CHAPTER 13: PRINTING WITH GDI: MOVING BEYOND THE SCREEN

This chapter explores the application of GDI for printing, delving into the similarities and differences compared to using it for the video display. While the concept of device independence remains largely applicable, some key distinctions emerge.

Similarities:

- GDI Functions: Many of the same GDI functions used for drawing on the screen can be utilized for printing text, lines, shapes, and other graphics elements on paper.
- Device Independence: The device-independent nature of GDI allows programs to write code without worrying about specific printer hardware details. The underlying system handles the translation and formatting for different printers.
- Drawing Concepts: Fundamental concepts like line styles, colors, brush styles, and text rendering are consistent between screen and printer output.

Differences:

- Printer Capabilities: Printers are not simply a paper-based video display. They have limitations in terms of speed, graphics support (some cannot handle bitmaps), and paper handling (e.g., page ejection).
- Output Speed: Printing is significantly slower than drawing on the screen. Programs need to be mindful of potential performance bottlenecks and optimize accordingly.
- Page Management: Unlike the reusable video display surface, printed pages require separate handling. Each page needs to be initiated with StartPage, completed with EndPage, and ejected upon completion.
- Document Organization: Unlike the windowed environment of the screen, printing output from different applications requires separation into distinct documents or print jobs.
- Printer-Specific Functions: GDI offers additional functions like StartDoc and EndDoc specifically for managing printer output and organizing pages.

Additional Resources:

- Chapter 15: Printing Bitmaps.
- Chapter 17: Printing Formatted Text.
- Chapter 18: Printing Metafiles.

These chapters provide further information on printing specific data formats using GDI and its related functions.

PRINTING PROCESS: A DETAILED BREAKDOWN

This section dives deep into the intricate process of printing in Windows, highlighting the interactions between the application program, the GDI module, the printer driver, and the print spooler.

Initiating the Process:

CreateDC or PrintDlg: The program acquires a handle to the printer device context, triggering the loading and initialization of the printer driver module (if necessary).

StartDoc: This function marks the beginning of a new document, handled by the GDI module. It subsequently calls the printer driver's Control function, preparing the device for printing.

Page Delimiters:

StartDoc/EndDoc: These calls bookend the normal GDI functions used for drawing page content.

StartPage/EndPage: These calls further delimit individual page boundaries within the document.

Metafile Creation:

For each page, the GDI module initially stores the drawing commands in a disk-based metafile (.EMF) for most printers.

This metafile acts as an intermediary representation of the page content.

Banding:

- High-resolution printers often use "banding" to divide the page into smaller, manageable sections.
- GDI obtains the band dimensions from the driver and sets a clipping region accordingly.
- The Output function within the driver then translates the metafile drawing commands for each band.
- This process, called "playing the metafile," ensures efficient handling of large print jobs.

Driver Output Generation:

- Each band requires translation into printer-specific output format.
- For dot-matrix printers, this involves control sequences and graphics commands.
- Laser printers with high-level languages like PostScript generate output in that specific language.

Temporary File Storage:

- The driver-translated output for each band is stored in another temporary file (.SPL) by the GDI module.
- This file acts as a buffer before handing over the entire print job to the spooler.

Print Spooler Interaction:

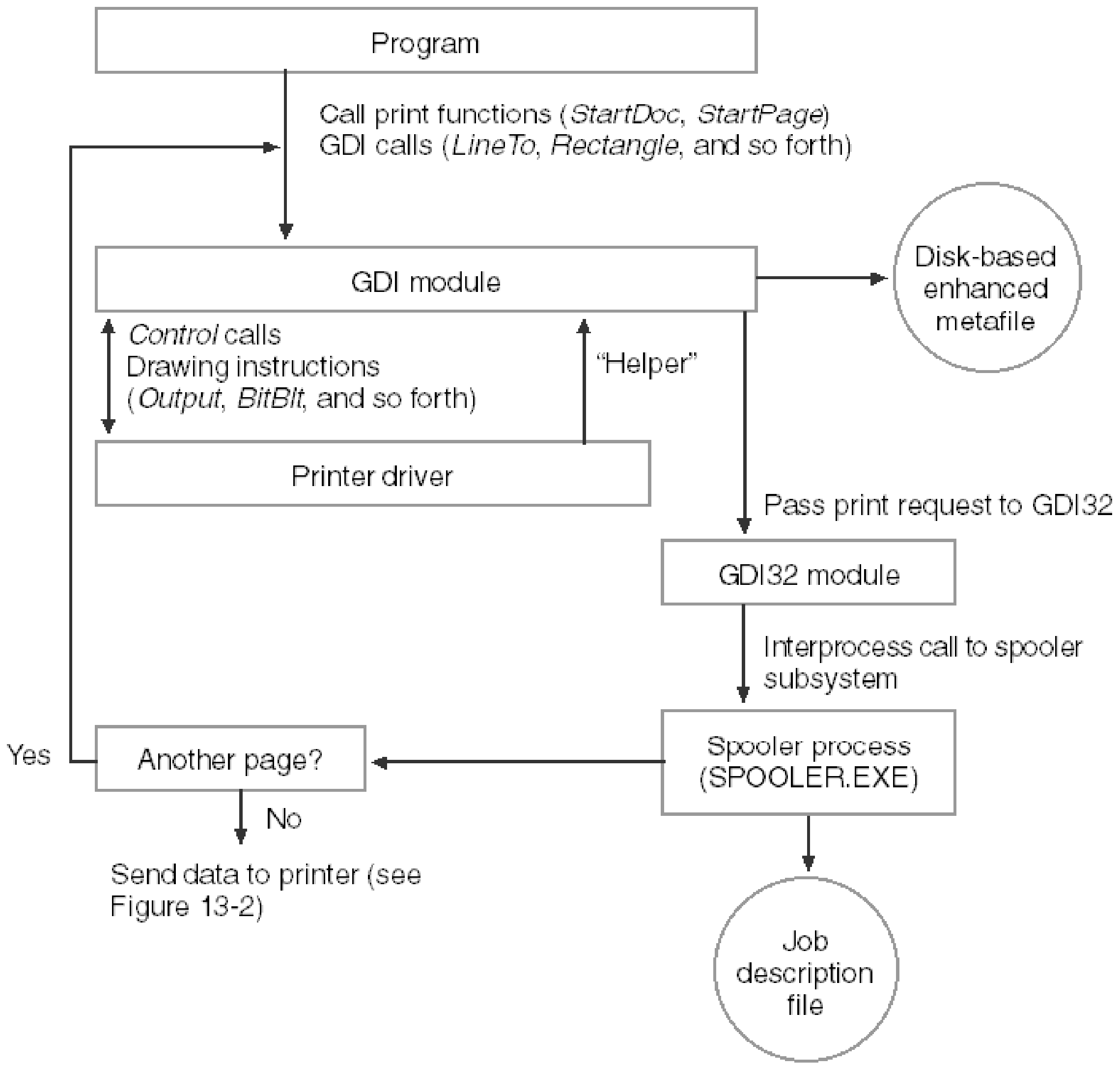
- Once the entire page is processed, the GDI module informs the print spooler about the new print job.
- This allows the spooler to manage and prioritize printing tasks for multiple applications.

EndDoc and Completion:

- After all page processing is complete, the program calls EndDoc to signal the end of the print job.
- This allows for cleanup and finalization of the printing process.

Key Points:

- Printing involves a complex interaction between the application, GDI, printer driver, and print spooler.
- Metafiles act as an intermediate representation of page content for efficient processing.
- Banding helps handle large print jobs on high-resolution printers.
- Driver translation converts drawing commands into printer-specific output formats.
- Temporary files store intermediary data for efficient spooling and management.


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

The Windows print spooler is a complex system consisting of various components working together to ensure efficient and smooth printing. Here's an in-depth breakdown of each component and its respective function:

1. Spooler:

Acts as the central hub, receiving print requests from applications and managing the entire printing process.

Routes the data stream containing the print job to the appropriate print provider.

2. Local Print Provider:

Handles print jobs destined for locally connected printers.

Creates and manages spool files containing the print job data in a format suitable for the specific printer.

3. Network Print Provider:

Responsible for handling print jobs directed towards network printers.

Similar to the Local Print Provider, it creates and manages spool files but for network destinations.

4. Print Processor:

Performs the crucial step of "despooling," which involves converting the device-independent spool file data into a format specifically understandable by the target printer.

This ensures compatibility and proper interpretation of the print job by the printer hardware.

5. Port Monitor:

Manages the communication port to which the printer is physically connected.

Oversees data transfer between the spooler and the printer, ensuring accurate and reliable transmission of the print job.

6. Language Monitor:

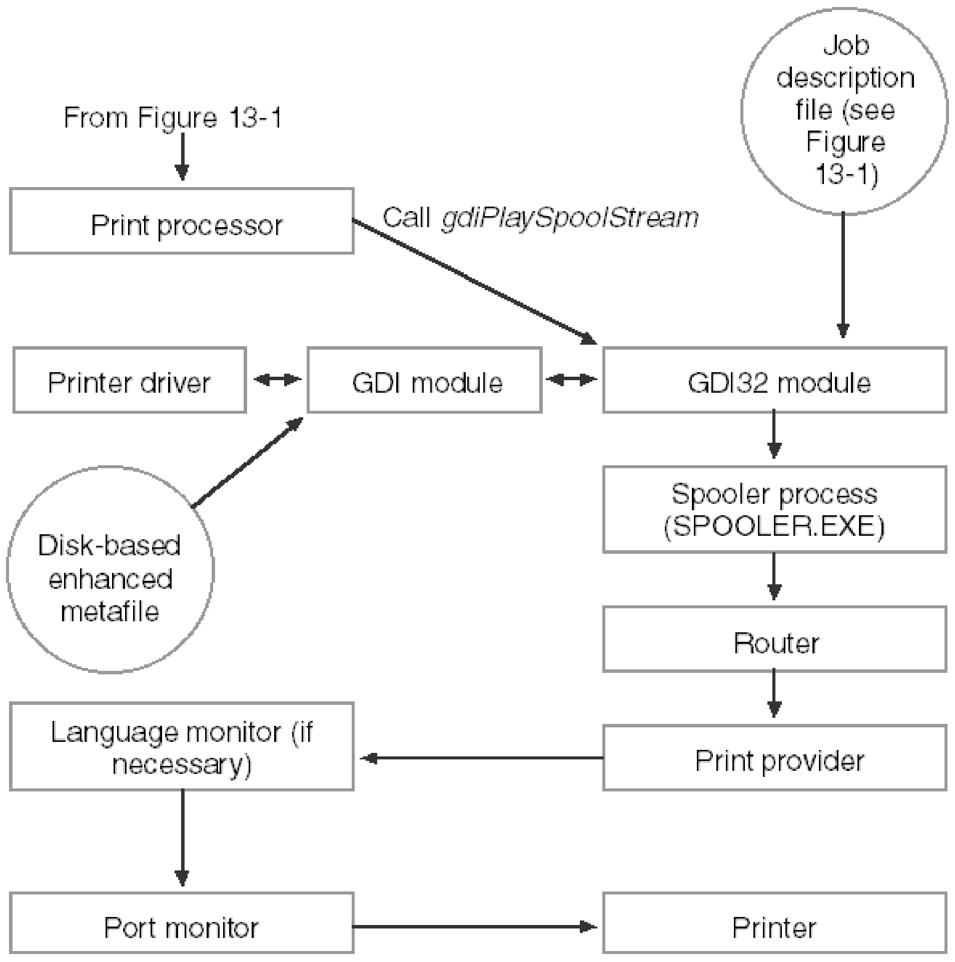
Applies to printers capable of two-way communication.

