Microsoft® Windows

Adaptation Guide

Microsoft Corporation

Document No. xxx-xxx-xxx
Part No. xxx-xxx-xxx

Contents

Chapter 1
Introduction

1.1 Overview

This guide explains how to adapt Microsoft® Windows to computer systems. In particular, it describes how to build the Microsoft Windows virtual machine and associated device drivers.

In order for Microsoft Windows to communicate with the hardware of a given computer system, a set of machine-dependent support modules must be present. This set is called the Microsoft Windows Virtual Machine. It is created by the OEM. The virtual machine is the foundation on which Microsoft Windows builds its extensions to the computer's capabilities.

A device driver is the support module that forms the interface between Windows and a particular piece of peripheral hardware (*e.g.*, a printer or plotter.) Windows can only communicate with such a device through a driver, so a driver must be written for each peripheral in the system if Windows is to use it. A device driver need be loaded only when its corresponding device is to be used. Display devices can only be added during **Setup**.

Printer and serial communications drivers can be added either during **Setup** or by use of the CONTROL.EXE program, which runs under Windows.

1.2 Adapting Windows

You can adapt Windows to your computer by combining the machine-independent Windows modules provided by Microsoft with the virtual machine you develop for your computer's hardware. You can enhance this system by developing additional device drivers for other devices.

To adapt Windows to your computer, you must:

- Make sure your computer meets the minimal hardware requirements
- Write a virtual machine that meets the functional specifications given in this guide
- Install Windows on your computer by using the **Setup** program

To create device drivers for a computer system, you must create a support module that meets the functional specifications for output devices, given in this guide.

1.2.1 Hardware Requirements

To adapt Windows to your computer, you must have:

- At least 640K of RAM.
- A graphics display device.
- A keyboard.

A mouse is not absolutely required, but is strongly recommended.

You can also adapt Microsoft Windows to use the following optional hardware devices:

- A timer.
- A serial port (required if a serial mouse is used).
- An interrupt controller.

Additional hardware devices, such as parallel ports or plug-in cards, are typically controlled through loadable device drivers and not by Windows directly.

1.2.2 Windows Modules

Windows has several machine-independent modules that take control of your computer's resources and maintain the user interface for application programs. These modules are developed by Microsoft and are ready for use with your computer. No modification is necessary.

The following modules form the heart of Windows:

kernel.exe	This module controls and allocates all machine resources for use with Windows. It works with your computer's operating system to manage memory, load applications, and schedule execution of programs and other tasks.
user.exe	This module 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.
gdi.exe	This module is the Graphics Device Interface (GDI). It generates the graphics operations needed to create images on the system display

and other display devices.

Microsoft also provides several Windows application modules. These applications include an operating system shell that gives a user access to the computer's disk system.

1.2.3 Support Modules

A complete Windows system has the following OEM-dependent modules:

Module	Description
system.drv	Supports system timer, information about system disks, and access to OEM-defined system hooks.
keyboard.drv	Supports keyboard input.
mouse.drv	Supports mouse or other pointing device input.
display.drv	Supports system display and pointing device cursor.
fonts.fon	Contains system font resources.
comm.drv	Supports serial and parallel device communications.
sound.drv	Supports sound generation and system speaker.
winoldap.mod	Supports loading and execution of standard applications.
winoldap.grb	Supports data exchange between standard applications and Windows.

Note

Several of these files are renamed at **Setup** time. DO NOT name your display driver *display.drv*, as this name is reserved. Instead, you should use some unique descriptive name, ending in *.drv*. (The high resolution EGA display driver provided with Windows is called *egahires.drv*.) Other files that are renamed are *keyboard.drv*, *mouse.drv*, *fonts.fon*, *comm.drv*, and *sound.drv*.

1.2.4 The Virtual Machine

Windows never communicates with a computer's hardware resources directly. It always uses the virtual machine to carry out any machine-dependent actions. This makes Windows machine-independent and much easier to develop for new computers and peripherals.

The interface between Microsoft Windows and the virtual machine is identical on each computer. The interface between the virtual machine and a computer's hardware is entirely up to the OEM. Each virtual machine is intended to be hand-tailored by the OEM to be the most effective and efficient for his computer.

The OEM develops all routines that require a detailed understanding of the hardware, since these are the most crucial to overall performance of the computer. Microsoft Windows provides the routines that either do not affect performance, or are involved strictly in maintaining the interface between Windows and application programs.

1.2.5 Device Drivers

Windows communicates with all output devices through a Graphics Device Interface (GDI) support module. GDI passes device-independent function calls to device drivers that have been developed for individual devices. The device driver translates the calls into device-dependent operations that draw images on the device's display surface.

GDI provides access to any output device that has an appropriate device driver. GDI recognizes two classes of output devices:

- Raster/frame-buffer type
- Vector/plotting type

These output device classes include system displays, printers, plotters, and even metafiles. Windows requires one device driver for each kind of output device in the system.

GDI is a rich graphics language that can carry out or simulate the full range of graphics operations on any device. GDI is designed to drive devices with a wide range of capabilities. It provides an efficient interface to display devices that have built-in graphics functions. It also provides a wide range of graphics operations on devices, such as dedicated displays on personal computers, that have few built-in graphics functions.

GDI simulates complex functions if they are not built into the device. A simulation is a sequence of less complicated operations, drawn from those that the device has available, that add up to the more complex function being simulated. For example, if a device has code that can draw circles,

rectangles, or other complex shapes, GDI passes to the device via the device driver the information required to draw these images. If the device does not have this code, GDI determines what operations are required based on what the device is capable of. It then passes these operations to the device via the device driver. GDI never requires the device driver to compute and carry out the complex operations such as drawing circles, rectangles, and other complex shapes, but it does permit it to do so.

Note

If your output device supports character rotations directly, GDI can instruct it to display or print rotated characters, but GDI does not otherwise simulate character rotations.

GDI requires the minimum set of functions for each raster and vector device driver. Ideally, you can determine the size and complexity of your device driver based on the capabilities of the hardware and what you want to be able to achieve with it. Chapter 5 describes device drivers in more detail.

1.3 Creating Support Modules and Device Drivers

Support modules and device drivers can be written in assembly language or in a Microsoft high-level language. Windows requires specific segment name and calling conventions that all Microsoft high-level languages provide. These conventions are available to assembly language writers who use the **Cmacros** macro package (see the *Microsoft Windows Programmer's Reference* for details). Special options may be required when invoking a high level language compiler.

Each support module and device driver must be a complete executable file that **Setup** can load and initialize. Each must define one or more routines and make these routines available to Windows and other modules by exporting them. The number and function of the routines depends on the support module. If a support module makes references to routines in other modules, it must import them. External references are resolved when the module is loaded.

1.3.1 Naming Conventions

Windows expects explicit names for the groups, segments, and classes defined by a program. It also has explicit rules for the spelling of routine names.

Table 1.1

Segment and Class Name Conventions

Segment Name	Class Name	Definition
_DATA	DATA	This segment contains all the static and global data used by the support module. The segment must be public and word aligned.
_TEXT	CODE	This segment contains the module's executable code. The segment must be public and byte aligned.

Table 1.2

Group Name Conventions

Group Names	Definition
DGROUP	This group contains all segments having DATA class. This includes a module's _DATA segment.
IGROUP	This group contains all segments having CODE class. This includes a module's _TEXT segment.

Support module routines should have the names defined in this guide. The names should be spelled as given except that upper- and lowercase letters can be used interchangeably.

1.3.2 Calling Conventions

Whenever Windows calls a support module routine, it uses a special calling convention. This convention is defined as follows:

- The **CS** register points to the called module's code segment, which must not be larger than 64K. Drivers cannot depend on the code segment to remain in any fixed position in physical memory.

- When the device driver module is first called, the **DS** register points to the caller's data segment. The standard Windows prolog (which is required) sets the **DS** register to point to the called module's DGROUP, which must not be larger than 64K. The data segment, once allocated, is not moved, and can be depended on to remain in place. A device driver in one module can save its data segment in full confidence that it will not be lost or modified.
- The **SS** register points to a stack segment, which may be different from the data segment.
- The module called must save and restore any of the following registers it uses: **DS**, **SS**, **SP**, **BP**, **SI**, **DI**.
- Values returned by a module call must be placed in **AX** if they are 16-bit, in **DX:AX** if they are 32-bit.
- All exported entries into a module are reached by far calls; each such entry must execute a far return.
- 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 routine is:

```
OEMFUNC         PROC    far
                mov     ax, ds          ;Windows support prologue
                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 epilogue 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.

Chapter 2
Input Support Modules

2.1 Overview

This chapter describes the functions performed by the input support modules for the Windows virtual machine. These include:

- Keyboard Handler
- System Support Module
- Mouse Support Module

2.2 Keyboard Handler

The Microsoft Windows keyboard handler preempts the normal keyboard hardware interrupt routine with one of its own. This flexibility in keyboard translation allows applications to be written which expect either the ASCII key codes returned by the OEM's usual keyboard routine **or** the virtual key codes used by Microsoft Windows. Applications that refer to virtual key codes will get correct results regardless of actual keyboard layout. Programs that refer to unstandardized virtual keys will be less portable, but may take advantage of special OEM keyboard arrangements.

In addition to handling the translation of key codes and the hardware interrupt-driven keyboard, the keyboard handler also includes entries for handling keyboard lights (keystate indicators) and sound. Each of these functions is described in more detail below. The exact functions are described in Chapter 3, "Input Functions".

2.2.1 KEYBOARD.DRV

This support module contains several entry points, including initialization code for the keyboard device and the hardware interrupt routine that handles keyboard interrupts.

The interrupt handler must translate the hardware scan code into a Microsoft Windows virtual key code and place the virtual key code in **AX** (with the sign bit set for an upgoing key transition, and reset for a downgoing key transition). It then saves **AX** and **BX**, and calls an event-handling procedure provided by Microsoft Windows. This event-handling procedure saves and restores all the other registers that it uses, and makes the keystroke information available to Microsoft Windows.

2.2.2 TOASCII

This entry is to the *ToAscii* routine which examines the current keyboard state and returns either the 8-bit extended ASCII code indicated by that state or an OEM-dependent translation of the state.

2.3 SYSTEM Support Module

The system support module provides information about timer resolution (in microseconds), system disk drives, and system printer ports. It also provides initialization and termination functions for the timer, and a method for dynamically installing an interrupt handler for the timer. Three calls have been added in version 2.0 for support of coprocessors.

1. This call returns the size (in bytes) of the area required to save the state of the coprocessor. If there is no coprocessor in the system then the size returned should be 0.
2. This call saves the state of the coprocessor in the area pointed to by lpSaveArea.
3. This call restores the state of the coprocessor from the area pointer to by lpSaveArea.

These calls are described in Chapter 3.

2.4 MOUSE Support Module

The mouse support module provides:

- Initialization and termination functions for the mouse
- Information about whether 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 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 these data are in place, the interrupt handler calls an event-handling procedure supplied by Microsoft Windows.

2.5 Communication Support Module

This support module has entries which assign and deassign serial device instances to ports, enable and disable interrupt handlers for input from assigned serial devices, and send characters to assigned serial output devices. The **$INICOM** entry takes parameters establishing the data transmission protocol to be used, and can attach the serial device to any output port.

Chapter 3
Input Functions

3.1 Introduction

This chapter describes the function performed by each entry in the input support modules required by Microsoft Windows.

3.2 Keyboard Entries

Inquire (*pKBINFO*)

Purpose This routine fills in the data structure with information about the hardware keyboard.

Parameters *pKBINFO* is a long pointer pointing to a **KBINFO** data structure. The routine must fill this data structure with information about the keyboard.

Return On return, **AX** holds the number of bytes actually written by the support module.

Enable (*pEventProc*)

Purpose Initializes the keyboard for use by Microsoft Windows. This routine should save the current hardware state in static storage so that it can be restored when the *Disable* routine is called.

Parameters *pEventProc* is a long pointer to a procedure which is called for each subsequent hardware interrupt.

Return On return, **AX** holds zero if unable to initialize the keyboard, –1 if the input devices were successfully initialized.

Notes This routine will be called once during startup of Microsoft Windows.

Disable ( )

Purpose Restores the keyboard to the state it was in prior to the start-up of Microsoft Windows.

Parameters None.

Return None.

Notes This routine will be called once during shutdown of Microsoft Windows.

ToAscii (*virtKey, scanCode, lpKeyState, lpState, Flags*)

Purpose Provides the default mapping from virtual key codes to extended 8-bit ASCII codes or to OEM-specific translations. This routine is called whenever Microsoft Windows wants the default translation for a virtual key code. For the IBM PC, this routine duplicates the functionality provided by the keyboard hardware interrupt routine in the IBM BIOS.

Parameters *virtKey* is the virtual keycode of the key being translated.

scanCode is the OEM-dependent scan code of the key to be translated.

lpKeyState is a long pointer to an array of bytes indexed by virtual key codes. If the high-order bit of a byte is set, that key is currently down; otherwise the key is up. If the low-order bit of a byte is set, that key is currently in a toggled state. The up/down bit for a key is set whenever the key goes down and is reset whenever the key goes up. The toggle bit for a key is inverted for every down transition of a key.

lpState is a long pointer to a state block, whose size is determined by the *kbStateSize* field of the *kbInfo* data structure. The state block is guaranteed to contain zeros the first time this routine is called.

Flags specifies whether or not a menu is currently being displayed. The low-order bit is 1 if a menu is displayed. Otherwise, the bit is 0. This argument is used whenever a menu affects the meaning of one or more keyboard keys.

Return The *ToAscii* routine returns the count of characters (stored as words) returned in the state block at *lpState*. *ToAscii* returns 0 if the *virtKey* has no translation. If *ToAscii* returns a negative number, the characters in the state block are assumed to be dead characters, and the number of words returned is the negated count.

ScreenSwitchEnable (*wEnable*)

Purpose Enables or disables screen switching under MS-DOS. Not usable with OS/2 (See section 4.6, "Dedicated Display Support Module" for more information.)

Parameters *wEnable* is zero for disable, non-zero for enable.

Notes This routine is called before and after crucial/uninterruptible functions (e.g., save and restore of EGA state). It prevents the operating system from switching the screen away from Windows, which requires a save-state in any event. If a save-state is underway, it cannot be started, so this function makes the operating system wait until it is safe to switch the screen.

3.3 System Functions

InquireSystem

InquireSystem (*Command, Device*)

Purpose This primitive returns information about the system's hardware.

Parameters *Command* is an integer value specifying the type of system information requested. It can be any one of the following:

0	Request for timer information.
1	Request for disk drive information.
2	Request for printer port information.

Device is an integer value specifying which device of the given type is to be examined.

Return The return value depends on *Command*:

Command =0. On return, **DX:AX** holds the timer resolution in milliseconds. Timer resolution is assumed to be a long integer. Device = 0.

Command =1. On return, **AX** holds one of four values:

AX is 0	The drive specified by *Device* either does not exist, or if **DX** is not 0, the drive is mapped to the drive given by **DX**.
AX is 1	The specified drive either does not exist, or if **DX** is not 0, the drive is mapped to the fixed disk given by **DX**.
AX is 2	The specified drive is removable (e.g., a floppy drive). **DX** must hold 0.
AX is 3	The specified drive is fixed (e.g., a hard disk). **DX** must hold 0.

Command =2. No return value. Reserved for future use.

CreateTimer

CreateTimer (*Period, lpProc*) : *hTimer*

Purpose This routine creates a timer with the specified period, in milliseconds. If *Period* is zero, then the timer will go off at the maximum rate allowed by the hardware.

Parameters *Period* is an integer value specifying the timer interval.

lpProc is a long pointer to the procedure to call whenever the timer goes off.

Return Value *hTimer* is a handle to the timer. **CreateTimer** returns NULL if it cannot create a timer.

Notes The *lpProc* procedure must obey the following conventions:

- It must preserve the state of the interrupt flag. It can only turn it off, and must restore the previous setting before returning.
- It must save/restore the **BP**, **DS**, **SI**, and **DI** registers.
- It must switch to a separate stack if more than 64 bytes of stack are to be used.

On entry, the **AX** register is set to the handle to the timer, and the **BP** register to the address of far frame that contains the return address of the timer interrupt.

Currently, **CreateTimer** can only create a fixed number of timers (8).

KillTimer

KillTimer (*hTimer*) : *wReturnVal*

Purpose This routine removes the passed timer from the timer table created by **CreateTimer**.

Parameters *hTimer* is a handle to the timer to be killed.

Return Value is NULL if the routine is successful. Otherwise, it is equal to *hTimer*.

EnableAllTimers

EnableAllTimers ( )

Purpose This routine enables the hardware interrupt hook that calls the original hardware timer interrupt routine and loops over the timer table created by **CreateTimer**. It updates the counts associated with each timer and calls those procedures whose timers have gone off. The hardware interrupt hook maintains a critical section flag so that if the interrupt hook is reentered by scanning the timer table, a second scan will NOT occur.

Parameters None.

Return None.

DisableAllTimers

DisableAllTimers ( )

Purpose This routine disables the hardware interrupt hook installed by **EnableAllTimers**. It does NOT destroy the contents of the timer table, and a subsequent **EnableAllTimers** will reenable the same timers.

Parameters None.

Return None.

GetSystemMsecCount

GetSystemMsecCount ( ) : *lCount*

Purpose Returns the number of milliseconds since Windows was booted

Return Value *lCount* is a long count of milliseconds

Get80X87SaveSize

Get80X87SaveSize ( ) : *Size*

Purpose To determine the size of the space required to save the state of the coprocessor.

Return Value *Size* is the size, in bytes, of the required space. **Get80X87SaveSize** returns NULL if no coprocessor is present.

Notes The save size for the 8087 and 80287 is 94 bytes.

Save80X87State

Save80X87State (*lpSaveArea*)

Purpose To save the state of the coprocessor

Parameters *lpSaveArea* is a long pointer to an area to save the state in

Return None.

Notes The size of the save area can be gotten by using **Get80X87SaveSize**, discussed above.

Restore80X87State

Restore80X87State (*lpSaveArea*)

Purpose To restore the state of the coprocessor

Parameters *lpSaveArea* is a long pointer to the area in which the state is stored.

Return None.

3.4 Mouse Functions

Inquire (*pMOUSEINFO*)

Purpose This primitive returns information about the mouse hardware.

Parameters *pMOUSEINFO* 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.

Return On return, **AX** holds the number of bytes actually written into the data structure.

Enable (*pEventProc*)

Purpose This primitive sets up the mouse to call the procedure whose address was passed for each mouse interrupt.

Parameters *pEventProc* is a long pointer to the procedure that is to be called for each mouse interrupt.

Disable ( )

Purpose This primitive removes the existing interrupt procedure from the mouse support module. After a call to this procedure, there will be no support for interrupts from the mouse module.

Addition to MOUSE.DEF

A call has been added to MOUSE.DEF for Windows 386 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 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, but you are not obliged to use it: you may replace it with your own.

Chapter 4
Output Support Modules

4.1 Introduction

This chapter describes the GDI support modules for devices and the way in which GDI support modules register their capabilities with GDI. GDI is designed to accommodate all device technologies and levels of performance as smoothly as possible: existing, relatively primitive, graphics devices as well as the much more capable devices that are becoming available.

Every system running Microsoft Windows must have a GDI support module for the dedicated system display. All other output graphics devices in a Microsoft Windows system, such as printers or plotters, must have their own GDI support modules.

While a complete GDI support module is reasonably complex, few GDI support modules will implement **all** GDI functions. The minimum set of functions required for each type of device is discussed below in section 4.5, "Minimal GDI Support Modules." Readers interested in getting a particular graphics output peripheral running under Microsoft Windows can use that section as a guide to reading the following section 4.2, "Overview," which describes all of GDI.

4.2 Overview

Microsoft Windows and its applications are coupled to graphics output devices through the GDI interface that allows all the output devices on a system to be uniformly driven from applications. Each graphics output device in a system (including the dedicated system display) has an associated GDI support module. The dedicated system display has additional routines that control the appearance and tracking of the cursor.

Each GDI support module translates the calls issued by GDI into actions that result in the appearance of an image on the output device to which the support module is attached. GDI support modules are responsible for performing device initialization and termination functions, translating logical attribute bundles into device-specific form, returning information about the state of the graphics peripheral and support module, and, optionally, for handling high-level geometrical primitives such as circles, rectangles, ellipses, and polygons.

The GDI support module for a graphics output device takes device-independent commands (with device coordinates) and arranges for those primitives to appear on the attached output device. Doing so requires deciding, for each combination of output primitive and attributes, whether to use an intrinsic device function, to build the intelligence into the support module, or to request that GDI interpret the primitive into a sequence of calls that achieve the desired result. For example, a GDI

support module may draw circles by calling the circle function in the device, by sending a sequence of lines to the device that produces the equivalent of the circle requested, or by requesting that GDI reduce circles to lines (in which case the support module never sees circles).

It is the responsibility of the support module writer to decide where actions will be performed, and to set up the simulation-request vectors so that GDI can decide how to handle output primitives for that device. GDI is sufficiently flexible for support modules to be written that are either very lean (leaving most of the work to the GDI), or very rich (capable of performing all of the complex GDI primitives). This flexibility simplifies the process of getting new output devices running under Microsoft Windows. You can get started by creating and testing a lean support module and then proceed by gradually adding capabilities to it. In that way, you can take advantage of special hardware that might make certain GDI functions run faster in the support module than in the GDI, and the support module can be built incrementally.

4.3 GDI Device Model

GDI is designed to drive all types of graphics peripherals. GDI's model of a device is based on both the technology of the graphics peripheral, and on the operations the peripheral can perform.

Each device driver is required to include a module that contains the GDIINFO data structure, which provides GDI with information about the technology and capabilities of the device. See section 10.2, "Information Data Structures" for further information.

Note that GDI drives devices, each of which is a combination of an output graphics peripheral and its associated support module. This section is devoted to the ways in which the graphics operations understood by one GDI device may differ from those understood by another. Section 4.5, "Minimal GDI Support Modules," discusses the minimum required set of capabilities for each GDI Device Class.

4.4 GDI Device Classes

Graphics output devices are generally broken into classes according to their underlying technology. Additionally, some of the "devices" in a system are not connected to view surfaces, but are "meta-devices" (such as a metafile or character-stream device). Meta-devices are included in the classification given here, because GDI will drive them as well. GDI device classes include:

1. Vector plotter
2. Raster display
3. Raster printer
4. Raster camera (COM - Computer Output on Microfilm devices)
5. Character-stream (NAPLPS)
6. Metafile (VDM, CORE-metafile)
7. Segment-storage (GKS/WISS)

Things would be very simple if the technology of a device fully specified the set of functions of which it is capable, but the instruction sets built into devices differ as much within a technology family as they do across families. That is, the support modules for a smart vector plotter and a high-speed raster display may have more capabilities in common than the support modules for a dumb raster display and a smart raster display, even though the latter two devices are based on the same technology.

Because the device technology fails to fully characterize the capabilities of a device, GDI includes a set of support module variables that specify to GDI exactly what functions can be performed at the support module level (or below). These support module variables allow the support module writer to customize the capabilities present in the support module to those which are supported by hardware, or which are naturally performed well by the support module, leaving GDI to do the rest of the work.

4.5 Minimal GDI Support Modules

Minimal support modules require line-drawing capabilities for both vector and raster devices. Modules for raster devices must support *bitblt* operations also. With this system, vector devices may even imitate raster devices if the support module provides the *Bitblt* primitive, and registers as "bitblt-capable." GDI makes no assumptions about how faithfully a device will imitate those primitives that are not native to the underlying technology. Image fidelity is the responsibility of the support module. This allows support modules to be written for vector-technology devices that produce pictures that come arbitrarily close to the appearance on a raster-technology device (though there is significant work involved in developing a support module with such capabilities).

Beyond a small set of functions common to all GDI support modules, minimal GDI support modules differ for different device classes. This is important because many functions can be omitted from a support module without compromising performance, but the minimal set beyond the "core" of functions differs for each device technology. This section discusses the minimal requirements for several functions common to all

GDI Support Modules:

Enable	Each support module must be able to initialize itself and the attached graphics peripheral, and to acquire any resources it requires.
Disable	Each support module must be able to terminate itself and the attached graphics peripheral, releasing any resources obtained while running. Note that support modules that send primitives to the peripheral asynchronously (relative to the actual stream of GDI commands), such as raster printers, may need to perform an *end_picture* action when a disable is received.
ColorInfo	Each support module must be able to return physical color information for use with pixels, pens, brushes, and text.
RealizeObject	RealizeObject converts an application-defined pen, brush, or font data structure to an OEM-defined format for use in drawing procedures.
Pixel(set)	Each support module must be able to set a given pixel to a given color on the given device or in a memory bitmap.
Output(line)	Each support module must be able to use a nominal pen to draw a line from a given startpoint to an endpoint. This operation, required on both devices and memory bitmaps, is used to draw borders. Printer drivers using the "brute-force" method do not need to implement this function.
Output(scanline)	Each module must be able to use a nominal pen or brush to draw scanlines on the given device, or in a memory bitmap. A scanline is a sequence of line segments on the same horizontal line. This operation is used to fill interiors.

4.6 Dedicated Display Support Module – DISPLAY.DRV

Note

For compatibility with MS-OS/2, display drivers must hook to *Int 2Fh* on enable; this is the multiplexed interrupt channel, and is used here for screen switching. A more detailed description is furnished at the end of this section.

Dedicated displays are assumed to be raster devices. Thus, in addition to the basic GDI functions described above, the minimal support module for the system display includes:

Bitblt	Each support module must be able to carry out bit manipulation using a *Bitblt* routine.
Strblt	Each dedicated display module must be able to display text using at the very least a *bitblt* from the monochrome font bitmap onto the screen at the specified location.
ScanLR	Dedicated display drivers need to be able to scan for colors within a single scanline. This scanning is used by several Windows applications (*e.g.*, Paint) for flood filling. The **ScanLR** routine performs this action, and is required.
Inquire	The dedicated display support module must be able to return to GDI the mickey-to-pixel ratios for its particular screen.
SetCursor	Each support module must be able to set the shape of the cursor.
MoveCursor	Each support module must be able to move the cursor at mouse interrupt time.
FastBorder	Each support module must be able to draw a bordered box.
SaveScreenBitmap	This function is optional, as stated below. You must, however, at least put a stub for it into your driver module. The stub is to return **AX** = 0.

Screen Switching

During display enable (late in initialization of Windows), the display driver must acquire the module handle for the window manager, query the Window Manager for the address of entry 275 in **USER.EXE**, the use of which is described below, and hook *int 2Fh*. (Likewise, on disable, the driver must unhook the interrupt channel.) This multiplexed interrupt channel carries, among other things, screen switching information.

If an *int 2Fh* occurs with **AH** = 40h, the display driver must, depending on the contents of **AL**, either:

- disable its cursor drawing code, save its state, clear the carry flag, and IRET

  or

- restore its state, enable its cursor drawing code, call entry 275 (which has no parameters) thus alerting the Window Manager to the need for a redraw, and IRET.

There are some conditions under which the display driver must not be interrupted. These are handled slightly differently in DOS 3.XX and OS/2.

Under Windows, if an interrupt with **AH** = 40h and **AL** = 01h is received at a time when the driver must not be interrupted, the *int 2Fh* handling code sets the carry flag. This tells Windows to wait.

Under OS/2, that approach is not viable. Instead, *before* the driver executes a save- or restore- state or any other procedure that must not be interrupted by a screen switch, it must call entry 100 of the keyboard driver, ScreenSwitchEnable, to disable screen switching. When the save, restore, or other item is finished, the driver must call entry 100 again, to re-enable screen switching.

Clearly, the driver must know which environment it is in. It should perform a DOSGetVer during initialization to acquire this information.

Note

In Windows 386, it is possible for the display driver output function to be called *after* the disable call to the OEM layer has been made. All routines that access the video adapter should check an internal flag to verify that the driver has access to the video adapter. The enable routine should set that flag, and the disable routine should clear it. All drawing routines, including the cursor drawing routine, must honor the flag.

4.7 Vector Plotter GDI Support Module

In addition to the basic GDI support functions, the minimal support module for a vector plotter may include:

Strblt	The *Strblt* routine enables the plotter to make use of any device-specific fonts.

Chapter 5
Output Functions

5.1 Introduction

This chapter describes the output functions used by Microsoft Windows. The functions performed by a GDI support module fall into the following groups:

Control:	Initialize and terminate devices.
Information:	Pass information about the graphics peripheral to which this support module 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 support module. Pass information about fonts or other data structures.
Output:	Perform actual output onto the device output surface, or to a bitmap in memory.
Attribute:	Handle creation of physical representations of attribute bundles suitable for the device. Those physical representations are the actual parameters to output primitive calls.
Cursor:	Provide position and visibility control of the cursor, and the ability to specify the bitmap to be displayed as the cursor. **Used for dedicated display modules only.**

5.2 Control Functions

The following are the control routines that Microsoft Windows uses to initialize and disable the physical display and to control various output operations.

Enable Function

Enable (*lpDestDev, Style, lpDestDevType, lpOutputFile, lpData*) : *wSize*

Purpose This routine initializes a support module or returns information about the module as defined by the value of *Style*.

This routine 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.

Parameters

lpDestDev is 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 is 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 is 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 is a long pointer to a null-terminated ASCII string giving the MS-DOS filename of the physical device. For example, "COM1" for a plotter, "FILE.MET" for a metafile, or a null string for the dedicated system display. This string will be the same string passed into the **CreateDC** routine as the desired physical device. It can be NULL.

lpData is 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.

Result

On return, **AX** holds zero if it is unsuccessful (e.g., hardware is not initialized). Otherwise, **AX** returns nonzero.

Notes

This routine will only be called once for most physical devices. It can be called more than once for a display, (*e.g.*, if an old application is run; this requires that the display be **Disable**d, and, when the application is terminated, re-**Enable**d.)

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 interaction between a dedicated display driver and the keyboard driver, every dedicated display driver should use the following procedure to assure that the keyboard works correctly under Windows:

The Enable function must save the value at address 40:49h, and put 06h into that location. This value is to be restored by the Disable function.

Disable Function

Disable (*lpDestDev*)

Purpose If the Windows session is ending or if an old application (e.g., Word) is being run, the display needs to be disabled. In the case of an old application, the display will be re-enabled later by a call to **Enable.**

This call disables the display for either purpose.

Parameter *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

Notes This routine does not return a value.

Note

This function must restore to address 40:49h the value that was saved by Enable.

Control Function

Control (*lpDestDev, Function, lpInData, lpOutData*) : *wReturnVal*

Parameters *lpDestDev* is a long pointer to the destination device bit-map.

Function The predefined subfunctions for Control are described in Appendix E.

lpInData is a long pointer to function-specific input data.

lpOutData is a long pointer to function-specific output data.

5.3 Environment Routines

The following routines are part of GDI and can be used by device drivers to access information about a device's operating environment.

DeviceMode Function

DeviceMode (*hWnd, hInstance, lpDestDevType, lpOutputfile*)

Purpose 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.

Parameters *hWnd* is a handle to the application's window.

hInstance is the instance handle of the application.

lpDestDevType is 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 is 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 None.

Notes **DeviceMode** should be included in and exported from any device driver that permits the user to change modes.

SetEnvironment Function

SetEnvironment (*lpDOSPort, lpBuffer, Size*) : *Bytes*

Purpose This routine copies the contents of the buffer specified by *lpBuffer* into the environment associated with the device attached to the system port specified by *lpDOSPort*. **SetEnvironment** overwrites any existing environment. If there is no environment for the given port, **SetEnvironment** creates it. If *Size* is 0, the existing environment is deleted and not replaced.

Parameters *lpDOSPort* is a long pointer to a null-terminated string specifying the name of the desired port.

lpBuffer is a long pointer to the buffer containing the new environment.

Size is 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 0 if there is an error. It is -1 if the environment is deleted.

GetEnvironment Function

GetEnvironment (*lpDOSPort, lpBuffer, MaxBuffer*) : *Bytes*

Purpose This routine copies the current environment associated with the device attached to the system port specified by *lpDOSPort* into the buffer specified by *lpBuffer*. 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 routine fails if there is no environment for the given port.

Parameters *lpDOSPort* is a long pointer to a null-terminated string specifying the name of the desired port.

lpBuffer is a long pointer to the buffer which receives the environment.

MaxBuffer is 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 0 if the environment cannot be found.

5.4 Output Functions

The output routines perform all the actual graphics operations on the display surface. There are the following routines:

Note

The output routines must not overwrite the display cursor while it is still on the display screen. Each routine 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 other items.

Bitblt Function

Bitblt (*lpDestDev, DstxOrg, DstyOrg, lpSrcDev, SrcxOrg, SrcyOrg, xext, yext, Rop, lpPBrush, lpDrawMode*)

Purpose

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 specification of all possible Boolean operations on three variables (source, destination, and the pattern in the brush). Note that the source and destination may overlap, so the implementation must be careful about the direction in which bits are copied.

Parameters

lpDestDev is a long pointer to a data structure of type **PDEVICE**.

DstxOrg and *DstyOrg* are two short integers specifying the coordinate origin of the destination rectangle on the destination device in device units.

lpSrcDev is a long pointer to a data structure of type **PDEVICE**.

SrcxOrg and *SrcyOrg* are two short integers specifying the coordinate origin of the source rectangle on the source device, in device units.

xext is a short integer, specifying the horizontal extent of the rectangle on both source and destination devices, in device units.

yext is a short integer, specifying the vertical extent of the rectangle on both source and destination devices, in device units.

Rop is a long integer specifying a raster operation code that defines the combining function to be used on the source, destination, and pattern information to produce the color to be placed at the destination for each pixel being rewritten. (See Appendix D, "Raster Operation Codes and Definitions.")

lpPBrush is 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 is a long pointer to **DRAWMODE** data structure. The color information in the structure is used to carry out color conversions in bitmaps.

Return None. .

Notes 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 routine 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

1. Convert white bits (1) to the background color.
2. Convert black bits (0) to the foreground color.

To convert a color bitmap to monochrome bitmap, **Bitblt** must:

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*.

Strblt Function

Strblt (*lpDestDev, DestxOrg, DestyOrg, lpClipRect, lpString, Count, lpFont, lpDrawMode, lpTextXform*)

Purpose This is an alternate entrypoint to **ExtTextOut**, provided for compatibility with Windows 1.XX.

Parameters Under Windows 2.0, **Strblt** should be implemented as a call to **ExtTextOut**, in the following manner:

