Adobe Developers Association

ADA news

In This Issue

Volume 4, Number 4

Interfacing with PostScript Printer Drivers on Multiple Platforms

There are at least four different Microsoft® Windows® platforms for which you can develop applications that output PostScript™ language code—Windows 3.x, Windows 95, Windows NT™ 3.1, and Windows NT 3.5. To address the differences in the support of PostScript printer escapes on each of these platforms, this article will discuss how you can develop platform-independent code for selected printer escapes.

Background

Today, your 16-bit or 32-bit application can run on Windows 3.x or in a Win32 environment, using special mapping layers. Diagram 1 depicts a 2x2 matrix that shows how your application can run under the various Operating Systems (OS). This is made possible through special interfaces.

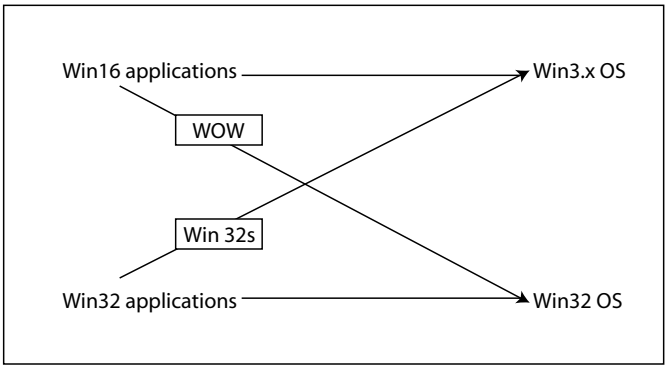


Diagram 1

A Win16 application can run in its native format under Windows 3.x, but needs a Win32 (WOW) Generic Thunking interface when running under a Win32 environment. This layer provides your application capabilities to make GDI calls into the Win32 dynamic-link libraries (DLL). Similarly, it is possible for your Win32 application to make 32-bit calls into the Windows 3.x environment using an interface called the "Win32s mapping layer".

The Win32s and WOW mapping layers are responsible for all graphics and windowing operations. Win32s relies on the existing Windows 3.x printer drivers for output and does not support the loading of 32-bit Windows NT printer drivers. Win32s supports all Windows 3.x escapes except those marked as obsolete in the Windows 3.1 SDK. Detail of the Win32s and WOW interfaces are available on the Win32 SDK CD-ROM from Microsoft.

continued on page 2

Adobe

How To Reach Us

DEVELOPERS ASSOCIATION HOTLINE:

U.S. and Canada:
(415) 961-4111
M–F, 8 a.m.–5 p.m., PDT.
If all engineers are unavailable, please leave a detailed message with your developer number, name, and telephone number, and we will get back to you within 24 hours.

Europe:
+31-20-6511-275

FAX:

U.S. and Canada:
(415) 967-9231
Attention:
Adobe Developers Association

Europe:
+31-20-6511-275
Attention:
Adobe Developers Association

EMAIL:

U.S.
devsup-person@mv.us.adobe.com

Europe:
eurosupport@adobe.com

MAIL:

U.S. and Canada:
Adobe Developers Association
Adobe Systems Incorporated
1585 Charleston Road
P.O. Box 7900
Mt. View, CA 94039-7900

Europe:
European Engineering
Support Group
Adobe Systems Benelux B.V.
P.O. Box 22750
1100 DG Amsterdam,
The Netherlands

Send all inquiries, letters and address changes to the appropriate address above.

PostScript Printer Drivers on Multiple Platforms

Supporting PostScript language on the various Windows platforms

Windows 3.1 supports 64 printer escapes, but Windows NT (3.1 and 3.5) only supports a small set of fundamental escapes (11 escapes). Although it is always a good practice to query for the presence of an escape, using the QUERYESCSUPPORT escape, it is more critical for your Adobe PostScript-language-centric application to do so now. The escapes that you have relied on may not be supported on certain Windows platforms.

Your challenge is to develop or modify your existing code so that it can run on any of the Windows platforms mentioned above. This may mean redesigning your escape-query logic and/or carefully assessing a return value before proceeding along a PostScript language printing path. To avoid printing problems, your application must now begin by testing all escapes before using them. But what if the escapes that you have relied on are not available? Your fallback is to proceed along a non-PostScript language path by generating equivalent GDI calls for your PostScript language output.