Enables configuration settings and status monitoring of the printer directly from the spooler.

This provides additional control and feedback about the printing process.

Benefits of the Print Spooler:

- Reduced workload on applications: By handling the intricacies of print job processing and communication with the printer, the spooler offloads this burden from applications, allowing them to focus on their primary functionality.
- Improved performance: Spooling allows applications to continue working while the print job is being processed and sent to the printer, resulting in faster response times and smoother workflow.
- Queued printing: The spooler manages a queue of print jobs, allowing multiple jobs to be submitted while ensuring they are printed in the correct order.
- Background printing: Users can continue working on their computers while the print spooler quietly transfers the print job to the printer in the background, maximizing productivity.


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

PRINTING PROCESS TRANSPARENCY AND VARIATIONS

While the printing process seems transparent to the application, understanding its nuances is crucial for developers. Here's a deeper look into the variations and potential implications:

1. Transparency for Applications:

Applications experience "printing" only during the time GDI saves the printer output to disk files.

This allows them to continue working while the spooler handles the actual printing, improving responsiveness.

2. Spooler Bypass:

Printing without the spooler is technically possible by disabling it in the printer properties.

Reasons for bypassing the spooler might include:

- Using a faster hardware or software spooler.
- Printing on a network with its own spooler.
- Avoiding the performance overhead of two spoolers.

This approach removes disk storage for print jobs but can potentially slow down the application until printing is complete.

3. GDI Direct Output:

When the spooler is inactive, GDI directly transmits printer output to the port, bypassing file storage.

While faster, this method can hold up the application program until printing finishes.

4. Metafile Variations:

GDI typically stores drawing functions in a metafile used for each band defined by the printer driver.

For drivers not requiring banding, the metafile is skipped, and GDI directly sends functions to the driver.

Alternatively, the application can manage band division, adding complexity but relieving GDI of metafile creation.

5. Potential Problems:

Printing can involve more overhead than video display usage.

Issues like GDI running out of disk space during file creation require handling and user feedback.

6. First Step: Obtaining a Printer Device Context:

The first step in printing from an application is acquiring a printer device context.

This involves functions like CreateDC or PrintDlg and triggers driver loading and initialization.

OBTAINING A PRINTER DEVICE CONTEXT

This section delves into the intricacies of acquiring a printer device context, crucial for printing from your application.

1. Device Context Handle:

Similar to interacting with the video display, printing requires a printer device context handle. This handle serves as the communication channel between your application and the printer driver, allowing you to issue drawing commands.

2. StartDoc and StartPage:

Before issuing drawing commands, you need to signal the start of a new document with StartDoc and the start of a new page with StartPage. These functions inform the system about your printing intentions and prepare the printer driver for receiving commands.

3. Creating the Device Context:

Two main approaches exist for obtaining the printer device context:

Standard Print Dialog (PrintDlg): This function displays a dialog box allowing the user to choose a printer and customize printing options. Upon selection, PrintDlg provides the application with a printer device context handle.

Direct Creation (CreateDC): This function offers more control and avoids displaying a dialog box. It requires the application to provide the printer's device name, which can be obtained through functions like EnumPrinters.

4. CreateDC Syntax:

The syntax for CreateDC when dealing with printers differs slightly from the video display:

```
hdc = CreateDC(NULL, szDeviceName, NULL, pInitializationData);
```

- szDeviceName: Pointer to a character string containing the specific printer's device name.
- pInitializationData: Generally set to NULL.

5. Finding Available Printers:

Since multiple printers can be attached to a system, determining the available options becomes crucial. This is achieved using the EnumPrinters function, which fills an array of structures containing information about each attached printer.

6. Getting the Default Printer Device Context:

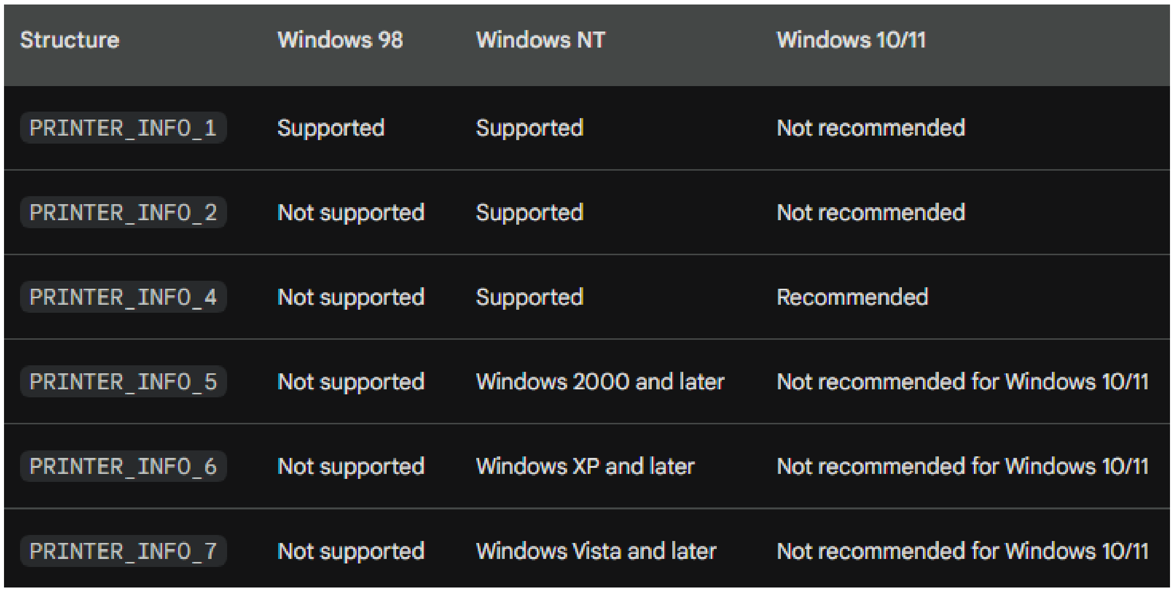
The GetPrinterDC function shown in Chapter 13 provides a platform-independent approach to retrieving the default printer's device context. It works under both Windows 98 and Microsoft Windows NT.

7. Choosing the Right PRINTER_INFO_x Structure:

The specific PRINTER_INFO_x structure to use with EnumPrinters depends on the desired level of detail and the operating system version:

- Windows 98: PRINTER_INFO_1
- Microsoft Windows NT: PRINTER_INFO_2 or PRINTER_INFO_4
- Windows 10 and 11: For Windows 10 and 11, the recommended PRINTER_INFO_x structure for using with EnumPrinters is PRINTER_INFO_4. This structure provides a good balance of information about each printer without being overly detailed.

Here's a breakdown of the available structures and their compatibility:



Structure	Windows 98	Windows NT	Windows 10/11
`PRINTER_INFO_1`	Supported	Supported	Not recommended
`PRINTER_INFO_2`	Not supported	Supported	Not recommended
`PRINTER_INFO_4`	Not supported	Supported	Recommended
`PRINTER_INFO_5`	Not supported	Windows 2000 and later	Not recommended for Windows 10/11
`PRINTER_INFO_6`	Not supported	Windows XP and later	Not recommended for Windows 10/11
`PRINTER_INFO_7`	Not supported	Windows Vista and later	Not recommended for Windows 10/11

Reasons for using PRINTER_INFO_4:

- Provides important information like printer name, driver name, and location.
- Relatively compact structure size compared to other options.
- Efficient for enumerating large numbers of printers.

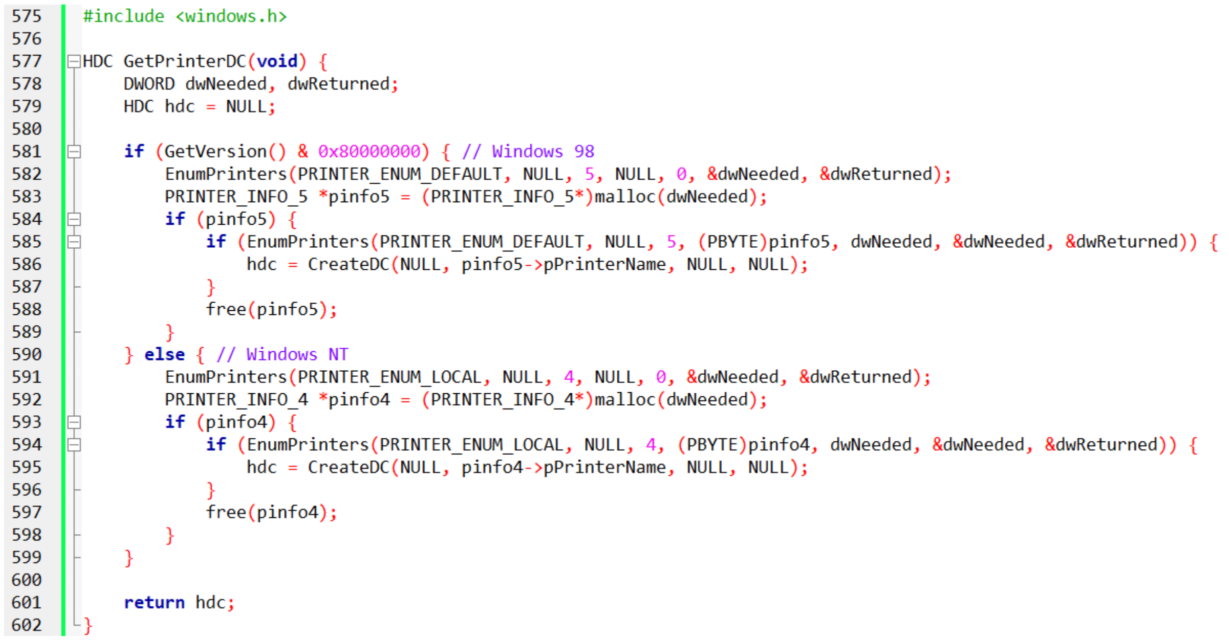
Alternatives to PRINTER_INFO_4:

- You can still use PRINTER_INFO_1 if you only require basic information like printer name.
- For more detailed information like printer properties and capabilities, consider using GetPrinterDriver and GetPrinter functions with the appropriate structures like DRIVER_INFO_x and PRINTER_INFO_2.