```
cProc       Strblt,<FAR,PUBLIC>,<SI,DI>
     parmd  lpDstDev
     parmw  DstxOrg
```

```
        parmw   DstyOrg
        parmd   lpClipRect
        parmd   lpString
        parmw   Count
        parmd   lpFont
        parmd   lpDrawMode
        parmd   lpTextForm
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   lpDstDev
        parmw   DstxOrg
        parmw   DstyOrg
        parmd   lpClipRect
        parmd   lpString
        parmw   Count
        parmd   lpFont
        parmd   lpDrawMode
        parmd   lpTextForm
        parmd   lpCharWidths
        parmd   lpOpaqueRect
        parmw   Options
cBegin
```

Return None.

Notes None.

ExtTextOut Function

ExtTextOut (*lpDestDev, DestXOrg, DestYOrg, lpClipRect, lpString, Count, lpFont, lpDrawMode, lpTextXform, lpCharWidths, lpOpaqueRect, Options*) : *lSuccess*

Purpose 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 character foreground, and a zero bit specifies character background. It is effectively the Windows 2.0 **Strblt** function.

Parameters *lpDestDev* is a long pointer to the destination device bitmap.

DestXOrg is the left origin of the string.

DestYOrg is the top origin of the string.

lpClipRect is 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 10, "Data Structures and File Formats".

lpString is a long pointer to the string itself.

Count has one of two 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 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.)

lpFont is a long pointer to a data structure of type **FONTINFO**, which represents the physical font in use.

lpDrawMode is 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 10 for a description of the text justification and character spacing.

lpTextXform is a long pointer to a data structure of type **TEXTXFORM** that describes text appearance that may differ from the actual values specified by *lpFont*. 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 *lpFont*. 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 nth character to character n+1. This number may be larger or smaller than the actual width of the nth character.

lpOpaqueRect, if non-null, is a long pointer to the opaquing rectangle.

Options is 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 the DrawMode. The area is to be filled regardless of opaque/transparent mode.Note that the text string bounding box and the opaquing rectangle are allowed to be disjoint rectangles.

Return Under certain circumstances, (*e.g.*, if the specified font is not supported), **ExtTextOut** returns **DX:AX** = 8000:0000h to signify an error.

Else, it returns **DX:AX** = 0000:0000h to signify success.

Notes Refer to the **GDIINFO** data structure for a description of how **ExtTextOut** registers its output capabilities and their meanings.

Rectangles are always given in bitmap coordinates, with upper left corner followed by lower right as the preferred ordering.

The upper left corner of the string is placed starting at the point defined by *DestOrg*. 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, background mode, 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, please refer to Chapter 10, "Data Structures and File Formats".

FastBorder Function

FastBorder (*lpRect, HorizBorderThick, VertBorderThick, RasterOp, lpDestDev, lpPBrush, lpDrawMode, lpClipRect*) : *wSuccess*

Purpose

This function draws a rectangle with border on the screen, subject to the limits imposed by the specified clipping rectangle. The border is drawn within the boundaries of the specified rectangle.

Parameters

lpRect is a long pointer to the rectangle to be framed

HorizBorderThick is the width, in pixels, of the left and right borders

VertBorderThick is the width, in pixels, of the top and bottom borders

RasterOp is the raster operation to be used

lpDestDev is a long pointer to a data structure of type **PDEVICE**, *i.e.*, the device to receive the output.

lpPBrush is a long pointer to a data structure of type **PBRUSH**

lpDrawMode is 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 10 for a description of the text justification and character spacing.

lpClipRect is a long pointer to the clipping rectangle.

Return FastBorder returns **AX** = 0 on error, **AX** = 1 on success.

Notes The specified rectangle should be given as (UpperLeftCorner, LowerRightCorner). If it is specified incorrectly, the example routine will draw the borders outside of the specified rectangle, instead of correctly drawing them inside.

Output Function

Output (*lpDestDev, Style, Count, lpPoints, lpPPen, lpPBrush, lpDrawMode, lpClipRect*) : *wReturnVal*

Purpose The output primitive group includes all the line-drawing routines.

Parameters *lpDestDev* is a long pointer to the destination device context.

Style is 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 is a 16-bit integer specifying the number of points in the points list.

lpPoints is 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.

lpPPen is a long pointer to a data structure of type **PPEN**.

lpPBrush is a long pointer to a data structure of type **PBRUSH**.

lpDrawMode is a long pointer to a data structure of type **DRAWMODE** that includes information required to decide how to alter the pixels. It includes a specification of a mode in which to draw the line (a logical function combining source and destination), a background mode, a physical background color.

lpClipRect is a long pointer to the clipping rectangle to be used to clip output. For polygons and lines, *lpClipRect* contains the bounding box of all 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* itself is zero, then the clipping rectangle is the entire display surface.

Return

Output returns

- **AX**= 1: success
- **AX**= 0: unrecoverable failure
- **AX**= –1: device driver could not support the passed style

The –1 return represents new support for "smart" devices in the Windows 2.0 GDI.

Notes

Refer to the **GDIINFO** data structure for a description of how **Output** registers its output capabilities and their meanings.

The only Output styles required by Windows 2.0 GDI are **ScanLines** and **PolyLines**. With certain "smart" devices it may be desirable to use the device's capability in order to draw complex figures. However, in many cases the device is either limited by the number of vertices in a polygon (etc.) or is not able to draw these complex figures into a main memory bitmap, though it can draw them to the screen. In these cases the device driver is now allowed to return –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 here, 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, 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., *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 of the styles are briefly described below:

OS_ARC (3)
: This style 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)
: This style 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.

OS_CHORD (39)
: This style 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)
: This style 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 of half the length between these two points. A zero radius colors a pixel at the center of the circle.

OS_ELLIPSE (7)
: This style 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)
: This style specifies a polygonal area which 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 is closed back to the first point, if necessary.

OS_TRAPEZOID (20)
: This style, formerly called "Winding Number Fill Polygon", specifies a polygonal area which 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 is closed back to the first point, if necessary.

OS_RECTANGLE (6)
: This style causes a rectangle to be drawn on the device, using two points passed as the two corners. See the **RECT** data structure.

OS_POLYLINE (18)
: This style 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)
: This style 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 scan line.

Output uses the current binary raster operation mode (ROP2) when drawing lines and scan lines. 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 (lines with gaps), the routine 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 brush pixel and background colors are equal. If *lpBrush* is NULL, the pen is used for the scan lines. A pen is considered to be the same as a solid brush.

Pixel Function

Pixel (*lpDestDev, X, Y, PhysColor, lpDrawMode*) : *PhysColor*

Purpose This routine sets or retrieves the color of the specified pixel. If *lpDrawMode* is not NULL, this routine sets the given pixel to the color given by *PhysColor*, using the binary raster operation given by *lpDrawMode*, or, if *lpDrawMode* is NULL, the routine returns the physical color of the pixel given by *X* and *Y*. (See *Return Value*, below.)

Parameters *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

X and *Y* are integer values specifying the device coordinates of the pixel to be acted on.

PhysColor is a physical color value of type **PCOLOR**.

lpDrawMode is a long pointer to a data structure of type **DRAWMODE** that includes the binary raster operation to carry out on the given pixel.

Return If *lpDrawMode* is NULL, the routine returns the physical color of the pixel given by *X* and *Y*.

If *lpDrawMode* is *non-NULL*, the *routine* returns **DX:AX** = 0000:0001.

On error, in either the SetPixel or GetPixel mode, the routine returns **DX:AX** = 8000:0000.

ScanLR Function

ScanLR (*lpDestDev, X, Y, PhysColor, Style*) : *wReturnVal*

Purpose This routine 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 "Paint" to perform flood fills.

Parameters *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

X and *Y* are integer values specifying the x and y coordinates of the pixel from which to start the scan.

PhysColor is a physical color value of type **PCOLOR**.

Style is 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. *PhysColor*.
- If bit 1 is cleared, scan for a pixel with color that doesn't match.
- If bit 2 is set, scan to the left.
- If bit 2 is cleared, scan to the right.

Return 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, **AX** = 8000h.

If no pixel is found that satisfies the given scan condition, **AX** = -1

SaveScreenBitmap Function

SaveScreenBitmap (*lpRect, Command*) : *wSuccess*

Purpose This function saves a single bitmap from the display, or restores a single (previously stored) bitmap to the display. It is used, e.g., when a menu is pulled down, to store the part of the screen that is "behind" the menu, until the menu goes away.

Parameters *lpRect* is a long pointer to the rectangle to use.

Command

0: save the rectangle

1: restore it

2: discard previous save, if there was one.

Return This routine 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)

Notes Because **SaveStringBitmap** can save only one bitmap at a time, the device driver must maintain a record of whether the save area is currently in use.

The bitmap is stored in 'shadow memory': memory for which the device has control of allocation. Thus, the device can save the bitmap in whatever form is most convenient for it, without the rest of Windows worrying about where it goes.

5.5 Information Functions

The following are the information functions:

ColorInfo Function

ColorInfo (*lpDestDev, colorin, lpPCOLOR*) : *rgbColor*

Purpose This routine 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 routine should choose the best possible physical color to match this color, 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 routine should return the corresponding RGB color value.

Physical colors returned by this routine are only used by GDI to set text colors, background colors, and pixel colors

using **Pixel**.

Parameters *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

colorin is 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.

lpPCOLOR is a long pointer to a variable of type **PCOLOR**. See the **PCOLOR** description in Chapter 6.

Return *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.

EnumObj Function

EnumObj (*lpDestDev, Style, lpCallbackFunc, lpClientData*) : *LastCallback*

Purpose This routine 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.

Parameters *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

Style is 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 is a long pointer to the user-supplied callback function.

lpClientData is a long pointer to the user-supplied data.

Return This routine returns the last value returned by the callback function.

Notes The callback function has the form:

CallbackFunction(*lpLogObj, lpClientData*);

where *LogObj* 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 enumerating brushes, the background color for hatched brushes is not returned.

Since there can be an almost infinite number of brushes, you may choose to enumerate only a few of them (e.g., all possible solid color brushes).

EnumDFonts Function

EnumDFonts (*lpDestDev, lpFaceName, lpCallbackFunc, lpClientData*)
: *LastCallback*

Purpose This routine 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.

Parameters *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

lpFacename is a long pointer which determines the method of enumeration. If *lpFacename* points to a string containing the name of a font face, all 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 is a long pointer to the user-supplied callback function. See notes, below.

lpClientData is a long pointer to the user-supplied data.

Return This routine returns the last value returned by the callback function.

Notes The callback function has the form:

CallbackFunction(*lpLogFont, lpTextMetrics, FontType, 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 which 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" can be ORed into *FontType* to indicate that the font is composed of a raster bitmap or vector strokes. If the device is capable of text transformations (Scaling, Italicizing, etc.) 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.

GetCharWidth Function

GetCharWidth (*lpDestDev, lpBuffer, FirstChar, LastChar, lpFont, lpDrawMode, lpFontTrans*) : *wSuccess*

Purpose 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.

Parameters *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

lpBuffer is a long pointer to the character width data, an array of 16-bit values.

FirstChar is the first character of the range.

LastChar is the last character of the range.

lpFont is a long pointer to a data structure of type **FONTINFO**.

lpDrawMode is 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 10 for a description of the text justification and character spacing.

lpFontTrans is a long pointer to a data structure of type **TEXTXFORM**

Return This function returns its information in a buffer, to which lpBuffer points. In the event of an error, it returns **AX** = 0.

Notes None.

DeviceBitmap Function

DeviceBitmap (*lpDestDev, Command, lpBitmap, lpBits*) : *wSuccess*

Purpose The call to this function is not yet implemented in GDI. It must be implemented as a stub.

Parameters *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

Command is an integer containing the number of the command.

lpBitMap is a long pointer to a data structure of type **BITMAP** containing a description of the device bitmap.

lpBits is a long pointer to the contents of the device bitmap.

Return This call is only a stub at this time. It may be used in future versions of Windows. It currently returns **AX** = 0.

Notes You must set up the stack frame correctly to assure correct returns to GDI should the stub ever be called.

5.6 Attribute Functions

The following are the display attribute functions:

RealizeObject Function

RealizeObject (*lpDestDev, Style, lpInObj, lpOutObj, lpTextXform*) : *wSize*

Purpose This primitive directs the support module to create an attribute structure that will be used when drawing output primitives. It may also direct the module 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. The *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 routine must translate the logical attributes into sufficient information to accurately describe a physical object for use by output routines when drawing.

If *lpOutObj* is NULL, the routine 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*.

Parameters

lpDestDev is a long pointer to a data structure of type **PDEVICE**.

Style is an integer that specifies the type of object to be realized. The predefined objects are:

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.

lpInObj is 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 is 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 *dfBreakChar* in the **FONTINFO** data structure. Additional information is copied to a data structure of type **TEXTXFORM** pointed to by *lpTextXform*.

lpTextXform is 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. Rotating the brush ensures that patterns drawn with the new origin match patterns already on the screen. 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** routine, allowing more capable devices to

make changes to the standard font.

Return If a device cannot realize an object, **RealizeObject** returns zero.

Notes None.

For further information on TEXTXFORM data structures, please refer to Chapter 10, Data Structures and File Formats.

SetAttribute Function

SetAttribute (*lpDestDev, StateNum, Index, Attribute*) : *wReturnVal*

Purpose The code which calls this function is not yet implemented in GDI.

Parameters *lpDestDev* is a long pointer to a data structure of type **PDEVICE**.

StateNum is an integer that specifies the state number.

Index is an integer.

Attribute is an integer.

Return At this time, this call is just a stub, and it returns **AX** = 0.

Notes You must set up the stack frame correctly to assure correct returns to GDI should the stub ever be called.

5.7 Cursor Functions

The following routines allow the 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.

Inquire Function

Inquire (*lpCURSORINFO*)

Purpose This routine returns the mouse's mickey-to-pixel ratio.

Parameters *lpCURSORINFO* is a long pointer to a device information block (data type **CURSORINFO**) that is filled in by the support module. The first word is the **X** mickey-to-pixel ratio, and the second word is the **Y** mickey-to-pixel ratio.

Return On return, **AX** holds the number of bytes (4) actually written into the data structure.

SetCursor Function

SetCursor (*lpCURSORSHAPE*)

Purpose This routine 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.

Parameters *lpCURSORSHAPE* is a long pointer to a data structure of type **CURSORSHAPE** that specifies the appearance of the cursor for the specified device.

Return None.

Notes 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 preserve the appearance of the screen as the cursor is replaced and ensures that at least some of the cursor is visible on all potential backgrounds.

MoveCursor Function

MoveCursor (*absX*, *absY*)

Purpose This routine 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 routine must ensure that screen bits under the current cursor position are restored and the bits under the new position are saved. The routine must move the cursor, even if the cursor is not currently displayed.

Parameters *absX* and *absY* are absolute X and Y screen coordinates of the new cursor position.

Return None.

Notes Microsoft Windows may specify a position where the cursor shape would lie partially outside of the display bitmap. The OEM routine is responsible for clipping the cursor shape to the display boundary.

The **MoveCursor** routine 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.

CheckCursor Function

CheckCursor ( )

Purpose This routine is called on every timer interrupt. It allows the cursor to be displayed if it is no longer excluded.

Parameters None.

Return Value None.

Chapter 6
Device Drivers

6.1 Overview

This chapter explains how to write a device driver to work with Windows. Device drivers communicate with Windows applications and Windows users through GDI, the Windows spooler, and the control panel application.

The following list defines the names used throughout this chapter:

Windows Application	A Windows application is any program that has been specifically designed to run under Windows. A Windows application communicates with a device driver by making calls to GDI, which then calls the device driver.
GDI	GDI (the Graphics Device Interface module of Windows) handles calls from the Windows application and passes them on to the device driver.
Spooler	The spooler program maintains a printing queue to ensure that all printing jobs are processed serially and to allow other applications to run while a job is printing. The spooler consists of two parts: the spooler "task" and the spooler "library". The spooler task is a Windows application that can be invoked by the user to set or change printing priorities. The spooler library is a set of routines called by the device driver to start, process, and end printing jobs.
Control Panel Application	The control panel application is a Windows application that lets the user change system settings, including printer assignments and characteristics.
GDI Library	The GDI library contains a set of supporting routines for device drivers. These utilities include versions of output routines such as **Bitblt** and **Strblt**, a **Transpose** routine for banding devices, and priority queue routines for daisy wheel printers.

6.2 Device Driver Modules

A device driver module is a Windows library module that contains the GDI routines needed to access a specified device or family of devices. The library (.DRV) file containing a device driver must also contain information naming the type of devices it can drive. These are the device types that can be defined in the WIN.INI file under the "[devices]" heading. To provide this information, each device driver must define a DDVR field. The field is added to the module by specifying a DESCRIPTION statement in the module definition file. The field should contain information in the following form:

DDVR*type*[,*type*]

where **DDVR** is the keyword that identifies the module as a device driver, and *type* is the name of a device type supported by the driver. This name must have the same spelling as would be used in the WIN.INI file. For example, the statement line

```
DESCRIPTION DDVR fx-80,fxG-80
```

defines two devices types: "fx-80" and "fxG-80."

Device driver modules are created using the same steps required for a Windows library.

6.2.1 Communications

A Windows application communicates with a device driver by first calling **CreateDC** to define the attributes of the display context, then issuing calls to GDI routines to produce output on the device. When the application calls **CreateDC**, GDI calls the **Enable** routine, supplied by the device driver. The **Enable** routine initializes the device for use by GDI routines.

The application calls the GDI routine **Escape** to control the actions of the device (for example, to start a print job, start a new page, or end a page). GDI translates the **Escape** calls into calls to the device driver's **Control** routine. Each **Escape** call gives an escape function number specifying the action to be performed.

The device driver's **Control** routine starts a print job by calling the spooler routine **OpenJob** to obtain a job number. All subsequent calls to the spooler use the job number.

Once the print job has been started, the application makes calls to GDI routines to output data on the device. GDI translates these calls into calls to the corresponding OEM-supplied output routines (described in Chapter 5 of this book.)

GDI has, in the past, supported two broad classes of devices: dot matrix devices (which require banding support) and daisy wheel devices.

With this release (version 2.0), GDI supports laser printers. Appendix E contains a list of printer escapes, including additions directed toward laser printer support.

If the device is a banding (dot matrix) device, as indicated by the *dpRaster* field in the **GDIINFO** structure, then two possibilities exist for printing. The application can perform its own banding, but if it does not, then GDI will. GDI builds a metafile from the application's GDI calls or bitmap. When the application sends a NEWFRAME escape function, GDI enters a banding loop, as follows:

1. First GDI calls the device driver **Control** routine with NEXTBAND as the escape function. The device driver initializes a **BITMAP** structure in **PDEVICE** to hold one band and returns a pointer to a **RECT** structure holding the device coordinates of the first band to GDI.
2. GDI then plays the metafile, making calls to output routines. GDI uses the coordinates returned by the device driver for clipping so that only the output appearing in the specified band is produced. The output routines (written by the OEM or called from the GDI library) write the metafile output to the band buffer.
3. After playing the metafile, GDI calls **Control** again with the NEXTBAND escape function. The device driver transforms the bitmap in the band buffer if necessary, sends it to the spooler, clears the buffer, and returns the device coordinates of the next band.
4. Steps 2 and 3 are repeated until the device driver returns 0,0,0,0 in the **RECT** structure as the device coordinates of the next band. This causes GDI to exit the banding loop. In general, the driver bands the entire page.
5. GDI sends the NEWFRAME call to the driver, which usually ignores it. The driver should be smart enough to know where the end of the page is, and it will, under ordinary circumstances, send a TOF to the printer on its own.

For any device that does not request banding support, GDI calls the OEM output routines directly, without building a metafile.

GDI assumes random access to the page image; the application is not required to output text in the same order that it appears on the page. Thus, some devices (for example, daisy wheel printers) may need to buffer a page and sort output lines to avoid backing up on the page. The GDI library provides priority queue routines to facilitate this.

6.2.2 Building a Dot Matrix Device Driver

This section provides a step-by-step overview of the requirements for building a dot matrix (banding) device driver. The routines introduced in this section are described in greater detail later in this chapter.

If your device requires banding support, here are the steps you must follow:

1. Initialize the **GDIINFO** structure with the correct data for your device. You must set the *dpRaster* field in **GDIINFO** to indicate that your device requires banding support.

   You can cause space to be allocated for your band buffer in the **PDEVICE** structure when the device is enabled by allowing space for the buffer in the *dpDEVICEsize* field of **GDIINFO**. The band buffer must have type **BITMAP**.

2. Provide the **Enable** routine for initializing the device and preparing it for use and the **Disable** routine for disabling the device and restoring it to its previous state.

3. Provide the **DeviceMode** routine for interacting with the user through the control panel application. Create a resource file to define the dialog box used for interaction. Change the information in **GDIINFO**, if necessary, in response to the user's input.

4. Provide definitions for any of the following output and information routines that your device supports, and stubs for the routines your device does not support.

   Bitblt
   Output
   Pixel
   Strblt
   ColorInfo
   EnumDFonts
   EnumObj
   RealizeObject
   ScanLR

   Instead of writing these routines yourself, you can implement these routines by calling supporting routines in GDI. The supporting routines are described in the section on the GDI Library, below.

5. Supply the **Control** routine, which starts, ends, and aborts jobs, handles banding and new pages, and provides information to the application.
6. Supply any routines needed to translate the bitmaps sent to the device driver by GDI into a series of bits the device understands. (The GDI library offers a **Transpose** utility for rearranging the order of bits.)
7. If the device can realize its own fonts, create font files for the device fonts.
8. Compile and link your routines into a single executable module.

6.2.3 Building a Daisy Wheel Device Driver

This section provides a step-by-step overview of the requirements for building a daisy wheel device driver. The routines introduced in this section are described in greater detail later in this chapter.

Here are the steps you must follow to write a daisy wheel device driver:

1. Initialize the **GDIINFO** structure with the correct data for your device.
2. Provide the **Enable** routine for initializing the device and preparing it for use and the **Disable** routine for disabling the device and restoring it to its previous state.
3. Provide the **DeviceMode** routine for interacting with the user through the control panel application. Create a resource file to define the dialog box used for interaction. Change the information in **GDIINFO**, if necessary, in response to the user's input.
4. Provide definitions for any of the following output and information routines that your device supports, and stubs for the routines your device does not support.

   Bitblt
   Output
   Pixel
   Strblt
   ColorInfo
   EnumDFonts
   EnumObj
   RealizeObject
   ScanLR

5. Supply the **Control** routine, which starts, ends, and aborts jobs, handles new pages, and provides information to the application.

6. If necessary, use the priority queue utilities supplied by GDI or your own routines to implement a sorting scheme for the lines on a page, avoiding the problem of backing up on a page.
7. Compile and link your routines into a single executable program.

6.3 Device Driver Routines and Data Types

The device driver consists of eighteen routines, as follows:

BitBlt
ColorInfo
Control
Disable
Enable
EnumDFonts
EnumObj
Output
Pixel
RealizeObject
StrBlt
ScanLR
DeviceMode
ExtTextOut
GetCharWidth
DeviceBitmap
SetAttribute
Inquire

These routines are described in the following sections.

Device drivers use the following data types:

GDIINFO
PDEVICE
BITMAP
RECT

These data types are described in Chapter 10, Data Structures and File Formats.

6.3.1 Enable

GDI calls the device driver's **Enable** routine when a Windows application calls **CreateDC** using the device name. **Enable** initializes the device for use by GDI. The **Enable** routine is fully described in Section 5.2 of this manual.

If the device is a banding device, the device driver can cause GDI to allocate space for a band buffer when the device is enabled by allowing space for the buffer in the *dpDeviceSize* field of the **GDIINFO** structure. The band buffer must have type **BITMAP**.

If the high bit of the *Style* parameter of the **Enable** call is set, GDI is calling **Enable** for informational purposes only; no output will be sent to the device. To conserve space, the device driver may not want to allocate a band buffer in this case. To allow for this decision, the device driver must take responsibility for allocating the band buffer itself, instead of allocating it through the *dpDeviceSize* field of **GDIINFO**.

6.3.2 Disable

GDI calls the **Disable** routine when the Windows application is finished with the device. **Disable** restores the device to its state prior to the **Enable** call. The **Disable** routine is fully described in Section 5.2 of this manual.

6.3.3 DeviceMode

The control panel application calls the device driver's **DeviceMode** routine when the user selects the device name from a list box. **DeviceMode** sets up a dialog box that displays the current device environments and permits the user to make changes. The **DeviceMode** routine is fully described in Section 5.2 of this manual.

6.3.4 Output and Information Routines

A device driver must provide the following output and information routines:

Output	Information
Bitblt	**ColorInfo**
Output	**EnumDFonts**
Pixel	**EnumObj**

Strblt **RealizeObject**

ScanLR

The dot matrix printers that Windows supports are treated as a special kind of memory display, and the GDI library provides output and information routines for these printers. These GDI routines are described in the secion on the GDI Library, below. The device driver must either call the GDI library routines or implement the routines as specified in Chapter 5 of this manual.

6.3.5 Control

Control (*lpDevice, EscNum, lpInData, lpOutData*) : *Result*

Purpose This routine performs the work of processing and printing. To control the device, a Windows application makes calls to the **Escape** routine, passing an escape number to signify the action to be performed. GDI translates the **Escape** call into a call to **Control**, which recognizes the escape number and takes the appropriate action.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, as defined by the device driver.

EscNum is an integer value specifying the escape number. The escape numbers are listed and described in Appendix E.

lpInData is a long pointer to data passed to the device driver from GDI.

lpOutData is a long pointer to data to be passed to GDI by the device driver.

Return *Result*, an integer value, should be positive to indicate success, negative to indicate an error or failure, and 0 for unimplemented escape numbers.

Subfunctions The subfunctions are described in Appendix E.

6.4 GDI Library

GDI provides a library of supporting routines for use by device drivers. Device drivers that call these routines must link to GDI.LIB and include the file printer.h.

The first section describes routines for use with dot matrix printers. The second section describes a priority queue system for sorting lines of output for daisy wheel printers.

6.4.1 Dot Matrix Support

Note

When using a GDI library routine, you must adjust the coordinates passed to the routine so that it (the routine) receives the coordinates for the band buffer (a memory bitmap) rather than the device coordinates.

GDI Library Routine	OEM Routine
dmBitblt	**Bitblt**
dmColorInfo	**ColorInfo**
dmEnumObj	**EnumObj**
dmEnumDFonts	Similar to **EnumDFonts**, but always returns 1. This tells GDI that the driver is not capable of realizing any fonts. GDI's font manager will select the best font it can find from its disk fonts. Device drivers that can realize their own fonts must supply their own version of this routine, or must add their device fonts to GDI's selection of disk fonts by using the **AddFontResource** routine.
dmOutput	**Output**
dmPixel	**Pixel**
dmRealizeObject	**RealizeObject**
dmStrBlt	**Strblt**
dmScanLR	**ScanLR**

In addition, the GDI library provides the following routine for transposing bits in a bitmap.

dmTranspose (*lpSrc, lpDst, WidthBytes*)

Purpose This routine copies 8 times the number *WidthBytes* (that is, 8 scan lines) from the source pointer to the destination pointer, transposing bits as it copies. The 8 most significant bits, one from each of the 8 scan lines, make up the first byte of the transposed line; the 8 next most significant bits make up the next byte, and so on.

Parameters *lpSrc* is a long pointer to the string to be copied and transposed.

lpDst is a long pointer to a buffer to hold the transposed string.

WidthBytes is a short integer specifying the number of bits in each scan line. Normally, the order in which bits from the 8 scan lines are packed into a byte is the same as the order of the scan lines. This order can be reversed by giving a negative value for *WidthBytes*. When *WidthBytes* is negative, the most significant bit from the first scan line becomes the eighth bit in the first byte of the transposed line, the most significant bit from the second scan line becomes the seventh bit, and so on.

Return None.

Note

This routine assumes that neither the source nor the destination bitmap exceeds 64K. The source and destination bitmaps must be disjoint; **Transpose** does not transpose in place.

6.4.2 Priority Queue Routines

The GDI library provides the priority queue data type, used in the routines described in this section. Priority queues are accessed through a two-byte value, the "key". Each key can have two bytes of information, called a "tag", associated with it.

CreatePQ (*size*) : *hPQ*

Purpose This routine creates a priority queue.

Parameters *Size*, a short integer value, is the maximum number of items to be inserted into this priority queue.

Return *hPQ* is a handle to the priority queue if the routine was successful. Otherwise, *hPQ* is zero.

MinPQ (*hPQ*) : *tag*

Purpose This routine returns the tag associated with the key with the smallest value in the priority queue, without removing this element from the queue.

Parameter *hPQ* is a handle to a priority queue.

Return *tag* is the tag associated with the key having the smallest value.

ExtractPQ (*hPQ*) : *tag*

Purpose This routine returns the tag associated with the key having the smallest value in the priority queue and removes the key from the queue.

Parameter *hPQ* is a handle to a priority queue.

Return *tag* is the tag associated with the key having the smallest value.

InsertPQ (*hPQ, tag, key*) : *Result*

Purpose This routine inserts the key and its associated tag into the priority queue.

Parameters *hPQ* is a handle to a priority queue.

tag is the tag to be associated with the inserted key.

key is the key to be inserted.

Return *Result*, a short integer value, is TRUE if the insertion is successful, ERROR (–1) otherwise.

SizePQ (*hPQ, sizechange*) : *Result*

Purpose This routine increases or decreases the size of the priority queue.

Parameters *hPQ* is a handle to a priority queue.

newsize is a short integer value specifying the number of entries to be added or removed.

Return *Result*, 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*

Purpose This routine deletes a priority queue.

Parameter *hPQ* is a handle to a priority queue.

Return *Result*, a short integer value, is TRUE if the queue is deleted, ERROR (–1) otherwise.

6.5 Spooler Routines

Windows provides background spooling for all the ports listed in the Windows initialization file (WIN.INI) that have printers attached to them. All printing jobs must go through the spooler for two reasons:

1. The printer spooler has its own thread of execution and yields at appropriate intervals to allow other applications to run while printing.
2. If two applications try to write to the same port directly, their data can be mixed up. The spooler ensures that all the printing jobs are processed serially.

The printer spooler is divided into two parts: the library routines and the spooler task. The library routines, which are called by the device driver, spool data to intermediate disk files, assuming that I/O to disk will be faster than I/O to physical devices.

The spooler task, implemented as a Windows application, reads data from the disk and writes to the ports when the ports are ready. In addition, the spooler task displays the printing jobs queued at each port. Through the spooler window the user can suspend or resume printing to a port or terminate a particular printing job.

The spooler library routines are described below.

OpenJob (*lpDosFile, lpDocName, hDC*) : *JobNum*

Purpose This routine assigns a unique job number to a print job. The device driver uses the job number in subsequent calls to the spooler.

Parameters *lpDosFile* is a long pointer to a null-terminated ASCII string specifying the name of a port or a disk file. If *lpDosFile* specifies the name of a port, the spooler will create and use an intermediate disk file. Otherwise, the spooler writes to a file with the given name. If *lpDosFile* is NULL, the spooler overwrites the file specified by the *lpDocName* parameter.

lpDocName is a long pointer to a null-terminated ASCII

string specifying the name of the document to be spooled. This name is used to display the document name in the spooler task window.

hDC is a handle to the display context associated with this print job, which is passed to the device driver through the SETABORTPROC escape.

Return *JobNum* is a short integer that represents a unique job number. This number must be used in all subsequent calls to the spooler.

The error code SP_NOPRINTER is returned if no printer is associated with the port specified by *lpDosFile*.

StartSpoolPage (*JobNum*) : *bResult*

Purpose This routine is used to spool large jobs into multiple files, allowing spool files to be deleted from the disk as their contents are sent to a port. If the *lpDosName* parameter of the **OpenJob** call specified a valid port, the call to **StartSpoolPage** opens a new temporary file ready to receive data. Otherwise, the call to **StartSpoolPage** has no effect.

Parameter *JobNum*, a short integer, is the job number returned by **OpenJob**.

Return *bResult*, a Boolean value, is TRUE if the routine is successful and FALSE otherwise.

EndSpoolPage (*JobNum*) : *bResult*

Purpose This routine closes a spool file opened by **StartSpoolPage**.

Parameter *JobNum*, a short integer, is the job number returned by **OpenJob**.

Return *bResult*, a Boolean value, is TRUE if the routine is successful and FALSE otherwise.

WriteSpool (*JobNum, lpData, size*)

Purpose This routine sends data to the spooler.

Parameters *JobNum*, a short integer, is the job number returned by **OpenJob**.

lpData is a long pointer to the data stream to be spooled under *JobNum*.

size is a short integer specifying the length of the data stream pointed to by *lpData*.

Return *Count*, a short integer, is the number of bytes written if the routine is successful. Otherwise, one of the following error codes is returned:

SP_APPABORT	The print job is being terminated because the application's abort procedure returned FALSE.
SP_USERABORT	The user aborted the print job through the spooler task.
SP_OUTOFDISK	No space is available for spooling, and no space will become available.

In case of error, the device driver calls **DeleteJob** to abort the print job and returns the error code to GDI, which passes the code to the application for appropriate action.

Note

The SP_OUTOFDISK error code is returned by **WriteSpool** to the device driver only when the space shortage is so severe that it is pointless to wait for more space to become available. In this case, the spooler does not call the application's abort procedure before returning the SP_OUTOFDISK error code.

If the disk space shortage can be solved by waiting for space to become available as spooled files are sent to the device, the spooler calls the application's abort procedure to ask whether the application wants to wait. If the application is not willing to wait, the print job is aborted and the SP_APPABORT error code is returned to the device driver.

WriteDialog (*JobNum*, *lpData*, *size*)

Purpose This routine sends a text message to the spooler, which displays the message in a message box on the user's screen. **WriteDialog** is useful for getting the user's attention during a print job (for example, when the font wheel on a daisy wheel printer needs to be changed).

Parameters *JobNum*, a short integer, is the job number returned by **OpenJob**.

lpData is a long pointer to the message to be displayed.

size is a short integer specifying the length of the message.

Return None.

DeleteJob (*JobNum*)

Purpose This routine deletes a print job from the spooler without printing it. All memory and disk space associated with the print job are freed.

Parameter *JobNum*, a short integer, is the job number returned by **OpenJob**.

Return None.

EndJob (*JobNum*)

Purpose This routine is used to end a print job. It informs the spooler that all data for the print job have been sent.

Parameter *JobNum*, a short integer, is the job number returned by **OpenJob**.

Return None.

6.6 Compiling and Linking

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

Resource file	Defines dialog box for **DeviceMode** routine.
Source files	Contain the device driver code, including the twenty required routines.
Include files	Contain definitions used by 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 supporting routines. At a minimum, device drivers must link with the C Windows library (SWINLIBC.LIB), USER.LIB, and GDI.LIB.

The following command line is recommended for compiling device driver

source files with C 4.0:

cl -d -c -Asnw -Gsw -Os -Zp [*source-file...*]

This command line specifies the following options:

-d	Display command lines for compiler passes.
-c	Compile only.
-Asnw	Use small memory model with SS not equal to DS and DS fixed.
-Gsw	Remove stack probes.
-Os	Optimize for code size and relax alias checking.
-Zp	Pack structure members.

For linking, the following options are recommended. They are shown in response file form.

object-file[, *object-file...*]
module.drv
module.map/*map*
\lib\swinlibc \lib\user \lib\gdi /NOD
module.def

The first line gives the name or names or the object files to be linked. The second line gives the name of the executable module. The third line specifies the /map option and gives the name of the map file. The fourth line lists the names of the libraries to be used and specifies the /NOD option to prevent linking with the default libraries. The fifth line gives the name of the module definition file.

6.6.1 Device Drivers

DESCRIPTION Line

DESCRIPTION 'DDRV <DeviceTypeList>: <FontTypeList>: <DescriptiveText>'

The DESCRIPTION line of a device driver must define the device type. The device type is used by the **Setup** program to associate fonts with the desired device.

The *DeviceTypeList* is a device type recognized by the device. It is the same as the second parameter to the **CreateDC** function. The *FontTypeList* can be the same as given in the DESCRIPTION line of a font resource file.