The rest of this article will demonstrate sample code and logic that you can apply to selected escapes in order to ensure successful support for PostScript language printing. The following sample code has been developed using the Visual C++ v1.0. Sample 1 has been successfully tested on Windows 3.1, WFW 3.1.1, Windows 95 (Beta 1), and Windows NT 3.5.

GETTECHNOLOGY

A Windows application uses the GETTECHNOLOGY escape API to determine the type of printer driver it is sending data to. If the selected print driver is a PostScript printer driver,GETTECHNOLOGY will return "PostScript" on all the Windows platforms.

NOTE: under Windows NT, GETTECHNOLOGY is only supported if your application uses the EXTESCAPE function call.

Hence, in order for your application to achieve portability among the various platforms, you will need to apply the "/D" or "define" switch on your compiler. Sample 1 demonstrates how to achieve support for the GETTECHNOLOGY escape with one set of code on the various Windows platforms.

The PostScript_Supported flag is first set to indicate a FALSE condition. This flag will be used in a later section of your code to verify PostScript language support. We first check to see if the GETTECHNOLOGY escape is supported, by calling the QUERYESCSUPPORT function. If GETTECHNOLOGY is supported, we proceed to check the compiler define switches to determine which escape call to use. Remember, under Windows NT, GETTECHNOLOGY is supported only if you use the EXTESCAPE function. To confirm that we are using a PostScript language printer driver, we will compare the retstr with the string "PostScript." A PostScript language printer driver will return "PostScript" in retstr.

PostScript Printer Drivers on Multiple Platforms

Sample 1

```
// TEST_ESCAPE: This routine checks for the support of the printer escapes
// passed in through the wEscape parameter.
char TEST_ESCAPE(short wEscape)
{
    HDC hdcPrint;
    short retesc;
    char retstr[128], PostScript_Supported=FALSE;
    char *szDevice, *szDriver, *szOutput ;

    // Get the handle to printer's device context.
    // e.g., szDriver = "ADOBEPS.DRV", szDevice = "name of printer", and szOutput ="lpt1:"
    hdcPrint = CreateDC (szDriver, szDevice, szOutput, NULL) ;

    if (hdcPrint == NULL) {
         // Create DC Failed
         // Call your fail routines
    } else {
         // Got a valid hdc.
         // now determine if the GETTECHNOLOGY escape is supported
         wEscape =GETTECHNOLOGY;
         if (Escape(hdcPrint, QUERYESCSUPPORT, sizeof(wEscape),(LPSTR)&wEscape, NULL)>0) {

             // different escape calls for NT to check if we are PostScript language
             // capable
#ifdef WINDOWSNT
             retesc = ExtEscape(hdcPrint, GETTECHNOLOGY, NULL, NULL, &retstr);
#else
             retesc = Escape(hdcPrint, GETTECHNOLOGY, NULL, NULL, &retstr);
#endif
                  if (retesc)
                      if (!strcmp("PostScript", retstr))
                           PostScript_Supported=TRUE;
         }
    }
    return PostScript_Supported;
}
```

PASSTHROUGH and SETGDIXFORM

The PASSTHROUGH escape is supported in Windows 3.x, Windows 95, and Windows NT. With this escape, PostScript language-centric applications can generate their own PostScript language code by telling the print driver to step out of the way.

However, there is a "danger" to having control over GDI, i.e. your application may not know the coordinate systems it is working with. In Windows NT 3.1, the default GDI coordinate system is different from the printing coordinate system, i.e. Windows 3.1 and Windows 95 have default GDI origins that start at the upper left corner with the *y* coordinates going down, while Windows NT 3.1 has default GDI origins at the lower left, with the *y* coordinates going up.

continued on page 4

PostScript Printer Drivers on Multiple Platforms

Because of this difference, it was necessary for PostScript printer drivers to call SETGDIXFORM to set the printing coordinate system to that of the default GDI coordinate system. Luckily, this oversight has been corrected in Windows NT 3.5 and above and the call to SETGDIXFORM is no longer necessary.

Therefore, to ensure that your application will print correctly under the different Windows platforms, look at the pseudo-code in Sample 2. This logic will set your application to print using the default GDI origin.

