Technical Note QD10

Picture Comments - The Real Deal

CONTENTS

This Note (formerly titled "Optimizing for the LaserWriter--PicComments") describes the picture comments defined and interpreted by the Apple printer drivers. Most of the picture comments are specific to PostScript, but we renamed the Note to emphasize that LaserWriter printers are not necessarily PostScript devices, and that QuickDraw printer drivers may implement their own picture comment handling. This Note has been completely rewritten and incorporates all additional insights gained during the last few years. We are also much more determined now to discourage the use of obsolete and problem-laden (although still supported) picture comments, and we carefully point out known problems or limitations of each comment.

[Nov 01 1988]

Introduction

The `QDProcs` record (see *Inside Macintosh* Volume I, page 197) reflects the foundations of the architecture of QuickDraw. The `commentProc` field points to a procedure that processes picture comments, as included in a picture by means of the `PicComment` procedure (*Inside Macintosh* Volume I, page 189). This allows applications to include application-specific additional information in the pictures they create.

The `QDProcs` record also is the key to understanding how Macintosh printer drivers work. When the application calls `PrOpenPage` and draws into the printing grafPort, the printer driver collects the drawing commands by hooking into the

QDProcs of the printing port. In particular, if an application calls the PicComment procedure while drawing into the printing grafPort, the printer driver gets a chance to capture and process the information contained in the kind and dataHandle parameters.

During the development of the original LaserWriter driver, it became obvious that applications should be able to take advantage of certain PostScript features that were not accessible through standard QuickDraw calls, such as rotated text and graphics, dashed lines, fractional line widths, and smoothed polygons. Also, certain applications needed a way to transmit their own native PostScript instructions to the printer. Picture comments seemed to be the ideal vehicle for providing these capabilities.

Unfortunately, there are conflicts with the device-independent nature of the Macintosh Printing Manager architecture. In this Note, we still want to document picture comments as completely and correctly as possible; and we want to tell you how to use the best of their features, while maintaining the important goal of device-independent printing and all- purpose PICTs. (This is why it has such a painful history!)

First, we give an overview of the picture comments as currently implemented by Apple's printer drivers. This leads us immediately to the problem section "Cohabitation of QuickDraw and PostScript," which also shows how to keep the QuickDraw and PostScript graphics states synchronized during printing. Finally, we discuss all the picture comments by subject, in the order suggested by Table 1.

Back to top

Picture Comments Repertoire

The following picture comments are recognized by all PostScript LaserWriter drivers version 3.1 and later.

Table 1. PostScript LaserWriter Picture Comments

	Type	Kind	Data Size	Data	Description
	TextBegin	150	6	TTxtPicRec	Begin text function
	TextEnd	151	0	NIL	End text function
	StringBegin	152	0	NIL	Begin string delimitation
	StringEnd	153	0	NIL	End string delimitation
	TextCenter	154	8	TTxtCenter	Offset to center of rotation
	LineLayoutOff	155	0	NIL	Turn LaserWriter line layout off
	LineLayoutOn	156	0	NIL	Turn LaserWriter line layout on
#	ClientLineLayout	157	16	TClientLL	Customize line layout error distribution
	PolyBegin	160	0	NIL	Begin special polygon
	PolyEnd	161	0	NIL	End special polygon
	PolyIgnore	163	0	NIL	Ignore following polygon data
	PolySmooth	164	1	PolyVerb	Close, Fill, Frame
	PolyClose	165	0	NIL	Close the polygon
	DashedLine	180	-	TDashedLine	Draw following lines as dashed
	DashedStop	181	0	NIL	End dashed lines
	SetLineWidth	182	4	Point	Set fractional line widths
	PostScriptBegin	190	0	NIL	Set driver state to PostScript
	PostScriptEnd	191	0	NIL	Restore QuickDraw state
	PostScriptHandle	192	-	PSData	PostScript data in handle
†	PostScriptFile	193	-	FileName	FileName in data handle
†	TextIsPostScript	194	0	NIL	QuickDraw text is sent as PostScript
†	ResourcePS	195	8	Type/ID/Index	PostScript data in a resource file
	PSBeginNoSave	196	0	NIL	Set driver state to PostScript
#	SetGrayLevel	197	4	Fixed	Call PostScript's setgray operator
	RotateBegin	200	4	TRotation	Begin rotated port
	RotateEnd	201	0	NIL	End rotation
	RotateCenter	202	8	Center	Offset to center of rotation
#	FormsPrinting	210	0	NIL	Don't clear print buffer after each page
#	EndFormsPrinting	211	0	NIL	End forms printing after PrCloseDoc

