CHAPTER 13: PRINTING (GDI)

Printing in Win32 is essentially "drawing to a piece of paper" instead of a window. While the GDI functions are the same, the **lifecycle** of the drawing is different.

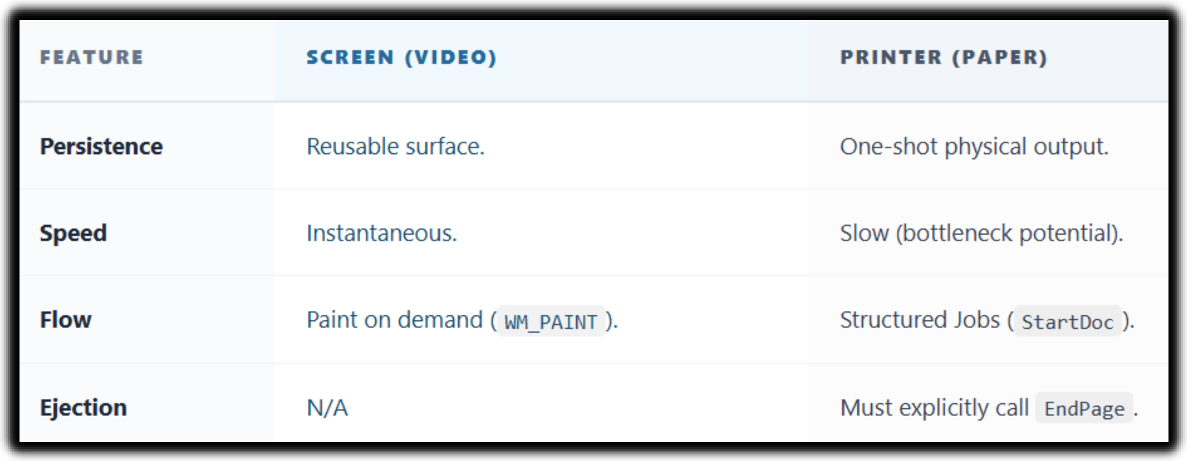
The Same: Device Independence

The beauty of GDI is that Rectangle() or TextOut() works the same way whether you're targeting pixels or ink.

- **Shared API:** You use the same pens, brushes, and fonts.
- **Abstraction:** You don't care if it's a laser printer or an inkjet; GDI handles the translation.

The Different: Pages vs. Pixels

You can't treat a printer like a monitor. It's slower, physical, and finite.



FEATURE	SCREEN (VIDEO)	PRINTER (PAPER)
Persistence	Reusable surface.	One-shot physical output.
Speed	Instantaneous.	Slow (bottleneck potential).
Flow	Paint on demand (`WM_PAINT`).	Structured Jobs (`StartDoc`).
Ejection	N/A	Must explicitly call `EndPage` .

The Printer Lifecycle

To print, you wrap your drawing code in "Job" and "Page" brackets:

1. **StartDoc**: Tells Windows, "I'm starting a new print job."
2. **StartPage**: "I'm starting a fresh sheet of paper."
3. **Drawing Code**: (Your usual TextOut, LineTo, etc.)
4. **EndPage**: "I'm done with this sheet. Eject it."
5. **EndDoc**: "The job is finished."

Printing is just GDI with **management overhead**. You use the same drawing tools, but you have to manually handle page breaks and document boundaries.

WINDOWS PRINTING PROCESS

Printing is significantly more complex than drawing to the screen. When you draw to a screen, you just change pixels in memory. When you print, you have to talk to a slow external device, translate your graphics into a language it understands (like PostScript), and manage massive amounts of data without freezing the computer.

Here is how Windows handles this heavy lifting.

---

1. The Players

To understand the process, you need to know who is involved:

- **The App:** Wants to print a document.
- **GDI (Graphics Device Interface):** The Windows graphics engine. It acts as the "Middleman."
- **The Driver:** The Translator. It converts Windows commands into specific byte codes for your specific HP, Canon, or Epson printer.
- **The Spooler:** The Waiting Room. It saves the print job to the hard drive so the user can get back to work while the printer takes its time.

---

2. The Step-by-Step Workflow

Phase A: The Setup (CreateDC, StartDoc)

The program asks for a "Printer Device Context." This loads the printer driver.

- **StartDoc:** Tells Windows, "I am starting a new job."
- **StartPage:** Tells Windows, "I am starting Page 1."

Phase B: The Recording (Metafiles)

When you call drawing functions (like TextOut or Rectangle), Windows does **not** send them to the printer immediately.

- **The EMF (Enhanced Metafile):** GDI "records" your commands into a file on the disk (.EMF).
- **Why?** It's faster to record "Draw a Circle" than to calculate the millions of pixels required to print that circle at 600 DPI.

Phase C: The Heavy Lifting ("Banding")

Printers have high resolution (600+ DPI). A full-page image can require 50MB+ of RAM. To save memory, Windows breaks the page into horizontal strips called **Bands**.

1. **Slice:** GDI asks the driver for the size of a band.
2. **Play:** GDI "plays" the Metafile recording *just for that top strip*. The driver translates it to printer code.
3. **Repeat:** GDI clears memory and plays the recording again for the *next strip* down.

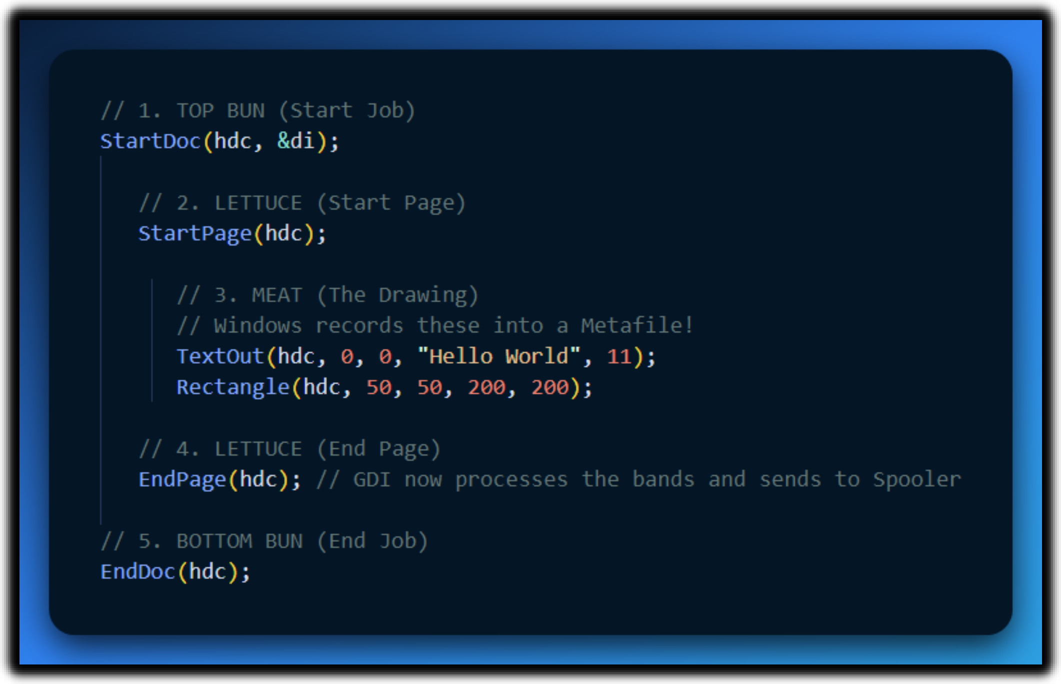
Phase D: The Handoff (Spooling)

- **Translation:** The driver converts the visuals into raw printer language (PCL, PostScript).
- **The .SPL File:** This raw data is stored in a temporary file (Spool File).
- **The Spooler:** Once the file is ready, the GDI tells the Print Spooler: "Here is the job. Send it to the printer whenever it's ready."
- **EndDoc:** The application is told, "Job sent!" and the user sees the "Printing Complete" notification.

---

3. The Code Sandwich

The structure of printing code always looks like this "Sandwich":



```
// 1. TOP BUN (Start Job)
StartDoc(hdc, &di);

    // 2. LETTUCE (Start Page)
    StartPage(hdc);

        // 3. MEAT (The Drawing)
        // Windows records these into a Metafile!
        TextOut(hdc, 0, 0, "Hello World", 11);
        Rectangle(hdc, 50, 50, 200, 200);

    // 4. LETTUCE (End Page)
    EndPage(hdc); // GDI now processes the bands and sends to Spooler

// 5. BOTTOM BUN (End Job)
EndDoc(hdc);
```

Explain Like I'm a Teenager: The Mural Painter

Imagine you are hiring a painter (The Printer) to paint a giant mural on a wall.

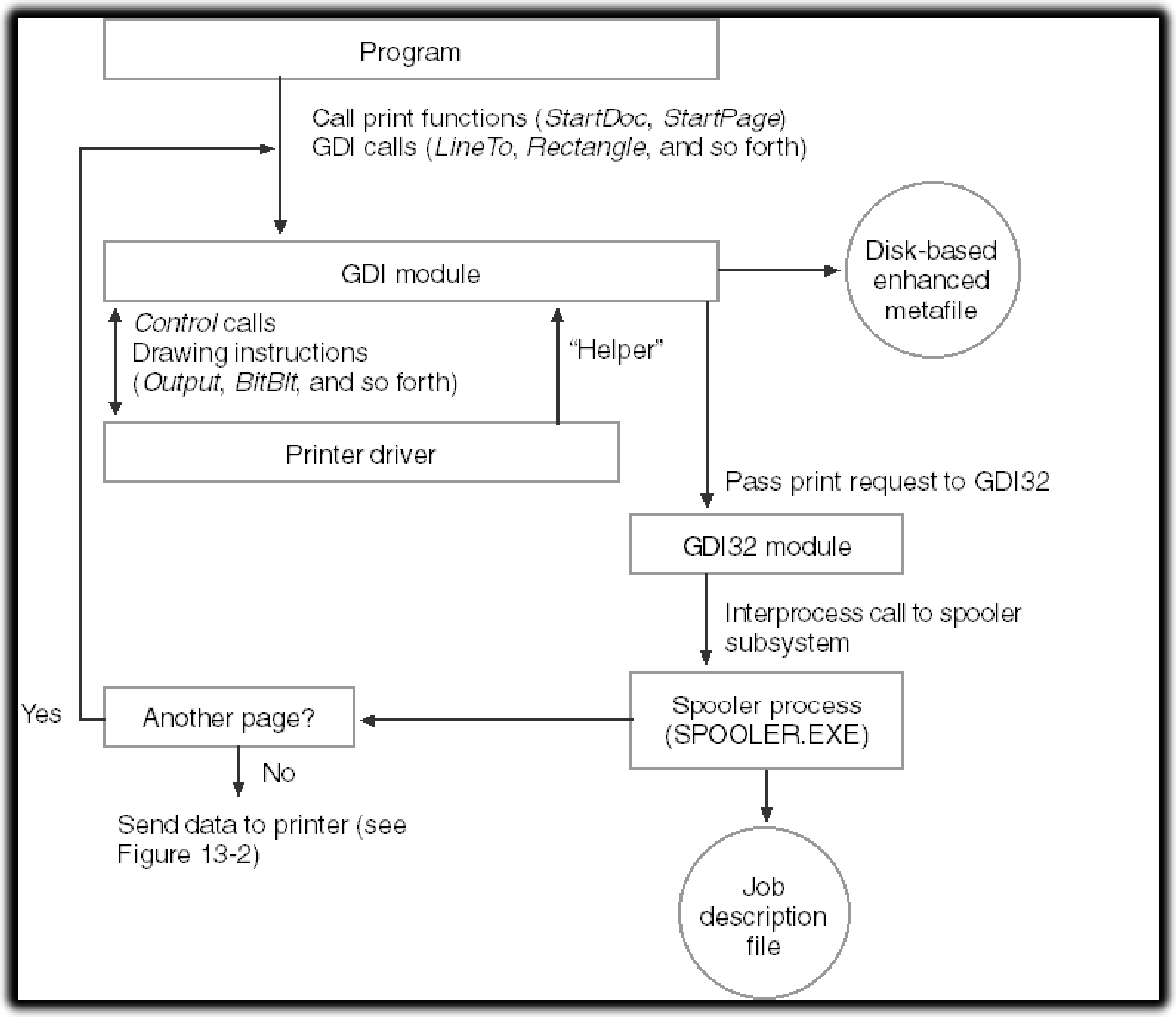
1. **The Recording (Metafile):** You don't stand there and tell the painter every brushstroke in real-time. You write down a list of instructions: "Draw a red tree on the left. Draw a blue sun on the right." This list is the **Metafile**.
2. **Banding:** The painter has a small ladder. He can't reach the whole wall at once.
   - He puts the ladder on the left and reads your *entire* list, but **only paints the parts he can reach**.
   - He moves the ladder to the middle, reads the *entire* list again, and paints the middle parts.
   - He moves to the right, reads the list again, and paints the rest.
3. **The Spooler:** You hand the painter the instructions and walk away. You don't wait for him to finish painting. He does the work in the background while you go get lunch.