```
DESCRIPTION 'DDRV Epson FX-80 : 83, 60, 72; 120, 72, 60 : EPSON 60x72 dpi'
DESCRIPTION 'DDRV Epson FX-80 : 167, 120, 72; 60, 72, 120 : EPSON 120x72 dpi
```

6.7 Realizing Fonts

If the device can realize its own fonts, the device driver must include font files that describe the device fonts. The device driver can either add these font descriptions to the Windows font selection by using the **AddFontResource** routine, or it can realize the fonts itself through the **RealizeObject** and **EnumDFonts** routines.

If the font descriptions are added to the Windows font selection, the device driver's **RealizeObject** routine returns zero when any font is selected, and the **EnumDFonts** routine returns 1 to signify that no device fonts are available. This causes GDI to select the best font from its own selection (which includes the added device fonts). The *devicename* field of the font file format allows GDI to identify fonts that are associated with particular devices.

If the device driver handles its device fonts itself instead of adding them to the Windows selection, it must provide versions of **RealizeObject** and **EnumDFonts** that support the device fonts.

6.8 Full Page Banding for Text

Banding devices that offer high-quality device fonts may benefit from making two complete passes over each page to be output. The first pass outputs all text on the page, while the second pass divides the pages into bands and outputs graphics. This method eases the memory requirements for the device, since text lines are handled by the device fonts instead of being treated as bitmaps.

When the device driver receives the first NEXTBAND call, it returns the device coordinates of the entire page to GDI. This means that no clipping takes places as GDI plays the metafile for the page. The device driver processes all **Strblt** calls for the page and ignores all other calls.

When GDI finishes playing the metafile and calls NEXTBAND again, the device driver initializes a buffer for the first band of graphics and returns the coordinates of the band to GDI. From this point on, banding is handled in the standard fashion, except that the device driver now ignores all **Strblt** calls, since they have already been processed. The device driver continues processing and outputting bands until the last band on the page is processed; the device driver then returns 0,0,0,0 to GDI.

Some pages may contain only text and no graphics. To detect this case, the device driver can maintain a flag during the first (text) pass over the page. If the GDI metafile makes calls to any routine besides **Strblt**, the flag is set (although the call to the routine is ignored). Before returning the coordinates of the first graphics band to GDI, the device driver checks the flag. If the flag is not set, no graphics are required and the device driver can simply return 0,0,0,0.

Chapter 7
Fonts and Their Structure

Introduction

7.1 Font Resources

Each Windows font file is a collection of character bitmaps or vector strokes with a unique combination of height, width, facename, character set, and other attributes.

A *font resource* is a group of individual fonts which have various combinations of heights, widths, and pitches. For example, the system font resource contains the ANSI character set and the terminal font resource contains the OEM character set used by the Windows system.

7.2 Creating Font Files

Windows font files have the **.FON** extension. Every **.FON** file consists of one or more **.FNT** files, a dummy code module, and a module definition file (with **.DEF** extension.) These are linked together to produce the **.FON** file, which is the final form actually used by Windows. The number, size, and type of **.FNT** files in a **.FON** file is up to you. Because when GDI loads a font, it must load all of the fonts within a particular .FON file, it is recommended that each .FON file contain only a few .FNT files with related aspect ratios or resolutions. Common ratios are 1 to 1 and 1.33 to 1. (The VGA and 8514 displays are 1:1, while the EGA is 1.33:1.) Some older devices were 1.5 to 1, and there are some displays at 2 to 1. When planning font sizes, remember that GDI can scale device-independent raster fonts by 1 to 8 vertically and 1 to 5 horizontally. GDI can also simulate bold, underlined, strikeout, and italic fonts. Although scaled or simulated fonts do not look as nice as actual fonts, they can save valuable memory resources. To create a **.FON** file:

- **.FNT** files may be created in one of two ways. The first way is to use an existing font bitmap, converting it to the format specified in section 2 of Appendix F, "The Font File Format".

  The second way to create **.FNT** files is by using the FONTEDIT program under Windows. This is described in the Microsoft Windows Software Development Kit volume, "Programming Tools", version 2.0. The output of the font editor is a **.FNT** file.

- The dummy code module can be copied verbatim from Section 7.3.1 of this book, and is also provided in the \FONTS\FONTS.ASM file included in this distribution.

- You can use the example Module Definition File in Section 7.3.2 of this book as a template from which to generate yours.

7.3 Creating Font Resource-Only Files

To create a resource-only module that contains only fonts, you must:

1. Create a dummy code module.
2. Create a module definition file that describes the fonts and the devices that use the fonts.
3. Create a .RC file.
4. Compile and link the sources.

7.3.1 Creating the Dummy Code Module

The dummy code module is used to provide the object file from which the resource file is made. The following assembly code illustrates the dummy code module:

```
        TITLE   FONTRES - Stub file to build FONTRES.EXE

.xlist
include cmacros.inc
.list

sBegin  CODE

sEnd    CODE

end
```

7.3.2 Creating the Module Definition File

The module definition file for the dummy executable should have the form:

```
LIBRARY FONTRES

          .
          .
          .
DESCRIPTION 'FONTRES ...'
          .
          .
          .
STUB 'WINSTUB.EXE'
DATA NONE
```

For a System Font file, the **LIBRARY** line should read

```
LIBRARY FONTS
```

For a Terminal (OEM) Font file, the **LIBRARY** line should read

```
LIBRARY OEMFONTS
```

For other types of fonts, you should use the standard Windows font file name, as for example, if you were creating **HELV.FON** you would use

```
LIBRARY HELV
```

or, to create **TMSRE.FON,**

```
LIBRARY TMSRE
```

Description Line

DESCRIPTION 'FONTRES <FontTypeList> : <DescriptiveText>'

Each DESCRIPTION line for a font resource file must specify the aspect ratio of the particular face from the corresponding **.FNT** file. The font type is used by the **Setup** program to associate the given font resource with a selected display or device driver.

The aspect ratio is given as three numbers, separated by commas. The first number is 100 * (AspectY/AspectX), rounded to an integer. The second number is the number of logical pixels per inch in X, and the third number is the number of logical pixels per inch in Y. (*AspectX*, *AspectY*, *LogPixelsX*, *LogPixelsY* are the values given in the device's GDIINFO structure.) For a font to be matched precisely to a particular device, these numbers must match the numbers specified for the device in its **.DEF** file.

In addition, there are two special keywords that may be used in place of the aspect ratios. If the font is a vector font that can be scaled indefinitely, you may specify CONTINUOUSSCALING. This tells GDI that the particular font can be realized correctly at any aspect ratio, and can therefore match any device.

If the font is tailored to a particular device and is not intended for use with any other device, you should specify DEVICESPECIFIC followed by the name(s) of the device(s) for which the font is intended.

```
DESCRIPTION 'FONTRES 133,96,72 : System, (Set #3)'
DESCRIPTION 'FONTRES 200,96,48 : Terminal (Set #2)'
DESCRIPTION 'FONTRES 200,96,48; 133,96,72; 167,120,72 : Courier'
DESCRIPTION 'FONTRES 200,96,48; 133,96,72; 83,60,72; 167,120,72 : Helv'
DESCRIPTION 'FONTRES CONTINUOUSSCALING : Modern, Roman, Script'
DESCRIPTION 'FONTRES DEVICESPECIFIC HP 7470A,·HP 7475A: HP 7470 plotters'
```

Note

The maximum length of a DESCRIPTION line is 127 characters.

Since Windows is capable of synthesizing attributes, such as bold, italic, and underline, you need not create separate .FNT files for fonts with these attributes. However, you are at liberty to do so if you wish.

Other fonts that do not correspond to the user's display aspect ratio may be used by Windows. These are generic raster fonts that are intended for output devices such as bitmap printers which rely on the display driver to draw their text.

7.4 Adding Fonts to Executable Files

The next step is to create an RC script that is used during the build process to bind the .FNT files together into a .FON file.

The RC script can, alternatively, add the .FNT files to a Windows library, a device driver, or a resource-only file. (A resource-only file contains icons, cursors, fonts, and other resources but no code). You should not add fonts to application modules, since an application's resources are available to the application only, which defeats part of the purpose of the font resource.

You add the resources to the file by using the **FONT** statement of the resource compiler, **RC**. That is, you place these statements in the resource script file of the executable module you are creating. The statement has the form:

number FONT *filename*

One statement is required for each font file to be placed in the resource. The *number* must be unique since it is used to identify the font later. The following is a typical resource script file for a font resource:

```
1  FONT FntFil01.FNT
2  FONT FntFil02.FNT
3  FONT FntFil03.FNT
4  FONT FntFil04.FNT
5  FONT FntFil05.FNT
6  FONT FntFil06.FNT
```

Fonts can be added to modules which contain other resources by merely adding them to the existing resource script. This means you can have icon, cursor, menu, and dialog box definitions in the resource script file as well as FONT statements.

7.5 Compiling and Linking the Font Resource

The following make file lists the commands required to compile and link the resource-only file:

```
fontres.obj: fontres.asm
    masm fontres;

fontres.exe: fontres.def fontres.obj fontres.rc fontres.exe \
                FntFi101.FNT FntFi102.FNT FntFi103.FNT \
                FntFi104.FNT FntFi105.FNT FntFi106.FNT
    link4 fontres.obj, fontres.exe, NUL, /NOD, fontres.def
    rc fontres.rc
    command /c rename fontres.exe fontres.fon
```

By convention, all font resource files have the .FON filename extension. The last line in the make file renames the executable file to FONTRES.FON.

7.6 Updating Windows 1.XX Font Files

If your files are in the format used by Windows v1.XX, there is a conversion utility that you can use to update them to the format required by Windows 2.XX. It is called 'NEWFON', and is invoked as follows:

NEWFON oldfilename.fon newfilename.fon

The System and Terminal .FON files are separate in Windows 2.XX. (In Windows 1.XX, they were combined in one file.) The generalized naming convention is as follows:

FONTxxyy.FON, where xx is a file descriptor, and yy, when present, is an identifier for terminal fonts configured to different language requirements.

Example: FONT40NO.FON would be a Norwegian version of a FONTS400.FON file.

7.7 Font File Formats

7.7.1 Raster Font File Format

In addition to the information in the header of the file, a raster font file contains a string of bytes which is 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 below.

7.7.2 Vector Font File Format

The header information for a vector font file is described below. 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 one-byte quantities, it is -128 (080H) and for two-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.

7.7.3 FONTINFO – 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 Appendix F, "The Font File Format" with two exceptions. First, **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.

```
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    reserved;
        short   dfCharWidth;
        char    Facename[n];
        char    Devicename[n];
        char    BitMaps[n];
}
FONTINFO;
```

The fields within the **FONTINFO** structure have the following meanings:

dfType — Two bytes specifying the type of fontfile. The low-order byte is for exclusive GDI use. If the low-order bit of the word is 0, 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, this bit is set to 0. 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 OPAQUE or TRANSPARENT mode. This may be zero at the designer's option.
dfItalic	One byte specifying whether the character definition data represent an italic font. The low-order bit is one if the flag is set. All other bits are zero.
dfUnderline	One byte specifying whether the character definition data represent an underlined font. The low-order bit is one if the flag is set. All other bits are zero.
dfStrikeOut	One byte specifying whether the character definition data represent a struck out font. The low-order bit is one 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 0.
dfPixWidth	Two bytes. For vector fonts, 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 familes 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:

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.

FF_DECORATIVE
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 *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, 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. Note that all characters with codes between *dfFirstChar* 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 *dfFirstChar* so that *dfDefaultChar* is the actual value of the character less *dfFirstChar*. *dfDefaultChar* should ideally be a visible character in the current font, e.g.. a period (.).
dfBreakChar	One byte specifying character which 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. *dfWidthBytes* is always an even quantity so that rows of the bitmap start on word boundaries.
dfDevice	Four bytes specifying the offset in 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 in the **FONTINFO** data structure to the null-terminated string which names the face.
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 begining 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 which 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 boundry. 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) which specify the strokes for each character of the font. *dfBitsOffset* must be even.

dfCharWidth

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 four-byte entry is divided into two two-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 **dfCharWidth** table. This is to allow you to calculate the width or number of bytes of definition of the last character. Though this only applies 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** routine, the *dfFace* and *dfDevice* fields must point to valid character strings containing the face and device names.

7.7.4 LOGFONT – 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:

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 which does not exceed the requested size, and if there is no such font, looks for the smallest font available.

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 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	This field is a 1-byte flag that specifies whether or 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	This field is a 1-byte flag that specifies whether 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	This field is a 1-byte flag that specifies whether 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 Appendix F for a table of the ANSI character set and the OEM character set distributed with the IBM PC.
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 values: OUT_DEFAULT_PRECIS (00H) 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 values:

CLIP_DEFAULT_PRECIS (00H)
CLIP_CHARACTER_PRECIS (01H)
CLIP_STROKE_PRECIS (02H)

lfQuality

This field is a one-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 physical font. Quality may be one of the values:

PROOF_QUALITY(02H)
The character quality of the font is more important than exact matching of the logical font attributes. For GDI fonts, scaling is inhibited so that it may not be possible to map as exact a size as at the lower qualities, but there will be no degradation of appearence. Bold, italic, underline, and strikeout will be synthesized if needed.

DRAFT_QUALITY(01H)
The appearence 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.

DEFAULT_QUALITY(00H)
Don't care.

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:

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.

FF_DECORATIVE
: 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 8

Resources and How to Build Them

8.1 Overview

This chapter presents the information you need in order to build resources and integrate them into device drivers for Windows.

8.2 How to Build Windows Resources

To build a complete Windows device driver, you must produce a set of resources that contains cursors, icons, and other bitmaps, as well as the definitions of the default colors, fonts, and border widths (etc.) to use. This is all compiled together into a **.RES** file, which is later concatenated to the final **.DRV** file by the resource compiler, **RC**.

To build a resource set (**.RES** file), you must complete the following steps:

1. Create a **FONTS.ASM** file.
2. Create a **CONFIG.ASM** file.
3. create icons, bitmaps, and cursors using the **ICONEDIT** program, or use the existing ones provided in the **\RESOURCE** directory on the accompanying diskettes.
4. Assemble, link, and **EXE2BIN** the **FONTS.ASM** and **CONFIG.ASM** files to create **FONTS.BIN** and **CONFIG.BIN**.
5. Use **RC** to compile all the resources together to form the **.RES** file.

8.2.1 Creating the FONTS.ASM File

The **FONTS.ASM** file tells Windows the characteristics of the System and OEM fonts that must be provided by the device driver. (The System font is the default font used by the MS-DOS executive, menus, dialog boxes, etc.) See Chapter 7, "Fonts and Their Structure", for details on how to do this.

This file also defines these same characteristics for the two fonts minimally required to run such programs as Write. These are the **ANSI** fixed-pitch (Courier) and variable-pitch (Helvetica) fonts. Most often, 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 four data structures that describe these same characteristics, adding one for the System font, and one for the Terminal font. Each data structure is of type **LOGFONT** (See Chapter 7). The order of the four 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)
4. **SYSTEM** font (of facename "System")

8.2.2 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, horizontal and vertical
- Scroll bar "thumb" sizes
- Cursor and icon compression ratios

Here is the 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   ?      ;Kanji window height (should be 0 for non-Kanji)
        dw   ?      ;cxBorder (thickness of vertical lines) (usually 1 pixel
                                                             thick)
        dw   ?      ;cyBorder (thickness of horizontal lines) (usually 1 pixel)

;       Default system color values

        db      03Fh,03fh,03fh,0        ;color of scroll bar
        db      000h,0FFh,0ffh,0        ;color of desktop
        db      000h,000h,0FFh,0        ;color of active title bar
        db      0ffh,0ffh,0ffh,0        ;color of inactive title bar
        db      000h,0FFh,0ffh,0        ;color of menu background
        db      0FFh,0FFh,0FFh,0        ;color of window background
        db      000h,000h,000h,0        ;color of window frame (caption)
        db      000h,000h,000h,0        ;color of menu text
```

```
        db      000h,000h,000h,0        ;color of text in window
        db      0FFh,0FFh,0FFh,0        ;color of text in a caption
        db      07Fh,07Fh,07Fh,0        ;color of text in active border
        db      07Fh,07Fh,07Fh,0        ;color of text in inactive border
        db      000h,080h,0FFh,0        ;color of text in application

        dw      0       ;       unused (reserved) words
        dw      0

OEM     ends
```

Note

The values shown in the example above are the default colors shipped with the EGA, VGA, and 8514 color displays, and are the recommended values.

The following describes each field in detail.

cnVertThumHeight	Is a 2-byte value specifying the height in pixels of the vertical scroll bar thumb.
cnHorizThumWidth	Is a 2-byte value specifying the width in pixels of the horizontal scroll bar thumb.
cnIconXRatio	Is a 2-byte value specifying the ratio by which the icon width is to be reduced before displaying.
cnIconYRatio	Is a 2-byte value specifying the ratio by which the icon height is to be reduced before displaying.
cnCurXRatio	Is a 2-byte value specifying the ratio by which the cursor width is to be reduced before displaying.
cnCurYRatio	Is a 2-byte value specifying the ratio by which the cursor height is to be reduced before displaying.
cnKanjiHeight	Is a 2-byte value specifying the height in pixels of the Kanji conversion window.
cnXBorder	Is a 2-byte value specifying the thickness in pixels of vertical lines.

cnYBorder	Is a 2-byte value specifying the thickness in pixels of horizontal lines.
cnScrollBarColor	Is a 4-byte RGB value specifying the default color of the scroll bar.
cnDesktopColor	Is a 4-byte RGB value specifying the default color of the Windows background.
cnActiveCapColor	Is a 4-byte RGB value specifying the default color of the caption in the active window.
cnInactiveCapColor	Is a 4-byte RGB value specifying the default color of the captin in an inactive window.
cnMenuBackgndColor	Is a 4-byte RGB value specifying the default color of the menu background.
cnWindowBackgndColor	Is a 4-byte RGB value specifying the default color of a window's background.
cnCaptionColor	Is a 4-byte RGB value specifying the default color of the a caption.
cnMenuTextColor	Is a 4-byte RGB value specifying the default color of the text in a menu.
cnWindowTextColor	Is a 4-byte RGB value specifying the default color of the text in a window.
cnCaptionTextColor	Is a 4-byte RGB value specifying the default color of the text in a caption.
cnActiveBorderTextColor:	Is a 4-byte RGB value specifying the default color of the text in an active border.
cnInActiveBorderTextColor:	Is a 4-byte RGB value specifying the default color of the text in an inactive border.
cnWorkSpaceTextColor:	Is a 4-byte RGB value specifying the default color of the workspace.
cnReserved	Is held for future use.

8.2.3 Creating Icons, Cursors, and Bitmaps

ICBs acceptable for use by many display resolutions and aspect ratios are provided in various subdirectories of the **\RESOURCE** directory on the accompanying diskettes. The **\RESOURCE\LORES** directory contains ICBs suitable for such displays as the CGA and Hercules; the. **\RESOURCE\MEDRES** directory contains ICBs suitable for EGA-type displays: the **\RESOURCE\VGARES** directory contains ICBs

suitable for VGA-type resolutions: and the **\RESOURCE\HIRES** directory contains ICBs suitable for very high resolution devices, up to 1280x1024 pixels.

If you want to create your own ICBs, you can do so using the Icon Editor provided. In creating ICBs, you should meet the criteria given in the table of Cursor, Icon, and Bitmap files below.

Note

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

8.3 Cursor, Icon, and Bitmap Files

Table 8.1

Cursor, Icon, and Bitmap Files

Resource Name	Type	Filename	Purpose
IDC_ARROW	cursor	NORMAL.CUR	An upward diagonal arrow used for the default mouse cursor in the work area.
IDC_UPARROW	cursor	UPARROW.CUR	An upward arrow used for the mouse cursor in menus and window captions.
IDC_ICON	cursor	ICON.CUR	An empty box used for the mouse cursor in the icon area.
IDC_SIZE	cursor	SIZE.CUR	A box used for the mouse cursor when sizing tiled windows.

IDC_IBEAM	cursor	IBEAM.CUR	An I-beam shaped cursor used for the mouse cursor in edit control windows.
32512	icon	SAMPLE.ICO	A white box with a black border used as the default icon for windows that normally paint their own icons.
32513	icon	HAND.ICO	An upraised hand used to indicate an error or condition that halts operation.
32514	icon	QUES.ICO	A question mark used when the program must query the user for a reply.
32515	icon	BANG.ICO	An exclamation mark used to emphasize the consequences of an operation.
32516	icon	NOTE.ICO	An asterisk used to indicate information that is useful but not central to the current issue.
IDC_WAIT	cursor	WAIT.CUR	An hour-glass used as when loading applications or carrying out lengthly computations.
IDC_CROSS	cursor	CROSS.CUR	A upright cross used as a marker when the mouse is used to select a region or object on the screen.

OBM_RGARROW	bitmap	RIGHT.BMP	A right arrow used in the horizontal scroll bar.
OBM_LFARROW	bitmap	LEFT.BMP	A left arrow used in the horizontal scroll bar.
OBM_CHECK	bitmap	CHECK.BMP	A checkmark used to checkmark menu items.
OBM_BTSIZE	bitmap	BTSIZE.BMP	A size box used between the intersection of two scroll bars.
OBM_UPARROW	bitmap	UP.BMP	An up arrow used in the vertical scroll bar.
OBM_DOWNARROW	bitmap	DOWN.BMP	A down arrow used in the vertical scroll bar.
OBM_SIZE	bitmap	SIZE.BMP	A box used as the sizebox in all windows.
OBM_CLOSE	bitmap	CLOSE.BMP	A box used as the closebox or system menu box in a window.
OBM_CHECKBOXES	bitmap	BUTTON.BMP	A box used as the button shape in dialog boxes.
OBM_BTNCORNERS	bitmap	BTNCORN.BMP	A box used as the cornered button in dialog boxes.

8.3.1 Assembling and Linking FONTS.ASM and CONFIG.ASM

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

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

(Substituting 'config' for 'fonts' as appropriate.)

8.3.2 Using RC to create the .RES File

Once you have completed all the steps listed above, you must create a script for the resource editor, **RC**. You may, if you wish, use the RC script in the **\RESOURCE** directory verbatim.

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.

Chapter 9

Communication and Sound Support Modules

9.1 Introduction

This chapter describes the functions of the communication and sound support modules.

9.2 Communication Module

The communication (COMM) module provides for assigning and deassigning serial device instances to ports, enabling and disabling interrupt handlers for input from assigned serial devices, and sending characters to assigned serial output devices. The data transmission protocol is communicated in a Device Control Block (DCB) when the serial device is assigned. See Chapter 8, "Data Structures and File Formats," for a description of the DCB structure.

For communication functions described in this chapter, the values of the **BP**, **DI**, **SI**, **SS**, and **DS** registers must be saved on entry and restored on exit. Other registers do not have to be preserved.

In case of error, the communication functions return either an initialization error code (the **$INICOM** and **$STACOM** functions) or an error word with one or more bits set to indicate the error. The initialization error codes are as follows:

NULL (0) No errors.

IE_BADID (-1)
Invalid or unsupported ID.

IE_OPEN (-2)
Device already open.

IE_NOPEN (-3)
Device not open.

IE_MEMORY (-4)
Unable to allocate queues.

IE_DEFAULT (-5)
Error in default parameters.

IE_HARDWARE (-10)
Hardware not present.

IE_BYTESIZE (-11)
Illegal byte size.

IE_BAUDRATE (-12)
Unsupported baud rate.

Initialization error codes above 1000 are available for OEM-specific initialization errors.

The error words returned by other functions can have the following settings:

NULL (0x0000)
No errors.

CE_RXOVER (0x0001)
Receive queue overflow.

CE_OVERRUN (0x0002)
Receive overrun error.

CE_RXPARITY (0x0004)
Receive parity error.

CE_FRAME (0x0008)
Receive framing error.

CE_CTSTO (0x0020)
CTS timeout.

CE_DSRTO (0x0040)
DSR timeout.

CE_RLSDTO (0x0080)
RLSD timeout.

CE_CANT (0x4000)
Requested function incomplete (character to be transmitted immediately not transmitted because another character was being transmitted immediately).

CE_MODE (0x8000)
Requested mode unsupported.

9.2.1 Setting Up Devices

The functions described in this section initialize and terminate an RS232 port and provide information about the status of a port.

$INICOM

Purpose This routine initializes an RS232 port and sets up the port with the attributes specified in the DCB. Communications buffers and interrupts for the port are also initialized.

Entry **ES:BX** points to Device Control Block with all fields set.

Exit **AX** contains the initialization error code.

$TRMCOM

Purpose This routine terminates an RS232 channel. $TRMCOM is called whenever a file associated with a communication channel is closed. The routine waits for any outbound data to be transmitted and disables interrupts from the device.

Entry **AH** contains the device ID.

Exit None.

$SETCOM

Purpose This routine reinitializes an RS232 port, allowing for a change in the configuration of the port. The port is set up with the attributes specified in the DCB, but buffers and interrupts are not initialized as they are in the **$INITCOM** call.

Entry **ES:BX** points to Device Control Block with all fields set.

Exit **AX** contains the initialization error code.

$STACOM

Purpose This routine tests and resets the device status, returning the number of free bytes in the input queue and the number of bytes queued for input.

Entry **AH** contains the device ID. **ES:BX** points to a status structure to be updated with the following form:

```
STAT STRUC
     STFLAGS   DB    ;Flag byte
     STRXCOUNT DW    ;Count of bytes in receive queue
     STTXCOUNT DW    ;Count of bytes in transmit queue
STAT ENDS
```

The STFLAGS byte contains the following flags:

```
FCTSHOLD   EQU  001H   ;Tx is on CTS hold
FDSRHOLD   EQU  002H   ;Tx is on DSR hold
FRLSDHOLD  EQU  004H   ;Tx is on RLSD hold
FXOFFHOLD  EQU  008H   ;Tx is on X-OFF hold
FXOFFSENT  EQU  010H   ;Rx is in X-OFF hold
FEOF       EQU  040H   ;Rx has received the
```

```
                                    ;defined EOF character
FTXIM       EQU  080H               ;There is a character to
                                    ;transmit immediate
```

Exit The structure pointed to by **ES:BX** is updated. **AX** contains the error word.

$DCBPTR

Purpose This routine returns a long pointer to the DCB for the requested device.

Entry **AH** contains the device ID.