Sample 2 (pseudo-code)

```
// 1) Use the sample code in Sample 1 to check for PostScript language support.
// 2) Modify the sample code in Sample 1 to check for support of SETGDIXFORM and
//     PASSTHROUGH escapes.

// ensure correct default GDI coordinates in Windows NT 3.1
#ifdef WINDOWSNT3.1
call Escape(..., SETGDIXFORM, ..., ..., ...);
#endif
...
// Now, you are safe with complete control under PASSTHROUGH
call Escape(..., PASSTHROUGH, ..., ..., ...);
```

As described in Sample 1, after you have determined that you are using a PostScript printer driver, and checked for the support of SETGDIXFORM and PASSTHROUGH escapes, you will call SETGDIXFORM escape, if you are running in Windows NT 3.1.

SETGDIXFORM will set your application to print in the default GDI coordinate system. If this step is omitted, your application will print incorrectly with the PostScript printer driver supplied in Windows NT 3.1 but you will not notice any difference in Windows 3.x, Windows 95, or Windows NT 3.5.

NOTE: The call to SETGDIXFORM is a no-op in Windows NT 3.5.

EPSPRINTING

In Windows 3.x and Windows 95, your application can tell the PostScript language print driver to download minimal PostScript language headers using this escape call. The PostScript language print driver in Windows NT has only one set of headers which it will or will not download based on your application's requirements (i.e., if you do not call the StartPage( ) or EndPage( ) function, the Windows NT header will not be downloaded).

NOTE: Although EPSPRINTING is not documented as being supported in Windows NT, applying QUERYESCSUPPORT on this escape under Windows NT 3.5 returns a value indicating it is supported.

PostScript Printer Drivers on Multiple Platforms

The pseudo-code below (Sample 3) shows how you can use the compiler definition to achieve platform independence with one set of code.

Sample 3 (pseudo code)

```
// 1) Use the sample code in Sample 1 to check for PostScript language support.
// 2) Use the pseudo code in Sample 2 to set to default GDI coordinate system for
//     Windows NT 3.1.
// 3) Modify the sample code in Sample 1 to check for support of EPSPRINTING.

// initialize flag
int Escape_Supported=FALSE;

#ifndef WINDOWSNT
// check to see if we have EPSPRINTING capabilities
if (Escape (..., QUERYESCSUPPORT, ..., EPSPRINTING, ...)) {
         //  escape supported, use it
         Escape_Supported=TRUE;
         Escape(..., EPSPRINTING, ..., ..., ...);
}
#endif

If (Escape_Supported || PostScript_Supported) {
#ifndef WINDOWSNT
         // Do not call the StartPage or EndPage function if the Windows NT PostScript
         // language header is not needed.
#endif
         // start printing
         StartDoc(..., ...);
} else {
         // escape not supported, call your alternate routine
}
```

You will use the compiler define switches to determine if you will be using the non-Windows NT pseudo logic, in other words, you will not need to query for the support of the EPSPRINTING escape under Windows NT. Because of this condition, the Escape_Supported flag cannot be used to determine your next step. Hence, you will have to rely on the sample code and the global boolean flag, PostScript_Supported, from Sample 1, to help you decide if you can proceed along the PostScript language path or an alternate printing solution path.

Next, with compiler define switch, you will determine if you want to invoke or omit the PostScript for Windows NT print driver header by calling the StartPage( ) or the EndPage( ) function or staying away from it. Once the platform-dependent checks are done, you can now start printing with the StartDoc function call.

continued on page 6

PostScript Printer Drivers on Multiple Platforms

OPENCHANNEL

The OPENCHANNEL escape will prevent the PostScript language printer driver from downloading a PostScript language header to the printer. NOTE: In Windows NT, the OPENCHANNEL escape is mapped to the StartDoc function call.