Conclusion:

- Choosing the appropriate PRINTER_INFO_x structure depends on your specific needs and the operating system you are targeting. For Windows 10 and 11, PRINTER_INFO_4 offers a good balance of information and efficiency for enumerating printers.
- Understanding the different methods for obtaining a printer device context and the intricacies involved in choosing the appropriate approach empowers you to effectively initiate printing from your application and interact with the chosen printer for subsequent drawing commands.

GETPRINTERDC FUNCTION: EXPLAINED



```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;
}
```

The GetPrinterDC function serves a crucial purpose in printing applications: it retrieves the device context handle for the default printer. Understanding its operation is crucial for effective printing.

1. Version Check:

The function first utilizes GetVersion to determine the operating system: Windows 98 or Windows NT.

2. Structure Selection and Memory Allocation:

Windows 98:

EnumPrinters is called twice:

- First call: Determines the required size for the PRINTER_INFO_5 structure.
- Second call: Allocates memory for PRINTER_INFO_5 and retrieves information about the default printer.

CreateDC creates the device context handle using the obtained printer name from pinfo5.

Allocated memory is then freed.

Windows NT:

Similar approach as Windows 98, but uses the PRINTER_INFO_4 structure instead.

3. CreateDC and Device Context Handle:

The chosen structure (pinfo5 or pinfo4) provides the printer name, which CreateDC utilizes to create the device context handle (hdc). This handle is ultimately returned by the function.

4. Efficiency and Documentation:

The function employs a two-pass approach with EnumPrinters to optimize memory usage. The specific structures (PRINTER_INFO_5 for Windows 98 and PRINTER_INFO_4 for Windows NT) are chosen based on their efficiency and ease of use, as mentioned in the Microsoft documentation.

5. Conclusion:

GetPrinterDC demonstrates the dynamic selection of appropriate structures based on the operating system and prioritizes efficiency while retrieving the necessary information for obtaining the default printer's device context handle. This functionality is essential for printing applications to interact with the chosen printer and issue drawing commands.

DEVCAPS2 PROGRAM

- The DEVCAPS2 program is a Windows application written in C that provides a graphical user interface to display various device capabilities information for both the video display and all printers attached to the system. It is an extended version of the original DEVCAPS1 program, displaying more detailed information obtained through the GetDeviceCaps function.
- Upon execution, the program creates a window with menus allowing the user to select either the video display or one of the available printers for which they want to view device capabilities. The user can also choose from different categories of information, such as basic information, other information, curve capabilities, line capabilities, polygonal capabilities, and text capabilities.
- The program utilizes the GetDeviceCaps function to retrieve information about the specified device context (either the video display or a printer). It covers a wide range of device capabilities, including dimensions (size in millimeters and pixels), color information (bits per pixel, color planes), and various other capabilities like the number of brushes, pens, markers, fonts, and available colors.
- The program dynamically populates the menu with the names of all local and remote printers attached to the system. It also provides an option to view printer properties, which opens a dialog displaying additional information about the selected printer.
- The graphical user interface is implemented using the Windows API, and the window displays the chosen device and information category. The information is presented in a readable format, and the program uses a fixed system font for consistent text rendering.
- In addition to displaying basic information, the program delves into more detailed capabilities, such as clipping, rasterization, curve capabilities, line capabilities, polygonal capabilities, and text capabilities. For each category, the corresponding capabilities are presented in a clear and organized manner, allowing users to understand the capabilities of the selected device.
- Overall, DEVCAPS2 serves as a practical tool for users and developers to explore and understand the capabilities of their display devices and printers and to gather valuable information about the features and limitations of the connected hardware. The program's structure is modular, with distinct functions handling different aspects of device capabilities, making it easy to extend or modify for future enhancements.
- The initialization process involves setting up the window class and creating the main window. The program responds to user actions such as selecting a different device or information category through menu options. It dynamically updates the menu with available printer names and provides a straightforward interface for users to interact with.

Window Procedure (WndProc):

The program defines a window procedure (WndProc) that handles various messages sent to the application window. It includes cases for window creation, command handling, initialization of menu items, painting the window, and handling the destruction of the window.

Initialization (WM_CREATE and WM_SETTINGCHANGE):

In the WM_CREATE case, the program retrieves device context information, such as the average character width and height, to determine the size of characters in the system font.

The WM_SETTINGCHANGE case is used to update the menu when the system settings change, such as when a new printer is added or removed.

Menu Handling (WM_COMMAND and WM_INITMENUPOPUP):

The program responds to menu commands in the WM_COMMAND case. It distinguishes between selections related to the display (IDM_SCREEN), printers, and properties. It also allows the user to choose different information categories, such as basic information, other information, curve capabilities, etc.

The WM_INITMENUPOPUP case enables or disables the "Properties" menu item based on whether the current selection is the screen or a printer.

Painting the Window (WM_PAINT):

In the WM_PAINT case, the program prepares and displays information about the selected device and information category. It creates a device context for information retrieval (hdcInfo), selects a system-fixed font, and uses TextOut to display the gathered information.

Basic Information (DoBasicInfo):

The DoBasicInfo function displays fundamental information about the selected device, such as size in millimeters, resolution in pixels, color depth, number of brushes, pens, markers, fonts, and available colors.

Other Information (DoOtherInfo):

The DoOtherInfo function provides additional details about the selected device, including driver version, technology type, and capabilities related to clipping and rasterization.

Bit-Coded Capabilities (DoBitCodedCaps):

The DoBitCodedCaps function handles bit-coded capabilities for curves, lines, polygons, and text. It interprets bit flags to determine whether specific capabilities are supported by the device.

Structures (BITS and bitinfo):

The program defines structures (BITS and bitinfo) to organize and store information about various capabilities, making the code more readable and maintainable.

Memory Management (Dynamic Allocation):

The program dynamically allocates memory to store information about printers using malloc and frees the memory using free when it is no longer needed.

Printing Properties (IDM_DEVMODE):

The program allows the user to view printer properties by selecting the "Properties" menu item. It calls PrinterProperties with the handle to the selected printer.

Cleanup (WM_DESTROY):

In the WM_DESTROY case, the program posts a quit message to terminate the application. Before exiting, it releases resources and cleans up by calling PostQuitMessage and returning 0.

Overall, the DEVCAPS2 program combines a well-organized structure with Windows API calls to create a graphical interface that efficiently presents detailed information about the capabilities of the selected display or printer.

The modular design allows for easy expansion or modification of functionality related to device capabilities on the Windows platform.

PRINTER PROPERTIES DEEP DIVE

The "Properties" option in DEVCAPS2's Device menu invokes a dialog box generated by the chosen printer driver. This dialog allows users to configure settings like paper size and orientation.

Dialog Origin:

- The dialog box is not part of DEVCAPS2 itself.
- It is triggered by the printer driver's ExtDeviceMode function.
- This function allows users to configure various settings, with at least paper size being a standard option.
- Most drivers also offer portrait and landscape orientation options.
- Selecting landscape mode swaps the horizontal and vertical size/resolution values reported by GetDeviceCaps.
- Color plotters might have more extensive property sheets, allowing configuration of pen colors and paper type.

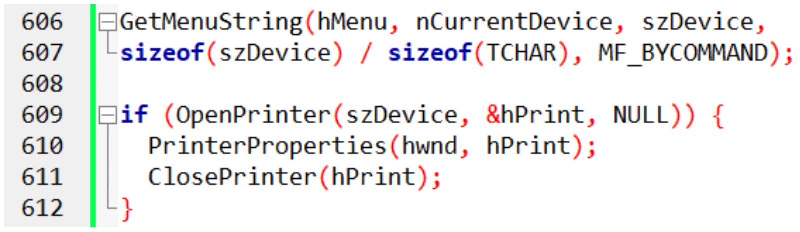
Configuration Persistence:

- Some printer drivers store user-configured settings in their own Registry section.
- This allows access to saved settings during subsequent Windows sessions.
- Programs like DEVCAPS2 can only access settings saved by the driver.

Program Interaction:

- Most programs use the PrintDlg function for printer selection and property configuration.
- This function handles user interaction and settings changes seamlessly.
- While directly calling the driver's ExtDeviceMode or ExtDeveModePropSheet functions is possible, it's not recommended.
- DEVCAPS2 demonstrates the preferred approach using PrinterProperties:
- This function requires a printer object handle obtained through OpenPrinter.
- Upon displaying the properties dialog, DEVCAPS2 uses ClosePrinter to release the handle.

Code Breakdown:



```
GetMenuString(hMenu, nCurrentDevice, szDevice,
sizeof(szDevice) / sizeof(TCHAR), MF_BYCOMMAND);

if (OpenPrinter(szDevice, &hPrint, NULL)) {
  PrinterProperties(hwnd, hPrint);
  ClosePrinter(hPrint);
}
```

- Retrieves the selected printer name from the menu and stores it in szDevice.
- Opens the printer using OpenPrinter and assigns the handle to hPrint.
- If opening succeeds, displays the property sheet with PrinterProperties.
- Finally, closes the printer handle using ClosePrinter.