Exit **DX:AX** contains a long pointer to the DCB (offset from **DS**.) If the given device ID was invalid, **AX** contains zero.

9.2.2 Setting Up Queues

The functions described in this section initialize and flush receive and transmit queues for a device.

$SETQUE

Purpose This routine sets pointers to the receive and transmit queues for a given device and initializes the queues. **$SETQUE** is called before any communication takes place.

Entry AH contains the device ID. **ES:BX** points to Queue Definition Block (QDB) with all fields set. The QDB defines the location and size of the transmit and receive circular queues used for interrupt transmit and receive processing. The QDB structure is defined as follows:

```
QDB STRUC
    QRXADR  DW  ;Offset of RX (receive) queue in segment
    QRXSEG  DW  ;Segment of RX queue
    CBQRX   DW  ;Length of RX queue in bytes
    QTXADR  DW  ;Offset of TX (transmit) queue in segment
    QTXSEG  DW  ;Segment of TX queue
    CBQTX   DW  ;Length of TX queue in bytes
QDB ENDS
```

Exit None.

$FLUSH

Purpose This routine empties both the transmit and receive queues, discarding all data.

Entry **AH** contains the device ID.

Exit **AX** contains the error word.

9.2.3 Sending and Receiving Data

The functions described in this section control the sending and receiving of data.

$RECCOM

Purpose This routine reads a byte from the receive queue if data are ready.

Entry AH contains the device ID.

Exit If data are available, **AL** contains the input byte and **PSW.Z** is reset. Otherwise, **AX** contains the error word and **PSW.Z** is set.

$SNDCOM

Purpose This routine sends a byte to the transmit queue.

Entry **AH** contains the device ID. **AL** contains the byte to be output.

Exit AX contains the error word.

$SNDIMM

Purpose This routine sends a character immediately, in one of two ways: either the character is sent immediately to the UART, or the character is placed in a special location that causes the next transmit interrupt to send the character before sending the characters in the transmit queue.

Entry **AH** contains the device ID. **AL** contains the byte to be sent.

Exit **AX** contains the error word.

$SETBRK

Purpose This routine clamps the transmitting data line low. **$SETBRK** does not wait for any characters to finish transmitting before clamping the line.

Entry **AH** contains the device ID.

Exit AX contains the error word.

$CLRBRK

Purpose This routine releases the BREAK clamp (if any) on the transmitting data line.

Entry **AH** contains the device ID.

Exit **AX** contains the error word.

9.2.4 Detecting Events

The functions described in this section retrieve and mask an event word.

$EVT

Purpose This routine sets up an event word and an event mask.

Entry **AH** contains the device ID. **BX** contains the event mask.

Exit **DS:AX** points to the event word after masking.

$EVTGET

Purpose This routine retrieves the value of the current event word, masks it, then clears the mask.

Entry **AH** contains the device ID. **BX** contains the event mask.

Exit **DS:AX** points to the event word after masking.

9.2.5 Extended Functions

The extended functions are short functions that are routed through a common entry point.

$EXTCOM

Purpose This routine provides an entry point for a number of extended functions. These functions are called by passing the function code on entry to $EXTCOM. The following extended functions are available:

Function (Code)	
SETXOFF (1)	Produces same effect as receiving an X-OFF character.

SETXON (2)	Produces same effect as receiving an X-ON character.
SETRTS (3)	Sets RTS high.
CLRRTS (4)	Sets RTS low.
SETDTR (5)	Sets DTR high.
CLRDTR (6)	Sets DTR low.

Entry **AH** contains the device ID. BL contains the function code.

Exit **AX** contains the error word.

9.3 Sound Module

Beep *lpBlock*

Purpose This routine 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.

Parameters *lpBlock* is a long pointer to a parameter block. If *lpBlock* is NULL, the routine generates an audible beep tone.

Return Value None.

Notes Currently, Windows always passes a NULL pointer.

This routine should save any registers that it uses.

Chapter 10

Miscellaneous Support Modules

10.1 Introduction

This chapter describes a number of support modules used by Windows to carry out tasks that are OEM-specific. It explains the following support modules:

Logo Driver Modules
Disk Formatter Modules

The Old Application Support Module, WINOLDAP, is covered in Chapter 12 of this book.

The following sections explain each module in detail.

10.2 Windows Logo Driver Module

A Windows logo driver, executed as part of Windows startup process, displays a Windows logo on the user's display screen. Its purpose is to tell the user that Windows is now loading and will soon appear.

10.2.1 Installation

The Windows **setup** program installs a logo driver as part of the **WIN.COM** startup program. WIN.COM is a concatenation of the binary file WIN.CNF and the logo driver that corresponds to the selected display driver. For example, if the user selects a display driver called "HERCULES.DRV," then **setup** concatenates the logo driver, HERCULES.LGO, to the WIN.CNF file to make the Windows startup program WIN.COM.

If **setup** finds no logo driver on the install diskette, it uses only WIN.CNF to make the WIN.COM program. Windows stills operates normally but no logo is displayed when Windows is started.

10.2.2 The Logo Driver

A logo driver is a pure binary file whose code origin begins at zero. It contains routines that prepare the display screen and draw the actual logo. The WIN.CNF module controls execution of the logo driver routines. To call these routines, WIN.CNF actually makes a far call to one of two jump instructions at the beginning of the logo driver. The beginning of the driver is organized as follows:

```
LOGO    SEGMENT
```

```
ASSUME   CS:LOGO,DS:LOGO,SS:NOTHING

ORG      0000H

DB       'LOGO'                      ; This identifies the file
JMP      NEAR PTR DrawLogo
JMP      NEAR PTR RestoreScreen
```

For example, to restore the screen mode to the startup condition, WIN.CNF makes a far call to offset 7 in the logo driver.

10.2.3 Logo Driver Routines

The following explain the actions Windows expects from the logo driver routines. Each routine must be defined as a far procedure. In assembly language, a far procedure definition has the form:

```
RestoreScreen   PROC    FAR
                .
                .
                .
RestoreScreen   ENDP
```

RestoreScreen ( )

Purpose This routine restores the screen mode to a startup condition.

Parameters None.

Return Value None.

Notes The routine should preserve the contents of the **DS**, **BP**, **SI** and **DI** registers.

DrawLogo ( )

Purpose This routine draws a pretty logo on the screen.

Parameters None.

Return Value **AX** register contains the last address used to draw the logo.

Notes The routine should preserve the contents of the **DS**, **BP**, **SI**, and **DI** registers.

WIN.CNF overlays all code after the returned address with code from the WIN200.OVL module.

10.2.4 Creating the Logo Driver

Once the logo driver routines are written, you can create the logo driver by doing the following:

1. Use the Macro Assembler, **MASM**, to assemble the source file.
2. Use the standard linker, **LINK,** to create the executable file.
3. Use the executable to binary command, **EXE2BIN**, to convert the executable file to a binary file.

For example, the following command sequence produces a logo driver named "HERCULES.LGO" from the source file "HERCLOGO.ASM:"

```
masm HERCLOGO;
link4 HERCLOGO;
exe2bin HERCLOGO.DRV HERCULES.LGO
```

Once the logo driver is created, place it on the Windows **setup** diskette. **Setup** uses the module to create the **WIN.COM**.

10.3 Disk Format Module

The MS-DOS Executive uses the routines of the disk formatting module, MSDOSD.EXE, whenever the user makes a formatting request. This Windows library module must contain two routines:

Format
Sys

The following describe the routines in detail.

Format (*dSrc, dDst, lpVID*) : *Word*

Purpose This routine formats the disk in the drive specifed by *dDst* using a boot sector from the disk in the drive specified by *dSrc*. If *pVID* is not NULL, it points to a volume ID which is placed on the destination disk after formatting.

Parameters *dSrc* is an integer value specifying the drive containing a disk with a valid boot sector. Default is 0.

dDst is an integer value specifying the drive containing the disk to be formatted.

lpVID is a long pointer to null-terminated character string representing the volume label of the disk to be formatted.

Return Value *Word* is 0 if the routine is successful. Otherwise, it is non-zero.

Sys (*dSrc*, *dDst*) : *Word*

Purpose This routine copies a system disk (i.e., boot sector, bios, dos, and command) from the source disk(ette) specified by *dSrc* to the destination diskette specified by *dDst*.

Parameters *dSrc* is an integer value specifying the drive containing a disk with a valid boot sector, bios, dos, and command. Default is 0.

dDst is an integer value specifying the drive containing the disk to be formatted.

Return Value *Word* is 0 if the routine is successful. Otherwise, it is non-zero.

Chapter 11

Data Structures and File Formats

11.1 Introduction

The virtual machine that supports Microsoft Windows is comprised of several sets of predefined routines. This chapter describes the data structures and file formats used by those routines. The data structures are presented in three sections. The first section is devoted to those data structures returned by the information calls in each support module used by Windows. The second section describes those data structures that are used as parameters to calls on support modules. The third section describes data structures that are device-dependent; these structures contain information about physical devices.

The data structure descriptions in this section are intended as a guide. In the C structure declarations that follow, the word "char" stands for an unsigned 8-bit integer. The word "short" stands for a signed 16-bit integer, and the word "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.

Note 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.

11.2 Information Data Structures

GDIINFO structure

This data structure describes the characteristics of the attached graphics device in sufficient detail that GDI can allocate space for the required data structures. It is used to describe the system screen to Microsoft Windows.

The currently defined fields are:

```
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;
}
GDIINFO;
```

11.2.1 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:

---

dpVersion specifies the version number. The low byte must be 0, and the high byte must be 1.

dpTechnology
specifies the device technology from the 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
: This field 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
: This field specifies the number of planes in frame-buffer memory. For a typical frame buffer with red, green, and blue bit-planes, this field would be 3.

dpNumBrushes
: This field specifies the number of device-specific brushes supported by this device.

dpNumPens
: This field specifies the number of device-specific pens supported by this device.

dpNumFonts
: This field specifies the number of device-specific fonts supported by this device.

dpNumColors
: This field specifies the number of entries in the color table for this device.

dpDEVICEsize
: This field specifies the size of the data structure of type **PDEVICE** that must be allocated for this device.

dpCurves
: This field specifies to GDI whether the support module can perform circles, pie wedges, chord arcs, and ellipses; whether 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 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 both wide and styled
bit 7 - set means can do interiors
```

The high byte must be 0.

dpLines This field specifies whether the support module can perform polylines, and lines; whether 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 lines
bit 1 - set means can do polyline
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 both wide and styled
bit 7 - set means can do interiors
```

The high byte must be 0.

dpPolygonals
This field specifies whether the support module can perform polygons, rectangles, and scanlines; whether 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 polygon
bit 1 - set means can do rectangle
bit 2 - set means can do winding number fill polygon
bit 3 - set means can do scanline
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 both wide and styled
bit 7 - set means can do interiors
```

The high byte must be 0.

dpText This field specifies what level of text support is provided by the support module. 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)
```

Each precision level (STRING, CHARACTER, NONE, 90, etc.) has a corresponding bit in dpText which 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. Hence, 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.
```

All of the available abilities are described in the sections following the description of the remaining **GDIINFO** fields.

dpClip This field indicates that clipping is available to the device. If the field is 1, the device can clip to a rectangle in the **Output** routine. If the field is 0, it cannot clip.

dpRaster This field specifies raster abilities.

```
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 Bytes
bit 4 - set means device supports the new (Windows 2.0) output functions
bit 5 - set means display context has state block
bit 6 - set means device can save bitmaps locally in 'shadow' memory
```

The new output functions mentioned in bit 4, above, are **ExtTextOut** and **FastBorder**.

dpAspectX, dpAspectY, dpAspectXY

These fields 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 screen these fields are 5, 12, and 13, respectively (that is, every pixel is a 5 by 12 rectangle). This corresponds 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
This field 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
This field is a constant specifying the width and height of the metric (low resolution) window. Width is *HorzSize**10; height is *VertSize**10.

dpMLoVpt This field is a constant specifying the horizontal and vertical resolutions of the metric (low resolution) viewport. Horizontal is *HorzRes*; vertical is -*VertRes*.

dpMHiWin This field is a constant specifying the width and height of the metric (high resolution) window. Width is *HorzSize**100; height is *VertSize**100.

dpMHiVpt This field is a constant specifying the horizontal and vertical resolutions of the metric (high resolution) viewport. Horizontal is *HorzRes*; vertical is -*VertRes*.

dpELoWin This field is a constant specifying the width and height of the English (low resolution) window. Width is *HorzSize**1000; height is *VertSize**1000.

dpEloVpt This field is a constant specifying the horizontal and vertical resolutions of the English (low resolution) viewport. Horizontal is *HorzRes**254; vertical is -*VertRes**254.

dpEHiWin This field is a constant specifying the width and height of the English (high resolution) window. Width is *HorzSize**1000; height is *VertSize**1000.

dpEHiVpt This field is a constant specifying the horizontal and vertical resolutions of the English (high resolution) viewport. Horizontal is *HorzRes**254; vertical is -*VertRes**254.

dpTwpWin
This field is 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 This field is 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 pixel per logical inch along a horizontal line on the display surface.

dpLogPIxelsY
This 2-byte field specifies the number of pixel per logical inch along a vertical line on the display surface.

dpDCManage
This 2-byte field contains the following bits:

```
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 it's 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.

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 measure, either metric or English. These numbers should be integers in 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.

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 graphics at 300 dpi are scaled to 75 dpi. Any device 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. Thus a scale factor of 2 means reduce by 4 and a scale factor of 1 means reduce by 2.

11.2.2 GDIINFO – dpText Field Precision Levels

OutputPrecision (STRING, CHARACTER, STROKE)

OutputPrecision specifies which font attributes the output routine 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:

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 doesn't 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 can't 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 others 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 don't 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 can't 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 **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. This ability is not affected by output precision.

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 output precision.

UnderlineAbility (UNABLE, ABLE)

UnderlineAbility is set ABLE if **Strblt** can underline a font. This ability is not affected by output precision.

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 output precision.

RasterFontAble (UNABLE, ABLE)

RasterFontAble indicates whether the device is capable of using raster format fonts.

VectorFontAble (UNABLE, ABLE)

VectorFontAble indicates whether 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.

11.2.3 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;
```

Below is a description of the fields in this structure:

msExist	This field is nonzero if the device initialization code was able to find and initialize a mouse device.
msRelative	This field is 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	This field identifies how many buttons are on the installed mouse. For the IBM PC, with a Microsoft Mouse, this field is set to 2.
msRate	This field specifies the maximum number of hardware interrupts per secont that the mouse can generate. For the IBM PC with the bus version of the Microsoft Mouse, this field is 34.
msXThreshold	This field 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.
msYThreshold	This field 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	This field is reserved.
msYRes	This field is reserved.

11.2.4 KBINFO – Keyboard Hardware Characteristics Structure

The values of the fields in this structure should be modified so that they accurately describe the characteristics of the attached keyboard. Some of the information will be filled in by the device initialization code. The order of these fields should not be changed, only their values, as a copy of this data structure will be given to Microsoft Windows when it starts up.

```
typedef struct  {
    char    kbFarEastRange[4];
    short   kbStateSize;
    short   kbNumFunc;
    short   kbHasBreak;
    short   kbRate;
}
KBINFO;
```

Below is a description of the fields in this structure:

kbFarEastRange	This 4-byte field specifies the two ranges of ASCII values form the keyboard. The bytes are organized as Begin_First_Range, End_First_Range, Begin_Second_Range, End_Second_Range. ASCII values within either range are the first byte in a double byte character.
kbStateSize	This field specifies the number of bytes of state information required by the *ToAscii* routine. For the routine supplied with the IBM PC, this field is 8.
kbNumFunc	This field identifies the number of function keys on the keyboard. For the IBM PC, this field is 10.
kbHasBreak	This field is nonzero if the keyboard hardware generates one code for a key going up and another code for a the same key

	going down. If this field is zero, keys generate events on down transitions only.
kbRate	This field specifies the maximum number of hardware interrupts per second that the keyboard can generate. For the IBM PC, this field is 10.

11.2.5 TIMERINFO – Timer Information Data Structure

This data structure contains information about the resolution of the system's timer.

```
typedef struct  {
    long   tiResolution;
}
TIMERINFO;
```

The field in this data structure has the following meaning:

tiResolution	32-bit integer that specifies the resolution of the timer, in microseconds. For the IBM PC, this is 54926 microseconds, or approximately 18.2 times per second.

11.2.6 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 meaning:

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.

11.2.7 DCB – Device Control Block Structure

RS232 configuration parameters are communicated in a Device Control Block. The Device Control Block (DCB) 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;          /* 0,1,2,3, 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:

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
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, disable 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, enable 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 will 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 character of being empty.

fPeChar — If set, indicates that characters received with parity errors are to be replaced with the specified PECHAR.

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.

11.3 Parameter Data Structures

These data structures are those actually used as parameters to routines described in this document.

11.3.1 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;
```

x	is the X-coordinate value of a point.
y	is the Y-coordinate value of a point.

11.3.2 RECT – Rectangle Data Structure

A rectangle is characterized by two points.

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

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.

Note

The "right, bottom" coordinates are actually one greater than the actual size of the rectangle would appear to require. Thus, 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).

11.3.3 RGB – Logical Color Specification

A logical color specifies the color desired by an application.

```
typedef long RGB;
```

The long integer is divided into four one-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: $r + 2^8*g + 2^{16}*b$.

That is, the lowest-order byte contains Red information, the next byte contains Green information, and the third byte contains Blue information. The fourth byte is not used.

Each byte represents an intensity level for the specified color (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 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 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.

11.3.4 DRAWMODE – Drawing Mode Specification

A drawing mode includes 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:

Rop2 Is a short integer value between 1 and 16 specifying the Boolean combining function (of source and destination colors) to be used. All 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      Functions of Dest and Pen
                                (or Pattern)
#define     R2_BLACK            1       /*  0       */
#define     R2_NOTMERGEPEN      2       /*  DPon    */
#define     R2_MASKNOTPEN       3       /*  DPna    */
#define     R2_NOTCOPYPEN       4       /*  Pn      */
#define     R2_MASKPENNOT       5       /*  PDna    */
#define     R2_NOT              6       /*  Dn      */
#define     R2_XORPEN           7       /*  DPx     */
#define     R2_NOTMASKPEN       8       /*  DPan    */
#define     R2_MASKPEN          9       /*  DPa     */
#define     R2_NOTXORPEN       10       /*  DPxn    */
#define     R2_NOP             11       /*  D       */
#define     R2_MERGENOTPEN     12       /*  DPno    */
#define     R2_COPYPEN         13       /*  P       */
#define     R2_MERGEPENNOT     14       /*  PDno    */
#define     R2_MERGEPEN        15       /*  DPo     */
#define     R2_WHITE           16       /*  1       */
```

BackgroundMode
Is 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
Is a physical color specifying the background color to be used.

TextColor Is a physical color specifying the text color to be used.

The remaining fields in **DRAWMODE** specify text justification as follows:

TBreakExtra Is 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.
BreakExtra	Is a short integer specifying the number of pixels to insert at every character break: div (*TBreakExtra, BreakCount*).
BreakErr	Is 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	Is a short integer specifying the remaining pixels to be scattered among the character breaks: mod (*TBreakExtra, BreakCount*).
BreakCount	Is a short integer specifying the number of character breaks into which the extra pixels specified by *TBreakExtra* must be inserted.
CharacterExtra	Is a short integer specifying in pixels (or device units) the amount of extra space to put between characters output by **Strblt**.
LogicalBGColor	(New field) Specifies the logical background color.
LogicalTextColor	(New field) Specifies the logical text color.

Notes

If no justification is required, *TBreakExtra* will be set to 0. 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
```

11.3.5 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

binary operators	*and*, *or*, and *xor*;
unary operator	*not*

on three variables. Those combining functions are listed in Appendix D, where the actual name and reverse Polish notation for each value are given. Note that the actual values used to denote the 256 functions are 32-bit unsigned integers, so the table of values is sparse, and one must be careful when specifying the **RASTEROP.**

Appendix D, "Raster Operations", 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                          */
```

11.3.6 CURSORSHAPE – Cursor Data Structure

A cursor 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:

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 hot spot can lie outside the cursor shape.
csWidth	Width of the cursor shape, in pixels.
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.

11.3.7 LOGPEN – Logical Pen Attribute Information

The logical pen attribute structure is used by the support module while drawing lines or perimeters.

```
typedef struct  {
    long    lopnStyle;
    POINT   lopnWidth;
    long    lreserved;
    long    lopnColor;
}
LOGPEN;
```

The fields within the **LOGPEN** structure have the following meanings:

lopnStyle	Is a long integer value specifying the type of interruptions to be used in generating the pen. Predefined pen styles include: solid, dashed, dotted, dash-dotted, dash-dot-dotted, and null, with indexes 0-5 respectively.
lopnWidth	Is 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.
lreserved	Is a long integer reserved for future use.
lopnColor	Is a long integer specifying the RGB color with which the pen is to be drawn.

11.3.8 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:

lbStyle	Selects the type of brush. Predefined brush types include: BS_SOLID, BS_HOLLOW, BS_HATCHED, and 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 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.
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:

```
HS_HORIZONTAL       - horizontal hatch
HS_VERTICAL         - vertical hatch
HS_FDIAGONAL        - 45-degree upward hatch from left to rig
HS_BDIAGONAL        - 45-degree downward hatch from left to r
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.

11.4 Physical Data Structures

This section describes data structures that contain information about physical devices. These data structures vary across devices.

11.4.1 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 support module. 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;
```

There is one constant field in this data structure, which 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* procedure sets this field to a nonzero value.

11.4.2 BITMAP – Physical Bitmap Data Structure

BITMAP – Physical Bitmap Data Structure

A physical bitmap describes a rectangle of bits in main memory (private bitmap).

```
typedef struct {
    short   bmMagic;
    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:

bmMagic	If the memory backing the bitmap is located 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, *bmMagic* is a unique number describing that physical display, and the remaining information in the data structure is as defined by the OEM (see **PDEVICE** structure above). 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 remaining fields are initialized by the OEM-supplied *enable* procedure 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 scan lines will be aligned on a word boundary. Note that *bmWidthBytes**8

	can exceed *bmWidth* in the case of a bitmap whose width in bits is not a multiple of 16.
bmPlanes	This field specifies the number of planes in frame-buffer memory.
bmBitsPixel	This field specifies the number of adjacent bits on each plane which are involved in making up a pixel.
bmBits	This is 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	This field specifies the width in bytes of each plane involved in making up the bitmap. It is equal to *bmWidthBytes*bmHeight*.
bmlpPDevice	This field is a long pointer to the **PDEVICE** structure of the device for which this bitmap is compatible.
bmSegmentIndex	for bitmaps that are greater than 64K bytes in length, this field is nonzero (in Windows 2.XX, 1000H). Otherwise, it is zero. It is used as a flag to tell you whether 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	This field 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 *bmHeight / bmScanSegment*. A raster line is equal to *bmWidthBytes* bytes. No segment may contain more than 64K bytes.
bmFillBytes	This field specifies the number of extra bytes in each segment. Segments are multiples of 16-byte paragraphs.

A Huge Bitmap Example

This example is for a display, such as the EGA, which registers itself (in the **GDIINFO** data structure) as having more than one bitplane. First, the bitmap in RAM:

Diagram 1: Huge Bitmap, in RAM

All planes of a particular scan line must fit within the segment, else we go to the next segment, leaving *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:

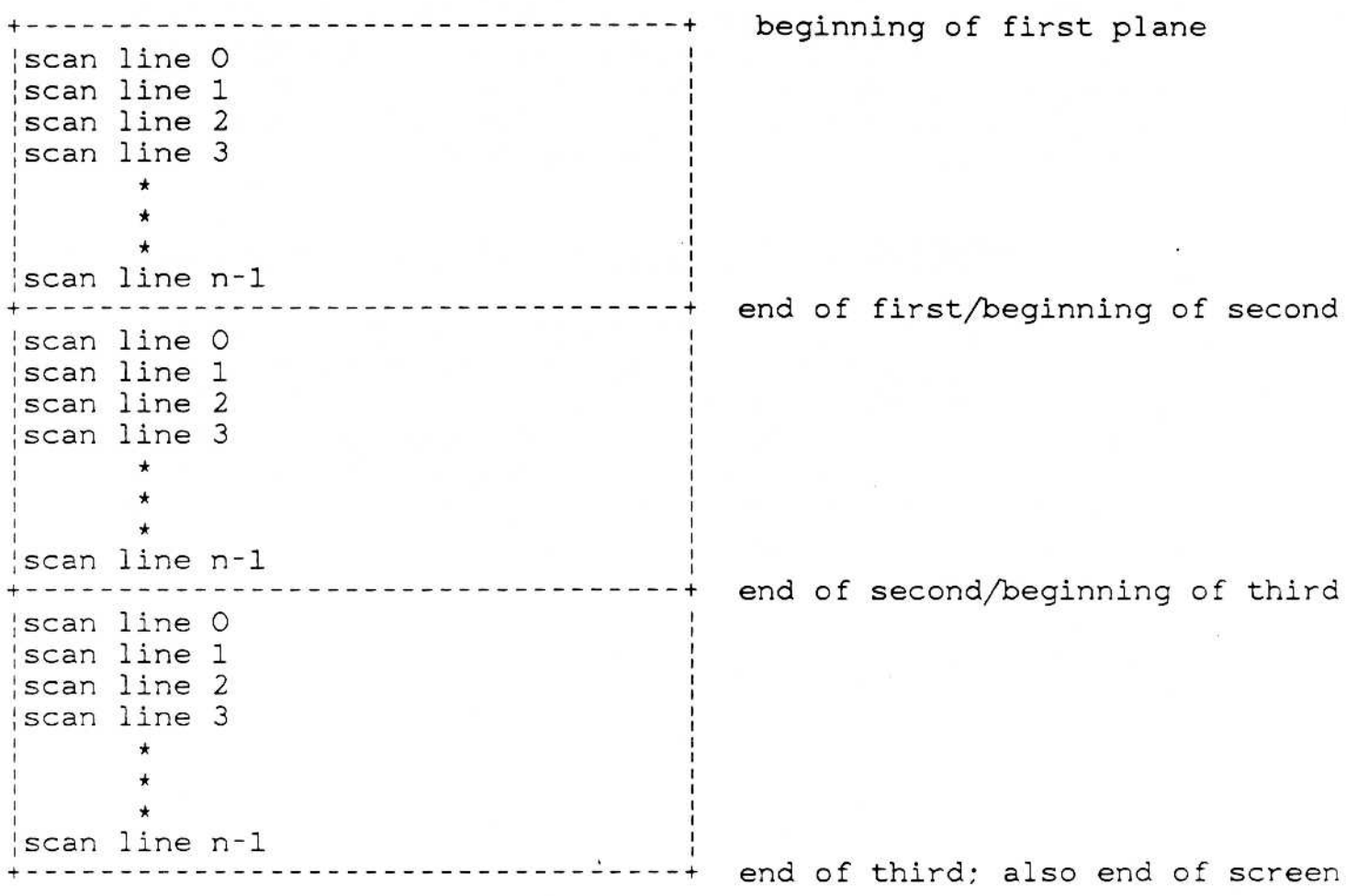


Diagram 2: EGA Format

11.4.3 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 appropriate output routines to be realized by the device support module.

11.4.4 PPEN – Physical Pen Data Structure

The **PPEN** structure is filled by the **RealizeObject** routine and passed to the **Output** routine to specify the physical pen to be used for drawing lines. The exact size and content of a physical pen depends on the device for which it is formed. The **RealizeObject** routine must fill this structure with appropriate values by translating the logical pen definition passed to it into physical pen specifications for the device. GDI allocates sufficient space for the structure before calling **RealizeObject**.

11.4.5 PBRUSH – Physical Brush Data Structure

The **PBRUSH** structure is filled by the **RealizeObject** routine and passed to the **Output** routine 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** routine must fill this structure with appropriate values by translating the logical brush definition passed to it into physical brush specifications for the device. GDI allocates sufficient space for the structure before calling **RealizeObject**.

Chapter 12

The Windows OEM Development Environment

12.1 Introduction

This chapter tells you how to create and use the OEM Development Environment to create device drivers.

12.2 The Development Environment and Device Drivers

The following sections describe the process of Device Driver construction in some detail.

Setting Up Your Environment

Note

In order to make sure that you build your device drivers correctly, you must set up your hard disk to a fairly rigid standard. If you have tools from the WINDOWS 1.xx device driver kit and/or software development kit, you will need to purge these tools, libraries, and include files before beginning work on WINDOWS 2.0 You must also take care not to mix the tools and libraries from the WINDOWS 2.0 Software Development Kit with those from this kit.

In addition to the tools and files included with this kit, you will also need the following products:

- The Microsoft Macro Assembler v5.00
- The Microsoft C Compiler v5.00 (if you are going to write printer drivers. This product is NOT needed if you are only going to write display drivers.)
- Microsoft WINDOWS v2.0
- A Microsoft Mouse or other mouse compatible with WINDOWS.

Note

It is NOT necesary to purchase a WINDOWS 2.0 Software Development Kit if you only wish to write device drivers. The Software Development Kit is only intended for use when writing WINDOWS applications.

Step 1 - Install the Languages

You should install the MASM and C compilers according to the instructions included with them. (Note to C users -- you only need the small model combined library SLIBCE.LIB to build the printer drivers included with this kit).

Step 2 - Install the Build Tools:

In the same directory as you installed the MASM and C executable files, you should now copy the files from the TOOLS diskette, directory \UTILS\BUILDBIN. These files supercede the files included with MASM and C and assure that you will correctly build the sample device drivers. Note that the new LINK.EXE included with this kit encompasses the features found previously in both the old LINK.EXE and LINK4.EXE. You should delete any copies of LINK4.EXE that you may have on your path.

Step 3 - Check Your Work

Start up the tools MAKE, LINK, MAPSYM, and RC to make sure that you have everything installed correctly. The version banners printed out should be:

```
MAKE -- v4.06 or above (the one included with MASM v5.0 is OK)
LINK -- MUST BE v5.01.02
RC   -- MUST BE v2.01
MAPSYM -- MUST BE v3.10
SYMDEB -- MUST BE WINDOWS v2.00
MASM -- Version 5.00 or above
C Compiler -- Version 5.0 or above
```

Step 4 -- Install the Libraries and Include Files

Now install the files from the \LIB directory on the diskettes to your LIB path. Note that the linker now finds libraries on your LIB environment variable that you probably set in your AUTOEXEC.BAT. Some of the printer and other driver sources have **.LIB** files of their own. You may copy these to your LIB environment as needed to build each driver. For display drivers you need only **SWINLIBC.LIB**.

Warning

MAKE SURE THAT YOU GET RID OF ANY OLD LIBRARIES FROM WINDOWS 1.xx!!!

Next, install the files from the \INCLUDE directory to your INCLUDE path. Note that MASM v5.00 is now able to search the INCLUDE environment variable set in you AUTOEXEC.BAT file. You should include the line in AUTOEXEC.BAT

```
SET INCLUDE=pathname
```

Where **pathname** is the full path to your INCLUDE directory.

Step 5 -- Try Out the WINDOWS 2.0 DEBUG Version

Included in this kit is the debugging version of WINDOWS 2.0 with support for the fully populated EGA color card and monitor, the Microsoft Mouse, and the standard IBM AT machines. You should set up a \WINDOWS\TEST directory and copy all of the files from the \WINBOOT directory to your hard disk. Then, type:

```
C:\WINDOWS\TEST> kernel
```

The debugging version of WINDOWS should come up on your screen. If it doesn't, you've done something wrong. You can run WINDOWS quite nicely from this debugging version. However, you'll need to get a bona-fide copy of WINDOWS v2.0 to run and test your driver with the WINDOWS standard applications. While writing your driver you should **NOT** run the WINDOWS 2.0 SETUP program into the \WINDOWS\TEST directory. Simply copy WINDOWS applications that you wish to test into the \WINDOWS\TEST directory from the Desktop Applications and Write disks included with WINDOWS 2.0.

Step 6 -- Try building a sample driver

The acid test of whether your environment is set up correctly is whether you can build the sample display and printer drivers included with this kit. We have included sample driver code that builds the:

```
EGA High-Resolution Color and VGA drivers
CGA and Hercules Drivers
EGA High-Resolution Monochrome driver
HP Laser Jet Driver
IBM Graphics Printer Driver
```

Each disk is fully self-contained. Simply copy the diskette which contains the driver that you wish to build into a sub-directory on your hard disk and follow the instructions in the MAKE file for that driver. It is very important that you read the header for the MAKE file before attempting to build the driver. It contains last-minute building information that you'll need to know about.