Sample 4 (pseudo code)

```
// 1) Use the sample code in Sample 1 to check for PostScript capabilities
// 2) Use the pseudo code in Sample 2 to ensure default GDI coordinates in Windows NT3.1.

// Initialize flag
int Escape_Supported=FALSE;
char retstr[128];

// check for OPENCHANNEL support
if (Escape(..., QUERYESCSUPPORT, ...,OPENCHANNEL, ...) {
         // OPENCHANNEL escape is supported
         Escape_Supported=TRUE;
         Escape(..., OPENCHANNEL, ..., ..., &retstr);
}

if (Escape_Supported) {
#ifndef WINDOWSNT
         // Start print job here. This call is redundant in Windows NT
         StartDoc();
#endif

         // Now do your PASSTHROUGH stuff here
         Escape(..., PASSTHROUGH, ..., ..., ...);

         EndDoc();
} else {
          // escape not supported, call your alternate routine here
}
```

As explained in Sample 1 and Sample 2, you will need to check for PostScript language capabilities and set your printer coordinates to the default GDI coordinates (for Windows NT 3.1 only) before proceeding to other printer escapes. The pseudo logic in Sample 4 will set the Escape_Supported flag to TRUE if the QUERYESCSUPPORT escape confirms a supported OPENCHANNEL escape. Using this flag, you can determine if you can proceed along a PostScript language printing path or go through an alternate printing route. Note: you will use the compiler define switches to determine if you will be using the non-Windows NT pseudo logic, in other words, you will not need to call the StartDoc( ) function under Windows NT.

The Sample 1 code described in this article has been tested on Windows 3.1, Windows For Workgroups 3.1, Windows 95, Windows NT 3.5. The other pseudo-code provided here is intended as a guideline and is by no means a complete solution. §

Developing With

Adobe PageMaker

What is the best method of passing parameters to Adobe™ PageMaker™ plug-in commands and queries?

There are several ways to pass parameters to PageMaker plug-in commands depending on what the parameters are. Adobe includes several macros in the PageMaker plug-in SDK that make it easier for you to move data into and out of the correct fields of the parameter block or data buffer. Most of these macros are located in the *cqparblk.h file*, which you'll find in the "ink" folder. The rest are located in the "*Utils.c*" source file.

The list below groups the macros by function:

• Move data in the binary format in and out of the parameter block when sending commands and queries:

PBBinCommand
PBBinCommandByShortValue
PBBinCommandByLongValue
PBBinQuery
PBBinQueryWithParms
PBGetReplyData

• Move data in the text format in and out of the parameter block when sending commands and queries:

PBGetReplyData
PBTextCommand
PBTextQuery
PBSetReplyUnits
PBSetRequestUnits

An important point to remember when passing parameters to a command or query is that PageMaker software is going to expect those parameters to end on an even byte, including the terminating '/0' for string parameters. If your parameter data does not conform to this alignment rule you can expect that the command or query will not work properly.

Passing binary data

Commands:

If you are passing a single short or long value to a command you can pass the value using PBBinCommandByShortValue and PBBinCommandByLongValue macros respectively. Following are examples of both methods:

```
/**[m******************************************************
* RedrawToggle
* Description:
*   This toggles redraw. When toggled on it will also update any
*       outstanding regions that need to be redrawn.
*
***m]******************************************************
RC RedrawToggle(LPPARAMBLOCK lpParamBlk, short redrawStatus)
{
    RC  rc;

    rc = PBBinCommandByShortValue(lpParamBlk, pm_redraw,
                                        redrawStatus);
    return rc;
}
/**[m******************************************************
* ChangeEnv
* Description:
*   Depending on the parameter, rebuilds one or all of the
*       following:
*       • The font metric information.
*       • The font submenu or popup menu.
*       • The plug-ins submenu.
*
***m]******************************************************
RC ChangeEnv(LPPARAMBLOCK lpParamBlk, long envChange)
{
    RC    rc;

    rc = PBBinCommandByLongValue(lpParamBlk, pm_changeenv,
                                        envChange);
    return rc;
}
```

continued on page 8

If you are passing parameters that consist of data greater that a single short or long value you would use the PBBinCommand macro. There are two possibilities here. If the parameters are numeric you can define a data structure and pass a pointer to it.

This is an excerpt from the header file.

```
    typedef struct {
        long left;
        long top;
        long right;
        long bottom;
    } PMRect;

    PMRect *PMSetRect(PMRect *r,
        PMCoord left, PMCoord top,
        PMCoord right, PMCoord bottom);
```

This is an excerpt from the source file.