† These comments are obsolete.
# These comments are not recommended.

Most of the comments in Table 1 were designed specifically for the original LaserWriter driver. In fact, the term *LaserWriter* has been (and often still is) used in the sense of "PostScript printer," and the LaserWriter driver is known to be basically a QuickDraw-to-PostScript translator. Meanwhile, however, QuickDraw-based LaserWriter models came out, so we should start being more careful in our terminology. This is why we insist on talking about PostScript drivers or PostScript printers when a picture comment applies to PostScript.

QuickDraw printer drivers may implement their own picture comments, or some of the above comments, in order to provide additional capabilities. Certain third-party printer drivers implement text rotation, for example, by supporting the `TextBegin/TextCenter/ TextEnd` picture comments.

Apple's QuickDraw printer driver for the LaserWriter SC supports the following three picture comments:

```
LineLayoutOff    155    0    NIL    Turn LaserWriter line layout off
LineLayoutOn    156    0    NIL    Turn LaserWriter line layout on
SetLineWidth    182    4    Point    Set fractional line widths.
```

The ImageWriter LQ driver and the first versions of the StyleWriter driver (prior to 7.2) implement the `LineLayoutOff` and `LineLayoutOn` picture comments. Even the ImageWriter driver reacts to picture comments:

```
BitMapThinningOff    1000    0    NIL    Turn off hi-res bitmap thinning
BitMapThinningOn    1001    0    NIL    Turn on hi-res bitmap thinning
```

The ImageWriter driver does the same toggling of the "bitmap thinning" of fat bitmaps in Best mode, when it encounters a `TextBegin` or `TextEnd` comment (undocumented feature**- -** never mind!). The ImageWriter LQ driver handles these comments similarly.

The current StyleWriter driver (version 7.2.2) and the personal LaserWriter LS driver do not support any picture comments at all.

The point of all this is:

It is impossible to determine which picture comments are supported by which printer driver.

In other words, your application should never assume a particular picture comment is available, but your application also should not defeat the device-independent design of the Macintosh printing architecture by writing printer driver-specific code!

Of course, you know (*Inside Macintosh* Volume II, page 152) that the high byte of the `prStl.wDev` field of the print record identifies a printer driver species, and that a value of $03 tells you the printer driver belongs to the PostScript LaserWriter driver ancestry. As a matter of fact, many applications use this information to achieve special printing features not available through the Printing Manager interface.

And, of course, you also know that we don't like this idea. One reason is future system software may allow spool files to be redirected to a printer other than the one chosen when you sent your printing instructions (including picture comments). Another reason is that picture comments usually are included in PICTs; documents containing such pictures should print with optimal results on any printer configuration. And, finally, you never know what the future holds for you, in terms of new printing devices or new printer drivers--or a new printing architecture!

Instead, if a picture (i.e., a sequence of imaging instructions) contains picture comments to enhance the output on devices that support them, it should also contain a standard QuickDraw representation as a fallback solution, in case the rendering device does not recognize the picture comments. The design and implementation of these picture comments should incorporate conventions to make this cohabitation of two representations in one picture possible.

