Adobe Developers Association

ADA news

In This Issue

Volume 4, Number 5

Version 1.1 of the Portable Document Format

Version 1.0 of the Adobe Portable Document Format (PDF) provides a rich feature set for representing electronic documents, and is used as the file format for version 1.0 of the Adobe Acrobat™ products. Version 1.1 of PDF expands the range of document features that PDF can represent, and adds features to support collections of documents. It is used by version 2.0 of the Adobe Acrobat products. This article describes many of the version 1.1 enhancements. Throughout this article, version numbers refer to the PDF specification version unless otherwise specified.

Before describing the new features, it’s worthwhile to mention that the PDF version number is 1.1 (instead of 2.0) because most features added in version 1.1 do not prevent viewing applications designed for version 1.0 from viewing the newer files. The major version of PDF will be incremented (i.e., there will be a version 2.0 of PDF) only if older viewers are unlikely to be able to read the new files.

Articles

Articles provide a convenient way to mark and read columns of related text, such as those commonly used for articles in newspapers and magazines. An article is represented by a sequence of rectangles that enclose the columns making up the article, and an optional title. The Adobe Acrobat 2.0 viewers (Exchange and Reader) have a special article-reading mode that lets users easily read and follow articles.

Actions

Actions are what happen when you click on a link or a bookmark. Version 1.0 supports only a single action type—one that changes the view to another location within the same document. Because there is only a single action type in 1.0, the concept of an action is not explicit. Version 1.1 makes the concept explicit, and adds several new action types. The action types supported in version 1.1 are:

- Change the view to a particular location in the current document
- Change the view to a particular location in another document
- Launch an application
- Begin reading an article

The first action type, present in version 1.0, is enhanced in version 1.1. It now supports several new destination types that fit the page’s bounding box (the rectangle enclosing all marks on a page) into the window.

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:
+31-20-6511-355

FAX:

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

Europe:
+31-20-6511-313
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.

Version 1.1 of the Portable Document Format

Cross-document links

Version 1.0 supports hypertext links between locations within a single document. This allows the creation of richly-linked documents. Version 1.1 adds support for hypertext links between documents, allowing the creation of richly interlinked collections of documents. In cross-document links, filenames are specified in a platform-independent format. They must be converted to the appropriate platform-specific name (e.g., by inserting colons and backslashes under Microsoft® Windows®, and colons on the Apple® Macintosh®) by the viewer program. Recognizing that the same file may have different names when seen from different platforms (Macintosh, Windows, and UNIX®), version 1.1 also allows three additional platform-specific filenames to be specified for each file. If present, the optional filename for the appropriate platform is intended to be used instead of the platform-independent name.

Named destinations

Named destinations address a file synchronization issue that arises in interlinked document collections. They allow each document in an interlinked collection to be updated independently. The easiest way to understand the need for named destinations and how they work is through an example.

Version 1.0 specifies destinations by a page number and a position on the page. This is problematic for links between files, although it is acceptable for links within a file. For example, suppose the file *srcfile.pdf* contains a link to Chapter 1 in the file *dstfile.pdf*. With version 1.0-style links, if *dstfile.pdf* is revised so that Chapter 1 begins on a different page or position, it is also necessary to revise the link in *srcfile.pdf* to point to the new location. This is clearly undesirable, particularly when you consider large, highly-interlinked collections of documents created by a number of authors. To address this, version 1.1 supports named destinations, which work as follows: the source of the link (the button on which you click) contains the name of the file to go to, and the name of a destination in that file (but not the page number and position). The file in which the destination is located contains a list of names, and the page number and position for each name. When a user clicks on a link whose destination is in *dstfile.pdf*, *dstfile.pdf* is opened and its list of destination names is searched to determine the page and position of the link's destination. Since *dstfile.pdf* contains all the information needed to convert the destination name into a page number and position, when *dstfile.pdf* is updated there is no need to update links in any other documents that point to it.

