBquads
```

NOTE The only format supported in Windows 3.0 will be DFF_FIXED and DFF_PROPORTIONAL.

12.4.2 LOGFONT - The Logical Font Descriptor

A logical font descriptor contains all the parameters for a logical font needed by the output primitives.

```
typedef struct {
        short lfHeight;
        short lfWidth;
        short lfEscapement;
        short lfOrientation;
        short lfWeight;
        BYTE  lfItalic;
        BYTE  lfUnderline;
        BYTE  lfStrikeOut;
        BYTE  lfCharSet;
        BYTE  lfOutPrecision;
        BYTE  lfClipPrecision;
        BYTE  lfQuality;
        BYTE  lfPitchAndFamily;
        BYTE  lfFaceName [32];
        } LOGFONT;
```

The fields within the LOGFONT data structure have the following meanings:

Field	Description
lfHeight	Specifies the height of the font in user units. The height of a font can be specified in three ways. If **lfHeight** is greater than zero, it is transformed into device units and matched against the cell height of the available fonts. If **lfHeight** is zero, a reasonable default size is used. If **lfHeight** is less than zero, it is transformed into device units and the absolute value is matched against the character height of the available fonts. For all height comparisons, the font mapper looks for the largest font that does not exceed the requested size and, if there is no such font, looks for the smallest font available.

Field	Description
lfWidth	Specifies the average width of characters in the font in user units. If **lfWidth** is zero, the aspect ratio of the device will be matched against the digitization aspect ratio of the available fonts looking for the closest match by absolute value of the difference.
lfEscapement	Specifies the angle, counterclockwise from the *x*-axis in tenths of a degree, of the vector passing through the origin of all the characters in the string.
lfOrientation	Specifies the angle, counterclockwise from the *x*-axis in tenths of a degree, of the baseline of the character.
lfWeight	Specifies the weight of the font ranging from 1 to 1000, with 400 being the value for the standard font. Passing a weight of zero signals the font mapper to choose any value.
lfItalic	A 1-byte flag that specifies whether or not the font is to be italic. If the low bit is set, the font is to be italic. All other bits are to be zero.
lfUnderline	A 1-byte flag that specifies whether or not the font is to be underlined. If the low bit is set, the font is to be underlined. All other bits are to be zero.
lfStrikeOut	A 1-byte flag that specifies whether or not the font is to be struck out. If the low bit is set, the font is to be struck out. All other bits are to be zero.
lfCharSet	Specifies the character set to be used. It can be either of the following:

ANSI_CHARSET	(00H)
OEM_CHARSET	(FFH)

The ANSI character set is recommended since it is constant across Windows machines and is available in more fonts than any other set. The OEM character set depends on the specific machine. See Chapter 15, "Miscellaneous Character Set Tables," for a table of the ANSI character set and the OEM character set distributed with the IBM PC.

Field	Description
lfOutPrecision	Specifies the required output precision for text. Output precision is described in detail in the GDIINFO data structure. Output precision may be one of the following values:

OUT_DEFAULT_PRECIS	(00H)

Field	Description	
	OUT_STRING_PRECIS	(01H)
	OUT_CHARACTER_PRECIS	(02H)
	OUT_STROKE_PRECIS	(03H)
lfClipPrecision	Specifies the required clipping precision for text. Clipping precision is described in detail in the GDIINFO data structure. Clipping precision may be one of the following values:	
	CLIP_DEFAULT_PRECIS	(00H)
	CLIP_CHARACTER_PRECIS	(01H)
	CLIP_STROKE_PRECIS	(02H)
lfQuality	A 1-byte flag that provides a hint to the font mapper as to what quality output is required. A hint is information that the mapper may use when it needs additional clarification to make a choice of which physical font to use. Quality may be one of the following values:	

Value	Description
DEFAULT_QUALITY(00H)	Don't care.
DRAFT_QUALITY(01H)	The appearance of the font is not as important. For GDI fonts, scaling is enabled so that more sizes are available, at the cost of appearance. Bold, italic, underline, and strikeout will be synthesized if needed.

Field	Description	
	PROOF_QUALITY(02H)	The character quality of the font is more important than the exact matching of the logical font attributes. For GDI fonts, scaling is inhibited. Therefore, it may not be possible to map as exact a size as at the lower qualities. However, there will be no degradation of appearance. Bold, italic, underline, and strikeout will be synthesized if needed.
lfPitchAndFamily	Specifies the font pitch and family. The low 2 bits specify the pitch of the font and can be any one of the following:	

DEFAULT_PITCH	(00H)
FIXED_PITCH	(01H)
VARIABLE_PITCH	(02H)

The high 4 bits of the field specify the font family. The constants are defined such that the proper value can be obtained by ORing together one pitch constant with one family constant. The font family name describes in a general way the look of a font. Family names are intended for specifying fonts when the exact facename desired is not available. The families are as follows:

Family	Description
FF_DONTCARE(00H)	Don't care or don't know.
FF_ROMAN(10H)	Proportionally spaced fonts with serifs. Times Roman, Century Schoolbook, Bodoni, etc.
FF_SWISS(20H)	Proportionally spaced fonts without serifs. Helvetica, Univers, Swiss, etc.

Field	Description	
	FF_MODERN(30H)	Fixed-pitch fonts. Pica, Elite, Courier, etc.
	FF_SCRIPT(40H)	Cursive or script fonts.
	FF_DECORATIVE(50H)	Novelty fonts. Old English, etc.
lfFaceName	An ASCII character string specifying the facename of the font. The size of this field is the length of the string plus a NULL terminator; it must not exceed 32, including the NULL. A string consisting of a single NULL indicates that any font face may be used.	

Chapter 13 The Font File Format

This chapter provides information on the three main data structures used to describe the physical font, its basic metrics, and the actual appearance of the text on the display device.

13.1 TEXTMETRIC - Basic Font Metrics

The TEXTMETRIC structure is a list of the basic metrics of a physical font. The structure is returned by the **GetTextMetrics** function.