Back to top

Cohabitation of QuickDraw and PostScript

Device-Independent Pictures

We can think of the Printing Manager's PrOpenPage and PrClosePage calls as being analogous to the OpenPicture and ClosePicture calls (which, by the way, reminds us to never call OpenPicture between PrOpenPage and PrClosePage; see *Inside Macintosh* Volume II, page 160). In both cases, a stream of imaging instructions is recorded for deferred rendering. We want to create pictures that include both QuickDraw and optimized PostScript representations so that we obtain the best results in all circumstances. We must take special care for third-party QuickDraw printer drivers that support picture comments originally intended for PostScript devices only.

Let's start with the easy ones.

The two picture comments PostScriptBegin and PostScriptEnd clearly suggest that any imaging instructions in between are intended exclusively for PostScript printing devices. In the case of the PostScript LaserWriter driver, the effect of PostScriptBegin is to disable all bottlenecks except commentProc, txMeasProc, getPicProc, and putPicProc. This means that QuickDraw's text, line, shape (Rects, RoundRects, Ovals, Arcs, Polygons) and bitmap drawing calls don't have any effect in the printing grafPort when enclosed by PostScriptBegin and PostScriptEnd. Instead, the PostScript LaserWriter driver expects to receive the imaging instructions as data enclosed in the PostScriptHandle comment. This way, both the PostScript and QuickDraw representation can coexist in the same picture without conflict. As a consequence, non-PostScript printer drivers, unable to interpret general PostScript text, must not imitate this behavior of ignoring QuickDraw instructions, even when they implement other picture comments such as TextBegin and TextEnd for text rotation. Otherwise, they would miss the QuickDraw representation of some PostScript imaging.

The text rotation picture comments (TextBegin, TextCenter, TextEnd) silently include the assumption that a printer driver that supports these comments:

a. ignores the QuickDraw clipping region between TextBegin and TextEnd
b. ignores the QuickDraw CopyBits instruction within TextBegin/TextEnd

This way, a bitmap representation of the rotated text can be included in the picture. It will be used only if the TextBegin/TextEnd comments are not supported. Conversely, the QuickDraw commands required to draw the text to be rotated by the printer driver are "hidden" from QuickDraw by setting the clipping region to empty, which is ignored by the driver supporting the comments.

The polygon picture comments provide another solution to the problem, in form of the special comment PolyIgnore. It allows one to include a QuickDraw representation of the smoothed polygon, ignored by a driver that supports polygon smoothing (such as via PostScript's curve operator). And for filled polygons, QuickDraw's region concept works around a conflict of who owns which polygon (see sample code later in this Note).

Some picture comments (such as the line layout comments) do not require a fallback solution in case they are not supported by a printer driver; or the feature, if absent, is not a big loss (such as SetLineWidth, provided you use it only for widths smaller than one 72-dot-per-inch (dpi) QuickDraw pixel).

But for the picture comments DashedLine/DashedStop and RotateBegin/ RotateCenter/RotateEnd, there is no general solution to the "cohabitation" problem; and we are distressed about it. It is obvious that they have been defined with the PostScript LaserWriter driver in mind, without anticipating a future furnished by some 150 third-party printer drivers. The only way to include both representations in these cases is indeed to assume that only PostScript drivers will support the picture comment, such that the PostScriptBegin and PostScriptEnd comments can be used to "comment out" the QuickDraw representation.

Even under the above assumption, we still need a trick to prevent the QuickDraw calls within the scope of the picture comments from showing up when the comments are not recognized. Fortunately, early Macintosh developers discovered a QuickDraw feature that, unintentionally, solves the problem. Passing the "magic" mode 23 to PenMode inhibits QuickDraw's normal drawing, but still lets the LaserWriter driver see the drawing instructions come through the bottlenecks, so that it can translate them into PostScript. Note that this pen mode always has been undocumented, and that using it was considered a compatibility risk and frowned upon for some time. Given the current state of affairs, however, there is no reason anymore to be paranoid about it.

Keeping QuickDraw and PostScript Synchronized

There are two situations, in the context of picture comments, where the design of the PostScript LaserWriter driver requires special precautions from the application programmer.