Device-independent color

Device-independent color attempts to ensure that colors are represented accurately on a wide variety of output device types. Version 1.0 supports only device-dependent color, while version 1.1 adds support for device-independent color. There are two parts to this device-independent color support: specifying colors in a manner that is independent of any particular output device; and specifying a style for mapping colors in an image to colors that an output device can display.

Version 1.1 of the Portable Document Format

Version 1.1 supports device-independent grayscale, RGB, and Lab color spaces. These are tied to the standard device-independent CIE 1931 (XYZ) color space. It also provides four color-mapping styles, called rendering intents. Each of these intents is best suited to certain image types:

- Absolute colorimetric—Requests an exact color (hue, saturation, and brightness) match. This is appropriate for uses that require exact matches, such as some line art or spot color usage. If the exact color cannot be displayed, a close available one is substituted.
- Relative colorimetric—Requests an exact hue and saturation match, but scales the brightness range so that no brightness values exceed the display device's maximum brightness. This is often appropriate for line art and spot color. As a result of the brightness scaling, the exact colors produced will differ on devices that have different brightness range capabilities. If the exact hue/saturation cannot be displayed, a close available one is substituted.
- Perceptual—Smoothly scales the hue, saturation, and brightness ranges to match those available on the output device. This generally provides a pleasing rendering of scanned images such as photographs. As a result of the scaling, all image colors are modified somewhat.
- Saturation—Emphasizes saturation instead of attempting to achieve a close hue and brightness match. This is appropriate for business graphics.

Security

Version 1.1 supports document encryption, to prevent users who do not know a document's password from viewing the document. This allows documents to be posted in a public location, without concern that everyone can read them. In addition, documents can be protected so that they cannot be printed, modified, copied to the clipboard, or any combination of these.

Font support

Support for two additional font-related features has been added. The first feature is font subsets. This affects embedded fonts by permitting the embedding of only those glyphs in a font that are used, rather than all the font's glyphs. Support for font subsets reduces the size of files that contain embedded fonts in which only a small fraction of the glyphs are used. The second new feature is support for embedding of TrueType™ fonts.

Annotation attributes

Several new attributes have been added to links and notes, both of which are annotation types. Link borders can now be dashed, and have a specified color. Each note can have a specified color, title string, and modification date.

continued on page 4

Version 1.1 of the Portable Document Format

Search-related

Version 2.0 of the Adobe Acrobat products supports cross-document search. This allows users to search for specific text or other information, such as title, in a collection of PDF files. The cross-document search is a two-step process: first, the collection of documents is indexed. In this step, all the searchable information is collected into a compact index file. Second, users access this index file whenever they perform a search. Version 1.1 provides two features to address issues that cross-document search introduces.

First, users often want to find a document with a specific title, or a document related to specific keywords, or any number of other characteristics of a document that cannot be directly or easily derived from the document's text. Version 1.1 expands the document info dictionary to help support this. Newly-added fields (all of which are optional) are:

- document title
- subject
- keywords
- modification date

Second, the machine on which a search is conducted is often not the same type of machine as that on which the index was built. Because of this, the filenames that are visible when the search is performed may not be the same as those seen when the index was created. For example, suppose a set of Macintosh files on a Novell server is indexed on a Windows machine, and a search is subsequently performed on a Macintosh computer. The Macintosh computer sees long filenames, but the index created on the Windows machine will contain 8.3 versions of the same filenames. To assist in locating documents under these circumstances, version 1.1 supports a document ID internal to the PDF file. The document ID is an array of two strings. The first string is assigned when the document is created, and the second is updated each time the document is modified. If a document has the correct first ID, but the incorrect second ID, it is an indication that it is essentially the correct document, but it has been edited in some fashion, for example, bookmarks or links have been added. The ID is most useful if no two documents have the same ID. It is up to the program that creates the PDF file to ensure its uniqueness. The Acrobat products attempt to do so by calculating the ID using an algorithm that includes the current time, pathname, size (in bytes), and all values in the document info dictionary.