Quick Review

Question 1: What is the purpose of "Banding"? *(Answer: To save memory. Instead of processing a massive 100MB page all at once, the computer processes it in small, manageable strips.)*

Question 2: What file format does GDI use to temporarily record your drawing commands? *(Answer: EMF - Enhanced Metafile.)*

Question 3: When does the actual printing happen? *(Answer: The application finishes quickly. The **Print Spooler** handles the actual slow transmission of data to the printer in the background.)*


Program
Call print functions (*StartDoc*, *StartPage*)
GDI calls (*LineTo*, *Rectangle*, and so forth)
GDI module
Disk-based enhanced metafile
Control calls
Drawing instructions
(*Output*, *BitBlt*, and so forth)
"Helper"
Printer driver
Pass print request to GDI32
GDI32 module
Interprocess call to spooler subsystem
Yes
Another page?
Spooler process (SPOOLER.EXE)
No
Send data to printer (see Figure 13-2)
Job description file

THE WINDOWS PRINT SPOOLER: A BREAKDOWN OF ITS COMPONENTS

How the Print Spooler Works (The "Pipeline")

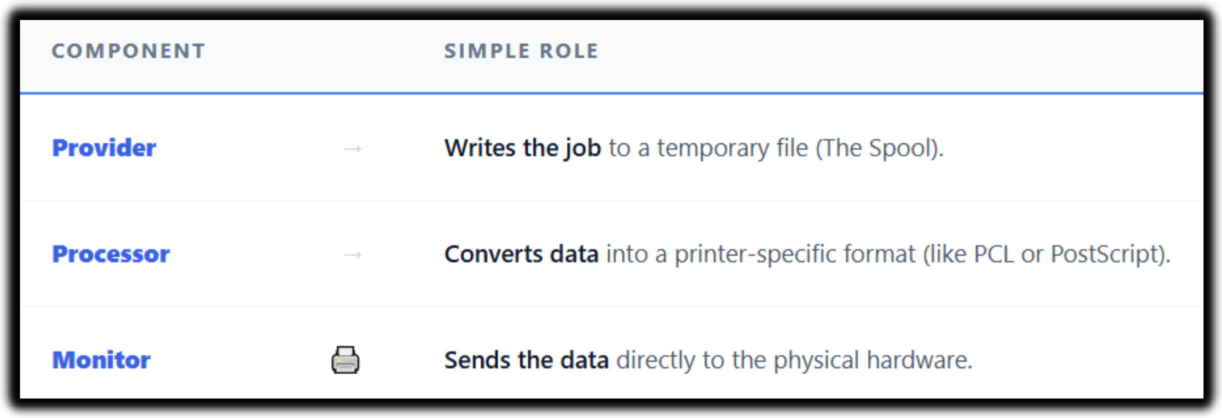
Instead of a list of definitions, here is the flow of a print job:

1. **The Hub (Spooler):** Your app dumps the data here and leaves. The Spooler decides if the job goes to a **Local** or **Network** provider.
2. **The Storage (Print Provider):** Creates a "Spool File." This is a temporary file on your disk so the job isn't lost if the power blips.
3. **The Translator (Print Processor):** This "despools" the file. It turns the generic GDI data into the specific language the printer understands (like PCL or PostScript).
4. **The Delivery (Port & Language Monitors):** The Port Monitor pushes the data through the cable (USB/Network), while the Language Monitor listens for "Out of Ink" or "Paper Jam" status.

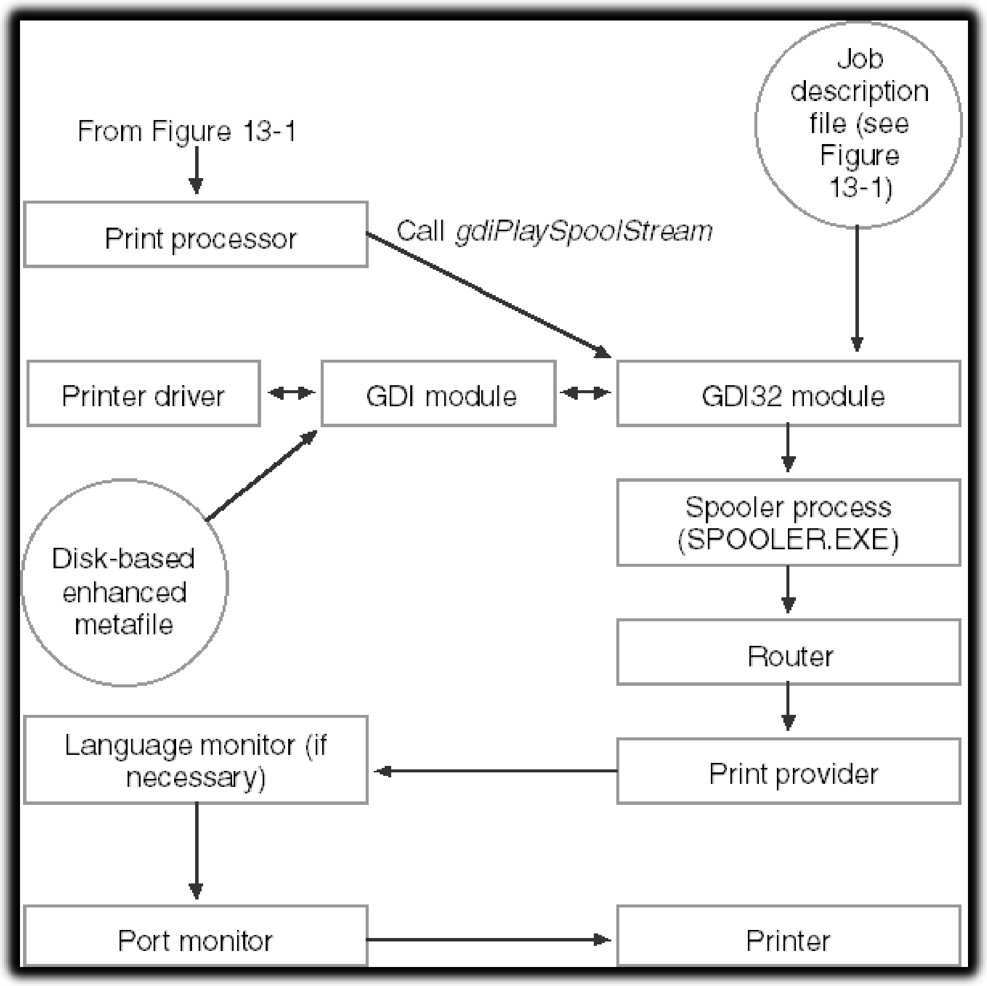
Why We Use It

- **App Freedom:** Your app finishes the "print" task in milliseconds because it only has to talk to the Spooler, not the slow physical printer.
- **The Queue:** Multiple apps can print at once; the Spooler lines them up like a polite traffic cop.
- **Background Work:** You can keep coding/browsing while the Spooler handles the heavy lifting in the background.

Essential Logic Summary

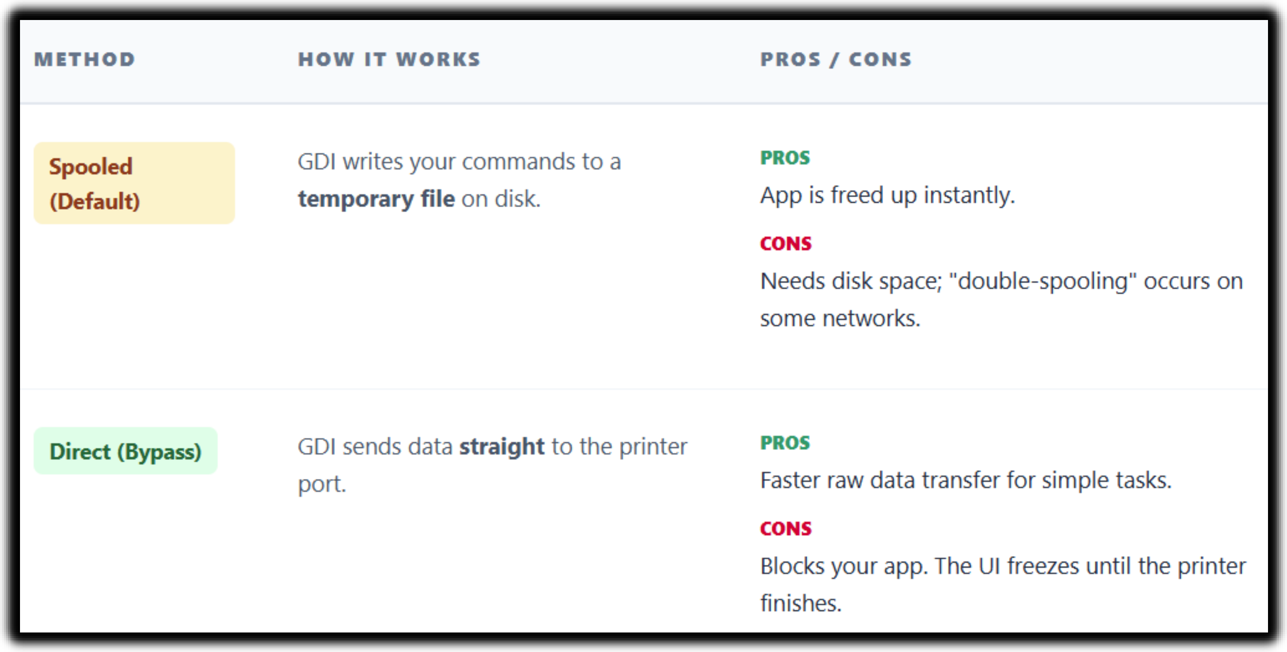


COMPONENT		SIMPLE ROLE
Provider	→	**Writes the job** to a temporary file (The Spool).
Processor	→	**Converts data** into a printer-specific format (like PCL or PostScript).
Monitor	🖨	**Sends the data** directly to the physical hardware.


From Figure 13-1
Print processor
Call *gdiPlaySpoolStream*
Job description file (see Figure 13-1)
Printer driver
GDI module
GDI32 module
Spooler process (SPOOLER.EXE)
Disk-based enhanced metafile
Router
Language monitor (if necessary)
Print provider
Port monitor
Printer

THE TWO PATHS: SPOOLED VS. DIRECT

In a perfect world, printing is **transparent**. Your app "paints" to a file on the disk, tells the Spooler "I'm done," and moves on. However, you can change this behavior:



METHOD	HOW IT WORKS	PROS / CONS
Spooled (Default)	GDI writes your commands to a **temporary file** on disk.	**PROS** App is freed up instantly. **CONS** Needs disk space; "double-spooling" occurs on some networks.
Direct (Bypass)	GDI sends data **straight** to the printer port.	**PROS** Faster raw data transfer for simple tasks. **CONS** Blocks your app. The UI freezes until the printer finishes.

Metafiles and "Banding"

Printers often have less memory than your screen. To handle high-resolution graphics, GDI uses **Banding**:

- **The Concept:** GDI records your drawing into a **Metafile** (a "tape recording" of your GDI calls).
- **The Execution:** It plays that recording back multiple times—once for each "band" (horizontal strip) of the page.
- **The Variation:** If a printer driver is smart enough or the page is simple, GDI skips the Metafile and sends commands directly to save time.

Real-World Risks

- **Disk Space:** Since "Spooled" printing saves to disk, GDI can actually crash if the hard drive is full.
- **Overhead:** Printing is "heavier" than screen drawing. You must handle errors (like SP_OUTOFDISK) that never happen on a monitor.

The First Step: Getting the DC

You can't do anything until you have a **Printer Device Context (DC)**.

- **PrintDlg**: The "correct" way. It pops up the standard Windows dialog so the user can choose the printer.
- **CreateDC**: The "hard-coded" way. Use this if you already know the printer name and want to skip the dialog.

OBTAINING A PRINTER DEVICE CONTEXT

The Goal: Getting the "Passport" (HDC)

Just like drawing to the screen requires a Handle to Device Context (HDC), printing requires a **Printer DC**.

- **Screen DC:** Tells Windows "Draw on the Monitor."
- **Printer DC:** Tells Windows "Draw on the HP LaserJet in the hallway."

Without this handle, your StartDoc and TextOut commands have nowhere to go.

The Two Ways to Get a Printer DC

Method 1: The Easy Way (PrintDlg)

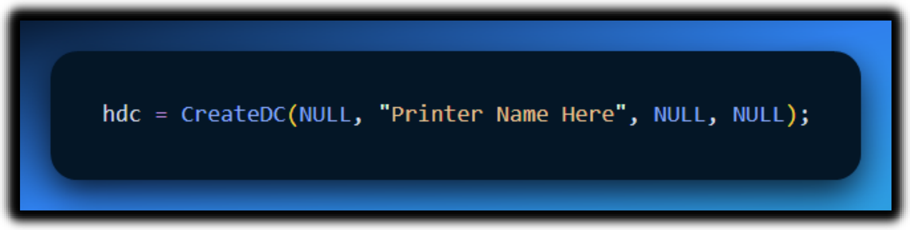
This is what 99% of apps use. You don't guess the printer; you let the user pick it.

- **Action:** You call the standard "Print" dialog box.
- **Result:** The user selects a printer, clicks "OK," and Windows hands you a ready-to-use HDC.
- **Benefit:** Zero headache. You don't need to know the printer's name or address.

Method 2: The Hard Way (CreateDC)

Used when you want to print silently (e.g., a receipt printer at a POS terminal) without popping up a dialog.