Before attempting to build a driver, you must first build the corresponding resource for the driver and copy it into the driver's build directory. If you have problems building the sample driver, check your environment. A common problem in building the driver results in a message during the MAKE that it can't find the xxxx.RES file for the driver. This is not a serious problem. Make sure that the resource has been built and is in the correct directory, and try again. The sample drivers were built and thoroughly tested before shipment, and are known to work.

Step 7 -- Try the Driver You Have Built

You now have a display driver called CGA.DRV, HERCULES.DRV, EGAHIRES.DRV etc. If you rename the .DRV file to DISPLAY.DRV and copy it into your \WINDOWS\TEST directory and run KERNEL, you'll see whether you built the driver properly. If you've built a printer driver, you'll have to run the WINDOWS program CONTROL.EXE (included in the WINDOWS 2.0 retail kit) to install your driver for testing (the instructions for CONTROL are included in the WINDOWS 2.0 retail kit).

Step 8 -- Modify the Driver Sources to Fit Your Device

Now that you have established and confirmed your working environment, it's time to write your own driver. It is probably not a good idea to start writing a driver from scratch since it's a fairly complex process. Display driver writers should note that you must support ALL Output, BitBlt, StrBlt, ScanLR, and Pixel operations to your screen in its color format and in converting monochrome format bitmaps into your screen's color format. You must also support these functions to main memory bitmaps in both your screen's color format and monochrome (1 plane 1 bit per pixel) format.

Because the EGA Monochrome driver is non-interlaced, 1 plane, 1 bit per pixel, you can "steal" its code to do drawing into monochrome formatted bitmaps. You should remove its cursor exclusion code and machine dependent code first, as these are unnecessary for drawing into main memory monochrome format bitmaps. You may also wish to remove some special casing from the StrBlt code since the ability to BLT strings to main memory is not too time critical and the special casing is large. After creating your display driver, rename it to DISPLAY.DRV, rename your symbol file to DISPLAY.SYM, and copy them to the \WINDOWS\TEST directory. You can then debug it using the DEBUG version of WINDOWS by

typing in the command:

```
C:\WINDOWS\TEST> SYMDEB DISPLAY.SYM KERNEL.EXE
```

SYMDEB will load in your display driver and symbols and you will then be able to debug your driver on a second monitor or terminal. See the SYMDEB documentation for details on how to use the second monitor or terminal.

12.2.1 Notes on the Resources and Fonts Included in this Kit

Because many device manufacturers are producing WINDOWS drivers at very high resolutions (e.g., 1024 by 768), we have included fonts and resources for a variety of resolutions.

Although we have not included the driver code for it in this kit, we have produced internally an IBM 8514 high-resolution device driver for WINDOWS. We have included in this kit the resources and system font (but not the terminal font) that we produced for the 8514 driver. The \RESOURCE\1024X768 directory contains resources suitable for display devices similar to the 8514's 1024 by 768 -- 1 to 1 aspect ratio display. Similarly, the system font included in the \FONTS directory is appropriate for this resolution. An appropriate terminal font will become available in the near future.

12.2.2 How To Install Fonts Using the DEBUG Version of WINDOWS

After building your font files, you can copy the system font to the \WINDOWS\TEST directory, renaming it to FONTS.FON. The terminal font is copied to OEMFONTS.FON. The DEBUG version will then recognize your new fonts and utilize them correctly.

12.2.3 How to Create a Shippable Device Disk

After completing the task of writing your WINDOWS driver, you will need to create a "Device Disk" to send to your customers. They will run SETUP (included with the WINDOWS 2.0 retail kit) and be prompted to insert your "device disk" when asked for the display device, the system font and terminal fonts. The following files must be on your device disk:

- a descriptive name for your device driver -- NOT "DISPLAY.DRV"!!

- a descriptive name for your System Font file -- NOT "FONTS.FON"!!
- a descriptive name for your Terminal font file -- NOT "OEMFONTS.FON"!!

  Since terminal fonts are specific to National Language adaptations of WINDOWS, the letters cc in the terminal font name should contain an abbreviation for the country that the font is intended to be used in. For example, for shipping an 8514 driver to be used in the U.S., it would be reasonable to name the terminal font file 8514US.FON. In any case, it must have a name distinct from that of the System font.
- the logo file that gives the pretty screen when WINDOWS is loading. Most people just rename EGA.LGO to the name that corresponds to their driver. The name of this file must be the same as that of the .DRV file.
- the old application support module. The name of this file must be the same as that of the .DRV file.

Chapter 13
Winoldap - Old Application Support

13.1 WINOLDAP.MOD - Support of old MS-DOS applications in Windows

The following topics cover the functional and implementation aspects of the Winoldap features which are incorporated in the Microsoft Windows 2.0 release.

These features can be organized into the following groups:

- EEMS/EMS support in the extended-mode memory manager.
- Winoldap DDE interface.
- OEM character set filter.
- MS-WORD support in Winoldap

This document assumes a strong familiarity with the Winoldap module.

13.1.1 MEMORY MANAGER

OVERVIEW

Winoldap supports an extended mode in which the Windows environment is swapped out to disc when executing a level-2 old application. The memory previously occupied by Windows is freed and can be used by the running old application. This approach thus allows large DOS applications to run and be context switched under Windows. Winoldap treats the swap disc just like a standard non-removable DOS block device (hard-disc, RAM disc).

Along with the implicit support of EEMS memory in Windows 2.0, it is apparent that configuring a large portion of the EEMS RAM as a swap disk for Winoldap is a wasteful and inefficient use of memory resources. Instead, Winoldap (in extended mode) should be enhanced to manage the EEMS memory dynamically. By taking advantage of the EEMS capability to bank switch user memory (below 640 Kb), Winoldap can context switch almost instantly between a large level-2 DOS application and Windows.

To understand how the memory manager exercises the bank switching scheme, it may be relevant to review briefly the way memory banks are allocated to Winoldap by the Windows kernel under an EEMS configuration. The following discussion is confined to the loading and execution of a level-2 DOS application in the extended mode.

Memory is offered to Winoldap in several phases:

1. At the start of a Winoldap instance: Windows allocates a set of memory pages to Winoldap code and data segments and maps them to some available physical system addresses. Code and instance data don't have to be contiguous.
2. During the loading of the level-2 application: Winoldap loads the grabber module either to the memory pages assigned to its code and data or to newly allocated pages by requesting a globalAlloc from Windows. Mapping of the grabber memory to the system address space is done transparently by Windows.
3. Before entering the extended mode: more memory pages are allocated to a group of data buffers needed mainly to save the machine state during a context switch. Those buffers are:
   - Switch buffer: for saving the screen and video states.
   - Grab buffer: for capturing a screen snapshot.
   - Copy buffer: for temporary use during a Copy operation.
   - HPSystem buffer: for saving the HP-specific machine state.
   - SFT buffer: for expanding the DOS System File Table.

In extended mode, Winoldap relocates itself and its data buffers to low memory and thus will overlay the Windows code. Once residing in low memory, Winoldap will massage the free DOS memory arena to the required size as specified in the DOS application PIF. Winoldap finally loads and executes the DOS application. Before overlaying the Windows code and memory arena, Windoldap has to save that context so that it can be restored after a context switch back from the DOS application. Without EEMS, that Windows context is saved into a unique swapping file. Winoldap will read that file when the Windows context needs to be restored. With EEMS present in the system, the same operation becomes more complex and can be broken down into the following steps:

1. To be able to make use of all logically available memory, Winoldap will try to relocate itself to the TopPDB (Process Data Block). This control block is maintained by DOS and is used to load Windows. The Windows environment is located after the TopPDB and grows toward the top of memory. In a configuration of both conventional and EEMS memory coexisting in the user space (below the 640 Kb range), the TopPDB will likely be located in the conventional memory range. Thus, relocating Winoldap is not a simple remapping of its EEMS pages. First, Winoldap must save the conventional memory range starting from TopPDB which will be overlaid by the Winoldap module. This is achieved by allocating some EEMS temporary pages, mapping them to the EMS pageframe window, and then copying the Windows context residing in the conventional memory to those pages via the pageframe window. A

portion of the Winoldap module which fits the conventional memory size starting from the TopPDB is then copied down to low memory.

2. Calls to the EEMS memory manager are made to save the current mapping context of the Windows environment. Pages assigned to the remaining part of the Winoldap module are then remapped starting from the beginning of the EEMS system space. In most cases, all Winoldap data buffers are remapped down to low memory except the switch buffer. The latter is only needed to save to video context after a switch back from the running DOS application. At that point in time, the entire Windows environment is already restored. Not having to relocate the switch buffer has the definite advantage to minimize the Winoldap size in the extended mode. The switch buffer can be as large as 32Kb in the case of the Multimode video subsystem.

3. Winoldap requests from the EEMS memory manager for enough pages to match the required size specified in the DOS application PIF. Those pages are then mapped to a contiguous system memory space right after the relocated Winoldap module.

Figure 1. Winoldap EEMS memory map in Extended mode

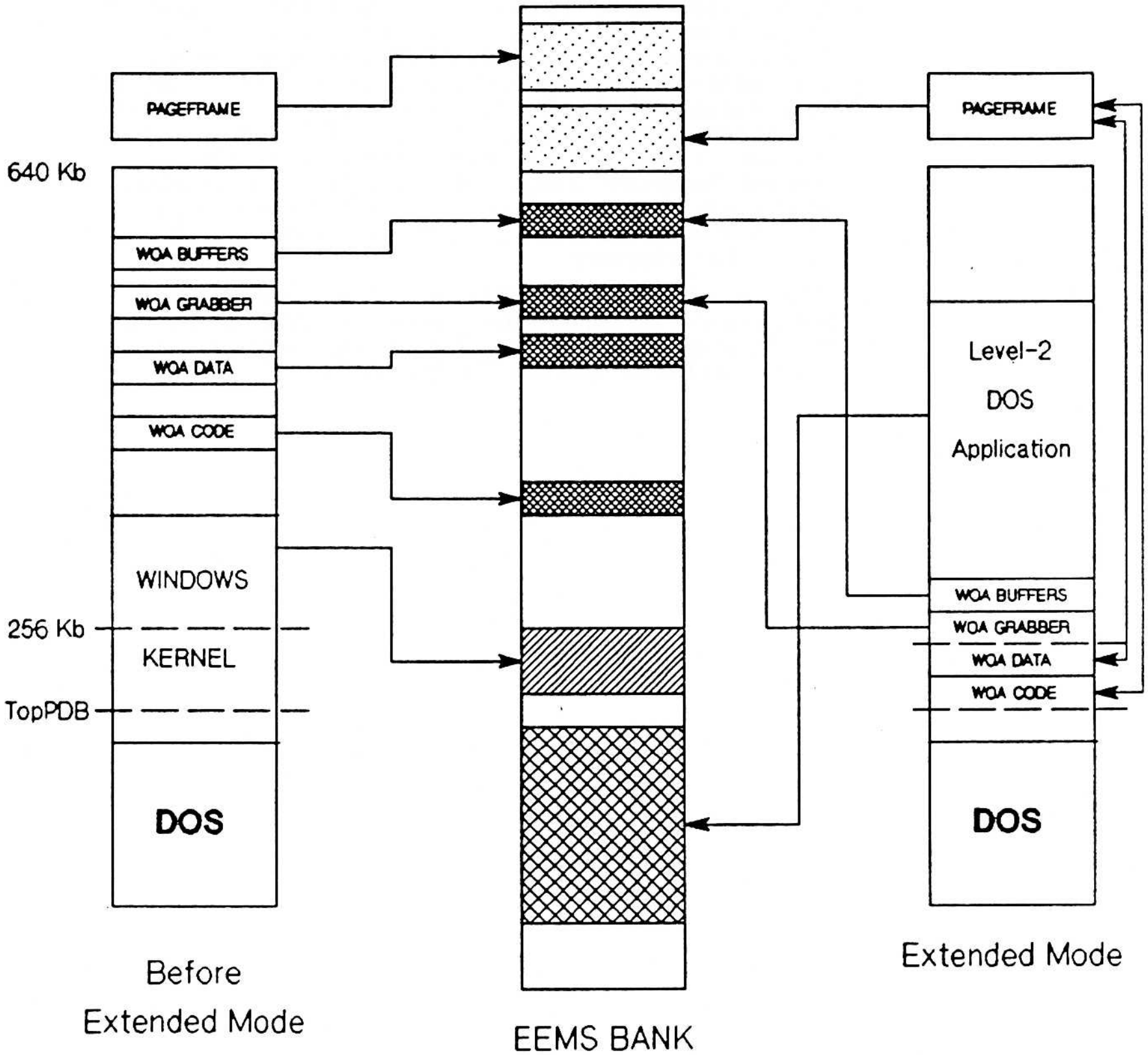


Figure 1 illustrates the system memory map before and after moving to the Extended mode. The picture is only for functional representation and is therefore drawn to scale. It is assumed that memory below the 256 Kb line is conventional, and above is EEMS. For clarity and simplicity, the Winoldap code is shown fitting nicely between the TopPDB and the 256Kb bounds, and the Winoldap components as well as the DOS application arena are mapped to a set of contiguous EEMS pages.

On a context switch back from the DOS application to Windows, a reverse memory management process takes place. The DOS application current mapping context is saved, including the EMS pageframe mapping (this is needed to support DOS applications which take advantage of expanded memory such as Lotus 123 Ver 2.x). If the DOS application memory map contains both conventional and EEMS memory, the conventional part is saved into EEMS temporary pages via the EMS pageframe physical window. Those pages will be copied back to the same conventional memory area when a context switch from Windows to the DOS application happens. The portion of Winoldap that resides in the conventional memory is copied back to its corresponding EEMS pages. The Winoldap original mapping is restored and control is transferred back to the original Winoldap. The Windows code which is previously located in the conventional memory is restored from its associated save EEMS pages via the pageframe window. Finally, the entire previous Windows context mapping is reinstated.

CODING RESTRICTIONS

Because of the particular nature of the Extended mode where Winoldap is relocated to low memory, some restrictions are imposed on all Winoldap code modules that may be accessed during the execution of the DOS application. These modules are the memory manager, DOS traps, Macro/Menu processor, Paste handler, and the screen grabber:

1. Do not issue any Windows system calls, because the Windows environment is already swapped out.
2. Do not make any **FAR** procedure calls or jumps inside Winoldap. Winoldap is relocated, so its code segment is not at a fixed location.
3. Do not make any variable references which require fix-up operations by the linker. Winoldap data is also relocated, and fix-ups resolved at loading time will not be valid in the extended mode.
4. Data blocks which are needed in the Extended mode must be allocated at startup time and their size must stay unchanged during the existence of the DOS application. Their starting address in the form Segment:0 must be saved in unique **PUBLIC** variables. References to those data blocks must be done by using the associated variables. The Winoldap memory manager keeps track of

these address variables and performs the necessary fix-ups when switching to the extended mode.

WINOLDAP DDE INTERFACE

INTRODUCTION AND OBJECTIVES

The WinOldAp DDE interface provides a programmatic interface that allows a Windows application, called a "shell", to manipulate an old MS-DOS program in the same manner as a user would under Windows. Furthermore, the interface contains features such as downloadable macros and menus which can assist the user. In addition, these features enables the shell to completely encapsulate the old application, allowing the user to only access the application through the shell. Both level 1 and 2 MS-DOS applications will work with shells. A shell can control more than one WinOldAp instance, but a WinOldAp instance can only be controlled by one shell at a time.

The interface's primary goal is to provide enough functionality, flexibility, and simplicity to warrant the writing of shells. Simplicity in this case means simplicity in the shell program. Functionality and flexibility means giving the shell writer enough power to write a useful shell. The protocol has a somewhat confilicting secondary goal to be extensible to allow for additional features which cannot be included in the first release due to schedule constraints. Finally, the interface tries to be consistent between different application levels.

SAMPLE SESSION

In the typical case, the shells will be active before the old application is invoked, so WinOldAp must notify each shell of its presence. It does so by starting a "Identify" conversation with the old application name as the topic. In the Identify conversation the shell and WinOldAp exchange information about each other.

In the Identify conversation the initiating WinOldAp is in control, contrary to natural order of the relationship where the shell should be in control. Once the shell and WinOldAp establish each other's identity, the shell can take control of the relationship by exchanging roles with WinOldAp; WinOldAp becomes the passive server and the shell becomes the active client. The exchange takes place when WinOldAp terminates the Identify conversation and the shell initiates a new Control conversation.

Once the Control conversation is established, the shell can send WinOldAp macros and menu items using the WM_DDE_POKE message. Usually, these exchanges will be done when the conversation is first established, although it can be done any time during the conversation within certain

limitations. The shell can also request information from WinOldAp like application data, current menu items, and current macros. Finally, the application can request to be advised when certain user actions are performed such as changes in the old application state and use of macros.

When the shell is ready to start the application, it issues an WM_DDE_EXECUTE which tells WinOldAp to continue execution of the old application. The execute message can also contain a command macro which will be executed before the user gets control of the old application.

As an example dialog, the diagram below outlines the messages sent when a shell adds 2 macros to a Winoldap menu before the application is executed. The ACK messages are not shown and should be implied.

```
          Shell                          WinOldAp
-------------------------------------------------
-- INITIATE ("AppName") ---     ; Establish identities
<-- REQUEST ("ShellInfo") --
-- DATA ("ShellInfo") ---
-- POKE ("WinOldApInfo") --
<-- TERMINATE --------

-- INITIATE ("AppName") --      ; Establish conversation
-- POKE ("Macro", Macro1) ->    ; Add menu macros
-- POKE ("Macro", Macro2) ->
-- POKE ("Menu", Menu1) ->      ; Add menu items
-- POKE ("Menu", Menu2) ->
-- ADVISE ("OldApState") ->     ; Advise when the app
-- EXECUTE ( ) ---------->      ; Start the old app

-- DATA ("OldApState") ---      ; User context switches

<-- TERMINATE --------      ; The old app wants to terminate
-- TERMINATE -------        ; The old app terminates
```

CONCEPTS

FULL DDE COMPLIANCE

This protocol intends to be in full compliance with the DDE standard. In this way, any enhancements to the DDE protocol, such as extensions for networks, will be leveraged. Among other requirements, compliance entails using all the DDE fields in their intended manner, adopting the execute command syntax for macros and using the proper message functions.

NAMING

Each data object, event, and action has a unique name. Objects are accessed using atoms formed from their name. Predefined objects can have english names (eg "WindowInfo") or integer names (eg "#1024"). Objects created for a shell have integer names which correspond to their identification number. Identification numbers of objects are returned in the acknowledgement message of their creation operation. For simplicity, names are not overloaded. That is, the same name does not refer to different objects depending on the operation. The currently recognized names are:

Name	Description
MenuList	List of current menu items
Menu	The menu manager
#2048 to #3071	Individual menu items
MacroList	List of current macros
Macro	The macro librarian
#1024 to #2047	Individual macros
WindowInfo	The state of the old applicaiton window
WinOldApInfo	WinOldAp information
ShellInfo	Shell information
OldApState	The state of the old application

SEQUENTIAL SHELLS

To prevent control conflicts, a WinOldAp instance can only be controlled by one shell at a time. However, under certain situations it may be desirable for many shells to have data objects present in a given WinOldAp instances. To accommodate this scenario, the protocol includes a sequential shell concept where WinOldAp deterministically gives a sequence of atemporaryn shells control before it gives control to a final apermanentn shell. Temporary shells perform their WinOldAp operations and terminate the control conversation before activating the old application with an execute command. WinOldAp gives control to temporary shells in the following sequence:

Type	Description
General	Shell is used for lots of applications

Specific	Shell has application specific knowledge
Execing	Shell which invoked the old application

If more than one permanent shell requests final control, WinOldAp prioritizes the shells in the following manner:

Type	Description
Execing	Shell which invoked the old application
Specific	Shell has application specific knowledge
General	Shell is used for lots of applications

If more than one shell in a given type wants permanent control, then a pseudo-random selection is made.

WinOldAp uses the SHELLINFO data structure which is passed during the Identify conversation to determine the shell's type. If the shell execs an old application, the hChild field should contain the AX value returned by the exec function. WinOldAp uses this field to identify it parents which gets the highest priority.

WINOLDAP STATES AND ACTIONS

The old application may be in three states: StartUp, Active, and InActive. The Startup state is entered immediately after the WinOldAp is started but before the old application is exec'ed. The StartUp state allows the shell perform actions such as adding menu items before the application has started. It is left when no shell sends a positive ACK to the WinOldApInfo poke, a shell posts a WM_DDE_EXECUTE message, or when a shell which has commited to starting an initiate conversation destroys its window.

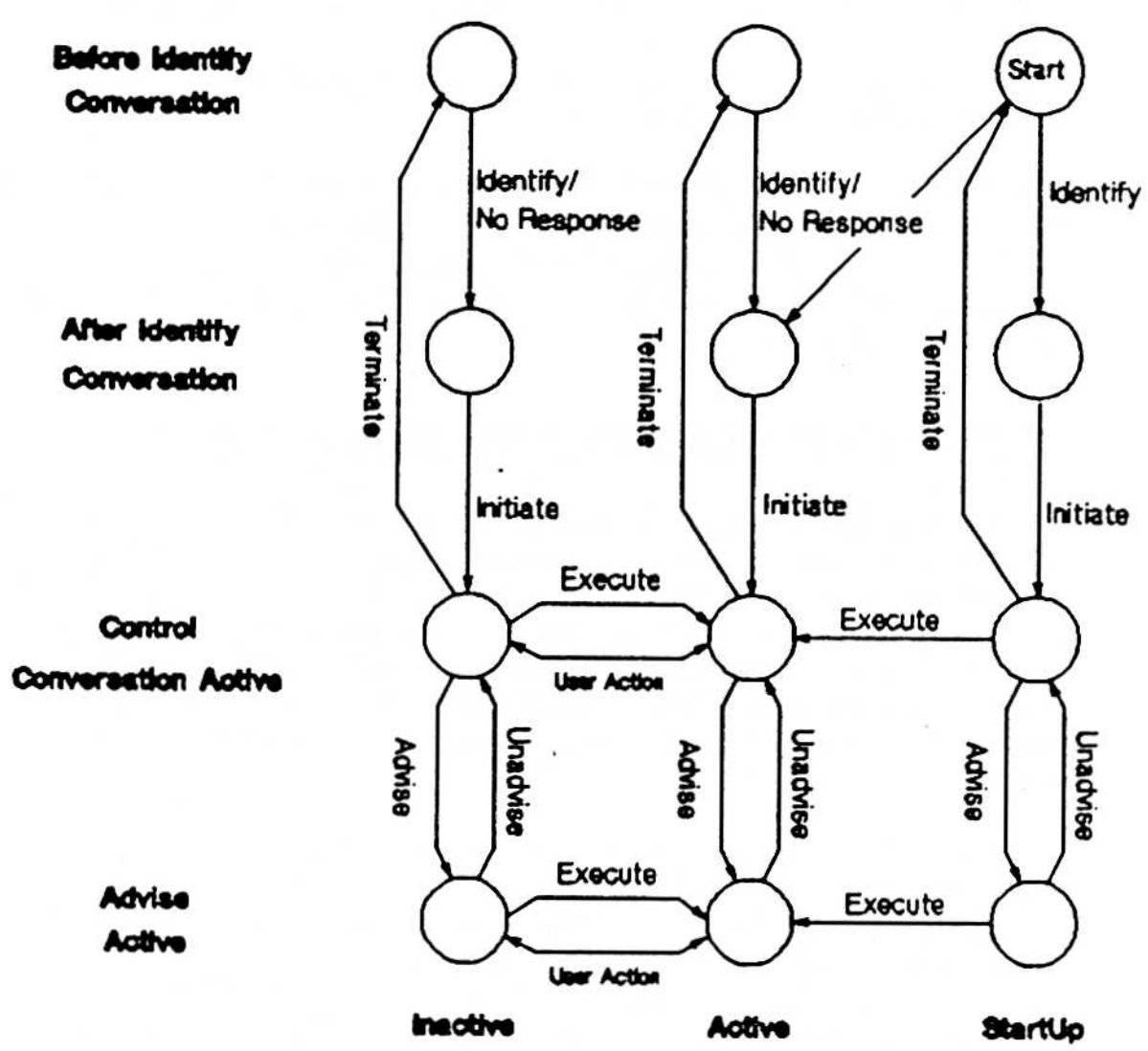
After WinOldAp leaves the StartUp state it exec's the old application and enters the Active state. In the Active state the old application has input focus allowing the user to use the application in the normal manner. The Active state is left for the Inactive state when the user presses Alt-Esc or Alt-Tab or a [Switch] macro opcode is executed. In the Inactive state the old application no longer has keyboard focus. The Inactive state is left for the Active state when the user context switches to the old application or a WM_DDE_EXECUTE message is received by the old application window. A flag bUserLock in the OldApState data structure can be set to prevent the user from moving an application from the Inactive to the Active state.

The DDE conversation can be in five states: before an identify conversation, during an identify conversation, after an identify conversation and before a control conversation, during a control conversation with no advisories set, and during a control conversation with advisories set. All

conversations are terminated when the old application exits.

The relationship between the WinOldAp states and the DDE conversation states is diagramed below. Basically, a change in DDE conversation state does not affect old application state except for the execute and the initiate actions.

Figure 2 - WinOldAp State Diagram.



NOTIFICATION EVENTS

Notification messages are sent when a event occurs on which advice has been requested by the shell. Currently, the shell can only request notification on changes in OldApState caused by the user and execution of macros. Notification events do not change the old application state.

DDE MESSAGES

There are three conversations established between the shell and Winoldap: Identify, Control and System. Each conversation has a different set of messages which are valid. All the messages in the conversations follow the DDE rules outlined in the DDE document.

IDENTIFY CONVERSATION

WM_DDE_INITIATE ( )

Sender WinOldAp

Purpose To notify shells of the invocation of this particular old application.

Parameters wDDEServerID - Atom containing "Shell".
wDDETopicID - Atom containing the name of the application.

Response If a shell exists which wants to control WinOldAp, it should acknowledge. Once the conversation has been established, WinOldAp will send a poke message with the application's information. If no shell acknowledges, then WinOldAp continues execution in a normal manner (see the "WinOldAp States and Control" subsection).

WM_DDE_REQUEST ( )

Sender WinOldAp

Purpose To request the characteristics of the shell for shell sequencing and conflict resolution. If only on shell wants control of WinOldAp, this message may not be sent.

Parameters wDataID - the atom named "ShellInfo."
wFormat - the "Binary" clipboard format.

Response The shell sends a WM_DATA_MESSAGE containing a SHELLINFO data structure. See the "Shell Information" subsection for more details. If a shell does not have a SHELLINFO data structure, it sends an negative acknowledgement. WinOldAp will assume the shell is a permanent general type.

WM_DDE_DATA ( )

Sender Shell

Purpose To respond to a request for shell information.

Parameters wDDEID - the atom named "ShellInfo."
hDDEData - a ShellInfo data structure which is explained in the "Shell Information" subsection. fACKRequired and fClientRelease may be set according to the shell's desire. wFormat must be the "Binary" clipboard format.

Response WinOldAp will acknowledge if the fACKRequired bit is set.

WM_DDE_POKE ( )

Sender WinOldAp

Purpose To send the command string to the shell. This information might be necessary for some servers to determine if they should continue with the DDE session.

Parameters wDataID - the "WinOldApInfo" atom.
hDDEData - the WOAINFO data structure with fACKRequired set and fServerRelease cleared. wFormat will be "Binary".

Response A positive acknowledgement by the shell commits it to initiating a control conversation. If no shell commits, WinOldAp will execute the application in a normal manner.

WM_DDE_TERMINATE ( )

Sender WinOldAp (normal) or Shell (abort)

Purpose To terminate the identify conversation.

Parameters None.

Response The shell does not respond to a normal terminate.

Since termination is a part of the normal conversation sequence, WinOldAp ignores the abort terminate except that it does not expect a control conversation to be initiated by the shell.

WM_DDE_ACK ( )

Sender Shell

Purpose To acknowledge the initiate and poke.

Parameters wDDETopic and wDDEServer in response to initiates; wStatus in response to poke.

CONTROL CONVERSATION

WM_DDE_INITIATE ( )

Sender Shell

Purpose To establish the control conversation between the shell and WinOldAp.

Parameters wDDEServerID - An atom containing the "WinOldAp". wDDETopicID - An atom containing the name of the application.

Response WinOldAp will send a positive acknowledgement if the shell is meant to control this WinOldAp instance during this identify-control cycle.

WM_DDE_TERMINATE ( )

Sender Shell (normal) or WinOldAp (abort)

Purpose Terminates the control conversation.

Parameters None.

Response Terminating the conversation does not remove the any data items that may have been added during the conversation.

WM_DDE_POKE ( )

Sender Shell

Purpose To add, change or delete a data item in WinOldAp.

Parameters Valid wDataId are:

Name	Description
Macro	The macro librarian
Menu	The menu manager
OldApState	The UserLock flag
WindowInfo	The application window info

hDDEData contains the actual information. A poke of a OldApState structure will only affect the UserLock flag of the application's structure. See the "Macro" and the "Menu" subsections.

Response WinOldAp will send a positive or negative acknowledgement depending on the success of the poke. See the WM_DDE_ACK subsection for error codes.

WM_DDE_REQUEST ( )

Sender Shell

Purpose To find some information about the current state of WinOldAp.

Parameters wDataID contains an atom identifying a WinOldAp data item. Valid atoms are:

Name	Description
MenuList	List of current menu items
#2048 to #3071	Individual menu items
MacroList	List of current macros
#1024 to #2047	Individual macros
WinOldApInfo	WinOldAp information
WindowInfo	The application window info
OldApState	The current OldApState

For more information see the appropriate subsection.

All objects should be requested in the "Binary" format.

Response If the wDataID and wFormat are valid, WinOldAp will send a WM_DDE_DATA message with fACKRequired cleared and fClientRelease set. A negative WM_DDE_ACK is sent otherwise.

WM_DDE_ADVISE ( )

Sender Shell

Purpose To setup notification about some user action occurring in WinOldAp.

Parameters wDDEID contains an atom associated with a user event. The following objects are recognized:

Name	Description
'"OldApState"'	This structure contains a lastUserAction field which changes every time the user does a context switch. A notification is sent when this field changes.
'"#1024" to "#2047"'	A notification will be sent after the specified macro is executed.

hOption is structured as follows:

Table 13.1
hOption Structure

Word	Name	Contents
1	fACKRequired	If bit 15 is set, the shell wants an acknowledgement of the request

	fNoData	If bit 14 is set, the shell just wants alarms (ie. data messages with no data). If bit 14 is cleared, WinOldAp will send data messages with just a data header. The data header will have fACKRequired cleared, fClientRelease set and a "Binary" clipboard format. Typically, the shell programmer will have this bit set.
		Bits 13-0 are reserved.
2	wFormat	The "Binary" clipboard format.

As a convenience to the programmer, hOption can be null which will be interpreted to mean fACKRequired cleared and fNoData set.

Response If fACKRequired is set, a WM_DDE_ACK message will be sent indicating the success of the advise request. If the advise is successful, WM_DDE_DATA messages are sent as notification when the advise event occurs.

WM_DDE_UNADVISE ( )

Sender Shell

Purpose To disable WinOldAp notifications.

Parameters wDDEID contains an atom associated with a user event. See WM_DDE_ADVISE for a list of valid events.

Response A WM_DDE_ACK message will be sent indicating the validity of the wDDEID.

WM_DDE_DATA ( )

Sender WinOldAp

Purpose A response to a request or a user event.

Parameters wDDEID is either an advise or a request atom. See WM_DDE_ADVISE and WM_DDE_REQUEST for a list of valid atoms.

To minimize work required of the shell programmer, the fACKRequired flag of the hDDEData object will be cleared and the fClientRelease flags will be set. hDDEData may also be NULL, if the shell request an "alarm" type notification.

Response No response is required of the shell.

WM_DDE_EXECUTE ( )

Sender Shell

Purpose Places the old application in an Active state. Inaddition, executes the passed command.

Parameters If hCommands is NULL, no command are executed, but the old application is still placed in an Active state. Any passed command is executed before the user gets control. The execute command follows the syntax laid out in the DDE document and the "Command Language" sub-section.