Binary data

Version 1.0 requires all binary data to be 7-bit ASCII encoded when written into a PDF file. Files that contain only 7-bit ASCII, however, do not necessarily pass unaffected through commonly-used transmission mechanisms. Because of this, and the fact that ASCII encoding

Version 1.1 of the Portable Document Format

increases the size of binary data by at least 12%, version 1.1 drops the requirement that all data be 7-bit ASCII encoded; PDF files may now contain raw binary data.

Pass-through PostScript language code

Version 1.1 allows sections of PostScript language code to be embedded into PDF files. These sections are ignored when the file is viewed or printed to a printer which is not a PostScript printer, but included when the file is printed to a PostScript printer. Pass-through PostScript language code must be used very carefully, and should only be used when there is no other way to accomplish the task using PDF. If used inappropriately, it can produce a file that does not print properly.

Distinguishing between version 1.0 and version 1.1 PDF files

The first line in a PDF file, known as the header, specifies the version of PDF to which the file contents conform. For version 1.0, the header is `%PDF-1.0`, while for version 1.1 it is `%PDF-1.1`. Note that a version 1.0 file that also conforms to the version 1.1 specification may have either a version 1.0 or a version 1.1 header.

Extensibility

The PDF file format is designed to be extensible. Version 2.0 of Acrobat Exchange has APIs that let you write plug-ins to take advantage of this in a number of areas. Plug-ins can add:

- New action types
- New annotation types
- New user authentication schemes for security
- Almost any other data that you might want to include

Conclusion

The rich capabilities of version 1.0 of PDF have been significantly expanded in version 1.1. See Technical Note #5156, "Updates to the Portable Document Format Reference Manual" for additional details on version 1.1. The note is available in both the Adobe Acrobat SDK and the Acrobat Plug-ins SDK, and is also available on Adobe's WEB site at:

`http://www.adobe.com/Support/TechNotes.html.`

By writing plug-ins for Acrobat Exchange, you can take advantage of the extensibility of PDF. The Acrobat Plug-ins SDK contains all the information you'll need to begin writing plug-ins. §

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:
Nicole Frees

Art Director:
Karla Wong

Designer:
Lorsen Koo

Contributors:
Tim Bienz, Nicole Frees, Mike Mitchell, Ed Taft

Part Number ADA0058 5/95

Adobe PostScript™

Developing With Adobe PageMaker

Last month we discussed passing parameters to binary commands in the Adobe PageMaker plug-ins API. This month's column addresses parameter passing and retrieving data using binary queries. If you are not familiar with passing parameters to binary commands, please see last month's column for some basic information on the subject.

The following macros move data in the binary format in and out of the parameter block when sending queries:

- **PBBinQuery** (this macro is used for queries that do not have parameters)
- **PBBinQueryWithParms**
- **PBGetReplyData**

The following macros move data in the text format in and out of the parameter block when sending queries:

- **PBGetReplyData**
- **PBTextQuery** (this macro can be used for queries having or not having parameters)
- **PBSetReplyUnits**
- **PBSetRequestUnits**

An important point to remember when passing parameters to a command or query is that Adobe PageMaker 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