- **Action:** You manually ask Windows to connect to a specific printer.
- **The Catch:** You must know the **exact device name** string (e.g., "Epson TM-T88V").



```
hdc = CreateDC(NULL, "Printer Name Here", NULL, NULL);
```

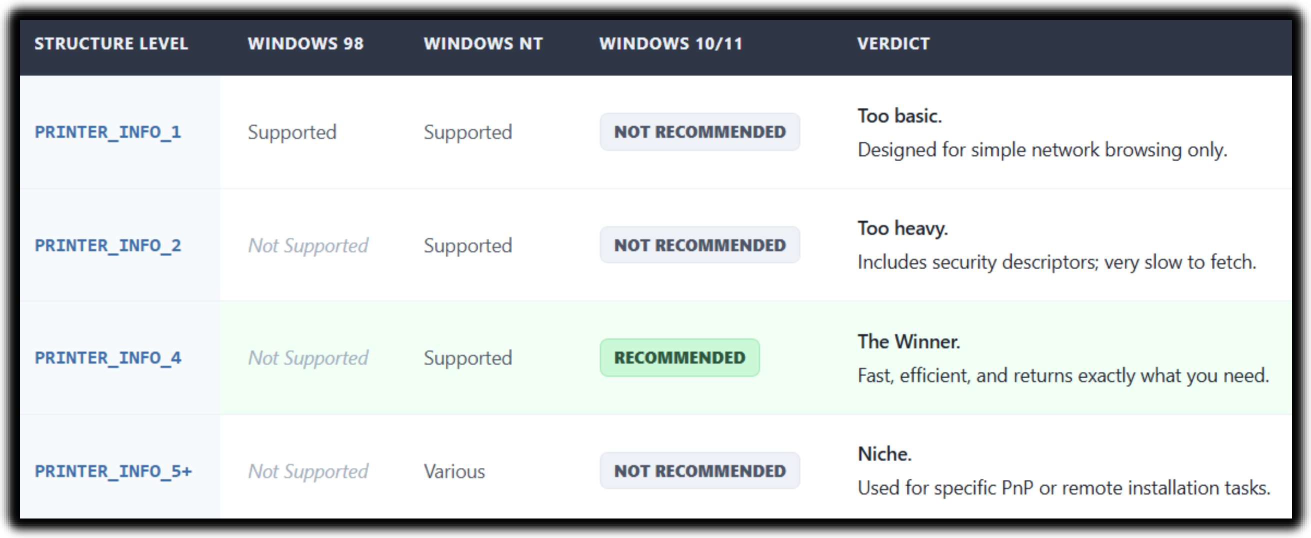
The Challenge: Finding the Printer Name (EnumPrinters)

If you use Method 2, how do you know the printer's name? You have to ask Windows for a list of all connected printers. This is where it gets messy because Windows has changed how it lists printers over the last 25 years.

You use the function EnumPrinters, but you must specify a **"Level"** (How much detail you want).

I. The "Structure Wars" (Which Level to Use?)

Different versions of Windows prefer different data structures (levels) to hold printer info. Your notes and the uploaded image outline this compatibility nightmare:



STRUCTURE LEVEL	WINDOWS 98	WINDOWS NT	WINDOWS 10/11	VERDICT
PRINTER_INFO_1	Supported	Supported	NOT RECOMMENDED	Too basic. Designed for simple network browsing only.
PRINTER_INFO_2	*Not Supported*	Supported	NOT RECOMMENDED	Too heavy. Includes security descriptors; very slow to fetch.
PRINTER_INFO_4	*Not Supported*	Supported	RECOMMENDED	The Winner. Fast, efficient, and returns exactly what you need.
PRINTER_INFO_5+	*Not Supported*	Various	NOT RECOMMENDED	Niche. Used for specific PnP or remote installation tasks.

II. Why is PRINTER_INFO_4 the Modern Choice?

1. **Speed:** It is "Relatively compact" and "Efficient for enumerating large numbers of printers."
2. **Essentials Only:** It gives you exactly what you need (Printer Name, Server Name, Attributes) without the bloat of detailed driver stats.

III. Explain Like I'm a Teenager: The Contact List

CreateDC: This is like trying to FaceTime someone. You need their exact contact name.

EnumPrinters: This is scrolling through your contacts to find that name.

The Structures (INFO_1 vs INFO_4):

- **PRINTER_INFO_2 (The Old Way):** This is like loading a full profile for every contact—photo, address, birthday, bio. It loads slowly.
- **PRINTER_INFO_4 (The Windows 10/11 Way):** This is "Compact View." It just lists the Names. It loads instantly. Since you only need the name to start the call (CreateDC), this is the one you should use.

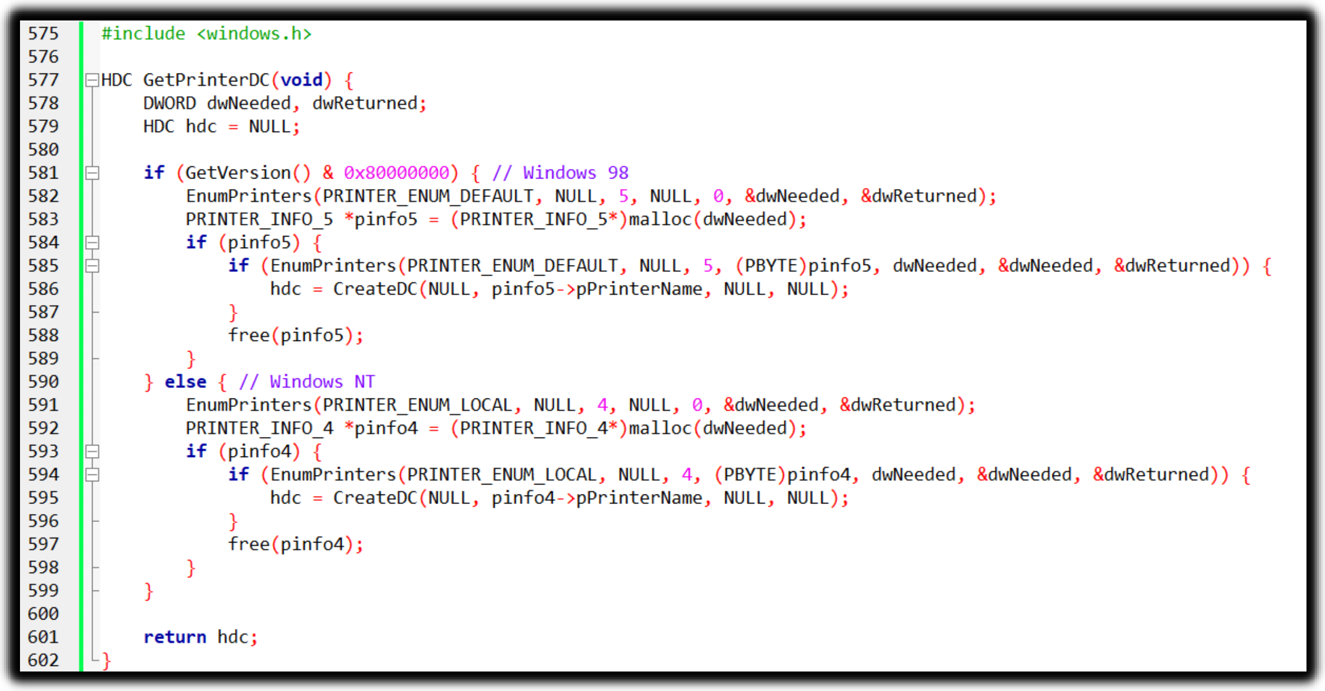
IV. Quick Review

Question 1: Which function opens the standard "Select Printer" popup for the user? *(Answer: PrintDlg.)*

Question 2: If you want to find a printer name on Windows 11, which structure should you ask EnumPrinters to return? *(Answer: PRINTER_INFO_4. It is the recommended standard for modern systems.)*

Question 3: Why is PRINTER_INFO_2 not recommended for simple enumeration on modern systems? *(Answer: It provides too much detail, making it slower and heavier than necessary just to get a list of names.)*

THE GETPRINTERDC LOGIC: 3 SIMPLE STEPS



```c
#include <windows.h>

HDC GetPrinterDC(void) {
    DWORD dwNeeded, dwReturned;
    HDC hdc = NULL;

    if (GetVersion() & 0x80000000) { // Windows 98
        EnumPrinters(PRINTER_ENUM_DEFAULT, NULL, 5, NULL, 0, &dwNeeded, &dwReturned);
        PRINTER_INFO_5 *pinfo5 = (PRINTER_INFO_5*)malloc(dwNeeded);
        if (pinfo5) {
            if (EnumPrinters(PRINTER_ENUM_DEFAULT, NULL, 5, (PBYTE)pinfo5, dwNeeded, &dwNeeded, &dwReturned)) {
                hdc = CreateDC(NULL, pinfo5->pPrinterName, NULL, NULL);
            }
            free(pinfo5);
        }
    } else { // Windows NT
        EnumPrinters(PRINTER_ENUM_LOCAL, NULL, 4, NULL, 0, &dwNeeded, &dwReturned);
        PRINTER_INFO_4 *pinfo4 = (PRINTER_INFO_4*)malloc(dwNeeded);
        if (pinfo4) {
            if (EnumPrinters(PRINTER_ENUM_LOCAL, NULL, 4, (PBYTE)pinfo4, dwNeeded, &dwNeeded, &dwReturned)) {
                hdc = CreateDC(NULL, pinfo4->pPrinterName, NULL, NULL);
            }
            free(pinfo4);
        }
    }

    return hdc;
}
```

Instead of manual searching, this helper function automates the discovery of your default printer.

I. The OS Check

The function calls GetVersion to see if it's running on **Windows 9x** or **Windows NT/Modern**. This is necessary because the "labels" on the printer information are stored in different structures depending on the version.

II. The Two-Pass Memory Trick

You can't just ask for printer info; you have to know how much "desk space" it needs first.

- **Pass 1:** Call EnumPrinters with a size of zero. Windows says, "That’s too small, you need X bytes."
- **The Swap:** You allocate exactly X bytes of memory.
- **Pass 2:** Call EnumPrinters again. Now that there’s room, it fills your memory with the printer's details.

III. Creating the Handle

- **The Goal:** Extract the pPrinterName string from the structure (PRINTER_INFO_4 for NT or PRINTER_INFO_5 for 9x).
- **The Result:** Plug that name into CreateDC.
- **Cleanup:** Free the temporary memory and return the HDC (your "Golden Ticket" to start drawing).

IV. Why is it written this way?

- **Efficiency:** It only uses as much RAM as the printer list requires.
- **Portability:** It doesn't care which version of Windows you are on; it adjusts its "labels" (PRINTER_INFO_x) automatically.

DEVCAPS2 PROGRAM

DEVCAPS2 is the "Professional Edition." It doesn't just ask "How big is the screen?"; it asks "Can you draw curves? Can you handle transparency? What about rotation?"

What is DEVCAPS2? (The Hardware Inspector)

DEVCAPS2 is a diagnostic tool. It interviews your hardware (Video Display or Printers) and forces it to confess its capabilities.

The Goal: To see exactly what GetDeviceCaps returns for different devices.

The Menu:

- **Device:** Lets you toggle between "Screen" and any connected "Printer."
- **Capabilities:** Lets you filter the data (Basic Info, Curves, Lines, Polygons, Text).

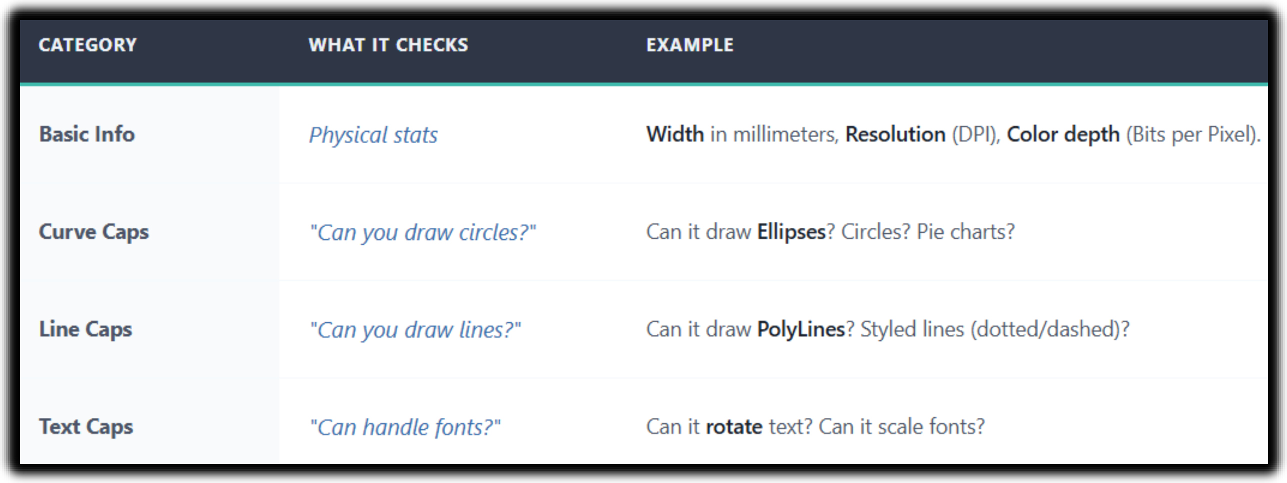
Key Feature: Dynamic Menus Unlike normal programs where the menu is fixed (File, Edit, View), DEVCAPS2 builds its menu **on the fly**. When you start it, it asks Windows, "List every printer currently connected," and adds them to the menu automatically.