```
typedef struct {
        short  tmHeight;
        short  tmAscent;
        short  tmDescent;
        short  tmInternalLeading;
        short  tmExternalLeading;
        short  tmAveCharWidth;
        short  tmMaxCharWidth;
        short  tmWeight;
        BYTE   tmItalic;
        BYTE   tmUnderlined;
        BYTE   tmStruckOut;
        BYTE   tmFirstChar;
        BYTE   tmLastChar;
        BYTE   tmDefaultChar;
        BYTE   tmBreakChar;
        BYTE   tmPitchAndFamily;
        BYTE   tmCharSet;
        short  tmOverhang;
        short  tmDigitizedAspectX;
        short  tmDigitizedAspectY;
        } TEXTMETRIC;
```

The TEXTMETRIC fields are described below. All the sizes are given in normalized units (i.e., they depend on the current mapping mode of the display context).

Field	Description
tmHeight	Specifies the height of characters (Ascent + Descent).
tmAscent	Specifies the ascent of characters (units above the base-line).

Field	Description
tmDescent	Specifies the descent of characters (units below the baseline).
tmInternalLeading	Specifies the amount of leading inside the bounds set by **tmHeight**. Accent marks may occur in this area. This may be zero at the designer's option.
tmExternalLeading	Specifies the amount of extra leading that the designer requests the application add between rows. Since this area is outside of the font proper, it contains no marks and will not be altered by text output calls in either the OPAQUE or TRANSPARENT mode. This may be zero at the designer's option.
tmAveCharWidth	Specifies the average width of characters in the font (loosely defined as the width of the letter "X").
tmMaxCharWidth	Specifies the maximum width of any character in the font.
tmWeight	Specifies the weight of the font.
tmItalic	If non-zero, specifies an italic font.
tmUnderlined	If non-zero, specifies an underlined font.
tmStruckOut	If non-zero, specifies a struckout font.
tmFirstChar	Specifies the value of the first character defined in the font.
tmLastChar	Specifies the value of the last character defined in the font.
tmDefaultChar	Specifies the value of the character that is to be substituted for characters that are not in the font.
tmBreakChar	Specifies the value of the character that is to be used to define word breaks for text justification.
tmPitchAndFamily	Specifies the pitch and family of the selected font. The low bit is set if the font is variable pitch. The high four bits give the family of the font. Refer to the LOGFONT structure for a description of the font families.
tmCharSet	Specifies the character set of the font.

Field	Description
tmOverhang	Specifies the per string extra width that may be added to some synthesized fonts. When synthesizing some attributes such as bold or italic, GDI or a device may have to add width to a string on both a per character and per string basis.

For example, GDI emboldens a string by expanding the intracharacter spacing and overstriking with an offset. It italicizes a font by skewing the string. In either case, there is an overhang past the basic string. For bold strings, it is the distance by which the overstrike is offset. For italic strings, it is the amount the top of the font is skewed past the bottom of the font.

The **tmOverhang** enables the application to determine the following:

1. How much of the character width returned by a **GetTextExtent** call on a single character is the actual character width, and how much is the per string extra width.

2. The actual width is the extent less the overhang. Alternately, **tmOverhang** is the difference between the width of a character when output singly versus in the interior of a string.

```
Character Width      Character Width
       |                    |
|------------|        |------------|

|         |                /         /|
|         |               /         / |
|         |              /         /  |
|         |             /         /   |
|---------|            /---------/    |
|         |           /         /     |
|         |          /         /      |
|         |         /         /       |
|         |        /         /        |
                              |-------|
|------------||------------|
                              Overhang
Character width      Character width
(including           (including
whitespace)          whitespace)

Overhang = 0         Overhang > 0
```

Field	Description
tmDigitizedAspectX, tmDigitizedAspectY	Specify the aspect ratio of the device for which this font was designed. The ratio of **tmDigitizedAspectY** to **tmDigitizedAspectX** can be compared against the ratio of *AspectY* to *AspectX* retrieved from **GetDeviceCaps.**

13.2 TEXTXFORM – Actual Text Appearance Information

This data structure describes the actual text appearance as displayed by the device. If there are differences between the TEXTXFORM and FONTINFO data structures, **StrBlt/Ext-TextOut** is responsible for accommodating the differences for which it has claimed abilities, as specified in the **dpText** field in the GDIINFO data structure. However, there may be more differences than the device can transform. In that case, GDI is responsible for simulating the required transformations.

Most of the fields in this data structure correspond to the fields in the LOGFONT data structure, but are expressed in device units. Notice that these fields may not correspond exactly to the logical font. For example, if the logical font specified a 19-unit font at string precision and the closest available is a 9-unit font on a device capable of doubling, then **Height** in the transform is 18.

```
typedef struct {
        short  Height;
        short  Width;
        short  Escapement;
        short  Orientation;
        short  Weight;
        char   Italic;
        char   Underline;
        char   StrikeOut;
        char   OutPrecision;
        char   ClipPrecision;
        short  Accelerator;
        short  Overhang;
        } TEXTXFORM;
```

The fields within the TEXTXFORM data structure have the following meanings:

Field	Description
Height	Specifies the height in device units from the bottom of the lowest descending character to the top of the tallest character.
Width	Specifies the width in device units of the bounding box of the letter "X."

Field	Description
Escapement	Specifies the angle in degrees counterclockwise from the X-axis of the vector passing through the origin of all the characters in the string.
Orientation	Specifies the angle in degrees counterclockwise from the X-axis of the baseline of the character.
Weight	Specifies the weight of the font ranging from 1 to 1000, with 200 being the value for the standard font.
Italic	This field is a 1-byte flag that specifies whether or not the font is to be italic. If the low bit is set, the font is to be italic. All the other bits are to be zero.
Underline	This field is a 1-byte flag that specifies whether or not the font is to be underlined. If the low bit is set, the font is to be underlined. All the other bits are to be zero.
StrikeOut	This field is a 1-byte flag that specifies whether or not the font is to be struck out. If the low bit is set, the font is to be struck out. All the other bits are to be zero.
OutPrecision	Specifies the required output precision for this font. Output precision is described in detail in the GDIINFO data structure. Output precision may be one of the following values: OUT_DEFAULT_PRECIS OUT_STRING_PRECIS OUT_CHARACTER_PRECIS OUT_STROKE_PRECIS
ClipPrecision	Specifies the required clipping precision for this font. Clipping precision is described in detail in the GDIINFO data structure. Clipping precision may be one of the following values: CLIP_DEFAULT_PRECIS CLIP_CHARACTER_PRECIS CLIP_STROKE_PRECIS
Accelerator	This field has a bit-for-bit correspondence with the **dpText** field in the GDIINFO data structure. Each bit in this field is set if the corresponding ability is required to transform the physical font (FONTINFO) into the displayed font (TEXTXFORM) as described by the logical font (LOGFONT).

Field	Description
	GDI uses the bitwise difference between the **Accelerator** field and **dpText** field to determine what abilities it should simulate. The device may use the **Accelerator** field to determine which attributes it should perform based upon what needs to be done, what it can do, and what GDI has simulated. By performing an AND operation on **Accelerator** and **dpText**, **StrBlt** can determine which transforms it is responsible for performing.
Overhang	This field has the same meaning as the **tmOverhang** field in the TEXTMETRIC data structure. This field is set by the device for device-realized fonts and is in device units. Notice that GDI uses additional overhang if it emboldens the font.

13.3 FONTINFO - The Physical Font

For Windows 3.0, FONTINFO also includes the glyph table in **dfCharTable**, which consists of structures that describe the bits for the characters in the font file and six new fields: **dfFlags, dfAspace, dfBspace, dfCspace, dfColorPointer**, and **dfReserved1**. The Windows 2.x version of FONTINFO, however, is still supported.