There are really very few queries that take parameters. Whether you are passing numeric or string data parameters, or a combination of both, the alignment rules for queries are basically the same as for API commands. In the following examples we will concentrate on passing parameters. We'll look at the different ways of obtaining the query results a little later.

```
/**********************************************************
* GetPaletteUpdtStatus - Shows passing one parameter
* Description:
*   This returns the update status of the specified
*   PageMaker palette.
*   1 - style palette
*   2 - color palette
*   3 - control palette
*
**********************************************************/
RC GetPaletteUpdtStatus(LPPARAMBLOCK lpParamBlk,
    short cPalette)
{
    RC        status = CQ_FAILURE;
    RC        rc;
    HANDLE    hReply = NULL;
    LPSTR     lpReply;
    short     arParms[1];

    arParms[0] = cPalette;

    rc = PBBinQueryWithParms(lpParamBlk,
        pm_getsuppresspaldraw, kRSPointer, arParms,
        sizeof(arParms), kRSHandle, NULL, NULL);

    hReply = PBGetReplyData(lpParamBlk);
    if (hReply)
    {
        lpReply = MMLock(hReply);

        LPGetShort(status, lpReply);

        MMUnlock(hReply);
        MMFree(hReply);
    }
    return status;
}
```

```
/**********************************************************
* GetStoryText - Shows passing two numeric parameters
* Description:
*   Gets the selected text and returns it in the
*   specified format.
*   nType   - 0 for raw text
*           - 1 for tagged text (see documentation)
*           - 2 for rich text format (RTF)
*   nFormat - 0 to keep all non-printing characters
*           - 1 to delete all non-printing characters
*           - 2 to replace all non-printing characters
*               w/spaces
*           - 3 to substitute (see documentation)
*
*   Returns: Handle to reply data in LPPARAMBLOCK
*            (note: calling function is responsible for
*            freeing handle)
*
**********************************************************/
HANDLE GetStoryText (LPPARAMBLOCK lpParamBlk,
    short nType, short nFormat)
{
    RC      rc;
    HANDLE  hReply = NULL;
    short   arParms[2];

    arParms[0] = nType;
    arParms[1] = nFormat;

    rc = PBBinQueryWithParms(lpParamBlk, pm_getstorytext,
        kRSPointer, arParms, sizeof(arParms), kRSHandle,
        NULL, NULL);

    return (PBGetReplyData(lpParamBlk));
}

/**********************************************************
* GetColorInfo - Shows passing two parameters -
* one short and one string
* Description:
*   Gets the color information for the specified color.
*   cModel - 0 for rgb
*          - 1 for cmyk
*          - 2 for hls
*   nFormat - Name of the color in quotation marks exactly
*   as it appears on the “Colors” palette.
*
*   Returns: Handle to reply data in LPPARAMBLOCK
*            (note: calling function is responsible for
*            freeing handle)
*
**********************************************************/
```

```
HANDLE GetColorInfo (LPPARAMBLOCK lpParamBlk,
    short cModel, char* sColorName)
{
    RC      status = CQ_FAILURE;
    RC      rc;
    HANDLE  hParms = NULL;
    HANDLE  hReply = NULL;
    LPSTR   lpParms, lpOrig;

    hParms = MMalloc(32);  /*allocate 32 bytes of*/
                           /*memory for the parameters*/
                           /*being passed*/

    if (hParms)
    {
        lpPrams = lpOrig = MMLock(hParms);
        lpParms += LPPutShort(lpParms, cModel);
        lpParms += LPPutString(lpParms, sColorName);
        rc = PBBinQueryWithParms(lpParamBlk,
            pm_getstorytext, kRSPointer, arParms,
            (lpParms - lpOrig), kRSHandle, NULL, NULL);

        return (PBGetReplyData(lpParamBlk));
    }
    else
        return NULL;
}
```

Earlier we mentioned that the Adobe PageMaker program is expecting the parameters to end on an even byte. In the above example we have a string parameter that could have an even number of characters so that when you include the null terminator for the string you will get and odd number of bytes in the parameter. For this reason we have provided you with the macro LPPutString which will pad the string placed in the buffer so that it ends on the correct byte count.

Some developers are tempted to pass a pointer to a structure containing the parameter list. This will work only if you do not embed string parameters inside numeric parameters.

This works:

```
typedef tagSTRUCT1
{
    short    cParm1;
    char     szParm2[32];
}STRUCT1;
```

This will not work:

```
typedef tagSTRUCT2
{
    short    cParm1;
    char     szParm2[32];
    short    cParm3;
}STRUCT2;
```

The problem is that as Adobe PageMaker parses through the parameter list it encounters two or more null terminators in szParm2 and thinks that the parameter list is done. So if you pass a pointer to STRUCT2 and szParm2 contains the string "Red" then PageMaker will count two null terminators after the string "Red" and conclude that it has reached the end of the parameter list. Because cParm3 is never reached, you would probably get a runtime error indicating that the third parameter was incorrect or missing.