Response A positive acknowledgement if any part of the command could be executed.

WM_DDE_ACK ( )

Sender WinOldAp

Purpose To acknowledge an advise, unadvise, poke, or execute message.

Parameters Bit 15 of wStatus indicates the success of the operation. If the operation is unsuccessful, bits 7 - 0 contains an error code.

Error	Name
1	Invalid object
2	Invalid format
3	Invalid fRelease parameter

4	Invalid fNoData parameter
5	Invalid fAckReq parameter
6	Invalid syntax
10	Unable to perform operation
11	Invalid operation
12	Out of memory
13	Invalid paramters
14	Invalid flags

SYSTEM CONVERSATION

The system conversation provdes a "System" topic in accordance to the DDE specification. The "System" topic enumerates information about an application in a standard manner.

WM_DDE_INITIATE ( )

Sender Shell

Purpose To start a System conversation.

Parameters wDDEServerID - Atom containing "WinOldAp". wDDETopicID - Atom containing "System".

Response WinOldAp will always acknowledge positively.

WM_DDE_REQUEST ( )

Sender Shell

Purpose To request items in the system topic.

Parameters wDataID - One of the following atoms:

Atom	Description
SysItems	A list of items supported under the System topic
Topics	A list of current topics. This will be current applications name.
Formats	The number of clipboard formats that WinOldAp can render. This will be "Binary".

wFormat the CF_TEXT clipboard format.

Response WinOldAp sends a WM_DATA_MESSAGE containing the requested list. Individual elements of each lists are delimited by tabs. If the requested item is not available, then a negative ACK is posted.

WM_DDE_DATA ( )

Sender WinOldap

Purpose To respond to a request for information.

Parameters wDDEID - the atom
hDDEData - fAckRequired will not be set. fClient will be set.

Response The shell must delete the passed list.

WM_DDE_TERMINATE ( )

Sender Shell (normal) or WinOldAp (abort)

Purpose To terminate the system conversation.

Parameters None.

Response The WinOldAp does not respond to a normal terminate.

WM_DDE_ACK ( )

Sender WinOldAp

Purpose To acknowledge the initiate and request.

Parameters wDDETopic and wDDEServer in response to initiates; wStatus in response to requests.

WINDOW INFORMATION

The WindowInfo structure controls the visiblity of the application's window as well as its icon. Using WindowInfo, a shell can change the application's icon to one it has supplied. Window infomation can be requested by sending a WM_DDE_REQUEST message with an atom named "WindowInfo". It can also be changed by sending a WM_DDE_POKE message.

Table 13.2

WindowInfo Structure

Word	Name	Contents
1	fACKRequired	Always cleared. Bit 14 is reserved. fClientRelease Always set. Bits 12-0 are reserved.
2	wFormat	The register clipboard format "Binary".
3	fAction	Bit 0 - Show the old application window Bit 1 - Hide the old appliction window Bit 2 - Used the default WinOldAp icon The default icon is a rectangle enclosing the first 3 letters of the applications name Bit 3 - Used the passed icon. Note that bits 0,1 and 2,3 are mutally exclusive.
4-5	Reserve	Must be 0
6-N	Icon	If bit 3 of fAction is set, then an Icon is passed with this structure. If bit 3 is reset, then these words are undefined. The Icon structure used is identical to Windows Icon structure. LockResource will return a long pointer to one of these structures.

WINOLDAP INFORMATION

Information about the old application can be requested by the shell by sending a WM_DDE_REQUEST message with an atom named "WinOldApInfo." WinOldAp will respond with a WOAINFO data structure. The same data structure is poked to the shell during the Identify conversation.

Table 13.3

WOAINFO Structure

Word	Name	Contents

1	fACKRequired	Set when poked during the Identify conversation. Cleared when requested during the Control conversation.
		Bit 14 is reserved.
	fClientRelease	Cleared when poked during the Identify conversation. Set when requested during the Control conversation.
		Bits 12-0 are reserved.
2	wFormat	The "Binary" clipboard format.
3	numReq	The number of shells which requested control in the last identify conversation.
4	numGranted	The number of shells which have been granted control.
5	numOther	The number of other instances of this application at the time this application was started. This number corresponds to the number of tick marks found in the default application icon.
6	pifBehavior	The pifBehavior word from the applications PIF file.
7	hTask	The instance handle of the old application instance. This field will match the AX value returned by the Exec function after loading the application.
8-9	reserved	Must be 0.

10-n	szCommandLine	The command line passed to the application when the application was started.

OLD APPLICATION STATE

Information about the old application state can be requested by the shell by sending a WM_DDE_REQUEST message with an atom named "OldApState." WinOldAp will respond with a STATE data structure.

Table 13.4

STATE Structure

Word	Name	Contents
1	fACKRequired	Always cleared. Bit 14 is reserved. fClientRelease Always set. Bits 12-0 are reserved.
2	wFormat	The "Binary" clipboard format.
3	OAState	0 - Old application not started. 1 - Old application running. 2 - Old application terminated. 3 - Old application has been closed.
4	LastUserAction	0 - User started the application 1 - User switched into the application 2 - User switched out of the application 3 - User terminated the application
5	UserLock	The user lock is set if this field is non zero.
6	wExitCode	Valid if OAState is set to closed.

The wExitCode field is only valid when wOAState is SF_CLOSED. It follows the convention set down the by MS-DOS WAIT (4D) function call with the addition of the new error class for WinOldAp failures. These WinOldAp failures are:

Value	Description

0x8000 "User canceled WinOldAp"

0x8001 "Need WINOLDAP files to run program"

0x8002 "Cannot run with other applications"

0x8003 "Initial directory not found"

0x8005 "Need more disk space"

0x8008 "Not enough memory to run"

0x8009 "Not enough memory for Clipboard"

0x800A "Cannot access PIF swap file"

0x800B "Bad EXE format encountered"

0x800C "Cannot swap to floppy"

0x800D "COM in use. Cannot swap Windows"

0x8085 "Need more disk space"

The typical shell which wants to examine wExitCode will establish a hotlink to OldApState. When the application closes, WinOldAp will post the OldApState structure to shell with the correct wExitCode. It should be noted that if a PIF file cannot be found for an application or not enough memory is available to load WinOldAp, the DDE conversation cannot be established and the error code cannot be retrieved.

SHELL INFORMATION

The shell information data structure is sent by the shell during the identify conversation to determine sequencing order of the shells. The SHELLINFO structure characterizes the future behavior of the shell.

Table 13.5

SHELLINFO Structure

Word	Name	Contents
1	fACKRequired	Set if the shell wants WinOldAp to acknowledge.
		Bit 14 is reserved.
	fClientRelease	Set if the shell wants WinOldAp to release the ShellInfo data structure.
		Bits 12-0 are reserved.
2	wFormat	The "Binary" clipboard format.
3	hChild	If the shell starts an old application, this field should be set with the AX value returned by the Exec function. Otherwise, it can be set to 0.
4	Bits 15 - 10	reserved

	fRequest	Bit 9. Set if the shell requests information.
	fAdd	Bit 8. Set if the shell adds data items.
	fPredefine	Bit 7. Set if the shell modifies data object which the shell did not add.
	fAdvise	Bit 6. Set if the shell sets advisories.
	fExecute	Bit 5. Set if the shell uses the execute command.
	fTemporary	Bit 4. Set if the shell terminates immediately.
	fExclusive	Bit 3. Set if the shell does not want any temporary shells to be given control be it is. This field is only valid if hChild matches the applications task handle.
	Bit 2	reserved.
	type	Bits 0-1. 0 for general; this shell works with a lot of different old applications. 1 for specific; this shell is meant for this old application.
5	wMacroSize	The total size in bytes of the MACRO data structures downloaded by the shell. WinOldAp will use this field to determine the size of the macro buffer used to store any downloaded macros. Setting this field does not guarantee that the specified size will be allocated. Shells should still use acknowledgement messages to certify the success of the downloading operation.
6	wMenuSize	The total size in bytes of the MENU data structures downloaded by the shell. WinOldAp will use this field to determine the size of the menu buffer used to store any downloaded menu items. Setting this field does not guarantee that the specified size will be allocated. Shells should still use acknowledgement messages to certify the success of the downloading operation.
7	Reserved	Must be set 0.

MACRO SUPPORT

Macro support uses the DDE execute command syntax and semantics. See the "Command Language" subsection for details. WinOldAp starts with a set of predefined macros in their default settings. The shell has the ability to add new macros, delete old macros and change existing macros.

The predefined macros are:

Table 13.6

Predefined Macros

Name	ID	Default	Description

Mark	1024	"[Mark]"	Used for default system menu items.
Copy	1025	[Copy]	Used for default system menu items.
Paste	1026	[Paste]	Used for default system menu items.
Scroll	1027	[Scroll]	Used for default system menu items.
Switch	1028	[Switch]	Used for default system menu items.
Close	1029	[Close]	Used for default system menu items.
Null	1030		Used for default system menu items.
SwitchOut	1031		Used before a switch out.
Exit	1032		Used after the application exits.
Execute	1033		Reserved

Each macro has a unique identification number between 1024 and 2047. The macro identification number can be used with the MAKEINTATOM C macro to form an atom name which refers to the macro.

The SwitchOut is executed every time the user tries to context switche out of an application. In level 2 applications, the macro is executed before the context switch occurs. The Exit macro is executed when the application exits. Keystrokes in the Exit macro have no effect since the application has terminated.

Macros are manipulated using the WM_DDE_POKE messages. The wDDEID parameter contains a atom whose name is the macro id. hData refers a the MACRO data structure.

Table 13.7

Macro Structure

Word	Name	Contents
1	fACKRequired	If bit 15 is 1, will WinOldAp will send a WM_DDE_ACK message.
		Bit 14 is reserved.
	fServerRelease	If bit 13 is 1, WinOldAp will free the global data object.
		Bits 12-0 are reserved.
2	wFormat	The register clipboard format "Binary".
3	wMacroID	The identification number of the macro the operation is referring to.

4	wAction	If 0, add the provided macro using wMacroID. If 1, replace the indicated macro with the provided macro. If 2, delete the indicated macro.
4-n	szMacro	The actual macro in the WM_DDE_EXECUTE format which is a NULL terminated ANSI string. If delete action is being performed this field is ignored.

If the poke is successful and fACKRequired is set, a WM_DDE_ACK message will be sent with the wDDEID containing "Macro" atom and bit 15 of wStatus set. Bits 7 - 0 will be undefined.

If the poke is unsuccessful and fACKRequired is set, a WM_DDE_ACK message will be sent with the wDDEID containing a macro identification atom and bit 15 of wStatus cleared. Bits 7 - 0 will contain an error code which follows the convention set forth in the "DDE Messages" subsection.

The shell can request the contents of an individual macro by sending a WM_DDE_REQUEST message with wDDEID containing a macro identification atom. WinOldAp responds with a WM_DDE_DATA message containing a MACRO data structure with fACKRequired cleared and fClientRelease set.

The shell can also request a list of all of the currently defined macros by requesting the "MacroList" object. The data message which replies to this request contains a handle to a MACROLIST data structure:

Table 13.8

MACROLIST Structure

Word	Name	Contents
1	fACKRequired	Always cleared. Bit 14 is reserved. fClientRelease Always set. Bits 12-0 are reserved.
2	wFormat	The register clipboard format "Binary".
3	numberMacros	Size of the passed macro list.

4-n	macroIds	An ascending list of all of the identification numbers of all the currently defined macros.

MENU SUPPORT

OVERVIEW

The DDE extensions in WinOldAp will allow a shell program to download menu information to WinOldAp. This information can be used to create, delete or modify menus and menu items in a level 1 or level 2 application. The user interface for level 2 applications is designed to match that of level 1 applications as closely as possible. The DDE messages sent to modify menus are modelled closely after the USER functions used for Windows applications.

WinOldAp will have predefined menu items in the control menu. For level 2 applications, The standard control menu items will exist with limited functionality. Both level 1 and 2 applications will have Mark, Copy, Paste and Scroll (Scroll being grayed in level 2 apps) items appended to the bottom of the menu. These items should not be modified by the shell programs since very unpredictable results may occur. The standard items in the control menu for level 1 apps may not be modified.

The predefined menu items are:

Table 13.9

Predefined Menu Items

Name	ID	MacroID	Default Style
System Menu	2048	NULL	MF_POPUP
Separator	2049	NULL	MF_SEPARATOR
Mark	2050	1024	MF_ENABLED
Copy	2051	1025	MF_GRAYED *
Paste	2052	1026	MF_GRAYED *
Scroll	2053	1027	MF_ENABLED #
Restore	2054	1030	MF_GRAYED &
Move	2055	1030	MF_GRAYED &
Size	2056	1030	MF_GRAYED &
Minimize	2057	1028	MF_ENABLED &
Maximize	2058	1030	MF_GRAYED &
Separator	2059	NULL	MF_SEPARATOR
Close	2060	1029	MF_GRAYED

An asterisk (*) indicates that WinOldAp sets the MenuStyle for these according to the current state of MCP. If there is not an area currently marked on the screen, the copy item is automatically grayed. If there is no data in the clipboard in the CF_TEXT clipboard format, the Paste item is grayed. WinOldAp handles these states directly. For this reaason these items should not be modified.

A number sign (#) means that this item is only enabled in level 1 applications. It has no functionality in level 2 applications.

An ampersand (&) indicates that these items are only referred to by these ID's in level 2 applications. Level 1 applications may not access these items. The Close option is initially MF_GRAYED but becomes MF_ENABLED when the application terminates. Again, these items should not be modified.

Each menu item has a unique identification number between 2048 and 3071. The menu id can be used with the MAKEINTATOM C macro to form an atom name which refers to the menu.

Menus are manipulated using the WM_DDE_POKE messages. The wDDEID parameter contains an atom whose name is the menu id. The DDE hData parameter refers to a structure in the following format:

Table 13.10

hData Structure

Word	Name	Contents
1	fACKRequired	If bit 15 is 1, WinOldAp will send a WM_DDE_ACK message containing a success or failure flag and the menu id.
		Bit 14 reserved
	fServerRelease	If bit 13 is 1, WinOldAp will free the global object.
		Bits 12-0 are reserved.
2	wFormat	The registered clipboard format "Binary"

3	wIDChangeItem	This items value depends on the flags set in the wChange parameter below. If MF_BYPOSITION is selected, this number contains the physical position within the menu to be changed. If MF_BYCOMMAND (default) is selected, this item contains the menu ID of the item to be changed. This parameter is the same as the wIDChangeItem used for the ChangeMenu command documented in the Windows Programmers Reference. Top level menu items (Menu Titles) may only be updated using the MF_BYPOSITION flag.
4	wIDNewItem	This is a new menu ID. See the section on popup menus below.
5	wMenuID	This is the menu ID of the popup menu to be modified. This is only used when modifying menu items within a popup menu. See the popup menu section below. If a top level menu item is being modified. this item is null.
6	wMacro	This is the macro ID which is to be associated with this menu item. If the MF_POPUP or MF_SEPARATOR flag is used, this item is null.

7	wChange	This is a combination of the menu flags listed below. These flags are combined using the bitwise OR operation. See the Programmmers Reference for a more detailed explanation of this parameter. Note the flags in the list below that are not supported for WinOldAp applications.
8 - 23	szItem	This is a null terminated ASCII string which is no longer than 31 characters including the null terminator. This is the string that will be displayed in the menu. This field is ignored if the MF_SEPARATOR flag is used. The & character is used, as in Windows applications, to precede the character in the string that is to be used as the mnemonic character for this item.

wChange = A combination of the following (These flags may be combined using the bitwise OR operator):

Flag	Description
MF_CHANGE	Change or replace the specified item.
MF_INSERT	Insert a new item just before the specified item. If wIDChangeItem is greater than or equal to the number of menu items, and the MF_BYPOSITION flag is set, the operation is aborted (i.e. when inserting a menu title, if wID-ChangeItem = 4 and there are 4 or less menu titles on the menu bar, the insertion will be

aborted). MF_APPEND should be used.

MF_APPEND	Append the new item to the end of the menu. wIDChangeItem is not used if this flag is set.
MF_DELETE	Delete the item. The system menu cannot be deleted. System menu items available to the shell, may be deleted.
MF_BYPOSITION	wIDChangeItem gives the position of the menu item to be changed. The first item is located at position 0.
MF_BYCOMMAND	wIDChangeItem gives the menu ID of the item to be changed. (default)
MF_GRAYED	Disable and gray the item to show that it cannot be selected.
MF_ENABLED	Enable the item, allowing it to be selected. (default)
MF_DISABLED	Disable the item without changing its appearance. The item cannot be selected.
MF_REMOVE	Remove the item from the menu but do not delete it. The item is not selectable.
MF_CHECKED	Place a checkmark next to the item. Valid only in a popup menu.
MF_UNCHECKED	Do not place a checkmark next to the item. (default)
MF_MENUBREAK	Unsupported.
MF_MENUBARBREAK	Unsupported.
MF_SEPARATOR	Draw a horizontal dividing line. Only valid in popup menus. This cannot be enabled, disabled, checked, grayed or highlighted.
MF_BITMAP	Unsupported.
MF_STRING	Use a string (wItem) as the menu item. (default)
MF_POPUP	Creates a popup menu with wItem containing the string to be used in the menu bar for the popup. This can only be used with top level menu items. See the popup menu section below for more details on the use of this flag.

The following rules apply to the usage of the above flags. MF_INSERT, MF_BYCOMMAND, MF_ENABLED, MF_UNCHECKED and MF_STRING are the default flags. The flags MF_CHANGE, MF_INSERT, MF_APPEND, MF_REMOVE and MF_DELETE should not be used together. The flags

MF_BYPOSITION and MF_BYCOMMAND should not be used together. The MF_GRAYED, MF_DISABLED, MF_ENABLED and MF_REMOVE flags should not be used together. The MF_STRING and MF_POPUP flags should not be used together. The MF_CHECKED flag and the MF_UNCHECKED flag should not be used together.

If the poke is successful and fACK is set, an WM_DDE_ACK message will be sent with the wDDEID containing a menu identifying atom and bit 15 of wStatus set. Bits 14 - 0 will contain the menu id.

If the poke is unsuccessful and fACK is set, an WM_DDE_ACK message will be sent with the wDDEID containing a macro identifying atom and bit 15 of wStatus cleared. Bits 14 - 0 will contain an error code.

POPUP MENUS

The protocol for creating and updating menus for WinOldAp is designed to mimic the Windows menu interface as closely as is possible while conforming to the DDE protocol. Popup menus are handled slightly differently from the Windows interface. To create a popup menu the menu title must be created first. This is done by creating a menu item, with a menu item ID set in wIDNewItem and using the MF_POPUP flag. The wMenuID must be zero when accessing items with the MF_POPUP flag set. This is the case since the MF_POPUP flag can only be used on top level (menu bar) items. The menu ID that you assign to this popup menu will now be used to insert or append items into the popup menu. This differs from the way that Windows handles popup menus in that, here, a popup menu is not created in advance and passed with the menu title upon creation.

The following is a high level scenario for building a popup menu:

Create the menu with the following data:

```
wIDChangeItem  = position if MF_INSERT is used
wIDNewItem     = your menu ID (i.e. 2500)
wMenuID        = 0
wMacro         = 0
wChange        = MF_BYPOSITION|MF_POPUP|MF_APPEND
wItem          = "My Popup Menu"  (Null terminated)
```

Insert menu items into this menu using:

```
wIDChangeItem  = position or another Menu ID in the
                 menu
wIDNewItem     = Menu item ID (i.e. 2501)
wMenuID        = menu ID (2500 in this case)
wMacro         = an existing macro ID
wChange        = MF_INSERT|MF_BYPOSITION
```

(MF_APPEND and MF_BYCOMMAND may also be used)
wItem = "My Menu Item" (Null terminated)

RETRIEVING MENU INFORMATION

The shell may obtain information about a particular menu item by using the WM_DDE_REQUEST message with wDataID specifying the menu ID of the item requested. WinOldAp will send back a WM_DDE_DATA which will contain a handle to the same structure used by the shell to poke menu commands to WinOldAP. The appropriate fields in this structure will be filled with the information about the menu item. The wID-ChangeItem will contain the position of the item in the menu specified by wMenuID. The MF_BYPOSITION flag will be set in wChange to identify this. If wMenuID is 0, this item is a top level (menu bar) item. The flags set in wChange identify the style of the menu.

The shell can also request a list of all of the currently defined menu items by requesting the "MenuList" object. The data message which replies to this request contains a handle to a list data structure:

Table 13.11

List Structure

Word	Name	Contents
1	fACKRequired	Always cleared.
		Bit 14 reserved.
	fClientRelease	Always set.
		Bits 12 - 0 are reserved.
2	wFormat	The regitered clipboard format "Binary".
3	MenuCount	The number of macro id codes passed back including separators.
4 - n	menu ids	This is a list of the menu id's availabe.

The "menu ids" use the following format:

MenuList
0
MenuList
0

:
:

where:

MenuList = Menu Title ID
Menu Item ID
Menu Item ID
:
:

The MenuList items are in the order that they appear from left to right across the screen. Therefor the first MenuList will pertain to the system menu. The second MenuList will pertain to the menu in position 0 the third to the menu in position 1 and so on. The system menu list will only contain the items specified by WinOldAp. Therefore, the standard Windows menu items cannot be modified or deleted in level 1 apps. The separators (0) are included in the MenuCount variable above.

COMMAND LANGUAGE

OVERVIEW

The WinOldAp Macro Command Language is a set of commands that provide access to WinOldAp user functions. An extension to the concept of keyboard macros, they provide an programable way to perform operations normally associated with the system menu.

COMMAND SET

COPY

Description The COPY command takes the currently marked area on the screen, and copies it into the Windows clipboard.

MACRO <macro id>

Description The MACRO command will begin the execution of the specified macro. This will allow for chaining macros together into one megamacro. Recursive calls to the same macro will not be supported.

<macro id> must be specified and must be an integer corresponding to a valid macro.

MARK <option>

Description The MARK command designates a portion of the screen to be copied into the clipboard. The display may be in either text or graphics mode. The MARK command will take the following parameters. If no parameters are given, the system is put into the interactive mark mode.

<option>	Description
SCREEN	Marks entire display.
X Y Xext Yext	Marks rectangular area of display. (X,Y) specifies the upper left corner of the rectangle, Xext and Yext specify the relative extents. These coordinates refer to the current video mode the system is in. Currently only text mode is supported.

MENU <menu id>,<menu flags>,<macro id>

Description The MENU command is used to modify the menus.

Parameter	Description
<menu id>	specifies the menuitem.
<menu flags>	Specifies the changes to be performed. Supported flags are: MF_GRAYED, MF_ENABLED, MF_DISABLED, MF_CHECKED, and MF_UNCHECKED. Combinations of flags may be set using the or ("|") operator.
<macro id>	Specifies the macro to be associated with this menu change.

NULL <count>

Description The NULL command is used to specify the number of times an application will un-successfully read the keyboard before control returns to the macro executive. This allows macro programmers to account for the behaviour of a particular application.

<count> is the number of times to fail, between 1 and 255.

PASTE

Description The PASTE command takes the text contents of the clipboard and directs it to the application. The PASTE command takes no parameters.

Note: Characters for level-2 apps are always converted to the OEM character set.

SCANCODE <number>

Description Inserts number as a scancode into the keyboard buffer. If certain applications need to read in codes that the IBM PC doesn't support, then this command may be used.

<number> is the decimal value of the scancode word to be inserted into the keyboard buffer.

SCROLL

Description The SCROLL command will enable the standard Windows scroll mode. In this mode, the user may use the keyboard arrow keys to scroll the application. This command is only valid for level-1 apps.

The SCROLL command takes no parameters.

SWITCH

Description The SWITCH command performs a context-switch from an application. This would be the equivalent of the user hitting Alt-Tab. At this point, any remaining macro commands will be ignored.

SYNTAX OF MACRO COMMANDS

The WinOldAp Macro command language follows the syntax of the WM_DDE_EXECUTE message. Section VIII of the "DDE Rules of the Road" presents the grammar for valid commands. This grammar explicitly allows for applications to extend this definition to support their own "application specific opcodes". It is through this type of extension that the WinOldAp macro command language has been defined. The resulting grammar is listed below.

```
<execute string> := [<token>]<execute string>
<token> := <keystroke> ¦ <app opcode>
<keystroke> := [<modifier>]<character> ¦ [<modifier>]<key equivalent>
<modifier> := [<shift>][<control>][<alt>]
<shift> := +
ESCAPE "*"
<control> :=
ESCAPE
<alt> := ~
<character> := <letter> | <digit> | <symbol> | <quoted character>
<letter> := a | A | ... | z | Z
<digit> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<symbol> := ` | ! | @ | # | $ | % | & | * | ( | ) | - | _ | = |  |
            " | " | ; | : | ' | """ | , | "<" | . | ">" | /
<quoted character> := <escape character1><escape character1>
ESCAPE "*"
<escape character> := [ | { | ] | } | + | ~ | ^
ESCAPE
<key equivalent> := {<keyword>}
<keyword> := TAB | ESC | ENTER | HOME | END | LEFT | RIGHT | UP | DOWN |
             PGUP | PGDN | NUM | SCROLL | SYS | PRTSC | BREAK |
             <function key>
<function key> := F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 | F12
<number> := <digit>[<number>]
<app opcode> := "["<WOA opcode string>"]"
<WOA opcode string> := COPY |
                       MACRO<macro id> |
                       MARK[<mark param>] |
                       MENU<menu id>,<menu style>,<macro id> |
                       NULL <number> |
                       PASTE |
                       SCANCODE <number>|
                       SCROLL |
                       SWITCH
<macro id> := <number>
<mark option> := SCREEN | <number>,<number>,<number>,<number>
```

```
<menu id> := <number>
<menu style> := <menu flag>["|"<menu style>]
<menu flag> := MF_GRAYED | MF_ENABLED | MF_DISABLED | MF_CHECKED |
               MF_UNCHECKED
```

WINOLDAP CLIPBOARD

Function Call Definitions

Access to all functions is via *Int 2FH*. The sub-function is placed in the **AX** register as detailed below. The IdentifyWinOldApVersion function call MUST be made first to verify that the functions are present.

Name IdentifyWinOldApVersion
Parameters AX = 1700H
Return Values AX = 1700H: Clipboard functions not available.
1700H: AL = Major Version Number
AH = Minor Version Number

Name OpenClipboard
Parameters AX = 1701H
Return Values AX = 0: Clipboard already open
0: OK

Name EmptyClipboard
Parameters AX = 1702H
Return Values AX = 0: Error Occured
0: OK. Clipboard emptied

Name SetClipboardData
Parameters AX = 1703H
DX = WinOldAp-Supported Clipboard format
ES:BX = Pointer to data
SI:CX = Size of data
Return Values AX = 0: Error Occured
0: OK. Data copied into allocated memory.
Notes The OldApp should call ClipboardCompact prior to this to determine if the data can be accomodated in memory.

Name GetClipboardDataSize
Parameters AX = 1704H
DX = WinOldAp-Supported Clipboard format
Return Values DX:AX = Size of the data in bytes. including any headers.

0 if data in this format is not in the clipboard.

Name GetClipboardData
Parameters AX = 1705H
DX = WinOldAp-Supported Clipboard format
ES:BX = Pointer to location to place data.
Return Values AX = 0: Error Occured (or data in this format is not in the clipboard)
0: OK

Name CloseClipboard
Parameters AX = 1708H
Return Values AX = 0: Error Occured
0: OK

Name ClipboardCompact
Parameters AX = 1709H
SI:CX = Desired memory size in bytes.
Return Values DX:AX = number of bytes of largest block of free memory
Notes The OldApp is responsible for including the size of any headers in the desired memory size.

Name GetDeviceCaps
Parameters AX = 170AH
DX = GDI information index
Return Values AX = integer value of desired item
Notes The implied hDC for this call will be for the display.

Structures

These structures mimic the actual Windows structures with one major difference: instead of including a handle or pointer to other memory containing the actual data, the data follows the structure. The structure information now behaves like a header prefacing the data.

Bitmap structure:

```
bmType        DW    ?  ;always 0
bmWidth       DW    ?  ;width of bitmap in pixels
bmHeight      DW    ?  ;height of bitmap in raster lines
bmWidthBytes  DW    ?  ;bytes/raster line
bmPlanes      DB    ?  ;# of color planes in the bitmap
bmBitsPixel   DB    ?  ;# of adj color bits to def pixel
bmBits        DQ    ?  ;points to byte following bmHigDim
bmWidDim      DW    ?  ;width of bitmap in 0.1 mm units
```

```
bmHigDim        DW      ?   :height of bitmap in 0.1 mm units
BitmapData                  :the actual data
```

MetaFilePict structure:

```
mm              DW      ?   :mapping mode
xExt            DW      ?   ;x extent
yExt            DW      ?   :y extent
MetaFilePictData            ;the actual data
```

OEM FILTER

In the Windows world, the clipboard and windows applications use the ANSI character set. In the MS-DOS world, applications use the native (or OEM) character set of the PC they are executing on. For example, the Vectra uses IBM PC-8, the HP150 uses Roman-8. The OEM and ANSI character sets usually contain a fair amount of overlap in contents. By using the Windows clipboard and the Mark/Copy/Paste technique, text can be easily transferred between applications.

When data is copied from an MS-DOS application (either level-1 or level-2) into the clipboard, a conversion to the ANSI character set is automatically performed. OEM characters that are copied into the clipboard are automatically converted to the ANSI character set.

This is fine if the intended destination is a Windows application, they only understand ANSI and a conversion would have to be performed eventually. However, if the intended destination is another MS-DOS application, characters that would have made sense to the receiving application may be garbled in transit.

WinOldAp 2.00 will remedy this situation. When an MS-DOS app is started up, WinOldAp will register a new clipboard format, CF_OEMText. If data is copied into the clipboard, WinOldAp will set the data in the clipboard in two formats: CF_TEXT and CF_OEMText. Then, if a Windows app wants to get data, CF_TEXT will be available. If an MS-DOS application wants to get data, WinOldAp will first get CF_OEMText if available. Otherwise it will get CF_TEXT and convert from ANSI to OEM as in Windows 1.x.

13.2 WINOLDAP.GRB - Display grabber, context switcher, and function library

OVERVIEW

The term "grabber" is Microsoft's name for that portion of the Windows old application support layer that allows Winoldap to capture data from the oldap screen, to context switch the video subsystem, and to display pop down menus and titlebars.

A a number of internal changes have been made in the grabbers of the Windows 2.03 release to allow better EGA context switching support, and provide increased efficiency, performance, and maintainability. However, the bulk of the enhancements have been made in an area known as "video access functions". These functions give Winoldap character-cell oriented tools specifically needed to implement Mark/Copy/Paste (MCP) and menuing capabilities. A primary design goal was to make these functions as general as possible so that Winoldap could easily manipulate the oldap display regardless of the screen mode or display adapter in use.

NAMING CONVENTIONS

The .GRB grabber files supplied in the retail release appear as shown below and are used for the display drivers indicated.

grabber name	display driver name
EGACOLOR.GRB	EGAHIRES.DRV, EGALORES.DRV, EGAHIBW.DRV
EGAMONO.GRB	EGAMONO.DRV
CGA.GRB	CGA.DRV
HERCULES.GRB	HERCULES.DRV
MULTIMOD.GRB	MULTIMOD.DRV

The corresponding entry in SETUP.INF appears as follows:

```
[grabber]
; this section defines the locations of the screen grabber files.
; (These files will be renamed to WINOLDAP.GRB when copied)
; These files may be aliased, like the .lgo files.

        (3:egacolor.grb,egahires.grb)
        (3:egacolor.grb,egalores.grb)
        (3:egacolor.grb,egahibw.grb)
        (3:egamono.grb)
        (3:cga.grb)
        (3:hercules.grb)
        (3:multimod.grb)
```

GRABBER ENTRY POINTS

STANDARD FUNCTION DISPATCH TABLE

Since the grabbers are loaded by Winoldap as a binary image, Winoldap transfers control to them via a jump table at offset 0 of the grabber code segment.

Note

With the exception of InitScreen, which is an optional entrypoint, the format of this table MUST remain fixed and MUST reside at offset 0!