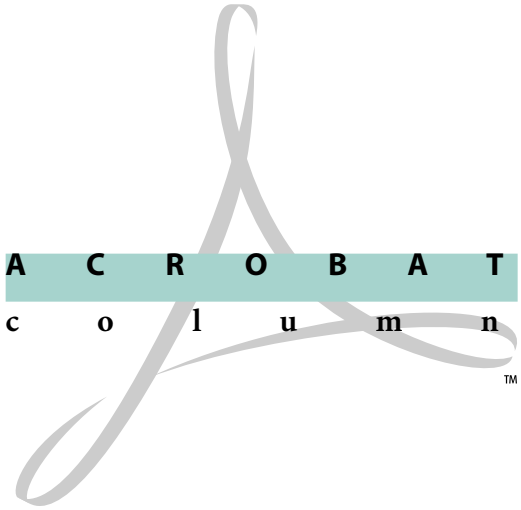
```
RC Box(LPPARAMBLOCK lpParamBlk)
{
    RC       rc;
    PMRect   rBox;

    rc = PBBinCommand(lpParamBlk, pm_box, kRSPointer,
        PMSetRect(&rBox,(2*INCH),(3*INCH),
        (4*INCH),(5*INCH)), sizeof(PMRect));

    return rc;
}

PMRect *PMSetRect(PMRect *r,
    PMCoord left,PMCoord top,PMCoord right,PMCoord bottom)
{
    r->left     = left;
    r->top      = top;
    r->right    = right;
    r->bottom   = bottom;
    return r;
}
```

If the parameters that you are passing contain strings you need to pack the parameters into a buffer and then pass a pointer to that buffer. As noted previously, PageMaker is expecting the parameter data aligns on even bytes. In order to make this easier the SDK provides a macro, LPPutString, that will append the string data and add any extra bytes so that the alignment is correct. The following is an example of parameters containing both numeric and string information.

```
RRC OpenDoc(LPPARAMBLOCK lpParamBlk)
{
    RC       rc = CQ_FAILURE;
    HANDLE   hParms = NULL;
    LPSTR    lpParms, lpOrgPos;
        /* open original document */
    short    cHowToOpen = 0;

        /* allocate 64 bytes of memory for parameters */
    hParms = MMAlloc(64);
    if (hParms)
    {
        /* lock the memory location */
    lpParms = lpOrgPos = MMLock(hParms);

    lpParms += LPPutString(lpParms, "c:\PM6\test.pm6");
    lpParms += LPPutShort(lpParms, cHowToOpen);

    rc = PBBinCommand(lpParamBlk, pm_open, kRSPointer,
                    lpOrgPos, (lpParms - lpOrgPos));

        /* unlock memory location */
    MMUnlock(hParms);
        /* free memory location */
    MMFree(hParms);
    }

    return rc;
}
```

Queries:

Next month we will explore how to pass parameters to queries and how to retrieve the data returned by the queries.



The Adobe Acrobat™ Plug-In Software Development Kit gives developers access to the Adobe Acrobat viewer plug-in Application Programming Interface (API). Using this interface, developers can create plug-ins for Acrobat Exchange with functionality including but not limited to:

• Customizing the Acrobat Exchange user interface.

• Augmenting existing Acrobat Exchange functions.

• Manipulating the contents of a Portable Document Format (PDF) file.

More than one plug-in can be installed simultaneously for use by Acrobat Exchange. Therefore, plug-in developers must be careful that their plug-ins do not conflict with others. Such conflicts can cause plug-ins to fail.

Developers must register a four-character company-specific prefix with the Adobe Developers Association. This prefix must be used within the names of various items. These items include tools, toolbar buttons, menus, menu items, annotation handlers, private data, HFT servers, selection servers, custom actions, and the unique extension name. The prefix for menus, menu items, and tools must be followed by a colon; the prefix used for other items must be followed by an underscore.

For example, the viewer API method that creates new menu items requires a language-independent name string for the menu item to be created. This name is independent of the title that appears in the user interface. To avoid conflicts in the menu item name space, it is important that each menu item name be unique.

The prefix used by Adobe for its own plug-ins is "ADBE". In a plug-in created by Adobe called SuperCrop, for example, the menu item name for the About SuperCrop menu item would be called "`ADBE:AboutSuperCrop`". The unique extension name would be "`ADBE_SuperCrop`".