First, certain QuickDraw instructions like Move, MoveTo, PenPat, and PenSize change the state of the grafPort, without going through the QDProcs bottleneck procedures. A Macintosh printer driver takes these changes into account only at the time it executes an actual drawing instruction. Remember, the printer driver hooks into the QDProcs to get execution time and only "sees" instructions coming through the QDProcs. Nothing is wrong with it--unless PostScript code is woven into the graphics instructions by means of picture comments. (Note that PostScript code may be generated transparently when the LaserWriter driver encounters certain picture comments.) If the PostScript code assumes that the current state of the grafPort corresponds to what you expect it to be, then you have to flush the state of the grafPort explicitly before inserting the PostScript code. This is easier than it sounds; just do something inoffensive that goes through the QDProcs.lineProc bottleneck, like in the following utility procedure:

```
PROCEDURE FlushGrafPortState;
{ This routine causes the state of the Printing Manager's grafPort to be }
{ flushed out to the LaserWriter, by making a dummy call through the     }
{ QDProcs.lineProc bottleneck. Pen size and pen location are preserved.  }

   VAR
      penInfo: PenState;

   BEGIN
      GetPenState(penInfo);                              { Save pen size. }
      PenSize(0,0);                                   { Make it invisible. }
      Line(0,0);                               { Go through QDProcs.lineProc. }
      PenSize(penInfo.pnSize.h, penInfo.pnSize.v);    { Restore pen size. }
   END;
```

Another unwanted effect is related to the PostScript LaserWriter driver's multiple internal buffering of generated PostScript code. The PostScript code generated for text drawing instructions (which usually involves font queries and, sometimes, font downloading) is buffered independently from the PostScript code inserted by means of picture comments. In certain cases, this results in apparently nonsequential execution of drawing instructions, and may affect clipping regions or may have side effects on the PostScript code you included in picture comments. In order to synchronize the sequence of QuickDraw instructions with the generation of PostScript code, you need to call the following procedure:

```
PROCEDURE FlushPostScriptState;
{ This routine flushes the buffer maintained by the LaserWriter driver. }
{ All PostScript, generated either by the app or by the LaserWriter     }
{ driver, will be sent to the device.                                   }
   BEGIN
      PicComment(PostScriptBegin, 0, NIL);
      PicComment(PostScriptEnd, 0, NIL);
   END;
```

In the following discussion of picture comments, we'll refer to these two utility routines as appropriate.

Back to top

Text Rotation

Comments: TextBegin, TextCenter, TextEnd

These comments give access to PostScript's capabilities of rotating, flipping, and justifying text. They are intended for applications likely to be used with PostScript printers (such as desktop publishing and advanced drawing applications), but which don't want to use PostScript explicitly. Note that some non-PostScript printer drivers support these comments as well. For situations where the comments are not supported (such as the QuickDraw screen, or most QuickDraw printer drivers), you must provide a bitmap representation of the rotated text as an alternative.