Under the Hood: The Core Logic

The program relies on one massive Windows function: GetDeviceCaps (Get Device Capabilities).

a) The Information Categories

The program breaks the data down into digestible chunks:



CATEGORY	WHAT IT CHECKS	EXAMPLE
Basic Info	*Physical stats*	**Width** in millimeters, **Resolution** (DPI), **Color depth** (Bits per Pixel).
Curve Caps	*"Can you draw circles?"*	Can it draw **Ellipses**? Circles? Pie charts?
Line Caps	*"Can you draw lines?"*	Can it draw **PolyLines**? Styled lines (dotted/dashed)?
Text Caps	*"Can handle fonts?"*	Can it **rotate** text? Can it scale fonts?

b) The "Bit-Coded" Secret

Most of these capabilities are not simple "Yes/No" variables. They are **Bit Flags**.

- *Example:* Windows returns a single integer like 0x0014.
- The program has to use **Bitwise AND (&)** operations to decipher it: "Does bit 3 equal 1? If yes, it supports Circles."

The Window Procedure (WndProc) Breakdown

Here is how the code manages the flow:

I. Startup (WM_CREATE)

The program measures the system font size so it knows how to space the text rows neatly.

II. Menu Updates (WM_SETTINGCHANGE)

If you plug in a new USB printer while the app is running, Windows sends this message. DEVCAPS2 catches it and rebuilds the printer menu instantly.

III. Painting (WM_PAINT)

This is where the reporting happens.

- It creates a "Device Context" for the selected hardware.
- It calls DoBasicInfo, DoCurveCaps, etc.
- These functions use TextOut to print the results on screen.

IV. The "Properties" Button (WM_COMMAND)

- If you select "Screen," the Properties button is disabled (Grayed out).
- If you select a "Printer," the Properties button lights up.

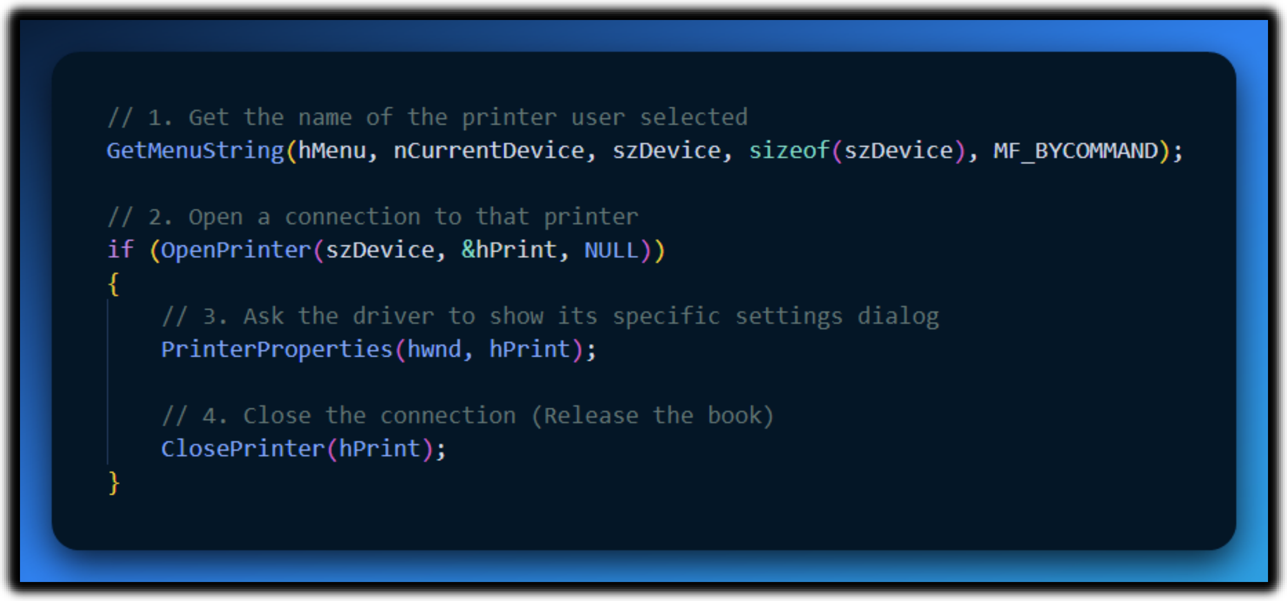
Deep Dive: Printer Properties

When you click "Properties" for a printer, a complex dialog box pops up where you can choose "Landscape," "Paper Size," or "Print Quality."

Crucial Concept: You (the programmer) do not create this dialog. It belongs to the **Printer Driver** (HP, Canon, Epson). Your program just politely asks the driver to show it.

The Code Workflow

To show this dialog, you cannot just call a simple function. You have to "Check Out" the printer like a library book.



```c
// 1. Get the name of the printer user selected
GetMenuString(hMenu, nCurrentDevice, szDevice, sizeof(szDevice), MF_BYCOMMAND);

// 2. Open a connection to that printer
if (OpenPrinter(szDevice, &hPrint, NULL))
{
    // 3. Ask the driver to show its specific settings dialog
    PrinterProperties(hwnd, hPrint);

    // 4. Close the connection (Release the book)
    ClosePrinter(hPrint);
}
```

Why Orientation Matters

If the user opens this dialog and changes the paper to **Landscape**, the GetDeviceCaps function will return different numbers!

- **Portrait:** Width = 2000, Height = 3000.
- **Landscape:** Width = 3000, Height = 2000.

Explain Like I'm a Teenager: The Car Mechanic

- **DEVCAPS2** is a generic OBD scanner you plug into a car.
- **GetDeviceCaps:** This is the scanner asking, "Does this car have ABS? Does it have a Turbo?"
- **Bit-Coded Flags:** The car replies with a code "101". You look up in the manual: 1st digit is ABS (Yes), 2nd digit is Turbo (No), 3rd digit is Sunroof (Yes).
- **Printer Properties:** This is like popping the hood. The scanner (your program) doesn't know how to fix the engine; it just tells the car (the Driver) to "Open the Hood" so the user can flip switches manually.

Quick Review

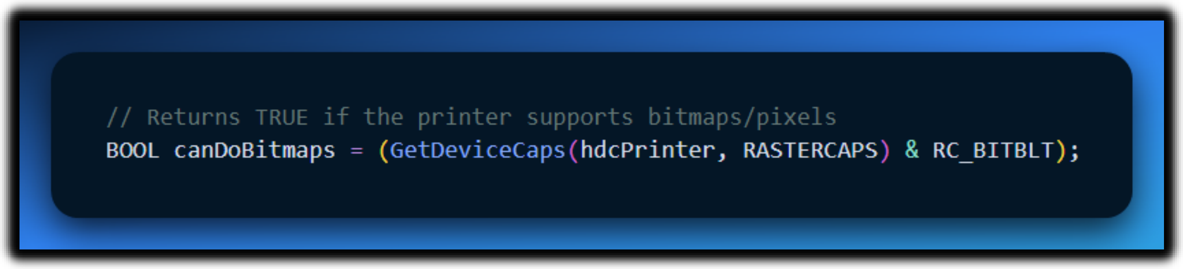
Question 1: How does DEVCAPS2 know if a device supports drawing circles?
Question 2: Who creates the "Printer Properties" dialog box?
Question 3: Why do we need OpenPrinter and ClosePrinter?

CHECKING FOR BITBLT: CAN THE PRINTER "DO" PIXELS?

While GDI usually fakes features you're missing, it cannot fake **BitBlt** (Bitmap Block Transfer). If a device is "vector-only" (like a plotter), it physically can't handle pixel data.

I. The 1-Line Test

Use GetDeviceCaps to check the RASTERCAPS bit:



```
// Returns TRUE if the printer supports bitmaps/pixels
BOOL canDoBitmaps = (GetDeviceCaps(hdcPrinter, RASTERCAPS) & RC_BITBLT);
```

II. What breaks if RC_BITBLT is missing?

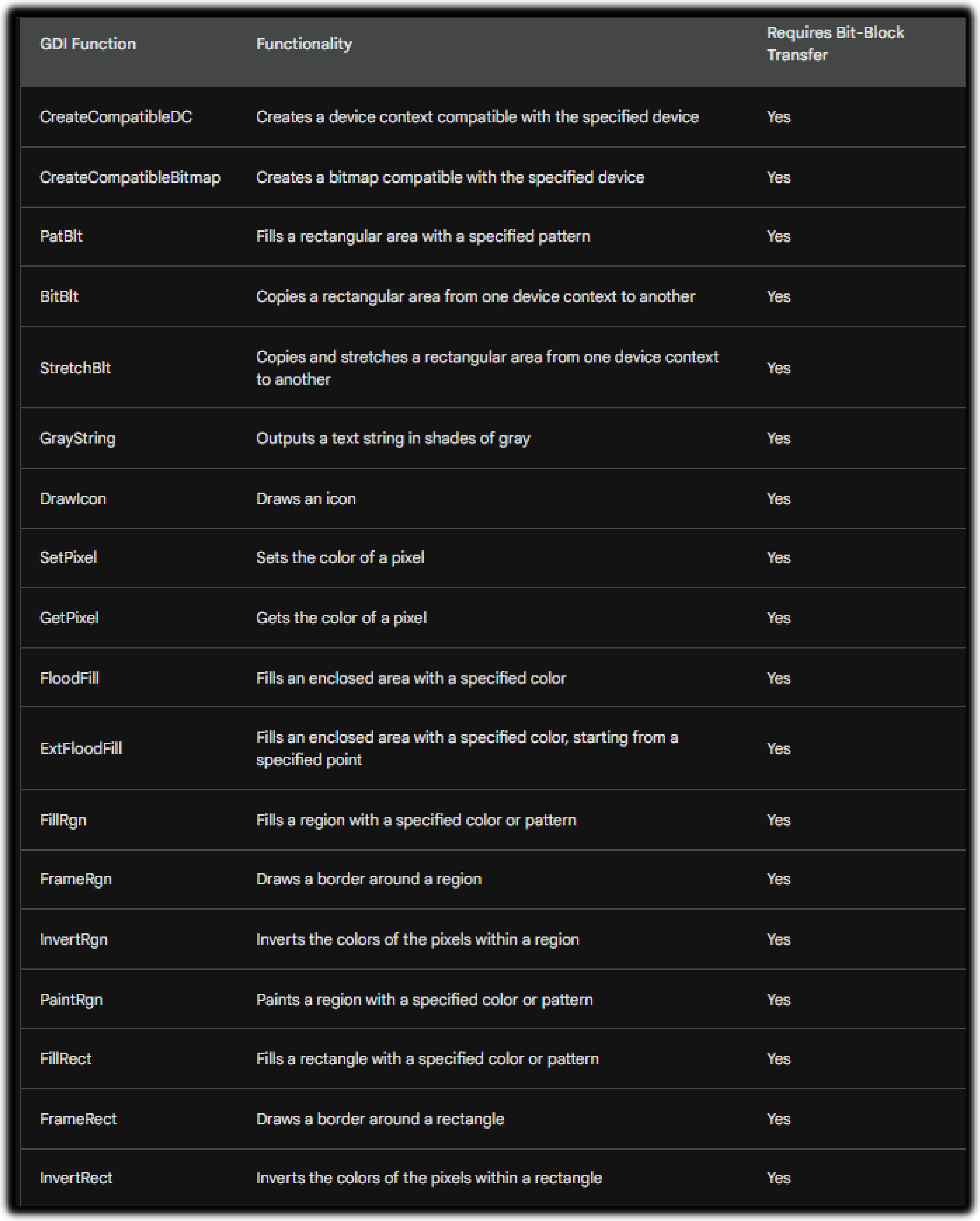
If this check fails, the following GDI categories are **useless**:

- **Images:** BitBlt, StretchBlt, DrawIcon, and CreateCompatibleBitmap.
- **Pixel-Level Work:** SetPixel and GetPixel.
- **Complex Fills:** FloodFill, InvertRect, and any Region functions (FillRgn, PaintRgn).
- **Drawing Helpers:** GrayString (which relies on bitmaps to dim text).

III. Summary for your Notes

- **Most Printers:** (Laser, Inkjet, Dot-matrix) support RC_BITBLT.
- **Plotters:** Usually do **not** support it (they only understand lines/vectors).
- **Dev Rule:** Always check this bit before trying to print a bitmap or an icon, or your app will just draw nothing.