```
typedef struct {
        dfVersion          dw    0
        dfSize             dd    0
        dfCopyright        db    60 dup (0)
        dfType             dw    0       ; Type field for the font.
        dfPoints           dw    0       ; Point size of font.
        dfVertRes          dw    0       ; Vertical digitization.
        dfHorizRes         dw    0       ; Horizontal digitization.
        dfAscent           dw    0       ; Baseline offset from character cell top.
        dfInternalLeading  dw    0       ; Internal leading included in font.
        dfExternalLeading  dw    0       ; Preferred extra space between lines.
        dfItalic           db    0       ; Flag specifying if italic.
        dfUnderline        db    0       ; Flag specifying if underlined.
        dfStrikeOut        db    0       ; Flag specifying if struck out.
                                         ; */ BYTE dfStrikeOut; /*
        dfWeight           dw    0       ; Weight of font.
        dfCharSet          db    0       ; Character set of font.
        dfPixWidth         dw    0       ; Width field for the font.
        dfPixHeight        dw    0       ; Height field for the font.
        dfPitchAndFamily   db    0       ; Flag specifying variable pitch, family.
        dfAvgWidth         dw    0       ; Average character width.
        dfMaxWidth         dw    0       ; Maximum character width.
        dfFirstChar        db    0       ; First character in the font.
        dfLastChar         db    0       ; Last character in the font.
        dfDefaultChar      db    0       ; Default character for out of range.
        dfBreakChar        db    0       ; Character to define wordbreaks.
        dfWidthBytes       dw    0       ; Number of bytes in each row.
dfDevice         dd    0        ; Offset to device name.
dfFace           dd    0        ; Offset to face name.
dfBitsPointer    dd    0        ; Bits pointer.
dfBitsOffset     dd    0        ; Offset to the beginning of the bitmap.
                                ; On the disk, this is relative to the
                                ; beginning of the file. In memory, this
                                ; is relative to the beginning of this
                                ; structure.
dfReserved       db    0        ; 1 byte reserved.
dfFlags          dd    0        ; Bit flags.
dfAspace         dw    0        ; Global A space, if any.
dfBspace         dw    0        ; Global B space, if any.
dfCspace         dw    0        ; Global C space, if any.
dfColorPointer   dd    0        ; Offset to color table, if any.
dfReserved1      dd    4 dup (0)
dfCharTable      dw    0        ; Area for storing the character widths
                                ; and offsets, face name, device name
                                ; (option), and bitmap.
                                ; unsigned short dfMaps[DF_MAPSIZE]
} FONTINFO
```

NOTE The constant "DF_MAPSIZE" must be defined prior to the INCLUDE statement of the GDIDEFS.INC file, or the array will default to one character element. This leaves room only for a single set of NULL to designate no typeface name, no device name, and no bitmaps.

The fields within the FONTINFO data structure have the following meanings:

Field	Description
dfVersion	Two bytes specifying the version of the file.
dfSize	Four bytes specifying the total size of the file in bytes.
dfCopyright	Sixty (60) bytes specifying copyright information.
dfType	Two bytes specifying the type of fontfile. The low-order byte is exclusively for GDI use. If the low-order bit of the WORD is zero, it is a bitmap (raster) fontfile. If the low-order bit is 1, it is a vector fontfile. The second bit is reserved and must be zero. If no bits follow in the file and the bits are located in memory at a fixed address specified in **dfBitsOffset,** the third bit is set to 1; otherwise, the bit is set to zero. The high-order bit of the low byte is set if the font was realized by a device. The remaining bits in the low byte are reserved and set to zero.

Field	Description
	The high byte is reserved for device use and will always be set to zero for GDI-realized standard fonts. Physical fonts with the high-order bit of the low byte set may use this byte to describe themselves. GDI will never inspect the high byte.
dfPoints	Two bytes specifying the nominal point size at which this character set looks best.
dfVertRes	Two bytes specifying the nominal vertical resolution (dots per inch) at which this character set was digitized.
dfHorizRes	Two bytes specifying the nominal horizontal resolution (dots per inch) at which this character set was digitized.
dfAscent	Two bytes specifying the distance from the top of a character definition cell to the baseline of the typographical font. It is useful for aligning the baselines of fonts of different heights.
dfInternalLeading	Specifies the amount of leading inside the bounds set by **dfPixHeight**. Accent marks may occur in this area. This may be zero at the designer's option.
dfExternalLeading	Specifies the amount of extra leading that the designer requests the application add between rows. Since this area is outside of the font proper, it contains no marks and will not be altered by text output calls in either the OPAQUE or TRANSPARENT mode. This may be zero at the designer's option.
dfItalic	One byte specifying whether or not the character definition data represent an italic font. The low-order bit is 1 if the flag is set. All the other bits are zero.
dfUnderline	One byte specifying whether or not the character definition data represent an underlined font. The low-order bit is 1 if the flag is set. All the other bits are zero.
dfStrikeOut	One byte specifying whether or not the character definition data represent a struckout font. The low-order bit is 1 if the flag is set. All the other bits are zero.
dfWeight	Two bytes specifying the weight of the characters in the character definition data, on a scale of 1 to 1000. A **dfWeight** of 400 specifies a regular weight.
dfCharSet	One byte specifying the character set defined by this font.

Field	Description
dfPixWidth	Two bytes. For vector fonts, specifies the width of the grid on which the font was digitized. For raster fonts, if **dfPix-Width** is nonzero, it represents the width for all the characters in the bitmap; if it is zero, the font has variable width characters whose widths are specified in the **dfCharTable** array.
dfPixHeight	Two bytes specifying the height of the character bitmap (raster fonts), or the height of the grid on which a vector font was digitized.
dfPitchAndFamily	Specifies the pitch and font family. The low bit is set if the font is variable pitch. The high four bits give the family name of the font. Font families describe in a general way the look of a font. They are intended for specifying fonts when the exact face name desired is not available. The families are as follows:

FF_DONTCARE (0<<4)	Don't care or don't know.
FF_ROMAN (1<<4)	Proportionally spaced fonts with serifs. Times ® Roman, Palatino ®, Century Schoolbook, etc.
FF_SWISS (2<<4)	Proportionally spaced font without serifs. Helvetica ®, Swiss ™, etc.
FF_MODERN (3<<4)	Fixed-pitch fonts. Pica, Elite, Courier, etc.
FF_SCRIPT (4<<4)	Cursive or script fonts. (Both are designed to look at least vaguely like handwriting. Script fonts have joined letters; cursive fonts do not.)
FF_DECORATIVE (5<<4)	Novelty fonts. Old English, etc.

Field	Description
dfAvgWidth	Two bytes specifying the width of characters in the font. For fixed-pitch fonts, this is the same as **dfPixWidth.** For variable-pitch fonts, this is the width of the character "X."
dfMaxWidth	Two bytes specifying the maximum pixel width of any character in the font. For fixed-pitch fonts, this is simply **dfPixWidth.**
dfFirstChar	One byte specifying the first character code defined by this font. Character definitions are stored only for the characters actually present in a font. Therefore, use this field when calculating indexes into either **dfBits** or **dfCharOffset.**
dfLastChar	One byte specifying the last character code defined by this font. Notice that all the characters with codes between **dfFirstChar** and **dfLastChar** must be present in the font character definitions.
dfDefaultChar	One byte specifying the character to substitute whenever a string contains a character out of the range. The character is given relative to **dfFirstChar** so that **dfDefaultChar** is the actual value of the character less **dfFirstChar**. The **dfDefaultChar** should indicate a special character that is not a space.
dfBreakChar	One byte specifying the character that will define word breaks. This character defines word breaks for word wrapping and word spacing justification. The character is given relative to **dfFirstChar** so that **dfBreakChar** is the actual value of the character less that of **dfFirstChar**. The **dfBreakChar** is normally (32 - **dfFirstChar**), which is an ASCII space.
dfWidthBytes	Two bytes specifying the number of bytes in each row of the bitmap. This is always even, so that the rows start on WORD boundaries. For vector fonts, this field has no meaning.
dfDevice	Four bytes specifying the offset in the file to the string giving the device name. For a generic font, this value is zero.
dfFace	Four bytes specifying the offset in the file to the NULL-terminated string that names the face.

Field	Description
dfBitsPointer	Four bytes specifying the absolute machine address of the bitmap. This is set by GDI at load time. The **dfBitsPointer** is guaranteed to be even.
dfBitsOffset	Four bytes specifying the offset in the file to the beginning of the bitmap information. If the 04H bit in the **dfType** is set, then **dfBitsOffset** is an absolute address of the bitmap (probably in ROM). For raster fonts, it points to a sequence of bytes that make up the bitmap of the font, whose height is the height of the font, and whose width is the sum of the widths of the characters in the font rounded up to the next WORD boundary. For vector fonts, it points to a string of bytes or words (depending on the size of the grid on which the font was digitized) that specify the strokes for each character of the font. The **dfBitsOffset** must be even.
dfFlags	Four bytes specifying the bits flags, which are additional flags that define the format of the Glyph bitmap, as follows:

```
DFF_FIXED           equ  0001h ; font is
                               ; fixed
                               ; pitch
DFF_PROPORTIONAL    equ  0002h ; font is pro-
                               ; portional
                               ; pitch
DFF_ABCFIXED        equ  0004h ; font is an
                               ; ABC fixed
                               ; font
DFF_ABCPROPORTIONAL equ  0008h ; font is an
                               ; ABC propor-
                               ; tional font
DFF_1COLOR          equ  0010h ; font is
                               ; one color
DFF_16COLOR         equ  0020h ; font is 16
                               ; color
DFF_256COLOR        equ  0040h ; font is
                               ; 256 color
DFF_RGBCOLOR        equ  0080h ; font is
                               ; RGB color
```

Field	Description
dfAspace	Two bytes specifying the global A space, if any. The **dfAspace** is the distance from the current position to the left edge of the bitmap.
dfBspace	Two bytes specifying the global B space, if any. The **dfBspace** is the width of the character.

Field	Description
dfCspace	Two bytes specifying the global C space, if any. The **dfCspace** is the distance from the right edge of the bitmap to the new current position. The increment of a character is the sum of the three spaces. These apply to all glyphs and is the case for DFF_ABCFIXED.
dfColorPointer	Two bytes specifying the offset to the color table for color fonts, if any. The format of the bits is like a DIB, but without the header. That is, the characters are not split up into disjoint bytes. Instead, they are left intact. If no color table is needed, this entry is NULL.
dfCharTable	For raster fonts, the **CharTable** is an array of entries each consisting of two 2-byte WORDs. The first WORD of each entry is the character width. The second WORD of each entry is the byte offset from the beginning of the FONTINFO structure to the character bitmap. There is one extra entry at the end of this table that describes an absolute-space character. This entry corresponds to a character that is guaranteed to be blank; this character is not part of the normal character set. The number of entries in the table is calculated as ((**dfLastChar** - **dfFirstChar**) + 2). This includes a spare, the *sentinel* offset mentioned below. For fixed-pitch vector fonts, each 2-byte entry in this array specifies the offset from the start of the bitmap to the beginning of the string of stroke specification units for the character. The number of bytes or WORDs to be used for a particular character is calculated by subtracting its entry from the next one, so that there is a *sentinel* at the end of the array of values. For proportionally spaced vector fonts, each 4-byte entry is divided into two 2-byte fields. The first field gives the starting offset from the start of the bitmap of the character strokes as for fixed-pitch fonts. The second field gives the pixel width of the character.
<facename>	An ASCII character string specifying the name of the font face. The size of this field is the length of the string plus a NULL terminator.
<devicename>	An ASCII character string specifying the name of the device if this font file is for a specific device. The size of this field is the length of the string plus a NULL terminator.

Field	**Description**
<bitmaps>	This field contains the character bitmap definitions. Each character is stored as a contiguous set of bytes. (In the old font format, this was not the case.) The first byte contains the first eight bits of the first scanline (i.e., the top line of the character). The second byte contains the first eight bits of the second scanline. This continues until what amounts to a first "column" is completely defined. The following byte contains the next eight bits of the first scanline, padded with zeros on the right if necessary (and so on, down through the second "column"). If the font is quite narrow, each scanline is covered by one byte, with bits set to zero as necessary for padding. If the font is very wide, a third or even fourth set of bytes can be present.

NOTE The character bitmaps must be stored contiguously and arranged in ascending order.

The following is a single-character example, in which we give the bytes for a 12x14 pixel character, as shown here schematically.