Winoldap computes the offset of the desired entrypoint using the knowledge that a near jump is 3 bytes of opcode. However, as Winoldap will be making a far call to the jmp, we MUST return far even though our functions are near. Winoldap must always set **DS** equal to the grabber's **CS** before making the jump.

Winoldap checks for the existence of the jmp opcode at offset 015h, and if it exists, assumes that the InitScreen entrypoint is present. If present, InitScreen will be called when an oldap starts up, and subsequently after every context switch from Windows to that oldap.

```
                org     0

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

EXTENDED FUNCTION DISPATCH TABLE

Because of the nature of the table above, any extensions to the Winoldap/grabber interface must be made via subfunction calls to an existing standard function entrypoint if we are to avoid crashing caused by Winoldap jumping to an entrypoint that does not exist. The standard function used for this purpose is the benign InquireGrab call. InquireGrab will dispatch control to one of the routines below if upon entry ax contains

a function number in the indicated range.

we are guaranteed not to crash with older Winoldaps written to the Microsoft OEM specification since **AX** will always contain 1 or 2 indicating an original InquireGrab buffer size request. Older grabbers used with the new Winoldap will not crash either, since they will either interpret an **AX** value other than 1 or 2 as an invalid request or blindly return an undefined buffer size. The GetID and GetVersion routines below are provided so that Winoldap can compare the return value with a standard and cease to call the extended interface if the value returned is inconsistent with the value expected.

```
ExtFuncTable    label   word
                dw      EndPaint                ;Func OFFF4h
                dw      BeginPaint              ;Func OFFF5h
                dw      MarkBlock               ;Func OFFF6h
                dw      PutBlock                ;Func OFFF7h
                dw      GetBlock                ;Func OFFF8h
                dw      RealizeColor            ;Func OFFF9h
                dw      GetVersion              ;Func OFFFAh
                dw      DisableSave             ;Func OFFFBh
                dw      EnableSave              ;Func OFFFCh
                dw      SetSwapDrive            ;Func OFFFDh
                dw      GetInfo                 ;Func OFFFEh
                dw      GetID                   ;Func OFFFFh
```

FUNCTION REFERENCE

InquireGrab

InquireGrab ( )

Purpose Returns size of grab buffer needed or dispatch extended call.

In Microsoft Windows 1.0x, this routine only handled grab buffer size requests. In HP's version of Microsoft Windows 1.03, 5 new subfunction numbers were added to perform various helper tasks to aid Winoldap in implementing HP's extensions (Mark/Copy/Paste and "extended mode" oldap memory management).

Microsoft Windows 2.0x adopted HP's 1.03 extensions as well as defining another 7 subfunctions to support the increased functionality of Winoldap 2.0x.

For an explanation of the new subfunctions, refer to the subfunction descriptions.

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

else
exit status depends on the extended call

Uses

if function number 1 or 2,
AX, **DX**, flags

else
register usage depends on the extended call

Notes

In order to get any useful work from the grabber, Winoldap must call either InquireGrab or InquireSave before any other call (InitScreen is a possible exception). We use this behavior as an opportunity to initialize our module if it is the first time one of these entrypoints has been called.

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

EnableGrab

EnableGrab ( )

Purpose

To enable screen snapshots.

Entry

DS = CS

if **ES:DI** = NULL,

We are to do whatever is necessary to enable a grab install the grab event hook into the interrupt system. Winoldap will take responsibility for detecting the grab event and will simulate a grab via a fullscreen MarkBlock GetBlock MarkBlock sequence.

This change to the grabber API was necessary to allow winoldap to provide support for certain "worse" apps such as WORD that take *int 9*.

Without this change, a call to DisableGrab would clobber the apps' *int 9* vector because we would restore it to the *int 9* handler we saved during EnableGrab, which is NOT the apps' *int 9*.

This change also implies that in addition to Block operations on alpha screens, the grabbers must also minimally support the fullscreen versions of MarkBlock and GetBlock

on graphics screens in order to provide the same functionality Windows 1.03 offered.

else

ES:DI -> grab buffer
AX = size of grab buffer allocated

We are to do whatever is necessary to enable a grab, including installing the grab event hook into the interrupt system. This is the original EnableGrab functionality, and must be retained to support snapshots for "UGLY" apps, since when running an "UGLY" app, Winoldap is not present to do the detection and snapshot simulation.

Exit none.

Uses **AX**, flags

DisableGrab

DisableGrab ( )

Purpose To disable screen snapshots.

Entry **DS** = **CS**

Exit None.

Uses **AX**, **DS**, flags

Notes If EnableGrab did not install any grab event hooks, DisableGrab must not attempt to remove them!

PromptGrab

PromptGrab ( )

Purpose To scan screen and display restart message if screen is non-blank.

This entrypoint is no longer called by Winoldap 2.0 since the new Winoldap uses menus and titlebars (similar to GoodOldAp operation) to let the user close the inactive application. It exists here as a stub to prevent older Winoldaps and WIN.COM from crashing if they attempt to call this function. WIN.COM calls the loads and calls the grabber during execution of an UglyOldAp.

Entry **DS** = **CS**

Exit None.

Uses No registers or flags are used.

InquireSave

InquireSave ( )

Purpose To return size of screen save buffer needed.

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

Uses **AX**, **DX**, flags

SaveScreen

SaveScreen ( )

Purpose To save 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)

Uses All except **BP**

Notes Winoldap guarantees that the offset portion of the screen save area will always be zero, so it is safe to omit **DI** from references to this area.

RestoreScreen

RestoreScreen ( )

Purpose To restore the 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)

Uses all except **BP**

Notes Winoldap guarantees that the offset portion of the screen save area will always be zero, so it is safe to omit **DI** from references to this area.

InitScreen

InitScreen ( )

Purpose To initialize screen to a known mode for the oldap.

This routine will be called by Winoldap once before the oldap starts up. Winoldap will also call this routine just before it returns control to the oldap after a context switch from Windows back to the oldap. The latter of these two cases is redundent, since the grabbers RestoreScreen routine will be called shortly thereafter to restore the application's display context. However, since we cannot discern these two cases, we always set the display to a known mode regardless.

Entry **DS** = **CS**

Exit None.

Uses None.

GetID

GetID ( )

Purpose Returns id to indicate presence of grabber extensions.

Entry None.

Exit **AX** = grabber id

Uses **AX**

Notes Winoldap 2.0 will NOT recognize the presence of the 2.0 grabber extensions unless the id returned matches 04A43h (JC).

In fact, this behavior makes GetID more like a version check, and seemingly makes the GetVersion call redundant. Since GetVersion is a 2.0 extension, the 1.03 Winoldap would, under other circumstances, have assumed it had found a 1.03 grabber; this assumption would have been fatal on the next call to GetInfo if a 2.0 grabber was actually present. GetVersion is still useful as a sanity check as

well as determining if minor enhancements are in place.

GetInfo

GetInfo ( )

Purpose Returns GrabInfo structure.

GetInfo fills a buffer pointed to by **ES:DI** on entry with data in GrabInfo format. Note that we just copy our InfoContext structure 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).

While Winoldap knows nothing about the reserved fields, we end up filling some of them with the extra data in InfoContext because it is convenient for debugging purposes. As these fields are still reserved, THEY MAY CHANGE AT ANY TIME.

For historical reasons, Winoldap insists that the return value from this call also indicate whether the display is in graphics or text mode. Essentially, this provides an easy way for Windoldap to determine whether or not it should attempt menuing operations, since the Get/Mark/PutBlock functions are not currently supported in their full generality for graphics modes. If they should ever become completely functional for graphics modes, this call must return a 1 in order for Winoldap to even attempt the menu operation.

Note that the block functions themselves will also check the current mode and return ERR_UNSUPPORTED to indicate non-support for the request, which is the prefered 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)

Uses **AX**, **DI**, **ES**, flags

SetSwapDrive

SetSwapDrive ( )

Purpose Sets the current swap drive and path.

The next SaveScreen call following this call will use 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 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 it is recommended that this call be done once, after Winoldap's 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:pathfname template for Windows temp file format
DS = **CS**

Exit None.

Uses **AX**, **BX**, flags

EnableSave

EnableSave ( )

Purpose Enables video context switching.

Analogous to EnableGrab, this entrypoint gives the grabber the chance to install any hooks needed for context switching.

Entry **DS** = **CS**

Exit None.

Uses **AX**, **DS**, flags

DisableSave

DisableSave ( )

Purpose Disables video context switching.

Analogous to DisableGrab, this entrypoint gives the grabber the chance to remove any hooks installed by EnableSave.

Entry **DS = CS**

Exit None.

Uses **AX**, **DS**, flags

GetVersion

GetVersion ( )

Purpose Returns version number of OEM extensions.

Entry None.

Exit **AX** = OEM version number

Uses **AX**

Notes Winoldap 2.0 will NOT recognize the presence of the 2.0 grabber extensions unless the version returned is greater than or equal to 2.01.

RealizeColor

RealizeColor ( )

Purpose Maps logical color to physical color.

Winoldap works internally with a set of device-independent logical colors for all menuing operations. Since the Put-Block function operates on attributes at a device level, these logical colors, which are defined in GRABBER.INC, must be mapped to physical colors for the display device in use.

These logical "colors" do not really represent points in an orthogonal RGB or HSB color space; rather they are simply members of a limited set of attributes which Winoldap might use to display text. For instance, a request to map logical color number 2 (LC_GRAYED) is simply asking, "Give me a foreground/background attribute pair that would give the impression that a text string displayed with that pair would appear to be disabled, grayed, or otherwise uninteresting with respect to the colors you have chosen to give me for my other RealizeColor requests."

For all grabbers written to date, this has been easily implementable for both color and monochrome via a statically addressed look-up table. In some cases (Multimode) it has been desirable to have one more level of indirection so that multiple translate tables could be maintained with a simple change of pointer. For now however, dynamic patching of the table entries has been sufficient to handle

color/monochrome conversions.

One problem that has persisted on the EGA occurs when an oldap changes the palette. In such cases, it is not worthwhile to attempt to work with the new palette entries via some complex mapping function. A grabber entrypoint called BeginPaint has been provided that will restore the default palette, among other things, so that calls to RealizeColor return usable values. Depending on how the oldap originally programmed the palette, it will be restored during the EndPaint call.

See also: PutBlock, BeginPaint, EndPaint, GRABBER.INC

Entry **BH** = logical color id to be mapped
DS = **CS**

Exit **BH** = mapped physical color
CF = 0

Error exit **BX** = 0
CF = 1 (implies logical color out of range)

Uses **AX**, **BX**, flags

GetBlock

GetBlock ( )

Purpose Copies a rectangular area of screen to 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:

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 to receive the screen data.
Xorg	Xorg specifies the unsigned x coordinate of the origin of the transfer in alpha coordinate space.
Yorg	Yorg specifies the unsigned y coordinate of the origin of the transfer in alpha coordinate space.

Xext	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	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 Get-Block operations is the fFormat field. If fFormat = FMT_NATIVE, the data is copied in the native screen format for the current screen mode. Thus alpha screens result in character/attribute pairs being copied, while graphics screens result in a bitmap. Note that native screen mode is most useful for saving areas where a popup menu will appear; using the data returned from GetBlock, a subsequent Put-Block with fFormat = FMT_NATIVE, will restore that area. Since the native format is screen dependent, it is recommended 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 alpha screens and gbType GT_NEWBITMAP format for graphics screens.
Char	Not used for GetBlock operations. ,LI Attr Not used for GetBlock operations.

Exit **AX** = number of bytes transfered

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

Uses **AX**, flags

Notes Note that when using FMT_OTHER in graphics mode, GetBlock attempts to reduce the amount of informationless data in the region by finding a minimum bounding box for the graphic image. Since the computation of the bounding box is based on the screen having been inverted by a Mark-Block request, the byte count returned when lpData = NULL will always reflect the size of the buffer required to capture a region of the requested size assuming no compression was possible. When lpData != NULL, it is assumed a standard grab is in progress, that the screen has been inverted, and that compression is desired. Thus in this case, the size returned may be LESS that that returned by a call with lpData = NULL, but no case will it exceed that size.

In text mode, the display adapter may contain multiple character sets or allow downloadble 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.

PutBlock

PutBlock ( )

Purpose PutBlock copies a specified buffer in a specified format to a rectangular block on the screen.

Entry **DS = CS**
ES:DI -> GrabRequest structure

In GrabRequest:

lpData	lpData must point to the buffer from which data will be copied.
Xorg	Xorg specifies the unsigned x coordinate of the origin of the transfer in alpha coordinate space.

Yorg	Yorg specifies the unsigned y coordinate of the origin of the transfer in alpha coordinate space.
Xext	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	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	Both the fFormat and fScreenOP fields of Style must be valid for Put-Block operations. fFormat: If fFormat = FMT_NATIVE, the data is copied in the native screen format for the current screen mode. This is useful for restoring sections of the screen previously saved using a GetBlock with fFormat = FMT_NATIVE. fScreenOp will be ignored for this format. If fFormat = FMT_OTHER, the fScreenOp field of Style specifies the screen operation requested and thus, explicitly specifies the format of the input data. fScreenOp: fScreenOp must be one of the 8 values defined in GRABBER.INC. The value of fScreenOp determines the expected format of the input buffer at lpData. Note that 5 of these 8 opcodes specify a fill operation with either a character operand, an attribute operand, or both. In these cases, the GrabRequest elements Char and Attr must contain the character or attribute to be used for the fill.

Char	If fScreenOp is either F_BOTH, F_CHAR, or C_ATTR_F_CHAR, this must contain the character code to be used for the fill operation.
Attr	If fScreenOp is either F_BOTH, F_ATTR, or C_CHAR_F_ATTR, this must contain the attribute code to be used for the fill operation.

Exit None.

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

Uses **AX**, flags

MarkBlock

MarkBlock ( )

Purpose MarkBlock toggles the color of a rectangular block of the screen such that the block can be differentiated from non-marked regions of the screen. The marking process is performed with a reversible operation (xor, not) such that a subsequent call to mark the same block effectively unmarks the the region. This allows Winoldap to perform a marking operation in a non-destructive manner so that the block need not be saved beforehand.

Entry **DS = CS**
ES:DI -> GrabRequest structure

In GrabRequest:

lpData	Not used for MarkBlock operations.
Xorg	Xorg specifies the unsigned x coordinate of the origin of the mark in alpha coordinate space.
Yorg	Yorg specifies the unsigned y coordinate of the origin of the mark in alpha coordinate space.
Xext	Xext specifies the unsigned x extent of the rectangular block to mark as measured from the origin specified by Xorg and Yorg. If Xext is zero, the entire width of the current screen is assumed.

Yext	Yext specifies the unsigned y extent of the rectangular block to mark as measured from the origin specified by Xorg and Yorg. If Yext is zero, the entire height of the current screen is assumed.
Style	Not used for MarkBlock operations.
Char	Not used for MarkBlock operations.
Attr	Not used for MarkBlock operations.

Exit None.

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

Uses **AX**, flags

BeginPaint

BeginPaint ( )

Purpose Prepare for Winoldap to call Get/Put/Mark functions

BeginPaint is something of a "catch-all" routine. It performs a number of loosely related functions that together help Winoldap maintain consistency and speed in menuing operations.

The Get/Put/MarkBlock functions do not attempt to validate the grabber's DC and IC information structures each time they are called, mainly for performance reasons (it may take hundreds of milliseconds just to validate these structures, especially on the CGAHERC-type of displays where the vertical retrace signal is sampled). The application may have changed screen modes or color palettes since the last time Winoldap had control of the processor; if these structures do not contain data that reflect the current state of the video subsystem, the results of the block operations will be undefined. Thus, it is crucial that these structures are validated just prior to calling the block functions. Since BeginPaint ALWAYS validates these structures, proper operation of the block routines can be insured by bracketing calls to them with calls to BeginPaint and EndPaint.

BeginPaint also allows us to undo any weirdness the oldap may have done to the adapter that might impair the menu displays. Specifically, no attempt has been made by the RealizeColor routine to dynamically track changes in a reprogrammable palette. Since RealizeColor performs the

logical to physical color mapping based on the default palette. BeginPaint will always switch the subsystem to the default. If the palette had been reprogrammed to non-standard values by the oldap, the effects of this call will be evident as a sudden change of color on the oldap screen. However, the menus will display as expected. (Currently, the EGA-style adapters are the only supported subsystems with this reprogrammable capability)

Finally, Windoldap provides its own software cursors during menuing operations. Thus a stray blinking hardware cursor is annoying and confusing, so BeginPaint hides it by sending it to an offscreen location.

BeginPaint attempts to save the current palette and cursor positions so that EndPaint can restore them. However, the save area is static, making BeginPaint non-reentrant. Until this is implemented with a stack, a nesting count is maintained that prevents clobbering data on nesting levels greater than 1.

Entry **DS = CS**

Exit None.

Uses **AX**, flags

EndPaint

EndPaint ( )

Purpose Cleans up after Winoldap is through painting.

EndPaint rectifies the action of BeginPaint. EndPaint attempts to restore the screen to its pre-menuing state. This includes restoring the original cursor position as well as the original palette (on EGA systems only).

Note that if the app has written directly to registers we can't read, it will be impossible to guarantee that the screen will look the same as it did. The cursor position is readable on all PC video adapters supported by Windows, so it is always possible to restore its location. The EGA palette is much more difficult; it can only be restored if: 1) EGA.SYS was installed, and 2) the oldap reprogrammed the palette via *int 010h* or the EGA Register Interface provided by EGA.SYS.

Entry **DS = CS**

Exit None.

Uses **AX**, flags

DATA STRUCTURES

GRABBER INFORMATION STRUCTURE

The extended grabber call GRAB_GETINFO fills a structure provided by Winoldap with the current video state information. This information can then be used by Winoldap to aid in mark/copy/paste operations, to determine the current display adapter in use, etc. All non-dimensionless quantities are 1-based.

```
GrabInfo        struc
  giDisplayId     db    ?       ;see below
  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
```

Codes for giDisplayId field in GrabInfo structure

```
DI_CGA          =       000h
DI_EGA          =       001h
DI_HERCULES       =       002h
DI_MULTIMODE        =       003h
DI_VGA          =       004h
DI_OLIVETTI       =       005h
```

Bitmaped codes for giScrType field in 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
```

GRABBER REQUEST PACKET STRUCTURE

Upon entry, all block functions expect es:di to point to a GrabRequest structure of the format below. Since not all 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
```

Codes for fScreenOps field of GrabRequest.Style

```
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
```

Codes for fFormat field of GrabRequest.Style

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

Error codes for block operations

```
ERR_UNSUPPORTED =     0FFh          ;block op not supported
```

```
ERR_BOUNDARY        =       0FEh        ;src or dest range error
```

GRAB BUFFER 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
```

Codes for gbType field of GrabSt

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

INFO CONTEXT STRUCTURE

InfoContext is a global structure shared by all the grabbers and may be considered the oldap analog of the GdiInfo structure for Windows drivers. It provides information needed by routines 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 routine. Note that it is essentially a superset of the GrabInfo structure returned by the GetInfo entrypoint. GrabInfo has plenty of reserved space so that this information may be made available to the Winoldap layer in the future as well as adding more fields 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 od pixels in Y
  icWidthBytes  db     ?           ;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
```

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.

DEVICE CONTEXT 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
```

DEVICE CONTEXT 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
```

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 hand 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 entrypoint GetInfo. Note 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]). Note 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 modes, the same cell-based convention applies so that Winoldap need not discern differences between graphics and alpha screens. The size of a character cell in graphics modes 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 specification of arbitrary block regions in graphics modes, it is still possible to request full-screen MarkBlock and GetBlock operations by using the procedure mentioned above. An attempt to specify any other block in graphics mode will return the ERR_UNSUPPORTED error code.

Although the ERR_BOUNDARY return code is provided in GRABBER.INC

for blocks that violate the screen boundaries, none of the block functions currently check for this condition. For now, the operation will proceed, possibly with undefined results.

COLOR MAPPING

The RealizeColor function maps Windoldap's logical menu colors to the display subsystem's physical colors. It does this via the PhysColorTable shown below. This is simply a translate table that defines a physical color attribute for each possible logical color defined in GRABBER.INC. Physical color assignments are left to the implementer's discretion, but some guidelines include:

1. The color assignments should produce the visual effect indicated by the name of the logical color. In other words, it should be clear to the user that menu items printed in LC_GRAYED are less bright (and therefore deactivated) than those items printed in LC_UNSELECTED.

2. Foreground/background saturation and intensity (contrast) are the most important parameters and depend upon the logical color being mapped; in general contrast should be high enough to discern text printed in any logical color.

3. The actual hues chosen are least important, but all combinations should be ergonomically pleasing. For monochrome displays that map hues to levels of gray, the guidelines in (2) should be followed. Also see Fundamentals of Interactive Computer Graphics, Foley, J.D. & Van Dam, A., pg 621.

4. When in doubt what effect a logical color should have, refer to menus generated by Windows itself or to the color selections chosen in the grabber source code.

5. If display hardware permits, LC_MNEMONIC should be an underlined version of LC_UNSELECTED. Otherwise, it should be an intensified version of LC_UNSELECTED.

```
LC_SELECTED          =     000h
LC_UNSELECTED        =     001h
LC_GRAYED            =     002h
LC_SELECTGRAY        =     003h
LC_TITLEBAR          =     004h
LC_SYSTEMUNSELECT    =     005h
LC_SYSTEMSELECT      =     006h
LC_APPTITLE          =     007h
```

```
LC_MNEMONIC         =       008h


PhysColorTable  label   word
            db      071h        ;PC_SELECTED:       blue on white
            db      017h        ;PC_UNSELECTED:     white on blue
            db      018h        ;PC_GRAYED:         grey on blue
            db      079h        ;PC_SELECTGRAY:     hi-blue on white
            db      06Fh        ;PC_TITLEBAR:       hi-white on brown
            db      00Fh        ;PC_SYSTEMUNSELECT: hi-white on black
            db      070h        ;PC_SYSTEMSELECT:   black on white
            db      00Fh        ;PC_APPTITLE:       hi-white on black
            db      01Fh        ;PC_MNEMONIC:       hi-white on blue
```

BUFFER SIZE CALCULATIONS

The low-level routines 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. due to differing requirements, the motivations for many of the assumptions may not be clear. Therefore, we present the following explanation in an attempt to clarify this process.

MAX_GBTEXTSIZE
MAX_GBGRPHSIZE

These two equates simply 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 is defined by MAX_VISTEXT and MAX_VISGRPH.

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 DC and 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 device specific BIOS areas, etc. For the

EGA. this includes the size of the EGA BIOS data area as well as the 4 bytes comprising the EGA SavePtr.

MAX_VISTEXT
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, and hence some video RAM is unused. Screen grabs are defined to copy only that portion of the screen which is visible. Unfortunately, the size of the visible page varies among display modes. Since InquireGrab can only return 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 wish to capture.

Note 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/linefeeds 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 hence the size of a visual page is unknown at the time of InquireGrab. One could simply specify a giant buffer, but that would be very 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.

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).

MAX_TOTTEXT
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 1 page of graphics. Thus saving the graphics page completely saves all 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, we could interpret the PIF setting to mean "multiple graphics/multiple text", and allocate space for the union of all text and graphics pages. Or we could elect to save the user some memory and allocate space for only the largest single graphics page (interpreting as "single graphics/multiple text"). Unfortunately both interpretations usually require prohibitive amounts of memory. On the EGA, the first method would require that we save all 256K, while the second would require that we save 128K, since the largest graphics page is in mode 010h (640x350x4) and requires 4 planes of 32K (including offscreen memory). Thus, the situation must be examined for each display adapter and a compromise reached.

As a compromise on the EGACOLOR grabber, we are forced to interpret the PIF setting as "single page of lores graphics/some of the text pages". This method requires we allocate only 16K, which is enough for one page of lores graphics or 4 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 takes far less of a toll on the user's memory. Unlike screen grabs, this technique is safe to implement because Winoldap must also swap to disk, and thus insures we do not reenter DOS for file i/o.

The only compromise required on the EGAMONO grabber is that we interpret the PIF setting as "single page of graphics/all text pages". This means it is possible to lose a graphics page if the oldap is maintaining 2 pages of graphics, but this is a rare enough 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, we are forced to 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 way too much memory to request. It is rare that the user runs an app that maintains images on both graphics pages, and it would be ashame to penalize him/her with 64K "just to be safe".

No compromise is required on the CGA and Multimode adapters since we save all 16K and 32K of the adapters' memory, respectively.

GrabTextSize
GrabGrphSize

The GrabTextSize variable holds the minumum buffer size we need to support a text grab, while the GrabGrphSize variable holds the minimum buffer size we need 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 which need consideration.

SaveTextSize
SaveGrphSize

The SaveTextSize variable holds the minumum buffer size we need to support a text context switch, while the SaveGrphSize variable holds the minimum buffer size we need 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 which 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 apps use the ERI to modify EGA registers (such as MS-WORD), in which case this context data will be more accurate than our DC and IC structures alone.

EXAMPLE FROM EGACOLOR GRABBER

```
MAX_GBTEXTSIZE  =     gbWidth
MAX_GBGRPHSIZE  =     gbBits

MAX_CDSIZE0     =     (SIZE DeviceContext) + (SIZE InfoContext)
MAX_CDSIZE1     =     (SIZE VideoBiosData) + (SIZE EgaBiosData) + 4
MAX_CDSIZE2     =     (SIZE HpBiosData)
MAX_CDSIZE      =     MAX_CDSIZE0 + MAX_CDSIZE1 + MAX_CDSIZE2

MAX_VISTEXT     =     80*43 + 02*43 + 2     ;43 lines + crlf each + null
MAX_VISGRPH     =     (640*200)/8           ;same as 16*1000
```

```
MAX_TOTTEXT     =       80*43*2 - 256       :256 takes us to end of page
MAX_TOTGRPH     =       16*1024

GrabTextSize    dw      MAX_GBTEXTSIZE + MAX_VISTEXT
GrabGrphSize    dw      MAX_GBGRPHSIZE + MAX_VISGRPH
SaveTextSize    dw      MAX_CDSIZE     + MAX_TOTTEXT
SaveGrphSize    dw      MAX_CDSIZE     + MAX_TOTGRPH
```

Appendix A
Terms and Abbreviations

A.1 Introduction

This appendix defines some of the terms and abbreviations used in this guide.

A.2 Glossary

Clipping	The process of removing any portion of a graphic image which extends beyond a specified boundary.
Device	For purposes of output, a GDI Device is a combination of a GDI Support Module for a particular graphics peripheral, a communications port or direct memory access, and the actual peripheral. GDI Devices are thus equivalent to GKS Workstations or CORE display surfaces (and include the pseudo-display surfaces such as metafiles).
Device Driver	The software that provides the interface between GDI functions and the graphics output device.
Direct Color	A color representation scheme in which the color values are specified directly, without requiring an intermediate mapping via a color table.
GDI	Graphics Device Interface - A set of graphics routines that provides the interface between application programs and output devices.
Indirect Color	A color representation scheme in which the color index is used as an index to the corresponding color values. The color table (red, green, blue) is used in mapping from the index to the color values.
Metafile	A metafile stores the primitives which define the picture.
MS-DOS	Microsoft Disk Operating System.
OEM	Original Equipment Manufacturer.
Pixel	The smallest element of a physical display surface which can be independently assigned color or intensity.

Pixel Array	A matrix of pixels which defines the color for a region on an actual display. There is exactly one pixel definition for each addressable picture element of a raster display covered by the pixel array.
PLP	Presentation Level Protocol - used for transmitting high quality text.
Primitive	A basic graphic function to be performed.
Raster Device	A raster device uses a matrix of pixels covering the entire screen area (display surface) to draw graphics. Pixels (points) are turned on/off, bit-by-bit.
Resolution	Number of visibly distinct dots that can be displayed in a given area of the screen. Typical resolution = 100 dots/inch.
RGB	Red, Green, Blue values from a color table.
Support Module	A support module is a dynamically loadable unit of code and data which is required to support a function or set of functions for Microsoft Windows.
Scaling	Coordinate scaling transforms points from one level to another. GDI scales coordinates from NDC space to values which are appropriate for your graphics device.
VDI	Virtual Device Interface - the ANSI graphics interface upon which the GDI is based. The VDI is a standard interface between device-dependent and device-independent code in a graphics environment. VDI makes all device drivers appear identical to the application program.
VDM	Virtual Device Metafile.
Vector Device	A vector device draws graphics with lines. Beginning and ending points are set and a line is drawn.
Window	Rectangular region on a display screen in which the contents of an application are displayed.
Workstation	The combination of a GDI device driver, a physical device driver and an actual display.

Appendix B
Reference Information

B.1 Introduction

This appendix lists the manuals, guides, and documents referenced in this guide.

B.2 Applicable Documents

MS-DOS	Operating System - Programmer's Reference Manual.
ANSI-VDM	Draft Proposed American National Standard for the Virtual Device Metafile, ANSI-X3H33.
ANSI-VDI	Evolving draft for the Virtual Device Interface, ANSI-X3H33.
CORE	SIGGRAPH/ACM CORE - Status Report of the Graphics Standards Planning Committee. V. 13, #3, August 1979.
GKS	Graphical Kernel System, ISO/DIS 7942, Version 7.2 - 1982.
NAPLPS	North American Presentation Level Protocol Standard, X3L2/82-135.

B.3 Compliance with Standards

GDI grows out of existing graphics standards, including ANSI-VDI and VDM interim specifications, NAPLPS, and the graphic languages CORE, GKS, and PLOT-10. GDI device drivers will comply with the relevant ANSI standards to the extent that they have been defined.

Appendix C
Microsoft Windows Key Codes

C.1 Introduction

This appendix describes the Windows virtual key set. The virtual key set consists of 256 virtual keys, numbered from 0 to 255. The keys from 0 to 127 are called the Windows-defined virtual keys. The keys from 128 to 255 are called OEM-defined virtual keys. The Windows-defined virtual keys are divided into three groups: standard, extended, and reserved.

Every OEM adaptation of Windows must be capable of generating:

Standard Keys
: The Windows-defined standard virtual keys.

Letter Combination Keys
: Key combinations consisting of a SHIFT, CONTROL, or CONTROL-SHIFT and a letter key (65 through 90 decimal, or 41h through 5Ah).

Cursor Combination Keys
: Key combinations consisting of a SHIFT, CONTROL, or CONTROL-SHIFT and a cursor key.

Function Keys
: A minimum of 10 function keys, f1 through f10. It is possible to query the OEM layer to determine the actual number of supported function keys. (The first 10 function keys are numbers 112 through 121, or 70h through 79h, as shown in the table below.)

Function Combination Keys
: Key combinations consisting of a SHIFT, CONTROL, or CONTROL-SHIFT and a function key.

Implementation of all other keys is entirely OEM-dependent.

C.2 Detailed Listing

In the listing below, the unused entries are empty, the extended keys are marked with an asterisk (*), and all others are standard. The extended keys 96 through 111 are numeric key pad digits and operators found on a keypad.

The numbers 0 through 7 across the top are the three high bits, and 0 through F down the left side are the four low bits in the seven bit code.

	0	1	2	3	4	5	6	7
0		shift	space	0		P	*numpad0	f1
1	*lbutn	control	prior	1	A	Q	*numpad1	f2
2	*rbutn	menu	next	2	B	R	*numpad2	f3
3	cancel	*pause	*end	3	C	S	*numpad3	f4
4		capital	home	4	D	T	*numpad4	f5
5			left	5	E	U	*numpad5	f6
6			up	6	F	V	*numpad6	f7
7			right	7	G	W	*numpad7	f8
8	backsp		down	8	H	X	*numpad8	f9
9	tab		*select	9	I	Y	*numpad9	f10
A			*print		J	Z	* *	*f11
B		escape	execute		K		* +	*f12
C	*clear				L		* ,	*f13
D	return		ins		M		* -	*f14
E			del		N		* .	*f15
F			*help		O		* /	*f16

If an OEM has a keyboard that has the following symbols on a single key-cap, then the following are suggested as part of the country-specific OEM-defined virtual key set.

	U.S.	Japan	France	Germany	Spain	Italy	Sweden
BA	; :	: *	$ *	< >	; :	< >	' *
BB	= +	; +	= +	+ *	= +	+ *	+ ?
BC	, <	, <	, ?	, ;	, ?	, ;	, ;
BD	- _	- =	- _	- _	- _	- _	- _
BE	. >	. >	; .	. :	. !	. :	. :
BF	/ ?	/ ?	: /	ss ?	< >	' ?	< >
C0	bq ~	@ bq	bq lb	ac gr	gr cr	ugr sec	ac gr
DB	[{	[{	) deg	Aum	ac um	agr #	Aum
DC	\ |	yn |	ugr %	Oum	Ntl	ogr @	Oum
DD	] }	] }	< >	Uum	Ccd	egr eac	Arn
DE	' "	^ ~	cr um	# ^	' "	igr ~	um cr
DF		_					

ss-German sharp s
bq-back quote
lb-British Pound
yn-Japanese Yen
sec-section symbol
deg-degree symbol
ac-acute accent
gr-grave accent
cr-circumflex
cd-cedilla
um-umlaut (diaeresis)
rn-ring
tl-tilde

Notes

Windows uses the OEM-supplied **ToAscii** routine to convert virtual

key codes into the corresponding ASCII value. To make this converusion simple, some virtual keys have the same value as the corresponding ASCII character. For example, the virtual key code for VK_A is equal to 65, the ASCII value of the capital letter "A."

Note

Since Windows can be run using the keyboard interface only, the mouse keys, 1 and 2, are not part of the standard set.

Appendix D
Raster Operation Codes and Definitions

D.1 Introduction

This appendix 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:

```
S       Source bitmap
P       Paintbrush or Pattern currently selected
D       Destination bitmap
```

The Boolean operators used in these operations are:

```
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.