Retrieving query data

There are three methods of getting the information that is returned from an API query. Because of the changes that are taking place in the LPPARAMBLOCK, and the possibility of changes in the future, we will only discuss one method. If you have any questions about alternative methods, you may contact the Adobe Developers Association.

Because of the changes that are being made for version 6.0 of the Adobe PageMaker application, we strongly suggest that you allow the PageMaker software to help you when you are attempting to get information through an API query. For Example, say you want to know what state the PageMaker application is in. You could find that out through the getpmstate query in the following manner:

```
/**********************************************************
* GetPMState - Shows recommended method of retrieving
* information from a PageMaker API query
* Description:
*   Gets the current PageMaker state.
*
*   0 - for help
*   1 - for no publication open
*   2 - for nothing selected or no text highlighted
*   3 - for text highlighted with text tool
*   4 - for object (graphic or text block selected)
*   5 - for place mode
*   6 - for story editor
*
**********************************************************/
RC GetPMState (LPPARAMBLOCK lpParamBlk)
{
    RC      rc;
    RC      state;
    HANDLE  hReply = NULL;
    LPSTR   lpReply;


    /* by passing kRSHandle we are telling PageMaker to */
    /* allocate the memory space for the results and */
    /* return us a handle */
    rc = PBBinQuery(lpParamBlk, pm_getpmstate,
         kRSHandle, NULL, NULL);

    hReply = PBGetReplyData(lpParamBlk);
    if (hReply)
    {
        lpReply = MMLock(hReply);
        LPGetShort(state, lpReply);

        MMUnlock(hReply);
        MMFree(hReply);

        return state;
    }
    else
        return CQ_FAILURE;
}
```

By telling Adobe PageMaker to take care of creating the handle to be returned, you don't have to worry about reserving memory for the return values. §

PostScript Language Technologies

In this month's column, we respond to a question about VM consumption that we received recently from a driver developer.

Q In our PostScript language output we have the following call to the setcolortransfer operator:

```
{ 255 mul cvi [ 0.000 0.002 0.003 .... 1.000 ] exch get }
{ 255 mul cvi [ 0.000 0.002 0.003 .... 1.000 ] exch get }
{ 255 mul cvi [ 0.000 0.002 0.003 .... 1.000 ] exch get }
{ 255 mul cvi [ 0.000 0.002 0.003 .... 1.000 ] exch get }
setcolortransfer
```

where the brackets enclose 256 values for each array, to be used for color look-up.

We found that this code consumes around 2.9 megabytes of VM. How can 10 Kbytes of code that calls one PostScript operator consume so much memory?

A You have encountered a common mistake that people make when attempting to define procedures for transfer functions or spot functions. As a matter of fact, any time you define a procedure to obtain data from arrays, you have the potential of reproducing this problem.

The problem is that the array constructor operations, **[...]**, are being re-executed every time the color transfer functions are called. **[** and **]** are special syntax for names that, when executed, invoke PostScript operators to construct an array. Since the color transfer functions are called many times, this causes many copies of the array to be created. Therefore execution of the code consumes enormous amounts of VM.

As a general rule, you should avoid creating composite objects within a procedure that will be executed many times, due to the memory implications as well as the degradation of performance that will result. Here are two suggestions for avoiding this situation:

1. Define the data array as a separate operation, for example:

```
/TransferFunctionArray [ 0.000 0.002 0.003 .... 1.000 ] def
{ 255 mul cvi TransferFunctionArray exch get }
...
setcolortransfer
```

2. Define the array as an embedded executable array. Executable arrays, unlike literal arrays, are constructed when the array is scanned rather than when the array is executed. The fact that the array is executable does not interfere with its use as a data array. Here is an example of a color transfer procedure which uses an executable rather than a literal array:

```
{ 255 mul cvi { 0.000 0.002 0.003 .... 1.000 } exch get }
...
setcolortransfer
```