```
............
.....**.....
....*..*....
...*....*...
..*......*..
..*......*..
..*......*..
..********..
..*......*..
..*......*..
..*......*..
............
............
............
```

The bytes are given here in two sets, because the character is less than 17 pixels wide.

```
00 06 09 10 20 20 20 3F 20 20 20 00 00 00
00 00 00 80 40 40 40 C0 40 40 40 00 00 00
```

Notice that in the second set of bytes, the second digit of each is always zero. It would correspond to the 13th through 16th pixels on the right side of the character, if they were present.

NOTE The character bitmaps must be stored contiguously and arranged in ascending order.

The glyph entries become dependent on the format of the Glyph bitmap.

```
DFF_FIXED
DFF_PROPORTIONAL

GlyphEntry      struc
geWidth         dw      ?       ; width of character bitmap in pixels
geOffset        dd      ?       ; pointer to the bits
GlyphEntry      ends

DFF_ABCFIXED
DFF_ABCPROPORTIONAL

GlyphEntry      struc
geWidth         dw      ?   ; width of character bitmap in pixels
geOffset        dd      ?   ; pointer to the bits
geAspace        dd      ?   ; A space in fractional pixels (16.16)
geBspace        dd      ?   ; B space in fractional pixels (16.16)
geCspace        dd      ?   ; C space in fractional pixels (16.16)
GlyphEntry      ends
```

The fractional pixels are expressed as a 32-bit signed number with an implicit binary point between bits 15 and 16. This is referred to as a 16.16 (“sixteen dot sixteen”) fixed-point number.

The ABC spacing here is the same as defined above. However, here there are specific sets for each character.

```
DFF_1COLOR
DFF_16COLOR
DFF_256COLOR
DFF_RGBCOLOR

GlyphEntry      struc
geWidth         dw      ?   ; width of character bitmap in pixels
geOffset        dd      ?   ; pointer to the bits
geHeight        dw      ?   ; height of character bitmap in pixels
geAspace        dd      ?   ; A space in fractional pixels (16.16)
geBspace        dd      ?   ; B space in fractional pixels (16.16)
geCspace        dd      ?   ; C space in fractional pixels (16.16)
GlyphEntry      ends

DFF_1COLOR means 8 pixels per byte

DFF_16COLOR means 2 pixels per byte

DFF_256COLOR means 1 pixel per byte

DFF_RGBCOLOR means RGBquads
```

NOTE The only format supported in Windows 3.0 will be DFF_FIXED and DFF_PROPORTIONAL.

Chapter 14

Raster Operation Codes and Definitions

This chapter provides a table of raster operation codes and their definitions. The raster operation codes define the ways in which **BitBlt** combines the bits in a source bitmap with the bits in a brush or pattern bitmap and the bits in the destination bitmap.

The operands used in the operations are as follows:

S	Source bitmap
P	Paintbrush or Pattern currently selected
D	Destination bitmap

The Boolean operators used in these operations are as follows:

o	Bitwise OR
x	Bitwise Exclusive OR
a	Bitwise AND
n	Bitwise NOT (invert)

The operations are presented here in reverse Polish notation. For example, the operation **DPSoo** performs a logical "OR" on the source and pattern and, then, performs another logical "OR" with the destination. The result is then stored in the destination.

Notice that there are alternate spellings of the same function. Therefore, although a particular spelling may not be in the list, an equivalent form will be. For example, "DPoSo" is an equivalent form to "DPSoo."

In general, the functions are spelled in such a way that it is easiest to read them outward from the place at which they change from upper to lower case. For example, **PSDPSanaxx** may be read as follows:

```
PSD PSa naxx: 'and' source with pattern.
PSDPSa n axx: complement result.
PS D PSan a xx: 'and' with destination.
P S PDPSana x x: 'xor' with source.
P SDPSanax x: 'xor' with pattern.
```

The following is another, more complex example:(This expansion is of ROP 0017h from the table below.)

```
The ROP: SSPxDSxaxn
```

The expansion:

```
S SPx DSxaxn: 'xor' source and pattern.
SSPx DSx axn: 'xor' destination and source.
S [SPx][DSx]a xn: 'and' the bracketed items.
s SPxDSxa x n: 'xor' result of last step with source.
SSPxDSxax n: complement result, and put into destination.
```

14.1 The Operation Codes

Each raster operation code is a 32-bit integer value; the high-order WORD of which is a Boolean operation index, and the low-order WORD of which is the operation code. The 16-bit operation index is a zero-extended 8-bit value that represents the result of the Boolean operation on predefined pattern (P), source (S), and destination (D) values. For example, the operation indices for the *PSo*, *PSon*, and *DPSoo* operations are as follows:

P	S	D	PSo	PSon	DPSoo	Arbitrary function
0	0	0	0	1	0	1
0	0	1	0	1	1	0
0	1	0	1	0	1	0
0	1	1	1	0	1	1
1	0	0	1	0	1	1
1	0	1	1	0	1	0
1	1	0	1	0	1	1
1	1	1	1	0	1	0
Hex Opcode:			FC	03	FE	59

Any Boolean function can be represented by the string of 1's and 0's on the right side of such a table. In this case, *PSon* is the string 00000011 (read from the bottom up), which is hexadecimal 03. (Recall that this is then zero-extended to the left: 0x0003.) Notice the *PSon* function in line 4 of the table.

In general, any arbitrary function such as the one on the far right above, has a unique hexadecimal number associated with it (in this case, 0x59). By looking in the table, one then finds the appropriate Rop (in this case, 0x00590609) and a function that evaluates it (*DPSnox*).

The first four digits of each opcode determine the location of the raster operation in the table: the *PSo* operation is in line 252 (hex FC) of the table, *DPSoo* is in line 254 (hex FE), and so on.

The most commonly used Rops have been given special names. Therefore, it is recommended that programs define the common name to be the Rop number and, then, use the common name throughout, to be consistent with the current "no magic numbers" style.

14.2 The Operation Code List

The following is a list of the Boolean functions in hexadecimal and reverse Polish notation, along with the Hex Rop and common name.

Boolean function in HEX	HEX Rop	Boolean function in R Polish	Common name
00	00000042	0	BLACKNESS
01	00010289	DPSoon	-
02	00020C89	DPSona	-
03	000300AA	PSon	-
04	00040C88	SDPona	-
05	000500A9	DPon	-
06	00060865	PDSxnon	-
07	000702C5	PDSaon	-
08	00080F08	SDPnaa	-
09	00090245	PDSxon	-
0A	000A0329	DPna	-
0B	000B0B2A	PSDnaon	-
0C	000C0324	SPna	-
0D	000D0B25	PDSnaon	-
0E	000E08A5	PDSonon	-
0F	000F0001	Pn	-
10	00100C85	PDSona	-
11	001100A6	DSon	NOTSRCCOPY
12	00120868	SDPxnon	-
13	001302C8	SDPaon	-
14	00140869	DPSxnon	-
15	001502C9	DPSaon	-
16	00165CCA	PSDPSanaxx	-
17	00171D54	SSPxDSxaxn	-
18	00180D59	SPxPDxa	-
19	00191CC8	SDPSanaxn	-

Boolean function in HEX	HEX Rop	Boolean function in R Polish	Common name
1A	001A06C5	PDSPaox	-
1B	001B0768	SDPSxaxn	-
1C	001C06CA	PSDPaox	-
1D	001D0766	DSPDxaxn	-
1E	001E01A5	PDSox	-
1F	001F0385	PDSoan	-
20	00200F09	DPSnaa	-
21	00210248	SDPxon	-
22	00220326	DSna	-
23	00230B24	SPDnaon	-
24	00240D55	SPxDSxa	-
25	00251CC5	PDSPanaxn	-
26	002606C8	SDPSaox	-
27	00271868	SDPSxnox	-
28	00280369	DPSxa	-
29	002916CA	PSDPSaoxxn	-
2A	002A0CC9	DPSana	-
2B	002B1D58	SSPxPDxaxn	-
2C	002C0784	SPDSoax	-
2D	002D060A	PSDnox	-
2E	002E064A	PSDPxox	-
2F	002F0E2A	PSDnoan	-
30	0030032A	PSna	-
31	00310B28	SDPnaon	-
32	00320688	SDPSoox	-
33	00330008	Sn	-
34	003406C4	SPDSaox	-
35	00351864	SPDSxnox	-
36	003601A8	SDPox	-
37	00370388	SDPoan	-
38	0038078A	PSDPoax	-
39	00390604	SPDnox	-
3A	003A0644	SPDSxox	-
3B	003B0E24	SPDnoan	-

Boolean function in HEX	HEX Rop	Boolean function in R Polish	Common name
3C	003C004A	PSx	-
3D	003D18A4	SPDSonox	-
3E	003E1B24	SPDSnaox	-
3F	003F00EA	PSan	-
40	00400F0A	PSDnaa	-
41	00410249	DPSxon	-
42	00420D5D	SDxPDxa	-
43	00431CC4	SPDSanaxn	-
44	00440328	SDna	SRCERASE
45	00450B29	DPSnaon	-
46	004606C6	DSPDaox	-
47	0047076A	PSDPxaxn	-
48	00480368	SDPxa	-
49	004916C5	PDSPDaoxxn	-
4A	004A0789	DPSDoax	-
4B	004B0605	PDSnox	-
4C	004C0CC8	SDPana	-
4D	004D1954	SSPxDSxoxn	-
4E	004E0645	PDSPxox	-
4F	004F0E25	PDSnoan	-
50	00500325	PDna	-
51	00510B26	DSPnaon	-
52	005206C9	DPSDaox	-
53	00530764	SPDSxaxn	-
54	005408A9	DPSonon	-
55	00550009	Dn	-
56	005601A9	DPSox	-
57	00570389	DPSoan	-
58	00580785	PDSPoax	-
59	00590609	DPSnox	-
5A	005A0049	DPx	PATINVERT
5B	005B18A9	DPSDonox	-
5C	005C0649	DPSDxox	-
5D	005D0E29	DPSnoan	-