Table Summary

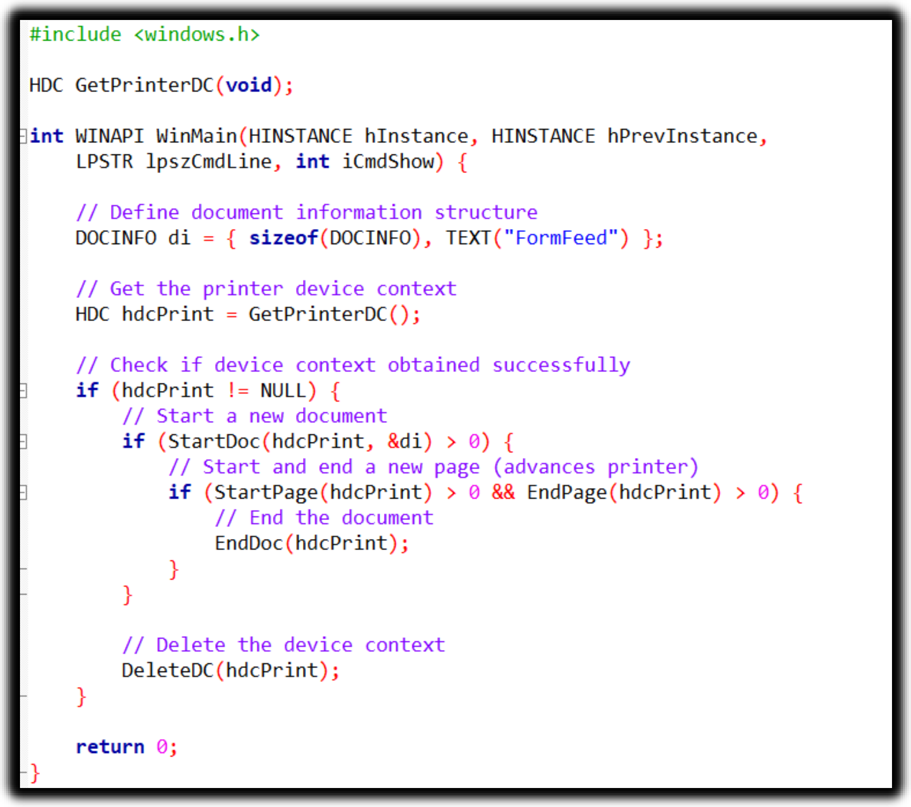


GDI Function	Functionality	Requires Bit-Block Transfer
CreateCompatibleDC	Creates a device context compatible with the specified device	Yes
CreateCompatibleBitmap	Creates a bitmap compatible with the specified device	Yes
PatBlt	Fills a rectangular area with a specified pattern	Yes
BitBlt	Copies a rectangular area from one device context to another	Yes
StretchBlt	Copies and stretches a rectangular area from one device context to another	Yes
GrayString	Outputs a text string in shades of gray	Yes
DrawIcon	Draws an icon	Yes
SetPixel	Sets the color of a pixel	Yes
GetPixel	Gets the color of a pixel	Yes
FloodFill	Fills an enclosed area with a specified color	Yes
ExtFloodFill	Fills an enclosed area with a specified color, starting from a specified point	Yes
FillRgn	Fills a region with a specified color or pattern	Yes
FrameRgn	Draws a border around a region	Yes
InvertRgn	Inverts the colors of the pixels within a region	Yes
PaintRgn	Paints a region with a specified color or pattern	Yes
FillRect	Fills a rectangle with a specified color or pattern	Yes
FrameRect	Draws a border around a rectangle	Yes
InvertRect	Inverts the colors of the pixels within a rectangle	Yes

THE SIMPLEST PRINTING PROGRAM: FORMFEED.C

This program demonstrates the minimum requirements for printing by simply causing a printer form feed.

Code Breakdown:



```c
#include <windows.h>

HDC GetPrinterDC(void);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpszCmdLine, int iCmdShow) {

    // Define document information structure
    DOCINFO di = { sizeof(DOCINFO), TEXT("FormFeed") };

    // Get the printer device context
    HDC hdcPrint = GetPrinterDC();

    // Check if device context obtained successfully
    if (hdcPrint != NULL) {
        // Start a new document
        if (StartDoc(hdcPrint, &di) > 0) {
            // Start and end a new page (advances printer)
            if (StartPage(hdcPrint) > 0 && EndPage(hdcPrint) > 0) {
                // End the document
                EndDoc(hdcPrint);
            }
        }

        // Delete the device context
        DeleteDC(hdcPrint);
    }

    return 0;
}
```

The code demonstrates basic Windows printing using GDI. It gets a printer device context, starts a print job with StartDoc, and advances pages using StartPage and EndPage.

A description string is shown in the printer queue. Each call checks for errors, with GDI automatically canceling the job on failure; GetLastError can be used for details.

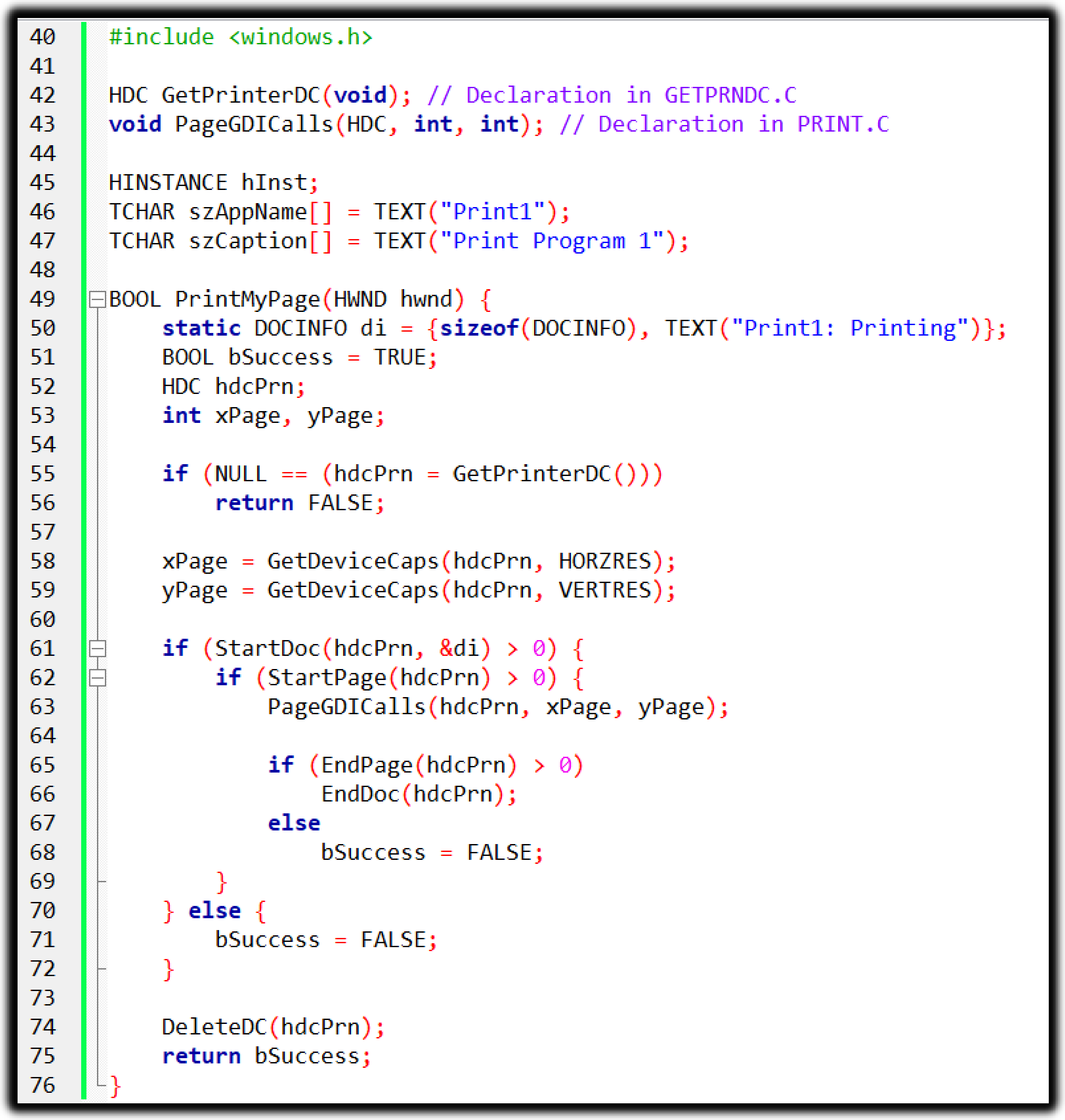
Directly writing form-feed commands to printer ports is **discouraged** due to portability and reliability issues. Using Windows printing APIs is the correct and robust approach.

Overall, the example shows the minimal structure needed to control printing and page feeds in a Windows application.

PRINT 1-BAREBONES PRINTING

The *Print.c* program is a simple Windows example that shows how basic printing works. It includes three versions—PRINT1, PRINT2, and PRINT3—each of which prints a single page containing text and graphics. Most of the printing logic is shared in *PRINT.C*, while *GETPRNDC.C* is used to obtain the printer's device context.

Together, these files provide a straightforward example of how a Windows application sets up and sends output to a printer.



```c
#include <windows.h>

HDC GetPrinterDC(void); // Declaration in GETPRNDC.C
void PageGDICalls(HDC, int, int); // Declaration in PRINT.C

HINSTANCE hInst;
TCHAR szAppName[] = TEXT("Print1");
TCHAR szCaption[] = TEXT("Print Program 1");

BOOL PrintMyPage(HWND hwnd) {
    static DOCINFO di = {sizeof(DOCINFO), TEXT("Print1: Printing")};
    BOOL bSuccess = TRUE;
    HDC hdcPrn;
    int xPage, yPage;

    if (NULL == (hdcPrn = GetPrinterDC()))
        return FALSE;

    xPage = GetDeviceCaps(hdcPrn, HORZRES);
    yPage = GetDeviceCaps(hdcPrn, VERTRES);

    if (StartDoc(hdcPrn, &di) > 0) {
        if (StartPage(hdcPrn) > 0) {
            PageGDICalls(hdcPrn, xPage, yPage);

            if (EndPage(hdcPrn) > 0)
                EndDoc(hdcPrn);
            else
                bSuccess = FALSE;
        }
    } else {
        bSuccess = FALSE;
    }

    DeleteDC(hdcPrn);
    return bSuccess;
}
```

WndProc

- Handles standard window messages.
- Adds a **Print** option to the system menu.
- Triggers printing when the menu option is selected.
- Repaints the window using PageGDICalls.
- Cleans up on exit.

PageGDICalls

- Contains all drawing logic.
- Uses GDI calls to render shapes and centered text on the page.
- Designed to work for both screen and printer output.

PrintMyPage

- Handles the print lifecycle.
- Gets the printer DC, starts the document and page, calls PageGDICalls, and closes everything.
- Returns success or failure based on printing results.

WinMain

- Standard Windows entry point.
- Registers the window, creates it, and runs the message loop.

Print1 Overview

Print1 shows the minimum setup required to print a single page using the Windows API. The program:

- Acquires a printer device context
- Starts a print job and page
- Draws content via PageGDICalls
- Ends the page and document cleanly

The design separates **printing control** from **drawing logic**, making it easy to extend in later versions (PRINT2, PRINT3).

UNDERSTANDING PRINT1 PRINTING & ABORT PROCEDURE

Obtaining Device Context and Page Dimensions:

Before anything prints, *PrintMyPage* first ensures it has a valid printer device context (DC) handle. If this fails, an error message is shown.
If successful, the program retrieves the page dimensions using GetDeviceCaps, specifically looking at HORZRES (horizontal resolution) and VERTRES (vertical resolution). These values give the actual printable area, not the full paper size, which is important for ensuring the content fits properly on the page.

Printing Process in PRINT1:

The overall flow of PRINT1's printing is similar to the earlier FORMFEED example, but with one key difference:
The **PageGDICalls** function is called between StartPage and EndPage, which handles the actual drawing on the page.
The document is only finalized with EndDoc if the previous steps—*StartDoc*, *StartPage*, and *EndPage*—are all successful. This ensures that the printing only completes if everything has gone right so far.

Canceling a Print Job (Abort Procedure):

Sometimes, during printing, you might need to stop a large document before it finishes—especially if something goes wrong. To handle this, we use an "abort procedure." This allows the program to monitor the printing process and cancel it if needed.

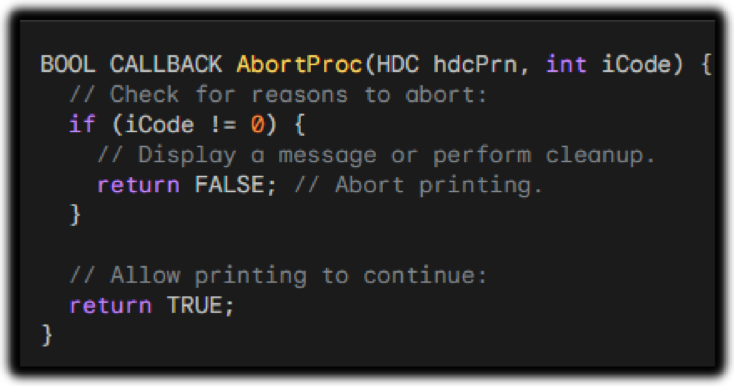
AbortProc Function:

The abort procedure takes two arguments:

- hdcPrn: The printer device context handle (essentially a link to the printer).
- iCode: A code that tells us why the procedure is being called. For example, 0 means success, while SP_OUTOFDISK signals a disk space issue.

Implementing the Abort Procedure:

1. **Registering AbortProc**:
   Before starting the document with StartDoc, the procedure is registered using SetAbortProc(hdcPrn, AbortProc). This sets up the program to be able to abort printing if necessary.
2. **Defining AbortProc**:
   The AbortProc function itself is where the program checks conditions like disk space or other errors. If any issue is found, it can stop the print job, saving time and resources.



```
BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode) {
  // Check for reasons to abort:
  if (iCode != 0) {
    // Display a message or perform cleanup.
    return FALSE; // Abort printing.
  }

  // Allow printing to continue:
  return TRUE;
}
```

GDI Calls AbortProc

While printing, during the processing of EndPage, **GDI calls the AbortProc function repeatedly**.

This gives the program a chance to check if any conditions (like errors or user cancellation) require halting the print job.