CHECKING FOR BITBLT CAPABILITY: A DEEP DIVE

While GetDeviceCaps provides valuable information about the printer's printable area, resolution, and other capabilities, GDI often handles limitations by simulating functionalities. However, one crucial capability applications should explicitly check is bit-block transfer.

BitBlt Capability and Its Impact:

Bit-block transfer capability is determined by the RC_BITBLT bit in the RASTERCAPS value returned by GetDeviceCaps.

Most printers (dot-matrix, laser, ink-jet) support bit-block transfers, but plotters do not.

Devices lacking this capability cannot utilize these GDI functions:

Bitmap Creation and Manipulation:

- CreateCompatibleDC
- CreateCompatibleBitmap

Pattern and Bit-Block Operations:

- PatBlt
- BitBlt
- StretchBlt

Text and Icon Drawing:

- GrayString
- DrawIcon

Pixel Access:

- SetPixel
- GetPixel

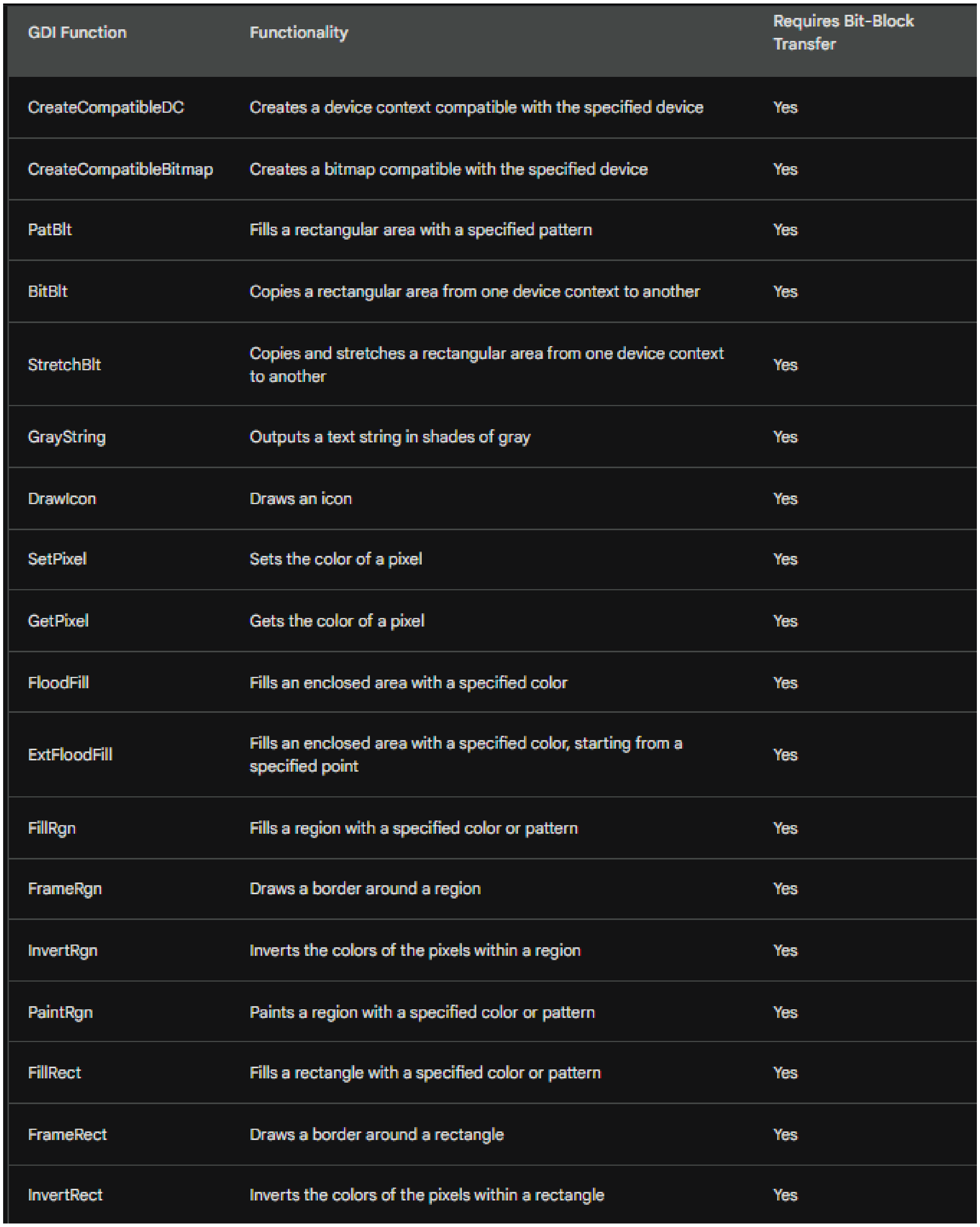
Region Filling and Operations:

- FloodFill
- ExtFloodFill
- FillRgn
- FrameRgn
- InvertRgn
- PaintRgn

Rectangle Operations:

- FillRect
- FrameRect
- InvertRect

Table Summary:

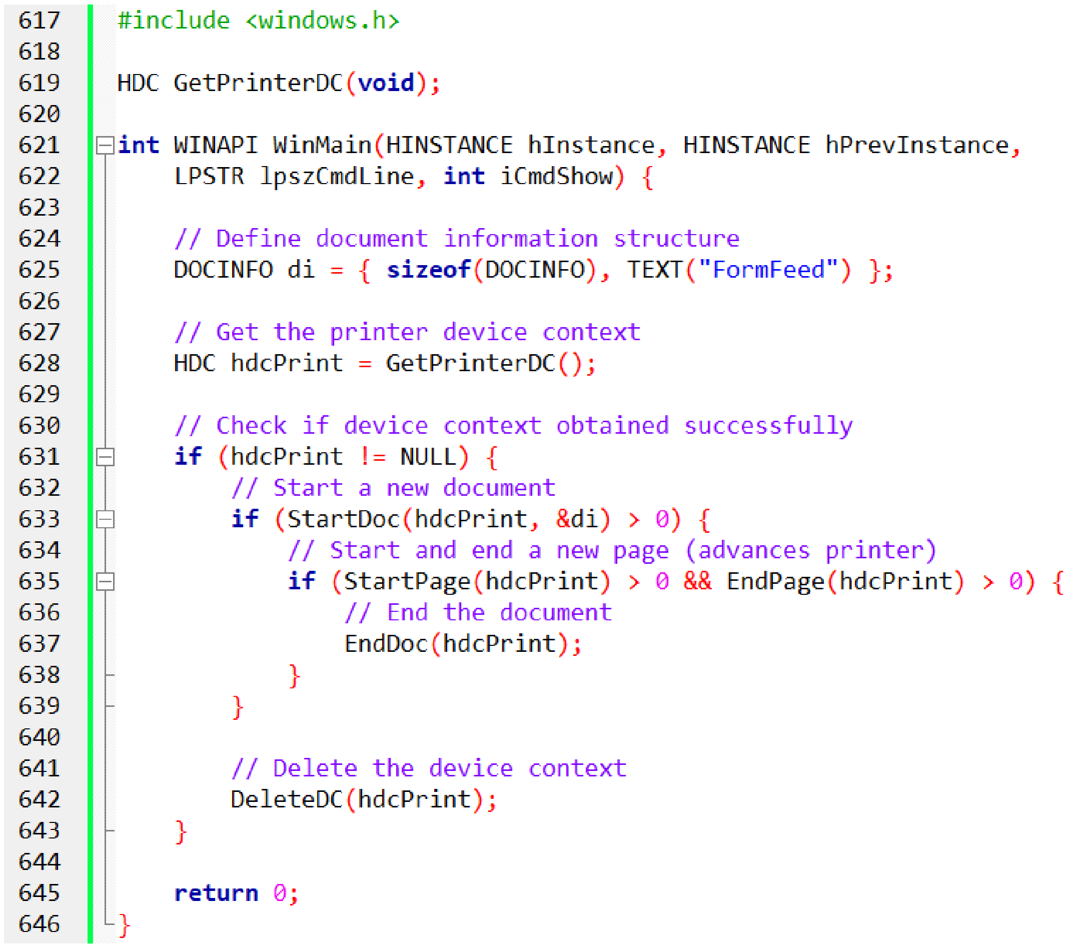


GDI Function	Functionality	Requires Bit-Block Transfer
CreateCompatibleDC	Creates a device context compatible with the specified device	Yes
CreateCompatibleBitmap	Creates a bitmap compatible with the specified device	Yes
PatBlt	Fills a rectangular area with a specified pattern	Yes
BitBlt	Copies a rectangular 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;
}
```

Key Points:

- GetPrinterDC: This function retrieves the printer's device context, allowing the program to interact with the printer.
- StartDoc: This function starts a new printing document and takes the DOCINFO structure as an argument.
- The DOCINFO structure includes the size of the structure and a descriptive text string ("FormFeed" in this case).
- This string appears in the printer's job queue during printing.
- StartPage/EndPage: These functions respectively start and end a printing page. Calling them consecutively advances the printer to a new page.
- Error Checking: The program checks the return values of each function to ensure successful execution.
- A positive return value indicates success, while an error code signifies an issue.
- GDI automatically aborts the document if an error occurs.
- To report specific errors to the user, call GetLastError.

Alternative Approaches:

- While tempting, directly accessing the printer port and writing ASCII code 12 for form feed is discouraged.
- Determining the appropriate port and ensuring exclusivity is complex.
- Different printers interpret form feed commands differently.
- Windows provides dedicated printing functions for reliable and robust operation.

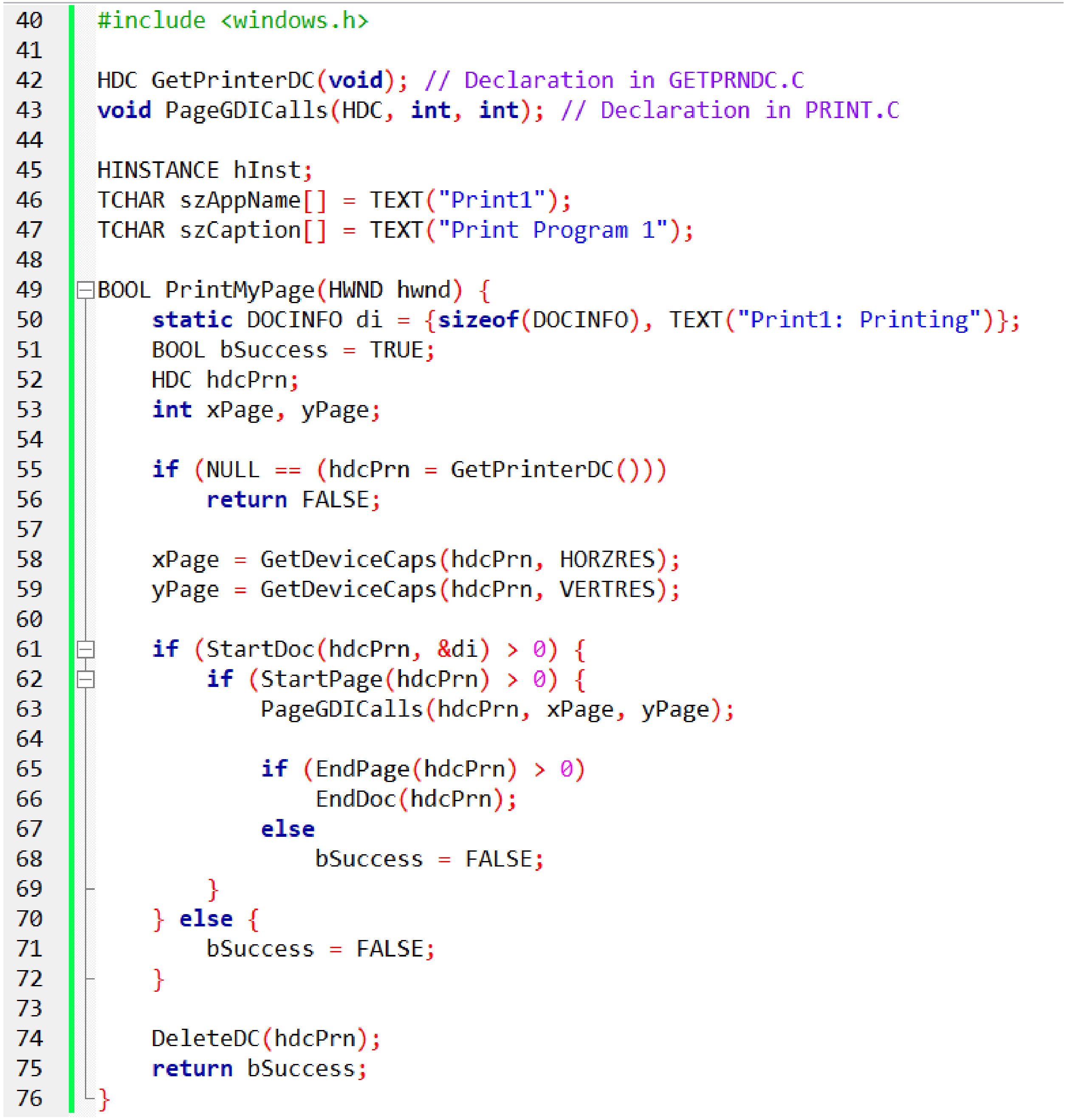
Overall:

FORMFEED.C demonstrates the fundamental principles of printing with Windows API functions. It provides a basic framework for understanding how to start documents, pages, and handle errors. This foundation can be used to build more complex printing applications.

PRINT 1-BareBones printing

The *Print.c program* is a Windows application that demonstrates basic printing functionality. It consists of three versions: PRINT1, PRINT2, and PRINT3. Each version is designed to print one page of text and graphics. The code is structured in a way that allows for reuse of common routines in the PRINT.C file, and it utilizes the GETPRNDC.C file to obtain the printer device context.

Let's break down the key components and functionalities of the program:



```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 Function:

- The WndProc function is the window procedure that handles messages for the main window.
- During the WM_CREATE message, the program modifies the system menu to include a "Print" option.
- The WM_SIZE message updates the client area dimensions when the window is resized.
- The WM_SYSCOMMAND message triggers printing when the "Print" option is selected from the system menu.
- The WM_PAINT message initiates the drawing of graphics on the window using the PageGDICalls function.
- The WM_DESTROY message handles the termination of the program.

PageGDICalls Function:

- The PageGDICalls function is responsible for drawing graphics on the printer device context.
- It draws a rectangle around the entire page, two lines between opposite corners, an ellipse in the middle, and the text "Hello, Printer!" centered on the ellipse.
- Graphics operations like Rectangle, MoveToEx, LineTo, SaveDC, SetMapMode, SetWindowExtEx, SetViewportExtEx, SetViewportOrgEx, Ellipse, SetTextAlign, TextOut, and RestoreDC are used.

PrintMyPage Function:

- The PrintMyPage function is called when the user selects the "Print" option from the system menu.
- It initiates the printing process and returns TRUE if the page is printed successfully, or FALSE if an error occurs.
- In the provided code, PrintMyPage simply calls PageGDICalls to draw graphics on the printer device context. In more advanced versions, this function could handle more complex printing tasks.

WinMain Function:

- The WinMain function is the entry point of the program.
- It registers the window class, creates the main window, and enters the message loop.

The Print1 program unveils its printing power through a well-orchestrated dance of functions and variables. Let's delve into its steps:

1. Setting the Stage:

Header Lineup: Essential Windows header files like windows.h are enlisted to provide the tools for printer interaction.

Function Friends: GetPrinterDC and PageGDICalls are declared, hinting at their crucial roles in the printing process. These functions reside in other modules, waiting for their turn to shine.

Global Introductions: The program introduces itself through hInst, a handle to its unique identity. szAppName proudly announces it as "Print1", while szCaption declares its purpose with "Print Program 1".

2. PrintMyPage: The Master of Ceremonies:

This function takes center stage, orchestrating the entire printing performance.

- Document Identity: A DOCINFO structure is declared, holding information like the document's name ("Print1: Printing"), ready to be shared with the printer.
- Success by Default: Optimistically, bSuccess is set to TRUE, hoping for a flawless printing experience.
- Meeting the Printer: A crucial handshake happens with GetPrinterDC, acquiring a handle to the printer's device context (DC). If this handshake fails, the printing dream fades.
- Sizing Up the Stage: The printer's page dimensions are measured using GetDeviceCaps, revealing the canvas for the upcoming artwork.
- Opening the Curtain: StartDoc initiates the printing process, introducing the DOCINFO to the printer. A negative response here signals an error, dimming the lights on the performance.
- A Fresh Page: With StartPage, a blank canvas is prepared for the magic to unfold. Again, a negative response brings the curtain down.
- The Act Begins: PageGDICalls steps into the spotlight, wielding the printer DC, page width, and page height like paintbrushes. This mysterious function, defined elsewhere, is where the actual content is drawn on the page.
- Wrapping Up the Act: EndPage signals the completion of the on-page masterpiece. If it fails, bSuccess is sadly lowered.
- The Grand Finale: If StartDoc succeeded, EndDoc gracefully closes the document, ensuring a complete performance.
- Releasing the Stage: Finally, the printer DC is released with DeleteDC, allowing others to use it.

3. PageGDICalls: The Hidden Artist:

While shrouded in mystery, this function receives the baton from PrintMyPage, armed with the printer DC, page dimensions, and its own artistic vision. It uses these tools to paint the desired content onto the page, bringing the document to life.

The Final Curtain:

With these steps, Print1 paints a picture on the chosen paper, proving its mastery of the printing art. Remember, this breakdown focuses on the orchestration of the process, leaving the specific artistic details of PageGDICalls to its own private performance.

Overall, the program showcases basic Windows graphics programming and introduces printing capabilities. The structure allows for incremental enhancements in the subsequent versions (PRINT2 and PRINT3) to introduce more advanced printing features.

DEEP DIVE INTO PRINT1 PRINTING AND ABORT PROCEDURE

Obtaining Device Context and Page Dimensions:

If PrintMyPage fails to get a printer device context (DC) handle, WndProc displays an error message.

If successful, it retrieves the page size (horizontal and vertical pixels) using GetDeviceCaps on HORZRES and VERTRES.

This value represents the printable area, not the full paper size.

PRINT1 Printing Logic:

Structurally similar to FORMFEED, but PageGDICalls is called between StartPage and EndPage.

EndDoc is only called if all previous calls (StartDoc, StartPage, EndPage) are successful.

Canceling Printing with Abort Procedure:

Large documents need a way to cancel printing during the process.

An "abort procedure" allows the program to control printing progress.

This function, named AbortProc, takes two arguments:

- hdcPrn: The printer device context handle.
- iCode: Indicates the reason for the call (0 for success, SP_OUTOFDISK for disk space issues).

Implementing Abort Procedure:

Register the procedure: Call SetAbortProc(hdcPrn, AbortProc) before StartDoc.