Boolean function in HEX	HEX Rop	Boolean function in R Polish	Common name
5E	005E1B29	DPSDnaox	-
5F	005F00E9	DPan	-
60	00600365	PDSxa	-
61	006116C6	DSPDSaoxxn	-
62	00620786	DSPDoax	-
63	00630608	SDPnox	-
64	00640788	SDPSoax	-
65	00650606	DSPnox	-
66	00660046	DSx	SRCINVERT
67	006718A8	SDPSonox	-
68	006858A6	DSPDSonoxxn	-
69	00690145	PDSxxn	-
6A	006A01E9	DPSax	-
6B	006B178A	PSDPSoaxxn	-
6C	006C01E8	SDPax	-
6D	006D1785	PDSPDoaxx	-
6E	006E1E28	SDPSnoax	-
6F	006F0C65	PDSxnan	-
70	00700CC5	PDSana	-
71	00711D5C	SSDxPDxaxn	-
72	00720648	SDPSxox	-
73	00730E28	SDPnoan	-
74	00740646	DSPDxox	-
75	00750E26	DSPnoan	-
76	00761B28	SDPSnaox	-
77	007700E6	DSan	-
78	007801E5	PDSax	-
79	00791786	DSPDSoaxxn	-
7A	007A1E29	DPSDnoax	-
7B	007B0C68	SDPxnan	-
7C	007C1E24	SPDSnoax	-
7D	007D0C69	DPSxnan	-
7E	007E0955	SPxDSxo	-
7F	007F03C9	DPSaan	-

Boolean function in HEX	HEX Rop	Boolean function in R Polish	Common name
80	008003E9	DPSaa	-
81	00810975	SPxDSxon	-
82	00820C49	DPSxna	-
83	00831E04	SPDSnoaxn	-
84	00840C48	SDPxna	-
85	00851E05	PDSPnoaxn	-
86	008617A6	DSPDSoaxx	-
87	008701C5	PDSaxn	-
88	008800C6	DSa	SRCAND
89	00891B08	SDPSnaoxn	-
8A	008A0E06	DSPnoa	-
8B	008B0666	DSPDxoxn	-
8C	008C0E08	SDPnoa	-
8D	008D0668	SDPSxoxn	-
8E	008E1D7C	SSDxPDxax	-
8F	008F0CE5	PDSanan	-
90	00900C45	PDSxna	-
91	00911E08	SDPSnoaxn	-
92	009217A9	DPSDPoaxx	-
93	009301C4	SPDaxn	-
94	009417AA	PSDPSoaxx	-
95	009501C9	DPSaxn	-
96	00960169	DPSxx	-
97	0097588A	PSDPSonoxx	-
98	00981888	SDPSonoxn	-
99	00990066	DSxn	-
9A	009A0709	DPSnax	-
9B	009B07A8	SDPSoaxn	-
9C	009C0704	SPDnax	-
9D	009D07A6	DSPDoaxn	-
9E	009E16E6	DSPDSaoxx	-
9F	009F0345	PDSxan	-
A0	00A000C9	DPa	-
A1	00A11B05	PDSPnaoxn	-

Boolean function in HEX	HEX Rop	Boolean function in R Polish	Common name
A2	00A20E09	DPSnoa	-
A3	00A30669	DPSDxoxn	-
A4	00A41885	PDSPonoxn	-
A5	00A50065	PDxn	-
A6	00A60706	DSPnax	-
A7	00A707A5	PDSPoaxn	-
A8	00A803A9	DPSoa	-
A9	00A90189	DPSoxn	-
AA	00AA0029	D	-
AB	00AB0889	DPSono	-
AC	00AC0744	SPDSxax	-
AD	00AD06E9	DPSDaoxn	-
AE	00AE0B06	DSPnao	-
AF	00AF0229	DPno	-
B0	00B00E05	PDSnoa	-
B1	00B10665	PDSPxoxn	-
B2	00B21974	SSPxDSxox	-
B3	00B30CE8	SDPanan	-
B4	00B4070A	PSDnax	-
B5	00B507A9	DPSDoaxn	-
B6	00B616E9	DPSDPaoxx	-
B7	00B70348	SDPxan	-
B8	00B8074A	PSDPxax	-
B9	00B906E6	DSPDaoxn	-
BA	00BA0B09	DPSnao	-
BB	00BB0226	DSno	MERGEPAINT
BC	00BC1CE4	SPDSanax	-
BD	00BD0D7D	SDxPDxan	-
BE	00BE0269	DPSxo	-
BF	00BF08C9	DPSano	-
C0	00C000CA	PSa	MERGECOPY
C1	00C11B04	SPDSnaoxn	-
C2	00C21884	SPDSonoxn	-
C3	00C3006A	PSxn	-

Boolean function in HEX	HEX Rop	Boolean function in R Polish	Common name
C4	00C40E04	SPDnoa	-
C5	00C50664	SPDSxoxn	-
C6	00C60708	SDPnax	-
C7	00C707AA	PSDPoaxn	-
C8	00C803A8	SDPoa	-
C9	00C90184	SPDoxn	-
CA	00CA0749	DPSDxax	-
CB	00CB06E4	SPDSaoxn	-
CC	00CC0020	S	SRCCOPY
CD	00CD0888	SDPono	-
CE	00CE0B08	SDPnao	-
CF	00CF0224	SPno	-
D0	00D00E0A	PSDnoa	-
D1	00D1066A	PSDPxoxn	-
D2	00D20705	PDSnax	-
D3	00D307A4	SPDSoaxn	-
D4	00D41D78	SSPxPDxax	-
D5	00D50CE9	DPSanan	-
D6	00D616EA	PSDPSaoxx	-
D7	00D70349	DPSxan	-
D8	00D80745	PDSPxax	-
D9	00D906E8	SDPSaoxn	-
DA	00DA1CE9	DPSDanax	-
DB	00DB0D75	SPxDSxan	-
DC	00DC0B04	SPDnao	-
DD	00DD0228	SDno	-
DE	00DE0268	SDPxo	-
DF	00DF08C8	SDPano	-
E0	00E003A5	PDSoa	-
E1	00E10185	PDSoxn	-
E2	00E20746	DSPDxax	-
E3	00E306EA	PSDPaoxn	-
E4	00E40748	SDPSxax	-
E5	00E506E5	PDSPaoxn	-

Boolean function in HEX	HEX Rop	Boolean function in R Polish	Common name
E6	00E61CE8	SDPSanax	-
E7	00E70D79	SPxPDxan	-
E8	00E81D74	SSPxDSxax	-
E9	00E95CE6	DSPDSanaxxn	-
EA	00EA02E9	DPSao	-
EB	00EB0849	DPSxno	-
EC	00EC02E8	SDPao	-
ED	00ED0848	SDPxno	-
EE	00EE0086	DSo	SRCPAINT
EF	00EF0A08	SDPnoo	-
F0	00F00021	P	PATCOPY
F1	00F10885	PDSono	-
F2	00F20B05	PDSnao	-
F3	00F3022A	PSno	-
F4	00F40B0A	PSDnao	-
F5	00F50225	PDno	-
F6	00F60265	PDSxo	-
F7	00F708C5	PDSano	-
F8	00F802E5	PDSao	-
F9	00F90845	PDSxno	-
FA	00FA0089	DPo	-
FB	00FB0A09	DPSnoo	PATPAINT
FC	00FC008A	PSo	-
FD	00FD0A0A	PSDnoo	-
FE	00FE02A9	DPSoo	-
FF	00FF0062	1	WHITENESS

Chapter 15 Miscellaneous Character Set Tables

This chapter defines the major character sets used with Microsoft Windows 3.0 and provides figures with samples of them.

One of the attributes of a Windows font is its *character set.* A character set is a mapping of byte values to graphic symbols. For example, the ASCII character set maps the number 65 to the letter "A."

The following are the two most common character sets for Windows fonts:

- ANSI, the standard Windows font
- OEM, which refers to the native character set of the computer on which Windows is running. For example, on IBM PC computers used in the United States, the OEM character set is the IBM PC character set.

Samples of these character set tables are provided in Figures 15.1 and 15.2.

A printer driver should attempt to support ANSI whenever possible. If necessary, the printer should implement a translation table to convert ANSI codes to the printer's (or printer font's) native character set. For example, the PCL/HP LaserJet driver supplied with Windows will convert ANSI into US ASCII, HP Roman 8, or ECMA 94, depending on the character set of the selected printer font. This requires that the printer font contain symbols similar to those used in ANSI.

The Microsoft Windows Graphics Device Interface (GDI) considers a font's character set as the most important attribute when selecting a font from those available. The character set has a high weight to ensure that the output is at least meaningful if not beautiful. To prevent unexpected results, drivers that implement their own font mappers in the **RealizeObject()** function should use criteria similar to GDI's when selecting fonts.

The Windows Symbol (CHARSET_SYMBOL) character set is new for Windows 3.0 and is shown in Figure 15.3. Notice that these are not fonts or type faces. The characters for this character set may be designed in a variety of type faces.

Other character sets are occasionally used for special purposes in specific applications or specialized drivers. If an application encounters a character set that it does not recognize when enumerating fonts, it should remember the font and character set index to allow the user to select the font and print it, unless it relies on the assignment of characters to byte values. The application should make no assumptions about character assignments.

0	■	32		64	@	96	`	128	■	160		192	À	224	à
1	■	33	!	65	A	97	a	129	■	161	¡	193	Á	225	á
2	■	34	"	66	B	98	b	130	■	162	¢	194	Â	226	â
3	■	35	#	67	C	99	c	131	■	163	£	195	Ã	227	ã
4	■	36	$	68	D	100	d	132	■	164	¤	196	Ä	228	ä
5	■	37	%	69	E	101	e	133	■	165	¥	197	Å	229	å
6	■	38	&	70	F	102	f	134	■	166	¦	198	Æ	230	æ
7	■	39	'	71	G	103	g	135	■	167	§	199	Ç	231	ç
8	■	40	(	72	H	104	h	136	■	168	¨	200	È	232	è
9	■	41	)	73	I	105	i	137	■	169	©	201	É	233	é
10	■	42	*	74	J	106	j	138	■	170	ª	202	Ê	234	ê
11	■	43	+	75	K	107	k	139	■	171	«	203	Ë	235	ë
12	■	44	,	76	L	108	l	140	■	172	¬	204	Ì	236	ì
13	■	45	-	77	M	109	m	141	■	173	-	205	Í	237	í
14	■	46	.	78	N	110	n	142	■	174	®	206	Î	238	î
15	■	47	/	79	O	111	o	143	■	175	¯	207	Ï	239	ï
16	■	48	0	80	P	112	p	144	■	176	°	208	Ð	240	ð
17	■	49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
18	■	50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
19	■	51	3	83	S	115	s	147	■	179	³	211	Ó	243	ó
20	■	52	4	84	T	116	t	148	■	180	´	212	Ô	244	ô
21	■	53	5	85	U	117	u	149	■	181	µ	213	Õ	245	õ
22	■	54	6	86	V	118	v	150	■	182	¶	214	Ö	246	ö
23	■	55	7	87	W	119	w	151	■	183	·	215	×	247	÷
24	■	56	8	88	X	120	x	152	■	184	¸	216	Ø	248	ø
25	■	57	9	89	Y	121	y	153	■	185	¹	217	Ù	249	ù
26	■	58	:	90	Z	122	z	154	■	186	º	218	Ú	250	ú
27	■	59	;	91	[	123	{	155	■	187	»	219	Û	251	û
28	■	60	<	92	\	124	|	156	■	188	¼	220	Ü	252	ü
29	■	61	=	93	]	125	}	157	■	189	½	221	Ý	253	ý
30	■	62	>	94	^	126	~	158	■	190	¾	222	Þ	254	þ
31	■	63	?	95	_	127	■	159	■	191	¿	223	ß	255	ÿ

■ Indicates that this character is not supported by Windows.

Figure 15.1 The ANSI Table

128	Ç	144	É	160	á	176	■	192	■	208	■	224	■	240	■
129	ü	145	æ	161	í	177	■	193	■	209	■	225	ß	241	±
130	é	146	Æ	162	ó	178	■	194	■	210	■	226	■	242	■
131	â	147	ô	163	ú	179	■	195	■	211	■	227	¶	243	■
132	ä	148	ö	164	ñ	180	■	196	■	212	■	228	■	244	■
133	à	149	ò	165	Ñ	181	■	197	■	213	■	229	■	245	■
134	å	150	û	166	ª	182	■	198	■	214	■	230	µ	246	■
135	ç	151	ù	167	º	183	■	199	■	215	■	231	■	247	■
136	ê	152	ÿ	168	¿	184	■	200	■	216	■	232	■	248	°
137	ë	153	Ö	169	—	185	■	201	■	217	■	233	■	249	■
138	è	154	Ü	170	¬	186	■	202	■	218	■	234	■	250	■
139	ï	155	¢	171	½	187	■	203	■	219	■	235	■	251	■
140	î	156	£	172	¼	188	■	204	■	220	■	236	■	252	■
141	ì	157	¥	173	¡	189	■	205	■	221	■	237	■	253	²
142	Ä	158	■	174	«	190	■	206	■	222	■	238	■	254	··
143	Å	159	■	175	»	191	■	207	■	223	■	239	■	255	■

■ Indicates that this character is not supported by Windows.

Figure 15.2 The IBM PC Extended Character Set

Index