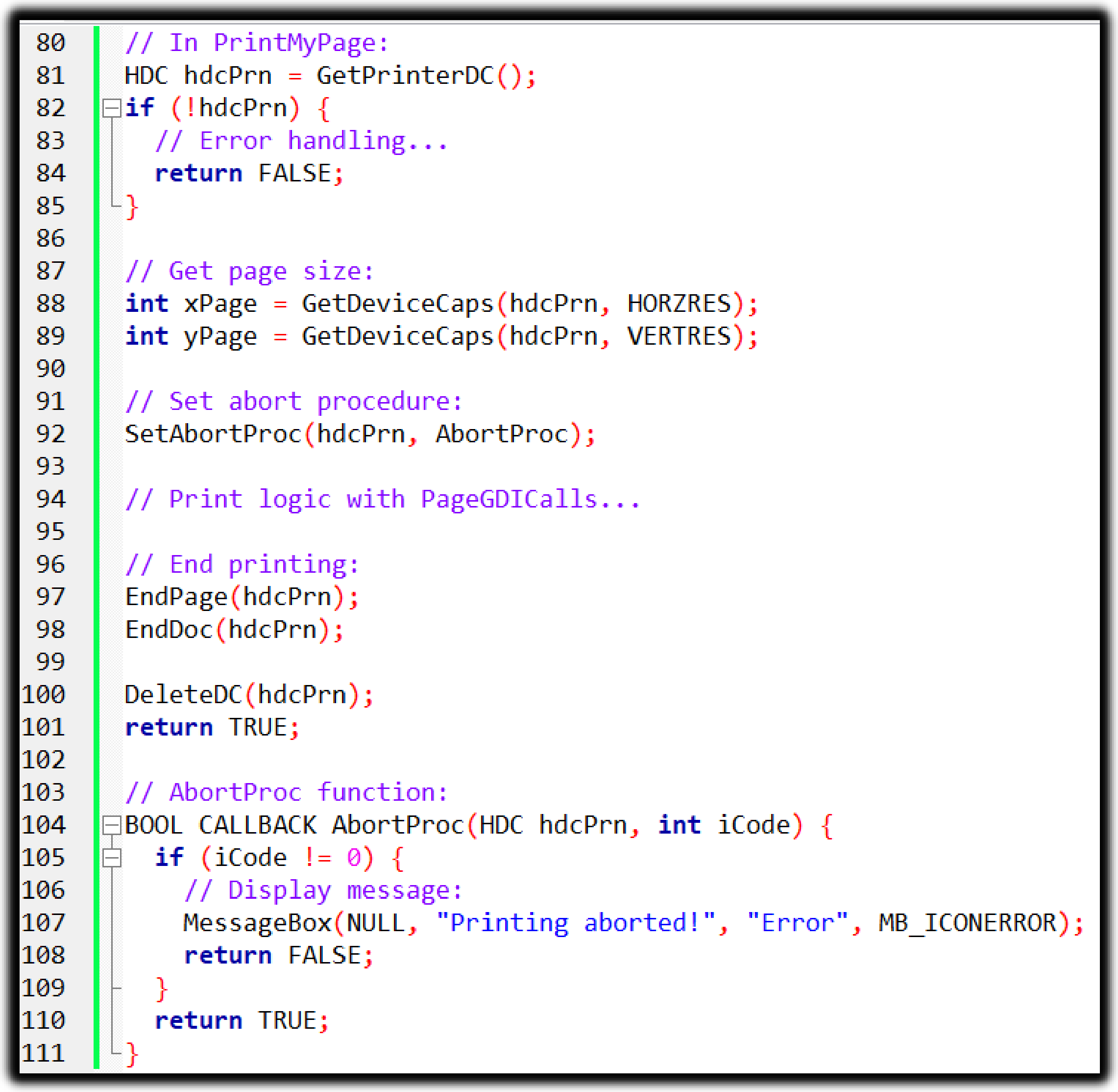
If the program decides that something's wrong, it can abort the print operation before continuing.

Keeping the Message Loop Active

The example uses **PeekMessage** instead of **GetMessage** in its message loop.

- **PeekMessage** allows the program to check for incoming messages without blocking the rest of the operations. This means the program can still process user input (like clicking or keyboard events) even while printing.
- The loop keeps running until there are no more messages to process, at which point it hands control back to Windows, letting the system manage everything else.

Code Examples:



```
// In PrintMyPage:
HDC hdcPrn = GetPrinterDC();
if (!hdcPrn) {
  // Error handling...
  return FALSE;
}

// Get page size:
int xPage = GetDeviceCaps(hdcPrn, HORZRES);
int yPage = GetDeviceCaps(hdcPrn, VERTRES);

// Set abort procedure:
SetAbortProc(hdcPrn, AbortProc);

// Print logic with PageGDICalls...

// End printing:
EndPage(hdcPrn);
EndDoc(hdcPrn);

DeleteDC(hdcPrn);
return TRUE;

// AbortProc function:
BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode) {
  if (iCode != 0) {
    // Display message:
    MessageBox(NULL, "Printing aborted!", "Error", MB_ICONERROR);
    return FALSE;
  }
  return TRUE;
}
```

Abort procedures enable print cancellation, while PeekMessage ensures responsive user input, adding flexibility and control to printing.

ABORT PROCEDURE (ABORTPROC)

This is the solution to a classic problem: **The White Screen of Death.** When your program calls EndPage(), Windows starts the heavy lifting (translating the Metafile to printer code).

This can take 5, 10, or 20 seconds. During this time, your application **freezes**. You can't move the window, resize it, or click "Cancel."

The AbortProc is the loophole that keeps your app alive during this heavy lifting.

I. The Bottleneck: EndPage

In the previous section, we learned that GDI "records" your drawing commands.

- **During TextOut:** Fast. Just writing to a list in memory.
- **During EndPage: SLOW.** This is when GDI actually processes that list, cuts it into bands, and translates it for the printer driver.

If you don't have an Abort Procedure, your entire program stops executing code until EndPage finishes. To the user, it looks like the app has crashed.

II. The Solution: The "Heartbeat" (AbortProc)

Windows knows EndPage is slow. So, while GDI is crunching numbers, it pauses every few milliseconds to call a specific function in your code: the **Abort Procedure**.

It is essentially GDI asking: **"I'm still working... Do you want me to continue?"**

- **You return TRUE:** GDI keeps printing.
- **You return FALSE:** GDI stops immediately (Cancels the print job).

III. The Magic Loop (PeekMessage)

To make the "Cancel" button work, your Abort Procedure needs to check if the user clicked it. You cannot use the standard GetMessage loop because it **waits** for input. You need PeekMessage.

The Logic:

1. GDI calls AbortProc.
2. You check the message queue: "Did anyone click Cancel?" (PeekMessage)
   - ✓ **Yes:** Process the click. If it was the Cancel button, set a flag to stop.
   - ✓ **No:** Return TRUE immediately so GDI can get back to work.

IV. Handling Errors (SP_OUTOFDISK)

Sometimes GDI calls AbortProc not to ask for permission, but to report a problem. The most common error passed to AbortProc is SP_OUTOFDISK.

- **Scenario:** The spooler is trying to write the temp file, but the hard drive is full.
- **Your Job:** The AbortProc doesn't fix the disk; it just decides whether to wait and retry or give up.

V. Explain Like I'm a Teenager: The Road Trip

Imagine you are driving your parents (GDI) on a long road trip (Printing).

The Freeze: If you don't talk, your parents just drive efficiently. But you (the User) are stuck in the back seat, unable to ask for a bathroom break.

The Abort Procedure: Every 10 miles, your dad turns around and asks, "Are we there yet? Should we keep going?"

The Cancel Button:

- If you say **"Yes"** (return TRUE), he drives another 10 miles.
- If you scream **"STOP!"** (return FALSE), he pulls over immediately, and the trip is over.

PeekMessage: This is you looking out the window for a split second to see if there is a "Rest Stop" sign. You don't stare out the window forever; you just peek and answer your dad.

VI. Diving Deeper into AbortProc Implementation

Recap:
AbortProc controls printing by checking messages and potentially stopping the job. It's registered with SetAbortProc before StartDoc.

The PeekMessage Trap:
The PeekMessage loop can cause issues like crashes if users interact during printing (e.g., re-printing or quitting).

Disabling Window Input:
To avoid problems, disable window input with EnableWindow(hwnd, FALSE) before calling SetAbortProc, and re-enable it with EnableWindow(hwnd, TRUE) after printing finishes.

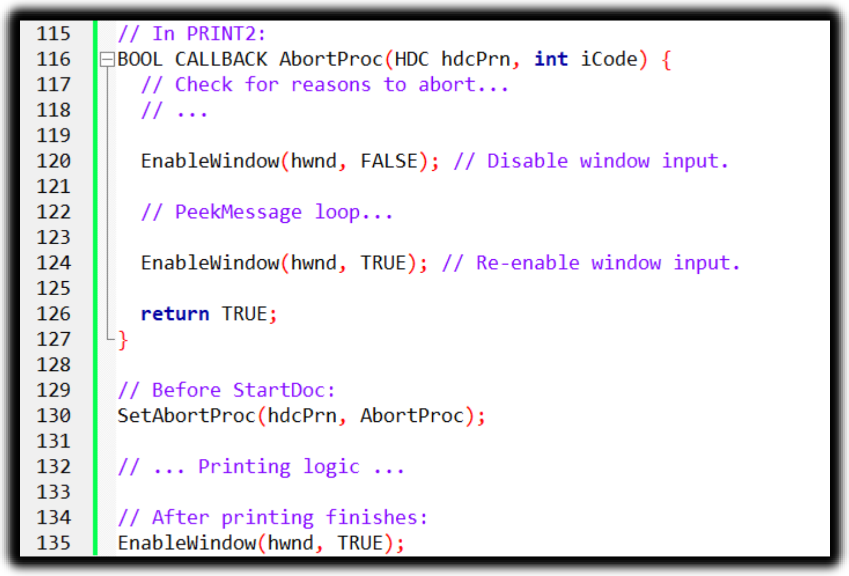
TranslateMessage and DispatchMessage:
DispatchMessage is crucial to handle WM_PAINT and prevent queue issues. TranslateMessage isn't essential but still used for message processing.

User Experience:
With the window disabled, users can switch apps while the program prints in the background, ensuring no interruptions.

PRINT2 builds upon PRINT1 by adding AbortProc and the necessary window handling.

It calls SetAbortProc and strategically uses EnableWindow to prevent user-induced chaos.

Code Example:

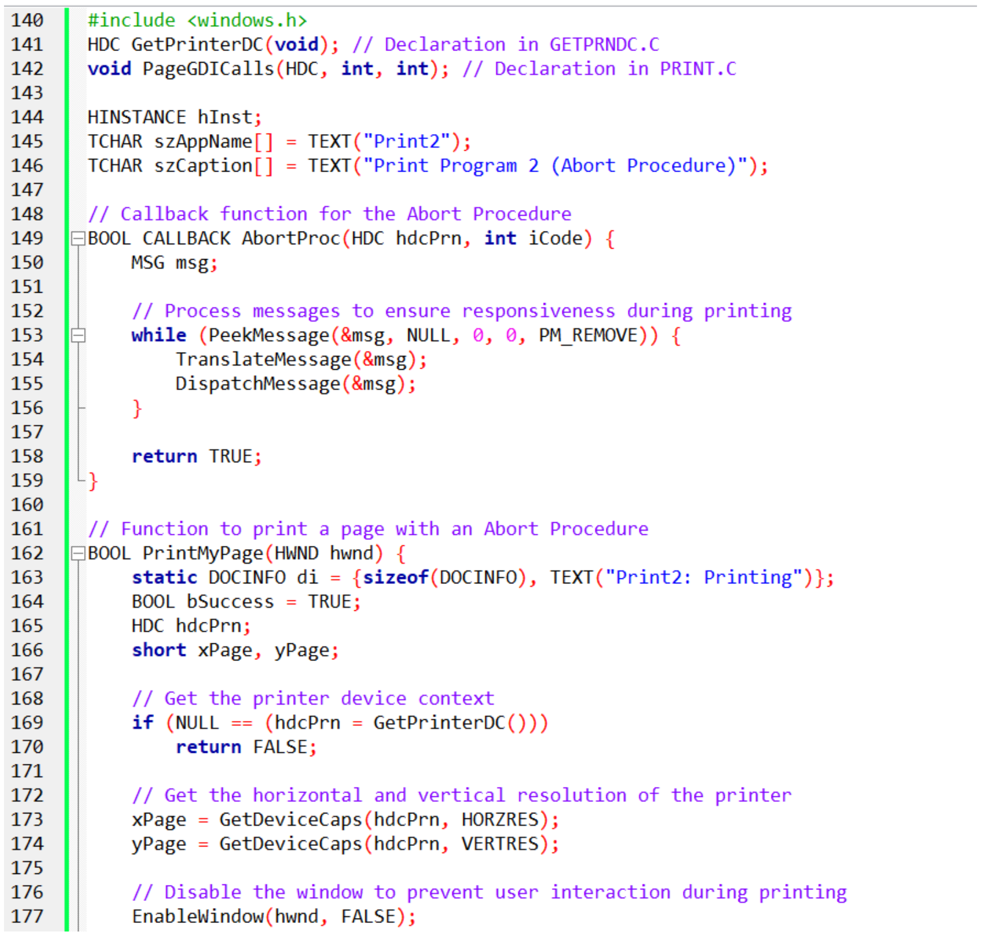


```c
// In PRINT2:
BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode) {
  // Check for reasons to abort...
  // ...

  EnableWindow(hwnd, FALSE); // Disable window input.

  // PeekMessage loop...

  EnableWindow(hwnd, TRUE); // Re-enable window input.

  return TRUE;
}

// Before StartDoc:
SetAbortProc(hdcPrn, AbortProc);

// ... Printing logic ...

// After printing finishes:
EnableWindow(hwnd, TRUE);
```