Note that there are alternate spellings of the same function, so although a particular spelling may not be in the list, an equivalent form will be. 'DPoSo' is an equivalent form to 'DPSoo', for example.

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.

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.

D.1.1 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 which represents the result of the Boolean operation on predefined pattern, source, and destination values. For example, the operation indices for the *PSo*, *PSon*, and *DPSoo* operations are:

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) Note 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), and by looking in the table one 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 and 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.

D.2 Operation Code List

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	-
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	-
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	-
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	PDSPDoaxxn	-
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	-
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	-
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	-
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	-
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

Appendix E
Printer Driver Escapes

E.1 Escapes

An escape is very much like a subfunction of the Control function (see Chapter 5, Output Functions), but is passed directly to the device driver, which supports it without help from GDI.

```
**************************************************************************
          Draft list of valid escapes for Windows 2.0
**************************************************************************
ABORTDOC
BANDINFO
DEVICEDATA
DRAFTMODE
DRAWPATTERNRECT
ENABLEDUPLEX
ENABLEMANUALFEED
ENABLEPAIRKERNING
ENBLERELATIVEWIDTHS
ENDDOC
EXTTEXTOUT
FLUSHOUTPUT
GETCOLORTABLE
GETEXTENDEDTEXTMETRICS
GETEXTENTTABLE
GETPAIRKERNTABLE
GETPHYSPAGESIZE
GETPRINTINGOFF
GETSCALINGFACTOR
GETTECHNOLOGY
GETTRACKKERNTABLE
MFCOMMENT
NEWFRAME
NEXTBAND
QUERYESCSUPPORT
SELECTPAPERSOURCE
SETABORTPROC
SETALLJUSTVALUES
SETCHARSET
SETCOLORTABLE
SETCOPYCOUNT
SETKERNTRACK
STARTDOC
STRETCHBLT

**************************************************************************
      Documentation for all escapes (both valid and invalid)
**************************************************************************
```

Control (*lpDevice, ABORTDOC, 0, 0*) : *Return*

Description 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 (SETABORTPROC escape), and to terminate printing operations that have not yet reached their first NEWFRAME or NEXTBAND call.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap.

Return The return value is positive if the function is successful, negative otherwise.

Notes On a printing error, the ENDDOC escape should not be used to terminate the printing operation, and the ABORTDOC escape should be used to terminate a next banding operation only.

Control (*lpDevice, BANDINFO, lpInData, lpOutData*)

Description 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 in order 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 there is graphics in the next band, a flag indicating whether there is text on the page, and a rectangle structure that contains a bounding rectangle for all graphics on the page.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* is set to NULL. *lpInData* is a long pointer to a data structure containing the following items:

Field	Type/Definition
# <fGraphFlag>#	**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 page.

lpOutData is a long pointer to a data structure containing the following items:

Field	Type/Definition
# <fGraphFlag>#	**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 current band.

lpOutData should be NULL if no data is returned.

Return 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.

Control (*lpDevice*, *DEVICEDATA*, *lpInData*, *lpOutData*) : *Return*
This function allows the application to send data directly to the printer, bypassing the standard print-driver code.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* is the number of bytes of device data to be passed to the printer. *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* is not used and can be set to NULL.

Return This is the number of bytes transferred to the printer if the function is successful; 0 if not or if the escape is not implemented. If the returned value is nonzero but less than *nCount*, 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, for example, downloading a font or a macro.

The driver should invalidate its internal state variables such as "current position," "current line style," etc., when executing this escape. The driver may issue a printer "save" command prior to transmitting the first of a sequence of DEVICEDATA escapes and issue a "restore" command prior to executing the first command after the last DEVICEDATA escape. In other words, the application must be able to issue multiple, sequential DEVICEDATA escapes without intervening "saves" and "restores" inserted by the driver.

Control (*lpDevice, DRAFTMODE, lpDraftMode, 0*)

Description 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *lpDraftMode* The return value is a 32-bit address to a short integer value specifying the draft mode. It is 1 for draft mode on, 0 for draft mode off.

Return The return value is positive if the function is successful, negative otherwise.

Comments The default draft mode is off.

Control (*lpDevice, DRAWPATTERNRECT, lpInData, lpOutData*)
: *Return*

Description This escape creates a pattern, gray scale, or solid black rectangle using the pattern rule capabilities of PCL printers. 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* is not used, and can be set to NULL. *lpInData* is a long pointer to a data structure containing the following items:

Field

Type/Definition
#<prPosition># **POINT** Is a point structure identifying the upper-left corner of the rectangle.
#<prSize># **POINT** Is 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:

Pattern	Value
Black Rule	0
Gray Scale	2
HP-Defined	3

#<prPattern># **WORD** Is ignored for a black rule. It represents the percent of gray for a gray scale pattern. It represents a one of six patterns for HP-defined patterns.

lpOutData is not used, and may be set to null.

Return The return value is one if the escape function is successful; otherwise it is zero.

Notes An application should use QUERYESCSUPORT to determine whether a device is capable of drawing patterns and rules before implementing this escape. 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. An application should use the function calls provided in GDI to obtain this effect.

Control (*lpDevice, ENABLEDUPLEX, lpInData, lpOutData*) : *Return*

Description 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* Specifies whether duplex or simplex printing is used. A non-zero value indicates that duplex printing should be enabled; zero indicates that simplex should be enabled. *lpInData* is not used, and may be set to null. *lpOutData* is not used, and may be set to null.

Return The return value is one if the escape function is successful; otherwise it is zero.

Notes An application should use the QUERYESCSUPPORT escape to determine whether 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 override of any values set in the driver specific dialog. If duplex printing is enabled and and uneven number of NEXTFRAME escapes are sent to the driver prior to the ENDDOC escape, the driver will add one page eject before ending the print job.

Control (*lpDevice, ENABLEPAIRKERNING, lpInData, lpOutData*)
: *Return*

Description This function enables or disables the driver's ability to kern character pairs automatically. When it is enabled, the driver will automatically kern those pairs of characters that are listed in the font's character-pair kerning table. The driver will reflect this kerning both on the printer and in **GetTextExtent** calls.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* contains the number of bytes used by *lpInData*. This value should be set to 2. *lpInData* is a long pointer to a short integer value that specifies whether automatic pair kerning is to be enabled (1) or disabled (zero). *lpOutData* is a long pointer to a short integer variable that will receive the previous automatic-pair-kerning flag.

Return The return value is 1 if the function is successful; zero if not or if the escape is not implemented.

Notes 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. In the case where the GETPAIRKERNTABLE escape is supported but the ENABLEPAIRKERNING escape is not, it is the application's responsibility to properly space the kerned characters on the output device.

Control (*lpDevice*, *ENABLERELATIVEWIDTHS*, *lpInData*, *lpOutData*)
: *Return*

Description This function enables or disables relative character widths. When it is disabled (the default), 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 allows applications to manually build an extent table 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* contains the number of bytes used by *lpInData*. This value should be set to 2. *lpInData* is a long pointer to a short integer value that specifies whether relative widths are to be enabled (1) or disabled (zero). *lpOutData* is a long pointer to a short integer variable that will receive the previous relative-character-width flag.

Return The return value is 1 if the function is successful; zero if not or if the escape is not implemented.

Notes The default state of this capability is zero, that is, relative character widths are disabled.

Enabling this feature causes values specified as "font units" and accepted and returned by the escapes described in this section 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.

It has not yet been decided whether this capability will ever become part of Windows GDI.

Control (*lpDevice, ENDDOC, 0, 0*) : *Return*

Description This escape ends a print job started by a STARTDOC escape.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap.

Return The return value is positive if the function is successful, negative otherwise.

Notes On a printing error, the ENDDOC escape should not be used to terminate the printing operation, and the ABORTDOC escape should be used to terminate a next banding operation only.

EXTTEXTOUT (*lpDevice, EXTTEXTOUT, lpInData, lpOutData*) : *Return*

Description This function provides a more efficient way for the application to invoke the GDI **TextOut** function when justification, letterspacing, and/or kerning is involved.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* contains the number of bytes used by *lpInData*. *lpInData* is a long pointer to a data structure described in the Notes section below. *lpOutData* is not used, and may be set to NULL.

Return The return value is 1 if the function is successful; zero if not or if the escape is not implemented.

Notes The structure pointed to by *lpIndata* contains the following items:

```
WORD X, Y;
LPSTR cch;
RECT clipRect;
WORD lpText;
WORD FAR *lpWidths;
```

The *X* argument specifies the *x*-coordinate of the upper-left corner of the string's starting point. The *Y* argument specifies the *y*-coordinate of the upper-left corner of the string's starting point. The *lpText* argument is a long pointer to an array of *cch* character codes. The *cch* argument is the number of bytes in the string (*cch* is also the number of words in the width array). The *lpWidths* argument is a long pointer to an array of *cch* character widths to use when printing the string. The first character appears at (*X*, *Y*), the second at (*X*+*lpWidths*[0], *Y*), the third at (*X*+*lpWidths*[0]+ *lpWidths*[1], *Y*), and so on. These character widths are specified in the font units of the currently

selected font. (The character widths will always be equal to device units unless the application has enabled relative character widths.)

The units contained in the width array are specified as font units of the device (see the EXTTEXTMETRIC escape, above) and are dependent on whether relative character widths are enabled (see the ENABLERELATIVEWIDTHS escape, above).

Control (*lpDevice, FLUSHOUTPUT, 0, 0 Return*)

Description This escape flushes output in the device's buffer.

Parameter *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap.

Return The return value is positive if the function is successful, negative otherwise.

Control (*lpDevice, GETCOLORTABLE, lpIndex, lpColor*) : *Return*

Description This escape retrieves an RGB color color table entry and copies it to the location specified by *lpColor*.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *lpIndex* is a 32-bit address to a short integer value specifying the index of a color table entry. Color table indices start at 0 for the first table entry. *lpColor* is a 32-bit address to the long integer to receive the RGB color value for the given entry.

Return The return value is positive if the function is successful, negative otherwise.

Control (*lpDevice, GETEXTENDEDTEXTMETRICS, lpInData, lpOut-Data*) : *Return*

Description This function fills the buffer pointed to by *lpOutData* with the extended text metrics for the currently selected font.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *InCount* is not used, and may be set to NULL. *lpInData* is a long pointer to a data structure described in the following Notes section. *lpOutData* is a long pointer to a data structure of type **EXTTEXTMETRIC**.

Return 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.

Notes The structure pointed to by *lpIndata* contains the following item:

```
WORD nSize;
```

The *nSize*parameter 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 the ENABLERELATIVEWIDTHS escape, above.

Control (*lpDevice*, *GETEXTENTTABLE*, *lpInData*, *lpOutData*) : *Return*

Description This function 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**. the destination device bitmap. *nCount* is not used, and mayh be set to NULL. *lpInData* is a long pointer to a data structure described below in the following Notes section. *lpOutData* is a long pointer to an array of type **short**. The size of the array must be at least (*chLast*– *chFirst* + 1).

Return The return value is 1 if the function is successful; zero if it is not or if the escape is not implemented.

Notes 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 the ENABLERELATIVEWIDTHS escape, above.

Control (*lpDevice, GETPAIRKERNTABLE, lpInData, lpOutData*)
: *Return*

Description This function fills the buffer pointed to by *lpOutData* with the character-pair kerning table for the currently selected font.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *InCount* is not used, and may be set to NULL. *lpInData* is not used, and may be set to NULL. *lpOutData* is a long pointer to an array of **KERNPAIR** structures. This array must be large enough to accomodate 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 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.

Notes The values returned in the **KERNPAIR** structures are affected by whether relative character widths are enabled or disabled. See the ENABLERELATIVEWIDTHS escape, above.

Control (*lpDevice, GETPHYSPAGESIZE, 0, lpDimensions*) : *Return*

Description This escape retrieves the physical page size and copies it to the location pointed to by *lpOutData*.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *lpDimensions* is 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 The return value is positive if the function is successful, negative otherwise.

Control (*lpDevice, GETPRINTINGOFFSET, 0, lpOffset*) : *Return*

Description This escape retrieves the offset, from the upper left hand corner of the physical page, where 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *lpOffset* is a 32-bit address to the **POINT** structure to receive the printing offset. The fields of the structure are filled as follows:

Field	Contents
int x	Horizontal coordinate in device units of the printing offset.
int y	Vertical coordinate in device units of the printing offset.

Return The return value is positive if the function is successful, negative otherwise.

Control (*lpDevice, GETSCALINGFACTOR, 0, lpFactors*) : *Return*

Description 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* if a scaling factor is 8.

Scaling factors are used by printing devices that cannot support graphics at the same resolution as the device resolution.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *lpFactors* is a 32-bit address to the **POINT** data structure to receive the scaling factor. The fields of the structure are filled as foolows:

Field	Contents
int x	Scaling factor for x axis.
int y	Scaling factor for y axis.

Return The return value is positive if the function is successful. negative otherwise.

Control (*lpDevice, GETTRACKKERNTABLE, lpInData, lpOutData*) : *Return*

Description This function fills the buffer pointed to by *lpOutData* with the track-kerning table for the currently selected font.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* is not used, and may be set to NULL. *lpInData* is not used, and may be set to NULL. *lpOutdata* is a long pointer to an array of **KERNTRACK** structures. 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** structure returned by the GETEXTENDEDTEXTMETRICS escape.

Return The return value is the number of **KERNTRACK** structures copied to the buffer. This value is zero is the font does not have kerning tracks defined, the function fails, or the escape is not implemented.

Notes The values returned in the **KERNTRACK** structures are affected by whether relative character widths are enabled or disabled. See the ENABLERELATIVEWIDTHS escape, above.

Control (*lpDevice, MFCOMMENT, lpComment, 0*) : *Return*

Description This escape is of type BOOL.

A windows application uses this function to add a comment to a metafile.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, containing the display context for the device on which the metafile appears. *nCount* contains the number of characters in the string pointed to by *lpComment*. *lpComment* is a 32-bit address to a null-terminated string containing the comment which will appear in the metafile. *lpOutData* is not used.

Return The return value is –1 if an error (such as insufficient memory or an invalid port specification) occurs. Otherwise, it is positive.

Control (*lpDevice, NEWFRAME, 0, 0*) : *Return*

Description This escape informs the device that the application has finished writing to a page.

This escape is typically used with a printer to direct the device driver to advance to a new page.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap.

Return The return value is positive if the escape is successful. Otherwise, it is one of the following:

SP_ERROR	General error.
SP_APPABORT	The job was aborted because the application's abort function returned zero.
SP_USERABORT	The user aborted the job through the spooler task.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.

Control (*lpDevice, NEXTBAND, 0, lpBandRect*) : *Return*

Description This escape informs the device driver that the application has finished writing to a band, causing the device driver to send the band to the spooler and return the coordinates of the next band.

This escape is used by applications that handle banding themselves.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *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 The return value is a positive if the escape is successful. Otherwise, it is one of the following:

SP_ERROR	General error.

SP_APPABORT	The job was aborted because the application's abort function returned zero.
SP_USERABORT	The user aborted the job through the spooler task.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.

Control (*lpDevice, QUERYESCSUPPORT, lpEscNum, NULL*) : *Return*

Description This escape finds out whether a particular escape function is implemented by the device driver.

The return value is non-zero for implemented escape functions, zero for unimplemented escape functions.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* is equal to 2 for this function. *lpEscNum* is a 32-bit address to a short integer value specifying the escape function to be checked.

Return The return value is non-zero for implemented escape functions, zero otherwise.

Control (*lpDevice, SELECTPAPERSOURCE, lpInData, lpOutData*) : *Return*

Description This function allows the application to determine the available paper sources and select among them.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* is the number of bytes used by *lpInData*. This value should be set to 2. *lpInData* is a long pointer to a short integer value containing the desired paper-source index. Index 0 specifies manual feed, if available. Index 1 specifies the primary or only paper tray. Indices greater than 1 refer to additional paper sources. *lpOutData* is a long pointer to a data structure described in the Notes section below.

Return The return value is 1 if the function is successful, –1 if the requested paper source does not exist, or zero if the escape is not implemented. The data structure pointed to by *lpOutData* is valid only if the function returns 1.

Notes The structure pointed to by *lpOutData* contains the following items:

```
WORD X, Y;
RECT imageRect;
WORD orientation;
```

The *X* argument specifies the horizontal paper size in device units. The *Y* argument specifies the vertical paper size in device units. The *imageRect* argument specifies the size of the rectangular region on a page that can be printed on. The *orientation* argument specifies whether the page orientation is portrait (1) or landscape (2).

The application may enumerate the available paper sources by invoking this escape with indices 0, 1, 2, etc., after which the escape returns a 1.

For the manual feed index (0), the returned values refer to the maximum paper size that can be fed manually. It is the user's responsibility to manually feed the proper size paper.

The driver defaults to the paper-source index (1), the primary paper tray. The driver will reset to this value if it returns a –1 because of an illegal paper-source request.

Paper sources can be changed only between frames.

Control (*lpDevice*, *SETABORTPROC*, *lpAbortFunc*, *0*) : *Return*

Description 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. The spooler 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *lpAbortFunc* is a long pointer to a procedure instance of the application-supplied abort function. See "Notes", below, for details.

Return The return value is positive if the function is successful, negative otherwise.

Notes The address passed as the *lpAbortFunc* parameter must be created using the **MakeProcInstance** function.

The abort function has the following form:

```
short  FAR PASCAL <AbortFuncName> (<hPr>, <code>)
lpDevice hPr;
```

```
short code;
```

where:

hPr	is a handle to the printer device context.
code	is zero if no error has occurred. It is SP_OUTOFDISK if the spooler is currently out of disk space, but more will become available if the application is willing to wait.

The return value should be non-zero if the print job is to be continued, and zero if it is cancelled.

Control (*lpDevice, SETALLJUSTVALUES, lpInData, lpOutData*)
: *Return*

Description This escape sets all of 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* is set to NULL. *lpInData* is a long pointer to a data structure containing the following items:

Field	Type/Definition
#<nCharExtra>#	**short** Specifies the total extra space in font units that must be distributed over *nCharCount* characters.
#<nCharCount>#	**WORD** Specifies the number of characters over which *nCharExtra* is distributed.
#<nBreakExtra>#	**short** Specifies the total extra space in font units that is distributed over *nBreakCount* break characters.
#<nBreakCount>#	**WORD** Specifies the number of break characters over which *nBreakExtra* units are distributed.

lpOutData is not used, and may be set to null.

Return The return value is one if the escape function is successful; otherwise it is zero.

Notes The units used for *nCharExtra* and *nBreakExtra* are the font units of the device (see the EXTTEXTMETRIC escape) and are dependent on whether relative character widths were enabled with the ENABLERELATIVEWIDTHS escape.

The values set with this escape will apply to subsequent calls to **TextOut**. The driver will stop distributing the *nCharExtra* amount when it has output *nCharCount* characters. Likewise, it will 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 **TextOut** call will be processed in the same manner.

In order to reenable justification with the **SetTextJustification** and **SetTextCharacterExtra** functions, an application should call SETALLJUSTVALUES and set the arguments *nCharExtra* and *nBreakExtra* to zero.

Control (*lpDevice, SETCOLORTABLE, lpColorEntry, lpColor*) : *Return*

Description 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *lpColorEntry* is a 32-bit address to a color table entry data structure. The structure has the following fields:

Field	Meaning
#<Index>#	**WORD** Is the color table index. Color table entries start at 0 for the first entry.
#<rgb>#	**LONG** Is the desired RGB color value.

lpColor is a 32-bit address to the long integer to receive the RGB color value selected by device driver to represent the requested color value.

Return The return value is positive if the function is successful, negative otherwise.

Notes A device's color table is a shared resource; changing the system display color for one window changes it for all windows.

The SETCOLORTABLE escape has no effect on devices with fixed color tables.

Control (*lpDevice*, *SETCOPYCOUNT*, *lpInData*, *lpOutData*) : *Return*

Description This function specifies the number of uncollated copies of each page that the printer is to print.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* The number of bytes used by *lpInData*. This value should be set to 2. *lpInData* ia a long pointer to a short integer value containing the number of uncollated copies to be printed. *lpOutData* is 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 The return value is 1 if the function is successful; zero if not or if the escape is not implemented.

Control (*lpDevice*, *SETKERNTRACK*, *lpInData*, *lpOutData*) : *Return*

Description 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 function is enabled, the driver will automatically kern all characters according to the specified track. The driver will reflect this kerning both on the printer and in **GetTextExtent** calls.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* The number of bytes used by *lpInData*. This value should be set to 2. *lpInData* is a long pointer to a short integer value that specifies the kerning track to use. A value of zero disables this feature. Values 1 to *nKernTracks* (see the **EXTTEXTMETRIC** structure in the "Data Types and Structures" section of this book) correspond to positions in the track-kerning table (using 1 as the first item in the table). *lpOutData* is a long pointer to a short integer variable that will receive the previous kerning track.

Return The return value is 1 if the function is successful; zero if not or if the escape is not implemented.

Notes The default state of this capability is zero, that is, 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 the SETKERNTRACK escape is not, it is the application's responsibility to properly space the characters on the output device.

Control (*lpDevice, STARTDOC, lpDocName, 0*) : *Return*

Description 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.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *nCount* contains the number of characters in the string pointed to by *lpDocName*. *lpDocName* is a 32-bit address to a null-terminated string specifying the name of the document. The document name is displayed in the spooler window.

Return The return value is –1 if an error occurs, such as, insufficient memory or an invalid port specification. Otherwise, it is positive.

Notes The correct sequence of events in a printing operation are 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. Begin each new page with NEWFRAME, or each new band with NEXTBAND.
5. End the printing operation with ENDDOC.

On a printing error, the ENDDOC escape should not be used to terminate the printing operation, and the ABORTDOC escape should be used to terminate a next banding operation only.

Control (*lpDevice, STRETCHBLT, lpInData, lpOutData*) : *Return*

Description This function implements the GDI function **StretchBlt** on the driver level.

Parameters *lpDevice* is a long pointer to a data structure of type **PDEVICE**, the destination device bitmap. *InCount* contains the number of bytes used by *lpInData*. *lpInData* is a long pointer to a data structure described below in the Notes section. *lpOutData* is not used, and may be set to NULL.

Return The return value is 1 if the function is successful; zero if not or if the escape is not implemented.

Notes The data structure pointed to by *lpInData* contains the following items:

```
WORD X, Y;
WORD nWidth, nHeight;
WORD XSrc, YSrc;
WORD nSrcWidth;
WORD nSrcHeight;
DWORD dwRop;
short bmType;
short bmWidth;
short bmHeight;
short bmWidthBytes;
short bmPlanes;
short bmBitsPixel;
short bmBits;
```

The members of this structure correspond to the arguments to the **StretchBlt** function described on page 117 of the *Microsoft Windows Programmer's Reference*. The *bmBits* member of this **BITMAP** structure is a long pointer to the bitmap bits. This pointer must be set by the application. This escape should be implemented only for devices with bitmap-stretching capabilities.

The driver need not implement all of GDI's raster operations, but must return zero if the specific ROP (raster operation) cannot be implemented. The driver must not substitute one ROP for another.

Appendix F
The Font File Format

F.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** routine.

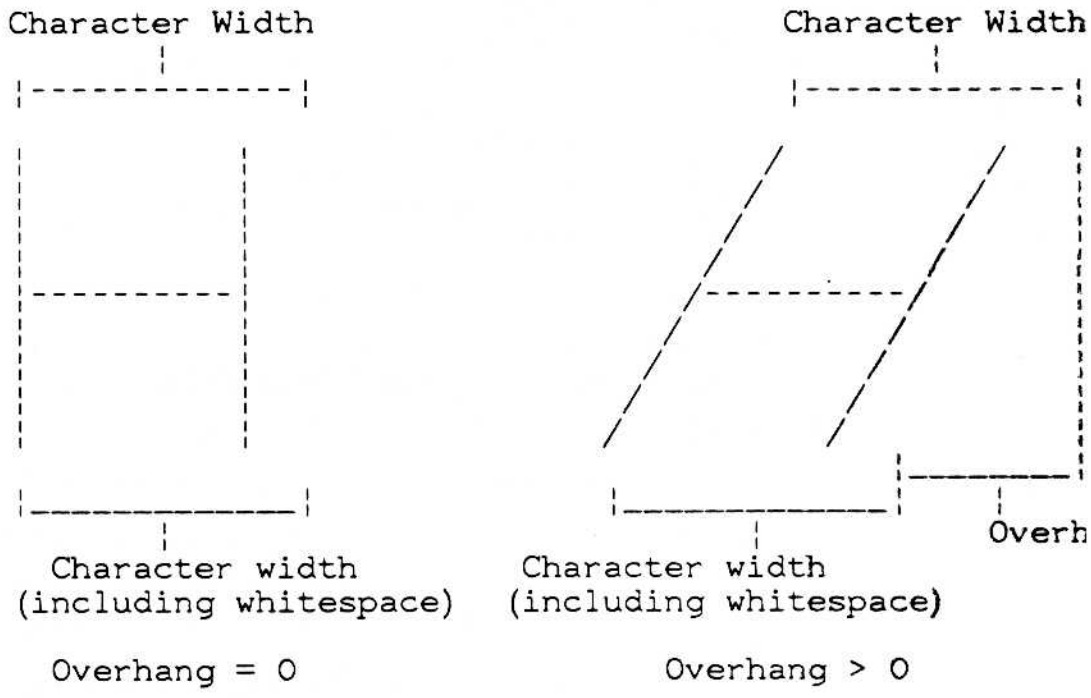
```
typedef struct {
        short tmHeight;
        short tmAscent;
        short tmDescent;
        short tmInternalLeading;
        short tmExternalLeading;
        short tmAveCharWidth;
        short tmMaxCharWidth;
        short tmWeight;
        BYTE  tmItalic;
        BYTE  tmUnderlined;
        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 sizes are given in normalized units (*i.e.*, they depend on the current mapping mode of the display context).

tmHeight	Specifies the height of characters (Ascent + Descent).
tmAscent	Specifies the ascent of characters (units above the baseline).
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

	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 which is to be substituted for characters which are not in the font.
tmBreakChar	Specifies the value of the character which 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 4 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.
tmOverhang	Specifies the per string extra width which may be added to some synthesized fonts. When synthesizing some attributes such a 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, and 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. *tmOverhang* allows the application to determine 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. The actual width is the extent less the overhang. Alternately, *tmOverhang* is the difference between the width of a character when output singly verses in the interior of a string.



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**.

F.1.1 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** is responsible for accommodating the differences for which it has claimed abilities, as specified in the *dpText* field in the **GDIINFO** data structure. There may be more differences than the device can transform, in which 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. Note 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:

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'.
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 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 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 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).

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. Note that GDI uses additional overhang if it emboldens the font.

F.2 Specifics of the Format:

```
FONTINFO
typedef struct  {
  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    d
  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.
  reserved            db      0   ; 1 byte reserved
  dfCharOffset        dw      0   ; Area for storing the character offsets,
                                  ; facename, 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 1 character element.

This leaves room only for a single set of NULL, to designate no typeface name, no device name, and no bitmaps.

---

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 0, 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 0. 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 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 OPAQUE or TRANSPARENT mode. This may be zero at the designer's option.

dfItalic	One byte specifying whether the character definition data represent an italic font. The low-order bit is one if the flag is set. All other bits are zero.
dfUnderline	One byte specifying whether the character definition data represent an underlined font. The low-order bit is one if the flag is set.
dfStrikeOut	One byte specifying whether the character definition data represent a struckout font. The low-order bit is one 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 of 1 to 1000. A *dfWeight* of 400 specifies a regular weight.
dfCharSet	One byte specifying the character set defined by this font.
dfPixWidth	Two bytes. For vector fonts, specifies the width of the grid on which the font was digitized. For raster fonts, if *dfPixWidth* is nonzero, 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 *dfCharOffset*array.
dfPixHeight	Two bytes specifying the height of the character bitmap (raster fonts), of 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:

```
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.
```

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 *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, so use this field when calculating indexes into either *dfBits* or *dfCharOffset*.

dfLastChar — One byte specifying the last character code defined by this font. Note that all 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*. *dfDefaultChar* should indicate a special character that is not a space.

dfBreakChar — One byte specifying the character which will define word breaks. This character defines word breaks for word wrapping and wordspacing justification. The 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 that of *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. 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 (0).
dfFace	Four bytes specifying the offset in the file to the null-terminated string that names the face.
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 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 which 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) which specify the strokes for each character of the font. *dfBitsOffset* must be even.
CharTable	For raster fonts, the **CharTable** is an array of entries each consisting of two two-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 absolute-space character. This entry corresponds to a character which is guranteed to be blank; this character is not part of the normal character set. The number of entries in the table is calculated as '((*dfLastChar* – *dFirstChar*) + 2)'. This includes a spare, the *sentinel* offset mentioned below. For fixed pitch vector fonts, each two-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 four-byte entry is divided into two two-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.

<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 zeroes 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, and 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.

Single-character example

We give the bytes for a 12x14 pixel character, as shown schematically here.

```
............
.....**.....
....*..*....
...*....*...
..*......*..
..*......*..
..*......*..
..*......*..
..*********..
```

```
..*.......*..
..*.......*..
..*.......*..
............
............
............
```

The bytes are given here in two sets, as 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

Note 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.

Appendix G
Expanded Memory Support

G.1 Expanded Memory Support in Microsoft Windows

G.1.1 Introduction

In 1984, Intel introduced a scheme which provided applications with the ablility to have many code or data segments map to the same physical addresses. In this way the applications are able to access an amout of memory totalling up to two megabytes. Intel called their design "expanded memory". This is distinctly different from "extended" memory, the memory above 1 Meg on the PC-AT.

Intel termed the definition of the software application programmer's interface to this expanded memory the "LIM" (Lotus/Intel/Microsoft) specification (also known as EMS - Expanded Memory Specification), and produced a hardware card that supports the specification (the Intel Above-Board). The LIM interface is based on a 64K window (four contiguous 16K physical pages) that can be located anywhere in the 1 megabyte physical address space of a PC. Any 16K "logical" page on the 2 Meg card can be banked into any physical page in this window, provided that only one logical page is banked into a physical page at a time. In this way, the physical address space of the PC remains the same, but code or data can be stored in the expanded memory of the card and can be banked into the physical address when needed.

AST followed Intel's introduction of LIM/EMS with a specification known as EEMS. The AST interface is a superset of LIM with some additional functions and an expanded definition of the banking window that permits it to be as large as 1 megabyte.

The current implementation of Windows (v 2.0) can be used with any expanded card regardless of which specification it is based, as long as it is compatible with the current version (4.0) of the LIM/EMS set. It allows the creation and execution of applications which use LIM for bank switching data (or code). In addition, the package includes a utility that enables the memory on the card to be configured as a RAM disk. Windows can then use the RAM disk as a swap device when it swaps standard DOS applications in and out of memory (Windows application code is not swapped).

The functions required of a memory manager for it to support Windows 2.0 are described in the LIM 4.0 specification, which is available from Intel.

Index