Contact the Adobe Developers Association to register your company-specific four-character prefix or for more information on using your prefix prior to distributing your plug-in.

Developing With

Adobe Fetch

Adobe Fetch™ is a multimedia database product for the Macintosh® that allows customers to catalog, browse, search, retrieve, and reuse graphic, movie, and sound files.

Macintosh Easy Open and Adobe Fetch software

In the *ADA News*, Vol. 4, Number 2 issue we told you about the ways the Adobe Fetch application can obtain thumbnails from files to display in its Gallery window. One way is through Fetch's built-in filters. Another way is by extracting the thumbnail from the 'pnot' resource of the file. The third way Fetch can get a thumbnail from a file is through Apple's Macintosh Easy Open technology.

What is Macintosh Easy Open (MEO)?

Macintosh Easy Open is an extension to System 7™ incorporating the Translation Manager application programming interface. MEO extends the Macintosh architecture to provide implicit file translation when a user attempts to open a file with an application which does not have the capability to read the file's format. Macintosh Easy Open doesn't perform the translation itself, but provides an effective, system-level method for managing access to third-party translation systems and applications, which perform the actual file translation for the user. MEO translation extensions are stored in the Extensions Folder and are registered with Macintosh Easy Open when the system is started.

How Fetch Uses the MEO Technology

The Adobe Fetch application relies on MEO translation extensions when there is no built-in filter for that file format and when there is no thumbnail stored in the 'pnot' resource of the file. When the Fetch application encounters a file for which it needs a translation extension, it builds and then searches through a list of registered translators to find one capable of converting the file from its original format to PICT or TEXT. The Fetch application will use the first capable translation extension it finds in the list.

Fetch passes the file specification to the MEO translation extension telling the translator to convert the file, saving the converted data to a temporary file in a specified location on the disk. When the translation extension returns, the Fetch application executes its normal cataloging code, using the temporary file as the source for thumbnail data. It then creates a record for the original file in its database and deletes the temporary file. A similar process is followed when a Fetch user previews a file by double-clicking on the thumbnail in the Gallery window.

What This Means To Customers

Customers who are creating large graphic databases are eager to have thumbnails for every file on their system. Fetch cannot possibly provide built-in filters for all of the file formats that exist today. While many graphic applications are beginning to store thumbnails in the 'pnot' resource for their native file formats, there are still many formats that Fetch is not able to obtain thumbnails for. Translation extensions make it possible for customers to obtain thumbnails for a larger number of file formats.

Developer Opportunities

As a developer, you can meet the needs of Adobe Fetch customers by writing Macintosh Easy Open translation extensions. The Fetch product team is willing to provide you information as to which file formats are most requested by customers. Several developers have jumped at the opportunity to provide MEO translation extensions for the more popular file formats. Translation extensions for Scitex® CT, Scitex LW, and Adobe

SuperPaint™ are now available for customers to purchase through the Adobe Plug-in Source. In addition, Adobe Fetch ships with MEO translation extensions for Targa, GIF, and coming soon, Adobe Acrobat. Keep in mind that MEO translation extensions can be used by other applications, too. Any application which takes advantage of the MEO technology will benefit from a translation extension residing on the machine.

If you are interested in writing MEO translation extensions, contact the Adobe Developers Association and ask to join the Adobe Fetch Developers Program. After joining, you will receive the Fetch Compatibility Toolkit along with technical support. You will also need to purchase the Mac OS Software Developers Kit from Apple. The Apple Developers Association can be reached electronically through the following addresses:

AppleLink:® APDA
America Online:℠ APDA
CompuServe:® 76666,2405
Internet: APDA@applelink.apple.com

Colophon

This newsletter was produced entirely with Adobe PostScript software on Macintosh and IBM® PC compatible computers. Typefaces used are from the Minion™ and Myriad™ families from the Adobe Type Library.

Managing Editor:
Jennifer Cohan

Technical Editor:
Jim DeLaHunt

Art Director:
Karla Wong

Designer:
Lorsen Koo

Contributors:
Mike Mitchell, Michelle Sellars, Ed Svoboda, Robert Teng

Part Number ADA0057 4/95

Adobe PostScript