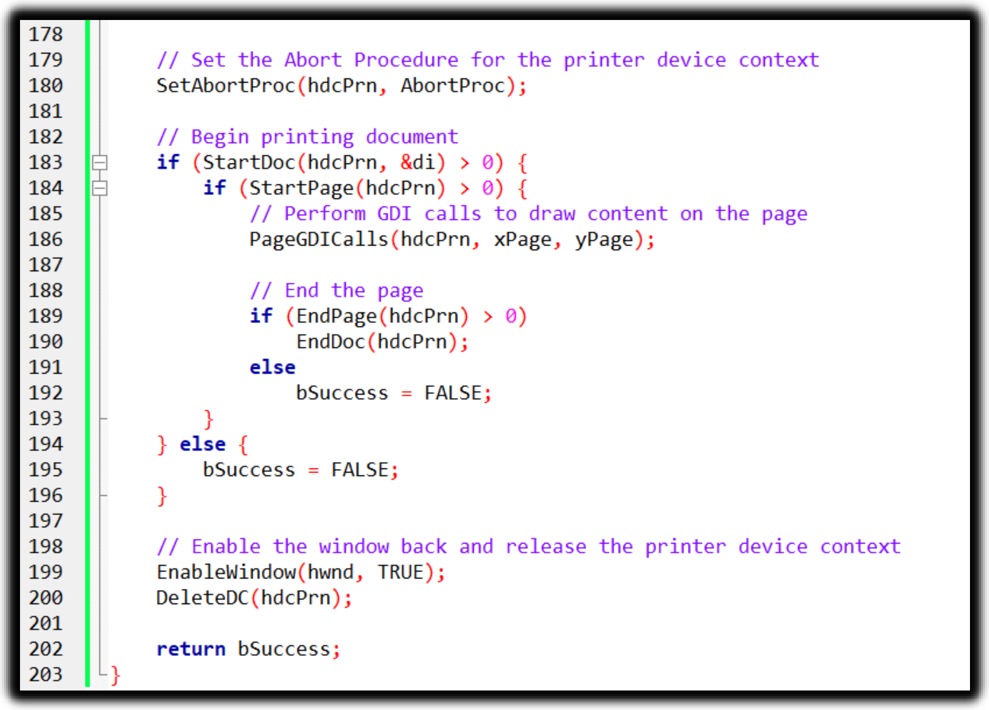
Takeaways:

- To use AbortProc effectively, you need to manage user interaction and handle potential errors carefully.
- Disabling the window during printing keeps things smooth and prevents unexpected issues.
- Understanding these details helps you create more reliable printing apps with easy cancellation.

This breakdown covers the key concepts, but how you implement and handle errors will depend on your app’s specific needs.



```c
#include <windows.h>
HDC GetPrinterDC(void); // Declaration in GETPRNDC.C
void PageGDICalls(HDC, int, int); // Declaration in PRINT.C

HINSTANCE hInst;
TCHAR szAppName[] = TEXT("Print2");
TCHAR szCaption[] = TEXT("Print Program 2 (Abort Procedure)");

// Callback function for the Abort Procedure
BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode) {
    MSG msg;

    // Process messages to ensure responsiveness during printing
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return TRUE;
}

// Function to print a page with an Abort Procedure
BOOL PrintMyPage(HWND hwnd) {
    static DOCINFO di = {sizeof(DOCINFO), TEXT("Print2: Printing")};
    BOOL bSuccess = TRUE;
    HDC hdcPrn;
    short xPage, yPage;

    // Get the printer device context
    if (NULL == (hdcPrn = GetPrinterDC()))
        return FALSE;

    // Get the horizontal and vertical resolution of the printer
    xPage = GetDeviceCaps(hdcPrn, HORZRES);
    yPage = GetDeviceCaps(hdcPrn, VERTRES);

    // Disable the window to prevent user interaction during printing
    EnableWindow(hwnd, FALSE);
```



```c

    // Set the Abort Procedure for the printer device context
    SetAbortProc(hdcPrn, AbortProc);

    // Begin printing document
    if (StartDoc(hdcPrn, &di) > 0) {
        if (StartPage(hdcPrn) > 0) {
            // Perform GDI calls to draw content on the page
            PageGDICalls(hdcPrn, xPage, yPage);

            // End the page
            if (EndPage(hdcPrn) > 0)
                EndDoc(hdcPrn);
            else
                bSuccess = FALSE;
        }
    } else {
        bSuccess = FALSE;
    }

    // Enable the window back and release the printer device context
    EnableWindow(hwnd, TRUE);
    DeleteDC(hdcPrn);

    return bSuccess;
}
```

PRINT2: ENHANCING PRINT1 WITH ABORT PROCEDURE

What’s New in PRINT2:

1. **AbortProc**:
   Defines a BOOL CALLBACK AbortProc function that checks for messages (e.g., Cancel button click) to potentially abort printing using a PeekMessage loop.
2. **Disabling Window Input**:
   The program disables window interaction with EnableWindow(hwnd, FALSE) before the abort procedure, preventing user interference during printing. It re-enables it after printing finishes.
3. **Integrating AbortProc with PrintMyPage**:
   PrintMyPage now calls SetAbortProc(hdcPrn, AbortProc) before starting the document, allowing the print job to be aborted if needed. The rest of the logic remains the same as PRINT1.

Key Benefits:

- **User Control**: Lets users cancel printing mid-process, saving paper and time.
- **Improved Experience**: Enhances user interaction with printing, offering more control.

Key Differences from PRINT1:

- Adds the AbortProc function and window disable/enable features.
- No major changes to the core printing logic.

What's Next:

- PRINT3 builds on PRINT2 by adding a Cancel button in a dialog box for easier user cancellation.

```c
#include <windows.h>
HDC GetPrinterDC(void); // Declaration in GETPRNDC.C
void PageGDICalls(HDC, int, int); // Declaration in PRINT.C
HINSTANCE hInst;
TCHAR szAppName[] = TEXT("Print3");
TCHAR szCaption[] = TEXT("Print Program 3 (Dialog Box)");
BOOL bUserAbort;
HWND hDlgPrint;

BOOL CALLBACK PrintDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam) {
    switch (message) {
        case WM_INITDIALOG:
            SetWindowText(hDlg, szAppName);
            EnableMenuItem(GetSystemMenu(hDlg, FALSE), SC_CLOSE, MF_GRAYED);
            return TRUE;

        case WM_COMMAND:
            bUserAbort = TRUE;
            EnableWindow(GetParent(hDlg), TRUE);
            DestroyWindow(hDlg);
            hDlgPrint = NULL;
            return TRUE;
    }
    return FALSE;
}

BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode) {
    MSG msg;
    while (!bUserAbort && PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
        if (!hDlgPrint || !IsDialogMessage(hDlgPrint, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return !bUserAbort;
}

BOOL PrintMyPage(HWND hwnd) {
    static DOCINFO di = {sizeof(DOCINFO), TEXT("Print3: Printing")};
    BOOL bSuccess = TRUE;
    HDC hdcPrn;
    int xPage, yPage;
    if (NULL == (hdcPrn = GetPrinterDC()))
        return FALSE;

    xPage = GetDeviceCaps(hdcPrn, HORZRES);
    yPage = GetDeviceCaps(hdcPrn, VERTRES);

    EnableWindow(hwnd, FALSE);
    bUserAbort = FALSE;
    hDlgPrint = CreateDialog(hInst, TEXT("PrintDlgBox"), hwnd, PrintDlgProc);
    SetAbortProc(hdcPrn, AbortProc);

    if (StartDoc(hdcPrn, &di) > 0) {
        if (StartPage(hdcPrn) > 0) {
            PageGDICalls(hdcPrn, xPage, yPage);
            if (EndPage(hdcPrn) > 0)
                EndDoc(hdcPrn);
            else
                bSuccess = FALSE;
        }
    } else {
        bSuccess = FALSE;
    }
    if (!bUserAbort) {
        EnableWindow(hwnd, TRUE);
        DestroyWindow(hDlgPrint);
    }
    DeleteDC(hdcPrn);
    return bSuccess && !bUserAbort;
}
```

Problems with PRINT2:

Lack of Feedback: Users don't know if printing is still going on or if it's finished unless they interact with the program.

No Cancellation: Users can't stop the printing once it starts.

I. Introducing the Printing Dialog Box in PRINT3

PRINT3 solves these issues by adding a modeless dialog box with a **Cancel** button, which:

- Lets users see the printing progress.
- Provides an option to cancel the print job.

II. Key Functions in PRINT3

1. **AbortProc**:
   The function stays similar to PRINT2, but now it sends messages to the dialog box window procedure to check if the user clicked **Cancel**.
2. **PrintDlgProc**:
   A new function that handles the dialog box's **WM_COMMAND** messages. It tracks if the user clicks the **Cancel** button.

III. User Interaction and Cancellation:

- If the user clicks **Cancel**, PrintDlgProc sets the bUserAbort flag to TRUE.
- AbortProc checks this flag, and if TRUE, it aborts the printing process. If not, it continues.

IV. Benefits of the Dialog Box:

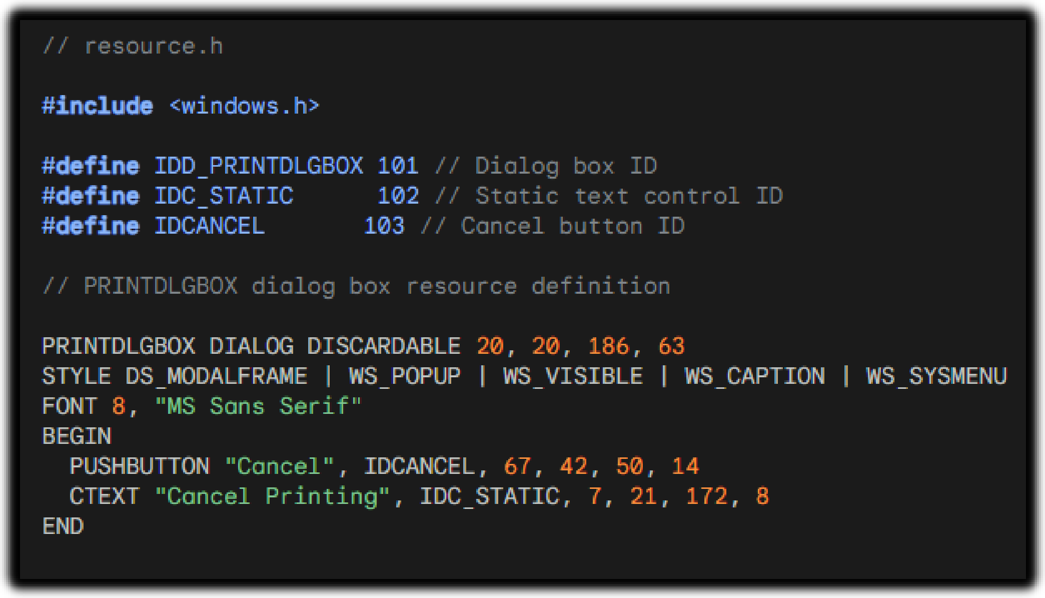
- **Better User Experience**: Users get immediate feedback on the printing process.
- **Improved Control**: The ability to cancel printing makes the process less frustrating.
- **Professional Touch**: Makes your app feel more polished and user-friendly.

IV. Key Differences from PRINT2:

- Added a **printing dialog box** with a **Cancel** button.
- Introduced the **PrintDlgProc** function to handle dialog interactions.
- Updated **AbortProc** to check for user cancellation via bUserAbort.
- PRINT3 takes a big step forward by adding user-friendly features like the modeless dialog box and cancellation options.
- It's a more interactive and polished printing experience, making your app feel more Windows-like and intuitive.

V. Next Steps:

- Dive into how PrintDlgProc works and how it handles the dialog box.
- Customize the dialog's look and content to fit your app.
- Explore other advanced printing features in Windows APIs for further improvements.



```
// resource.h

#include <windows.h>

#define IDD_PRINTDLGBOX 101 // Dialog box ID
#define IDC_STATIC      102 // Static text control ID
#define IDCANCEL       103 // Cancel button ID

// PRINTDLGBOX dialog box resource definition

PRINTDLGBOX DIALOG DISCARDABLE 20, 20, 186, 63
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
FONT 8, "MS Sans Serif"
BEGIN
  PUSHBUTTON "Cancel", IDCANCEL, 67, 42, 50, 14
  CTEXT "Cancel Printing", IDC_STATIC, 7, 21, 172, 8
END
```

This code snippet defines a resource named PRINTDLGBOX which is a dialog box with the following features:

- **IDD_PRINTDLGBOX:** Unique identifier for the dialog box.
- **DISCARDABLE:** Indicates the dialog can be discarded from memory when not needed.
- **20, 20:** Coordinates of the dialog box's upper left corner.
- **186, 63:** Width and height of the dialog box.
- **DS_MODALFRAME:** Makes the dialog modal, preventing interaction with other windows.
- **WS_POPUP:** Creates the dialog as a pop-up window.
- **WS_VISIBLE:** Makes the dialog box visible initially.
- **WS_CAPTION:** Displays a title bar for the dialog box.
- **WS_SYSMENU:** Shows a system menu with minimize and close buttons.
- Font: Sets the dialog box font to "MS Sans Serif" with size 8.
- **BEGIN/END:** Delimit the dialog box controls.
- **PUSHBUTTON:** Defines a button with the text "Cancel" and ID IDCANCEL.
- **CTEXT:** Defines a static text control with the text "Cancel Printing" and ID IDC_STATIC.