Define AbortProc:

```
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:

During processing of EndPage, GDI calls AbortProc repeatedly.

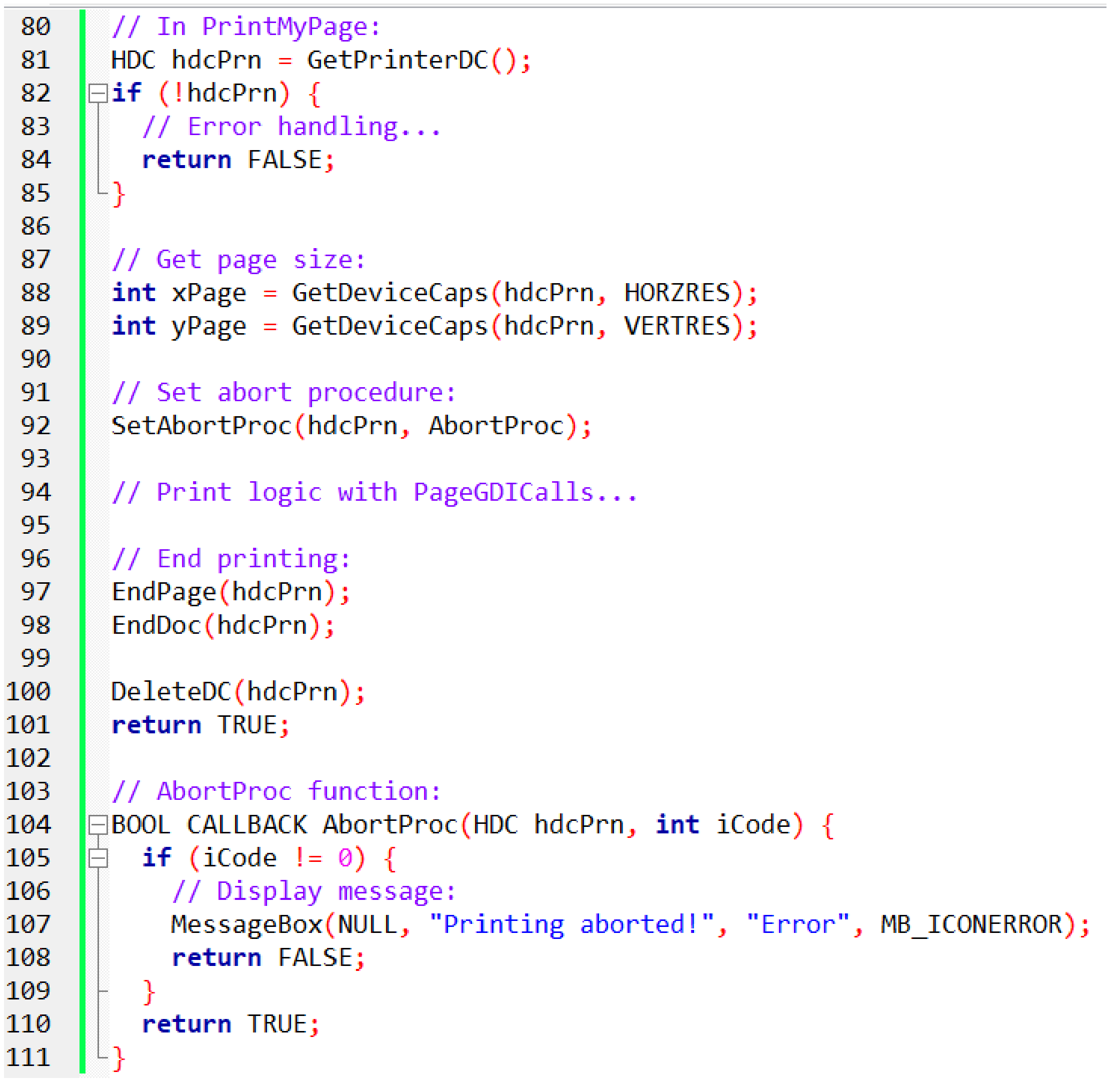
This allows the program to check and potentially abort the print job.

Keeping the Message Loop Going:

The provided example uses PeekMessage instead of GetMessage.

- PeekMessage retrieves messages without blocking, allowing user input to interrupt printing.
- The loop continues until no more messages are available, then returns control to Windows.

Code Examples:



```c
// 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;
}
```

Key Takeaways:

- Abort procedures allow interactive control over printing.
- PeekMessage enables responsive cancellation based on user input.
- Implementing abort logic adds flexibility and user-friendliness to printing applications.

This breakdown provides a deeper understanding of PRINT1's printing logic and the integration of an abort procedure for user control. Remember, these are just examples, and you can adapt them to your specific needs and desired functionality.

DEEP DIVE INTO WINDOWS' USE OF ABORTPROC AND PRINTING LOGIC:

Behind the Scenes of Printing:

The meat of printing happens during EndPage. Before that, GDI builds a metafile on disk, recording each drawing call.

On EndPage, GDI "plays" this metafile for each printer band, creating printer output stored in a file.

If the spooler isn't running, GDI directly sends this output to the printer.

The Role of AbortProc:

During EndPage, GDI calls your program's AbortProc (usually with iCode=0).

If GDI runs out of disk space (due to other temporary files), iCode=SP_OUTOFDISK.

AbortProc enters a loop to check its message queue using PeekMessage.

If no messages exist, PeekMessage returns FALSE, AbortProc exits the loop, and returns TRUE to GDI, allowing printing to continue.

Why AbortProc Exists:

GDI handles errors automatically, so AbortProc's primary purpose is user cancellation.

This requires a separate dialog box with a Cancel button (implemented in PRINT3).

Let's dissect this process step by step:

1. GDI Building the Metafile:

Between StartDoc and EndPage, every GDI drawing function adds a record to the metafile.

This file stores the sequence of drawing instructions for the page.

2. Playing the Metafile and Creating Printer Output:

On EndPage, GDI processes the metafile, translating each instruction into specific printer commands.

These commands are sent to the printer driver, which interprets them and generates the actual page image.

The printer driver also creates a temporary file containing the page data.

3. Disk Space and AbortProc:

If GDI runs out of space for temporary files due to other pending print jobs, it sets iCode=SP_OUTOFDISK when calling AbortProc.

This allows your program to react to this situation, potentially displaying a warning or canceling the print job.

4. AbortProc's Message Loop and Canceling Printing:

AbortProc enters a loop, repeatedly calling PeekMessage to check for messages without blocking.

This enables your program to respond to user input like clicking a Cancel button in a dialog box.

If a Cancel message is received, AbortProc can return FALSE to GDI, signaling to stop the printing process.

5. GDI Handling Errors and Cleanup:

GDI automatically handles internal errors and cleans up temporary files when necessary.

Your program's primary role in error handling is through AbortProc and user interactions.

Remember:

- AbortProc provides a way to interact with the printing process during EndPage.
- Effective cancellation requires a separate user interface element like a Cancel button in a dialog box.
- PRINT2 will demonstrate adding AbortProc, and PRINT3 will integrate the dialog box for user-driven cancellation.

This in-depth explanation clarifies the intricate interplay between GDI, AbortProc, and printing progress. By understanding these components, you can build printing applications with more control and user-friendly cancellation options.

DIVING DEEPER INTO ABORTPROC IMPLEMENTATION AND ITS PITFALLS:

A Quick Recap:

AbortProc allows you to control printing by checking for messages and potentially stopping it.

You define it as a function taking hdcPrn (printer DC) and iCode (reason for call) as arguments.

You register it with SetAbortProc(hdcPrn, AbortProc) before StartDoc.

The PeekMessage Trap:

While convenient, the PeekMessage loop in AbortProc has a major issue.

Since it runs during printing, user interactions can cause unexpected behavior.

Re-printing, loading new files, or even quitting the program can lead to crashes.

Disabling Window Input:

To prevent these issues, you need to disable your program's window with EnableWindow(hwnd, FALSE) before SetAbortProc.

This blocks keyboard and mouse input, keeping the user from interfering during printing.

After printing finishes, re-enable the window with EnableWindow(hwnd, TRUE) for normal interaction.

Mystery of TranslateMessage and DispatchMessage:

You might wonder why these functions are used if no input messages arrive.

While TranslateMessage isn't essential, DispatchMessage is crucial for handling WM_PAINT messages.

Without proper handling with BeginPaint and EndPaint, these messages can clog the queue and prevent AbortProc from finishing.

User Experience during Printing:

With the window disabled, your program becomes inactive on the screen.

Users can switch to other applications while the spooler sends output to the printer.

PRINT2: IMPLEMENTING ABORTPROC WITH SAFETY MEASURES:

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

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

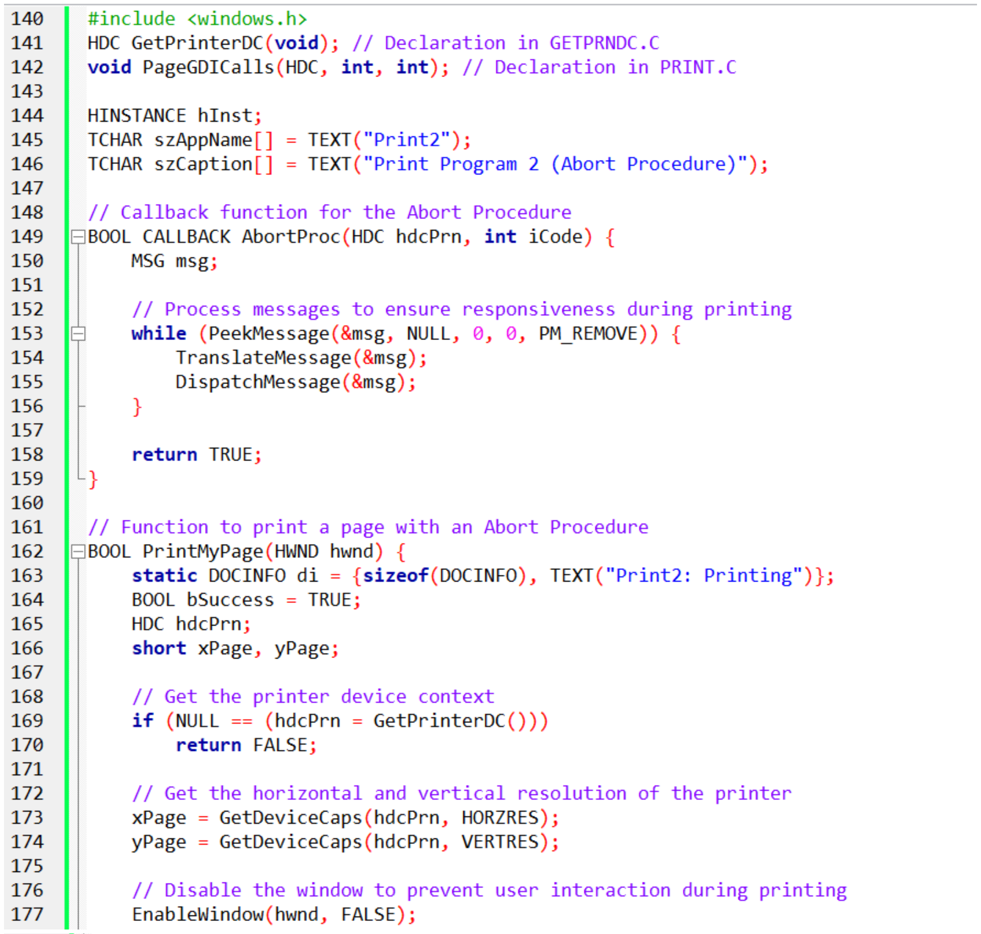
Code Example:

```
// 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);
```

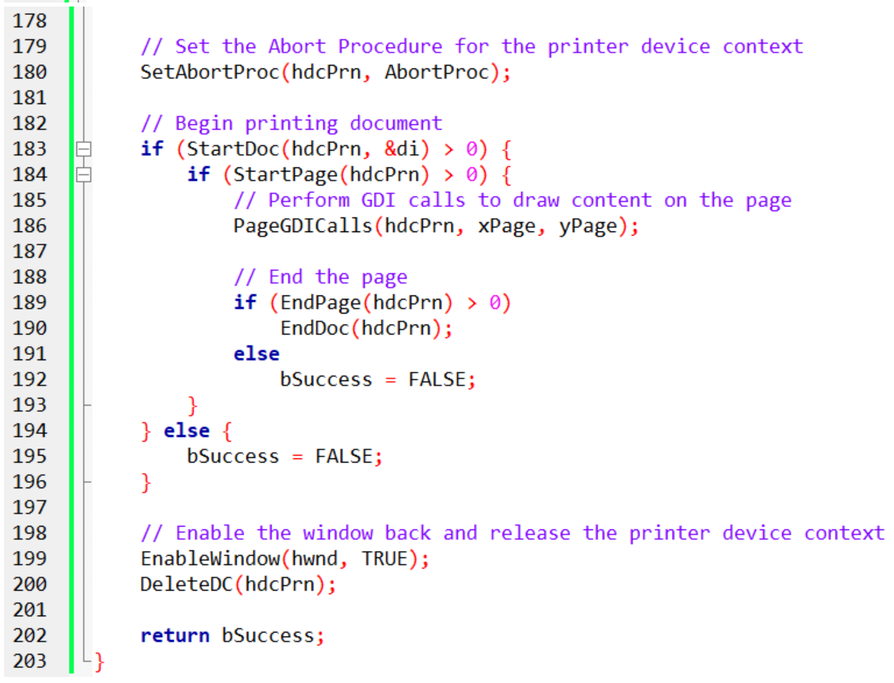
Takeaways:

- Using AbortProc effectively requires careful consideration of user interaction and potential errors.
- Disabling the window during printing ensures smooth operation and prevents unpredictable behavior.
- By understanding these nuances, you can build robust printing applications with user-friendly cancellation options.

Remember, this is just a detailed breakdown of the underlying concepts. The specific implementation and error handling will depend on your program's unique needs and desired functionality.



```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: BUILDING UPON PRINT1 WITH AN ABORT PROCEDURE

PRINT2 takes the foundation laid by PRINT1 and introduces a crucial feature: user control over printing through an abort procedure. This allows the user to cancel the printing process while it's still ongoing, preventing wasted paper and time. Here's a breakdown of what PRINT2 adds:

1. Introducing AbortProc:

Defines a BOOL CALLBACK AbortProc function that receives the printer DC and a code indicating the reason for the call.

Inside the function, a PeekMessage loop checks for messages in the program's message queue without blocking.

If a message is found (e.g., from clicking a Cancel button), it's translated and dispatched, potentially leading to aborting the print job.

2. Implementing Window Disable/Enable:

Before setting the abort procedure, EnableWindow(hwnd, FALSE) disables the program's window, preventing further user interaction.

This ensures the user can't accidentally trigger other actions or interfere with the printing process.

After printing finishes, EnableWindow(hwnd, TRUE) re-enables the window for normal interaction.

3. Integrating with PrintMyPage:

The PrintMyPage function now calls SetAbortProc(hdcPrn, AbortProc) before starting the document.

This registers the AbortProc function with the printer DC, making it available for potential cancellation calls.

The rest of the printing logic remains largely unchanged from PRINT1, utilizing the provided PageGDICalls function.

Benefits of AbortProc in PRINT2:

- Users can cancel long or unwanted printing jobs mid-process.
- This avoids wasting paper and printer resources.
- Enhances user experience and control over the printing process.

Key Differences from PRINT1:

- Addition of AbortProc function and its integration into PrintMyPage.
- Implementation of window disabling/enabling to prevent user interference during printing.
- No major changes to the core printing logic compared to PRINT1.

Overall, PRINT2 demonstrates how to add user-driven cancellation to a printing application through an abort procedure. This provides a more flexible and interactive printing experience, catering to the user's needs and minimizing wasted resources.

Further Exploration:

- PRINT3 builds upon PRINT2 by introducing a dedicated dialog box with a Cancel button for a more user-friendly cancellation experience.
- Consider exploring the details of PeekMessage and message handling for a deeper understanding of how AbortProc interacts with user input.

Let’s now deal with print3 program….

```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 can't tell if printing is ongoing or finished unless they interact with the program.
- No cancellation option: Users can't stop the printing process while it spools.

Introducing the Printing Dialog Box:

PRINT3 addresses these issues by adding a modeless dialog box with a Cancel button.

This dialog appears during printing, allowing users to monitor progress and cancel if needed.

Two Crucial Functions:

AbortProc: This function remains similar to PRINT2, but its PeekMessage loop sends messages to the dialog box window procedure.

PrintDlgProc: This new function handles the dialog box's WM_COMMAND messages, checking the Cancel button status.

Interaction and Cancellation:

If the user clicks Cancel, PrintDlgProc sets a global variable bUserAbort to TRUE.

AbortProc checks this variable and returns FALSE (aborting printing) if bUserAbort is TRUE, otherwise it returns TRUE to continue.

Implementation in PRINT3:

PRINT3 integrates these functions and dialog logic to provide a user-friendly printing experience.

It displays the dialog during the printing process, allowing users to monitor and potentially cancel.

Benefits of the Dialog Box:

Enhances user experience by providing feedback and control over printing.

Improves user satisfaction by allowing cancellation of unwanted or lengthy print jobs.

Makes your application feel more professional and user-friendly.

Key Differences from PRINT2:

- Addition of the printing dialog box with a Cancel button.
- Implementation of the PrintDlgProc function to handle dialog interactions.
- Modification of AbortProc to check for user cancellation through bUserAbort.

Overall, PRINT3 takes a significant step towards a more robust and user-friendly printing experience by incorporating a modeless printing dialog box with cancellation functionality.

This demonstrates how to build Windows-like printing applications with improved user control and feedback.

Further Exploration:

- Analyze the specific implementation details of PrintDlgProc and its interaction with the dialog box.
- Consider customizing the dialog box content and appearance to better suit your application's needs.
- Explore advanced printing features and options available in Windows APIs.

```
// 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. By referencing these IDs in your code, you can interact with the dialog box elements like the Cancel button and static text.

DEEP DIVE INTO PRINT3: PRINTING WITH USER CONTROL AND CANCELLATION

PRINT3 builds upon PRINT2 by introducing crucial features for user interaction and control over printing:

- Printing Dialog Box: A modeless dialog box with a "Cancel" button appears during printing, providing visual feedback and cancellation options.
- User-Driven Cancellation: Clicking the Cancel button sets a global variable bUserAbort to TRUE, signaling AbortProc to stop printing.
- Improved User Experience: Users can monitor the printing process and intervene if needed, offering greater flexibility and satisfaction.

Key Implementation Details:

Two Global Variables:

- bUserAbort: Boolean flag indicating user-initiated cancellation (initially False).
- hDlgPrint: Handle to the printing dialog box window.

PrintMyPage Function:

- Initializes bUserAbort to False.
- Disables the program's main window.
- Sets AbortProc and PrintDlgProc pointers for communication with GDI and the dialog box.
- Creates the printing dialog box with CreateDialog and stores its handle in hDlgPrint.

AbortProc's Message Loop:

- Checks bUserAbort before using PeekMessage to avoid unnecessary processing.
- Uses IsDialogMessage to send retrieved messages to the dialog box window procedure.
- Returns the inverse of bUserAbort: True continues printing, False indicates cancellation.

PrintDlgProc Function:

- Sets the dialog box caption and disables the Close option in the system menu during WM_INITDIALOG.
- Sets bUserAbort to True and handles cleanup (enable main window, destroy dialog box) on WM_COMMAND for the Cancel button.

EndPage Handling and Cancellation:

- AbortProc checks the return value from EndPage.
- If AbortProc returns False (due to bUserAbort), control returns to PrintMyPage.
- PrintMyPage calls EndDoc, effectively stopping the printing process.

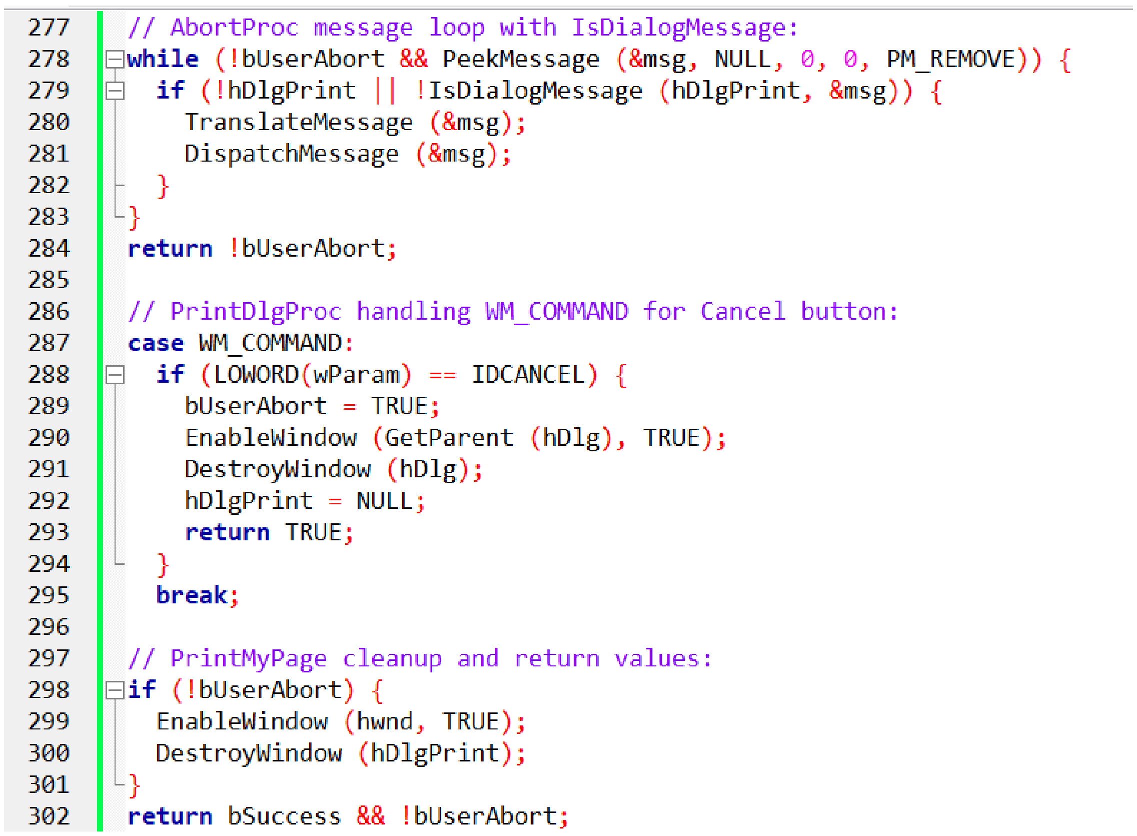
Cleanup and Completion:

- If not canceled, PrintMyPage reenables its window and destroys the dialog box.
- PrintMyPage returns a combination of bSuccess (indicating printing errors) and !bUserAbort (cancellation status).

Key Takeaways:

- PRINT3 demonstrates a user-friendly printing experience with a cancelable dialog box.
- The interplay between bUserAbort, AbortProc, PrintDlgProc, and EndPage controls the printing process and cancellation.
- Understanding these details allows you to build robust and interactive printing applications in Windows.

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. Adapting and expanding upon these concepts will allow you to create custom printing experiences tailored to your specific application and user needs.

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

POPPRNT.C program plays a vital role in transforming POPPAD from a simple text editor into a versatile document creation tool. It seamlessly integrates printing functionality, empowering users to preserve their work on paper. Let's dissect its key components:

1. The Global Stage:

bUserAbort: This boolean flag acts as a sentinel, instantly halting the printing process when users hit the "Cancel" button. It starts as False, granting full control, and flips to True when the user intervenes.

hDlgPrint: This handle serves as a bridge between the main program and the printing dialog box. It allows the program to manipulate the dialog's appearance and behavior, ensuring a smooth user experience.

2. The Dialog Box Maestro:

PrintDlgProc: This function acts as the brain behind the printing dialog box. It listens for various messages:

WM_INITDIALOG: When the dialog box first appears, it disables the "Close" option, preventing accidental dismissal and ensuring the user has full control over the printing process.

WM_COMMAND: When the user clicks "Cancel," the function springs into action: it sets bUserAbort to True, signaling immediate termination of printing. It also cleans up by enabling the main window and destroying the dialog box, ensuring a seamless transition back to editing.

3. The Printing Orchestrator:

PopPrntPrintFile: This function serves as the conductor of the printing symphony. It takes the user's edited text and transforms it into a physical document:

Setting the Stage: The function gathers essential information like total lines, page capacity, and character width to determine printing layout and efficiency. It also allocates memory for a temporary buffer to hold each line of text.

The User's Cue: A "Print" common dialog box appears, allowing users to choose their printer and configure options. This empowers them to personalize their printing experience.

The Printing Loop: Once the user confirms, the function embarks on a meticulous journey:

- Page by Page: It iterates through each page, ensuring proper collating if selected.
- Line by Line: Within each page, it meticulously reads each line from the edit control, storing it in the temporary buffer.

The Ink Touches Paper: Utilizing TextOut, the function meticulously paints each line onto the virtual page, gradually forming the document.

The Grand Finale: If the user didn't interrupt or an error didn't occur, the function gracefully concludes the printing process with EndDoc. It then cleans up resources, frees memory, and returns control to the user.

Beyond the Basics:

AbortProc: This function acts as a vigilant watchdog, constantly checking for user cancellation. It filters messages to ensure only relevant ones reach the dialog box, preventing unnecessary processing during printing.

Customization: POPPRNT.C provides a solid foundation for further customization. The printing dialog box can be enhanced with additional options, the layout can be fine-tuned, and error handling can be expanded.

POPPRNT.C improves POPPAD to transcend its digital boundaries and share its creations with the physical world. By understanding its intricate workings, you can unlock the potential for more advanced printing capabilities in your own applications.

Remember, this is just a starting point. As you delve deeper into the code, you'll discover even more nuances and possibilities for enhancing the printing experience. So, grab your curiosity and explore the fascinating world of POPPRNT.C!

UNVEILING THE SECRETS OF POPPAD'S PRINTING ENGINE: A DEEP DIVE INTO POPPRNT.C

POPPRNT.C stands as a testament to the power and simplicity of leveraging high-level Windows features. It seamlessly integrates printing functionality into POPPAD, allowing users to transform their digital creations into tangible documents. Let's delve deeper into its key aspects:

Unlocking the PrintDlg Magic:

PrintDlg: This function, residing in the common dialog box library, takes the driver's seat in initiating printing. POPPAD utilizes it to display a familiar dialog box, empowering users to choose page ranges, specify copies, and even toggle collation.

User-Centric Features: This dialog box goes beyond mere page selection. It offers options for multiple copies, collating order (stacked or interleaved), printer selection, and even device mode configuration (portrait or landscape).

Decoding the Return Values: Upon returning from PrintDlg, the PRINTDLG structure holds valuable information: the desired page range, collation preference, and even the printer device context handle – ready to be used for painting text onto the page.

PopPrntPrintFile: The Orchestrator of Printing:

Setting the Stage: This function, called upon user's "Print" selection, performs crucial calculations. It leverages GetDeviceCaps and GetTextMetrics to determine page resolution and character dimensions, allowing it to fit lines and pages efficiently.

Line by Line, Page by Page: POPPAD navigates the document meticulously. It sends EM_GETLINECOUNT to the edit control to determine the total lines, allocates memory for line buffers, and then iterates through each line.

The Ink Touches Paper: Using EM_GETLINE, POPPAD retrieves each line and stores it in the buffer. Then, with a flourish of TextOut, it paints the line onto the virtual page, gradually building the document.

Collation: A Matter of Order:

Two Loops for Precision: POPPAD handles both collated and non-collated printing. Two nested loops, controlled by iColCopy and iNonColCopy, ensure the copies are printed in the correct order, whether stacked or interleaved.

User Abort,Taking Control: Throughout the process, POPPAD constantly checks the bUserAbort flag. If set to True, indicating user cancellation, the printing immediately stops.

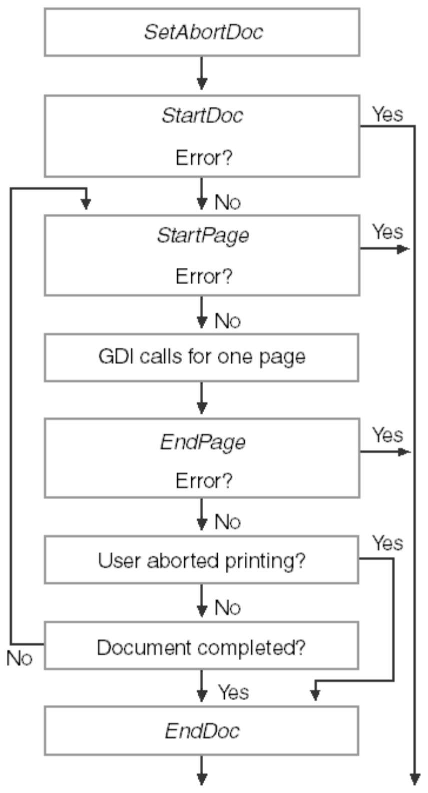
Beyond the Basics:

EndDoc: The Grand Finale: Upon successful printing, POPPAD gracefully concludes with EndDoc, ensuring a clean exit.

Error Handling: No Show, No Need: Interestingly, POPPAD avoids the AbortDoc function altogether. The user's control through the dialog box and print job window, combined with the prompt EndPage error checks, render AbortDoc unnecessary.

The Sequence of Calls: A Choreographed Dance:

Figure 13-11 illustrates the optimal sequence of printing functions for multi-page documents. Each EndPage call serves as a checkpoint, offering the perfect opportunity to check for user intervention via bUserAbort.



Remember, once an error occurs during printing, it's best to gracefully exit without relying on further functions like AbortDoc.

The image you sent me is a flowchart of the printing process in a Windows GUI application using WinAPI. It shows the different steps involved in printing a document, from the initial call to the StartDoc function to the final call to the EndDoc function.

Here's a breakdown of the steps involved:

StartDoc: The application calls the StartDoc function to initialize the printing process. This function sets up the printing context and tells the spooler that a new document is about to be printed.

StartPage: Before printing any content, the application calls the StartPage function to start a new page. This function allocates memory for the page and sets up the origin for drawing.

GDI calls for one page: The application then uses the GDI (Graphics Device Interface) to draw the content on the page. This can include text, images, lines, and other graphical elements.

EndPage: After all the content for the page has been drawn, the application calls the EndPage function to signal the end of the page. This function releases the memory allocated for the page and prepares for the next page.

User aborted printing?: After each page is printed, the application checks to see if the user has aborted the printing process. If the user has aborted printing, the application calls the AbortDoc function to clean up and exit the printing process.

Document completed?: After all the pages have been printed, the application checks to see if the document is complete. If the document is complete, the application calls the EndDoc function to signal the end of the printing process. This function cleans up any remaining resources and closes the printing context.

The flowchart also shows some error handling steps. If an error occurs during any of the steps, the application can handle the error and continue printing, or it can abort the printing process.

Here are some additional things to keep in mind about printing in WinAPI GUI C:

- The WinAPI provides a variety of functions for printing text, images, and other graphical elements.
- The application must handle all of the memory allocation and deallocation for printing.
- The application can use the AbortDoc function to cancel the printing process at any time.
- The application can use the SetAbortProc function to specify a callback function that will be called if the user aborts the printing process.

In Conclusion:

POPPRNT.C showcases how high-level Windows features can empower applications like POPPAD to offer user-friendly printing capabilities.