This resource definition provides the basic structure for the printing dialog box used in PRINT3.

Referencing these IDs in your code means you can interact with the dialog box elements like the Cancel button and static text.

DEEP DIVE INTO PRINT3: USER CONTROL AND CANCELLATION

PRINT3 builds on PRINT2 by adding:

- **Printing Dialog Box**: A modeless dialog with a "Cancel" button during printing, providing feedback and cancellation.

- **User-Driven Cancellation**: Clicking "Cancel" sets bUserAbort to TRUE, which stops the print job in AbortProc.

- **Improved Experience**: Users can monitor and cancel printing as needed for greater control and satisfaction.

Key Implementation Details:

Global Variables:

- bUserAbort: Tracks if the user cancels (initially FALSE).
- hDlgPrint: Handle to the printing dialog box.

PrintMyPage:

- Initializes bUserAbort and disables the main window.
- Sets up AbortProc and PrintDlgProc.
- Creates the printing dialog box with CreateDialog.

AbortProc Message Loop:

- Checks bUserAbort and processes messages with IsDialogMessage.
- Returns TRUE to continue printing or FALSE to cancel.

PrintDlgProc:

- Handles dialog box actions, including setting the title and handling "Cancel" button press.
- On "Cancel", sets bUserAbort to TRUE and cleans up.

EndPage and Cancellation:

- If AbortProc returns FALSE (cancellation), PrintMyPage stops printing by calling EndDoc.

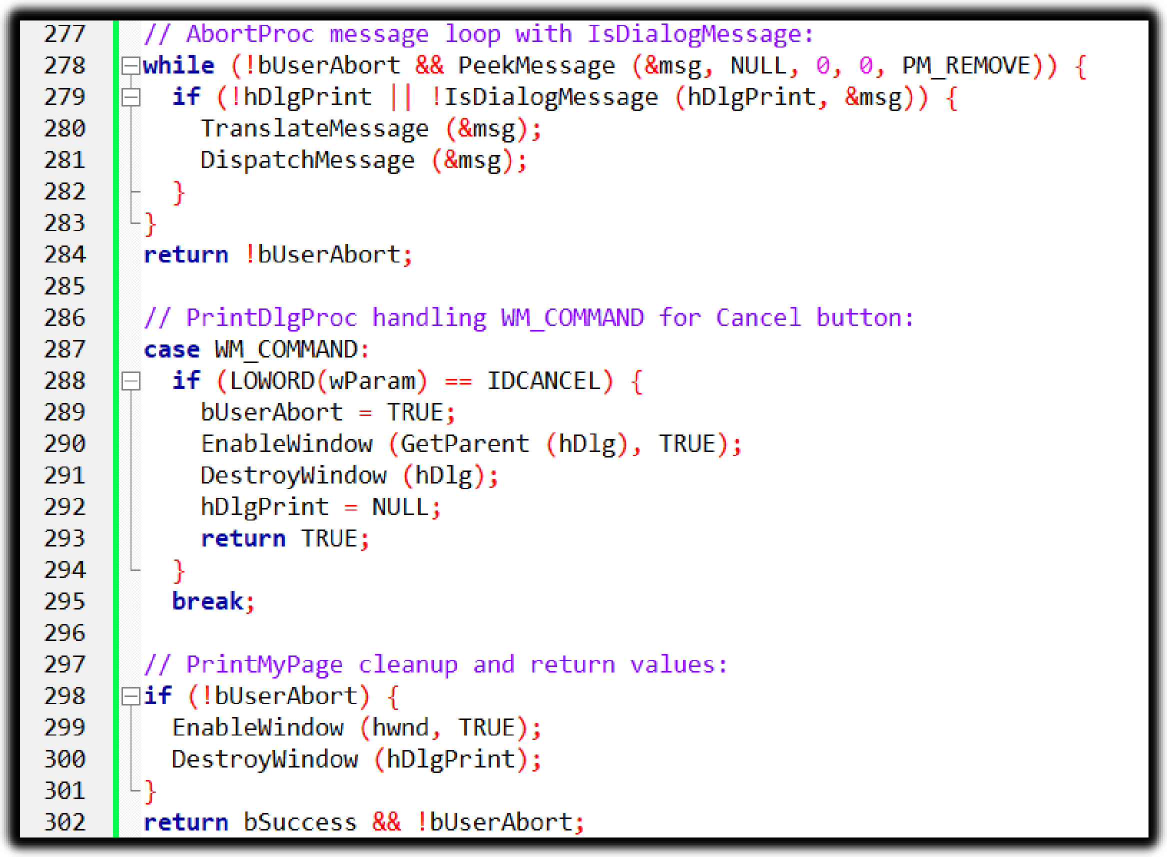
Cleanup:

- If not canceled, PrintMyPage reenables the window and destroys the dialog box.
- Returns printing success or cancellation status.

Key Takeaways:

- PRINT3 offers a more user-friendly printing experience with an interactive cancel option.
- The collaboration between bUserAbort, AbortProc, PrintDlgProc, and EndPage provides control over the printing process.
- With this setup, you can create flexible and interactive Windows printing apps.

Code Example:



```c
// AbortProc message loop with IsDialogMessage:
while (!bUserAbort && PeekMessage (&msg, NULL, 0, 0, PM_REMOVE)) {
  if (!hDlgPrint || !IsDialogMessage (hDlgPrint, &msg)) {
    TranslateMessage (&msg);
    DispatchMessage (&msg);
  }
}
return !bUserAbort;

// PrintDlgProc handling WM_COMMAND for Cancel button:
case WM_COMMAND:
  if (LOWORD(wParam) == IDCANCEL) {
    bUserAbort = TRUE;
    EnableWindow (GetParent (hDlg), TRUE);
    DestroyWindow (hDlg);
    hDlgPrint = NULL;
    return TRUE;
  }
  break;

// PrintMyPage cleanup and return values:
if (!bUserAbort) {
  EnableWindow (hwnd, TRUE);
  DestroyWindow (hDlgPrint);
}
return bSuccess && !bUserAbort;
```

Remember, this is just an example. Adapt and expand upon these concepts to create custom printing experiences.

DELVING DEEPER INTO POPPRNT.C: UNVEILING THE PRINTING ENGINE OF POPPAD

Delving Into POPPRNT.C: The Printing Engine Behind POPPAD

The **POPPRNT.C** file turns POPPAD from a simple text editor into a fully functional document creation tool by adding printing capabilities.

Here’s a breakdown of how it works:

Key Components:

I. Global Variables:

- **bUserAbort**: A flag that stops printing when the user hits "Cancel." It starts as FALSE and switches to TRUE when the user intervenes.
- **hDlgPrint**: A handle connecting the main program to the printing dialog box, controlling its appearance and behavior.

II. The Printing Dialog Box:

PrintDlgProc: The function behind the dialog box. It listens for messages:

- **WM_INITDIALOG**: Disables the "Close" option to prevent accidental cancellation.
- **WM_COMMAND**: If the user clicks "Cancel," it sets bUserAbort to TRUE, halting printing, and cleans up by re-enabling the main window and destroying the dialog.

III. The Printing Process:

PopPrntPrintFile: The function that orchestrates the printing process:

- **Setting Up**: It calculates page layout, page capacity, and allocates memory for each line.
- **User Input**: Displays a "Print" dialog for printer selection and settings.
- **Printing Loop**: Iterates through pages and lines, using **TextOut** to print the text to each page.
- **Completion**: After printing, it ends the document with **EndDoc**, frees memory, and returns control to the user.

IV. Abort Procedure:

AbortProc: Constantly checks for user cancellation, filtering messages to ensure only relevant ones are processed during printing.

V. Beyond the Basics:

POPPRNT.C provides a great foundation for customization. You can improve the dialog box with more options, adjust the layout, or expand error handling to enhance the printing experience.

Unveiling the Secrets of POPPAD's Printing Engine: A Deep Dive into POPPRNT.C

POPPRNT.C is a powerful and straightforward integration of Windows printing features, turning POPPAD into a tool that can transform digital documents into printed ones. Let's break down how it works:

I. Unlocking the PrintDlg Magic

- **PrintDlg**: This common dialog box function controls the print setup. POPPAD uses it to display a familiar print dialog, where users can select page ranges, adjust copies, and toggle collation.
- **User Features**: Beyond basic page selection, the dialog lets users set options like multiple copies, stacked or interleaved collation, printer choice, and orientation (portrait or landscape).
- **Return Values**: After PrintDlg returns, the PRINTDLG structure holds important info like the page range, collation preference, and printer device context for printing.

II. PopPrntPrintFile: The Orchestrator of Printing

- **Setting the Stage**: When the user hits "Print," this function kicks off, calculating page resolution and character dimensions with GetDeviceCaps and GetTextMetrics to optimize layout.
- **Line by Line, Page by Page**: It retrieves the total line count using EM_GETLINECOUNT, allocates memory for the text, and processes each line.
- **The Ink Touches Paper**: EM_GETLINE pulls each line, which is then printed on the virtual page with TextOut, creating the document one line at a time.

III. Collation: A Matter of Order

Two Loops for Precision: POPPAD handles both collated and non-collated printing by using two nested loops, making sure the copies print in the correct order (stacked or interleaved).

IV. User Control: Canceling the Print Job

User Abort: Throughout the printing process, POPPAD checks the bUserAbort flag. If it's set to TRUE (user cancellation), printing halts immediately.

V. Beyond the Basics

EndDoc – The Grand Finale: After the document is printed, POPPAD ends the process with EndDoc, ensuring a clean finish.

Error Handling: Interestingly, POPPAD skips the use of AbortDoc. Instead, user control through the dialog box and regular error checks with EndPage make it unnecessary.

The Sequence of Calls: A Choreographed Dance

Figure 13-11 illustrates the optimal sequence of printing functions for multi-page documents.

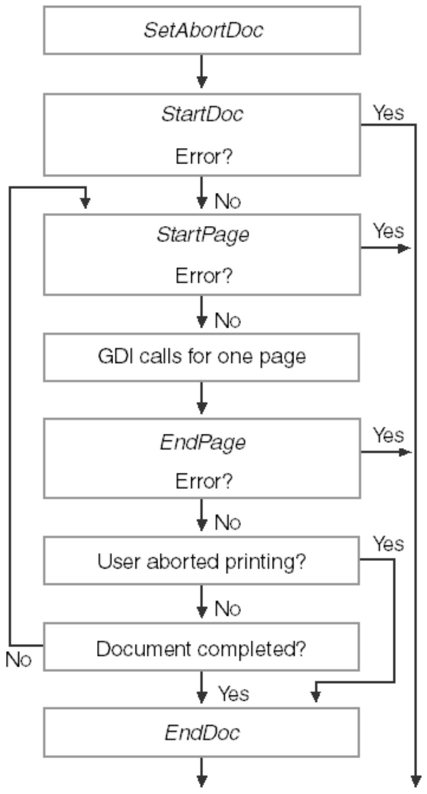
To simplify: think of multi-page printing as a **loop** where you check for a "stop" signal at every sheet of paper.

I. The Printing Loop

1. **StartDoc**: Open the connection.
2. **The Loop**: For every page:
    - ✓ Call **StartPage**.
    - ✓ Draw your content.
    - ✓ Call **EndPage**.
    - ✓ **The Check:** This is the best time to see if the user clicked "Cancel" (bUserAbort). If they did, break the loop.
3. **EndDoc**: Close the connection and send the job to the tray.

II. Why it's a "Dance":

You have to keep your program's "Cancel" dialog in sync with the printer. If the user hits cancel while the printer is busy, EndPage is the "checkpoint" where you officially stop sending data.



III. The Printing Loop Flow

Think of printing as a **nested loop**: The outer loop is the **Document**, and the inner loop is the **Page**.

1. **StartDoc**: Open the job.
2. **StartPage**: Prepare the paper.
3. **GDI Calls**: Draw your text/graphics.
4. **EndPage**: Eject the paper.
5. **Check Status**:
   - ✓ **User Cancelled?** Call AbortDoc to kill the job.
   - ✓ **Error Occurred?** Stop immediately; don't try to call more functions.
   - ✓ **More Pages?** Go back to step 2.
6. **EndDoc**: Finish the job if everything went perfectly.

IV. Essential Tips

- **Memory:** You are responsible for cleaning up GDI objects (pens, brushes) used during drawing.
- **The Abort Procedure:** Use SetAbortProc to create a "Cancel" window so the user can stop the printer if they realize they're printing 100 pages by mistake.
- **Failure Rule:** If any function fails, the "dance" is over. Don't try to "fix" it mid-job; just exit gracefully.

Printer Poppad
Notepad with printi

V. Final Takeaway: Why POPPRNT.C Matters

POPPRNT.C isn't just about printing text; it's the "Gold Standard" template for any professional Win32 print job. It solves the three biggest headaches:

1. **The Dialog:** It uses PrintDlg so the user feels in control.
2. **The Background Thread:** It handles the "Cancel" window so your app doesn't look like it "Crashed" while the printer is warming up.
3. **The Clean Exit:** It demonstrates exactly how to kill a job (AbortDoc) if the user changes their mind.

The Bottom Line: Don't reinvent the wheel. If you need to add printing to an app, copy the structure of POPPRNT.C, swap out the drawing logic, and you're done.