Let's look at sample code right away.

```
USES  PicComments;  { See Appendix; defines constants for just and flip and   }
                    { the structures referred to by TTxtPicHdl and TCenterHdl.}

PROCEDURE QDStringRotation(s: Str255; ctr: Point;
                           just, flip: Integer; rot: Fixed); EXTERNAL;
{ This routine should generate a bitmap of the rotated and flipped text       }
{ and use CopyBits to draw it to the grafPort. Left as an exercise ...        }

PROCEDURE DrawXString(s: Str255; ctr: Point; just, flip: Integer; rot: Fixed);
{ Draws the string s rotated by rot degrees around the current point, offset  }
{ by ctr, justifying and flipping according to the just and flip parameters.  }
{ If the printer driver supports the TextBegin, TextCenter, and TextEnd       }
{ picture comments, it rotates the text at device resolution; otherwise, the  }
{ external procedure QDStringRotation is called to image the rotated string.  }
{ The pen position is preserved. }

   VAR
      hT: TTxtPicHdl;     { Defined in PicComments.p - see Appendix. }
      hC: TCenterHdl;     { -"- }
      zeroRect: Rect;
      pt: Point;
      oldClip: RgnHandle;

   BEGIN
      GetPen(pt);  { to preserve the pen position }

      hT := TTxtPicHdl(NewHandle(SizeOf(TTxtPicRec)));
      hC := TCenterHdl(NewHandle(SizeOf(TCenterRec)));
      { No error handling: if these fail, we are in deep trouble anyway ...}
      WITH hT^^ DO BEGIN
         tJus      := just;
         tFlip     := flip;
         tAngle    := - FixRound(rot); { I like counterclockwise better. }
         tLine     := 0; { reserved }
         tCmnt     := 0; { used internally by the printer driver }
         tAngleFixed := - rot;
      END;
      hC^^.y := Long2Fix(ctr.v);
      hC^^.x := Long2Fix(ctr.h);

      PicComment(TextBegin,SizeOf(TTxtPicRec),Handle(hT));
      PicComment(TextCenter,SizeOf(TCenterRec),Handle(hC));
      { PostScript graphics state now has rotated/flipped coordinates. }

      oldClip := NewRgn;
      GetClip(oldClip);
      SetRect(zeroRect,0,0,0,0);
      ClipRect(zeroRect);  { Hides the following DrawString from QuickDraw }
      DrawString(s); { in the rotated PostScript environment. }
      ClipRect(oldClip^^.rgnBBox);

      { Now the "fallback" bitmap representation; see the comments above  }
      { at the declaration of the QDStringRotation procedure. }
      QDStringRotation(s, ctr, just, flip, rot);

      PicComment(TextEnd,0,NIL); { Set environment back to the original state }

      DisposHandle(Handle(hT));
      DisposHandle(Handle(hC));

      MoveTo(pt.h,pt.v);  { to preserve the pen position }
   END;
```

The preceding discussion about including both QuickDraw and PostScript representations and the comments included in the source code say it all: The conventions tied to the usage of the TextBegin and TextEnd picture comments allow you to take advantage of a printer driver's implementation of high-resolution text rotation, while including a bitmapped representation for where the comments are not supported.

Some Additional Hints

- Because of QuickDraw's orientation of the vertical coordinate axis, the rotation angle is measured clockwise. Nothing prevents us from using the negative angle if we are used to the counterclockwise orientation.
- The angle is measured in degrees (0..360), and passed as a Fixed type number (that is, if taken as a LongInt value, you have to divide it by 65536 to obtain the angle in degrees). For integer angles, it is possible to use a reduced TTxtPicRec structure that does not contain the tRotFrac field. The PostScript LaserWriter driver uses GetHandleSize(hT) to determine whether it must use the fractional angle in the tRotFrac field. To be safe, always set the tRot field to FixRound(tRotFrac) if you go with the extended TTxtPicRec (as we do here).
- It is convenient that clipping regions are ignored between the TextBegin and TextEnd picture comments, because it allows us to clip out the DrawString on printers that don't support these comments. Unfortunately, this also means that text rotated this way can't be clipped. If clipping of rotated text is required, you'll have to do it entirely within PostScript.
- Due to the LaserWriter driver's internal buffering of generated PostScript code, the effect of ignoring clip regions may be propagated to preceding sections of your drawing instructions. We recommend calling the FlushPostScriptState procedure described earlier immediately before the TextBegin comment.
- The tJus field in the TTxtPicRec, if different from tJusNone, tells the printer driver to maintain either the left, right, or center point of the string without recalculating the interword and intercharacter spacing. The tJusFull value specifies that the original length of the string (on the QuickDraw screen) must be maintained. This is important when the printer font has widths different from those of the screen font, and when you rotate justified blocks of text.
- The tFlip field in the TTxtPicRec specifies horizontal or vertical flipping about the center point specified by the TextCenter comment.
- The TextCenter comment specifies the center of rotation for any text enclosed within the TextBegin and TextEnd calls, as an offset to the location of the current point. The rotation is achieved by changing PostScript's coordinate system. A sequence of DrawString - MoveTo instructions is rotated as a whole until TextEnd is encountered.
- Some versions of double-byte Kanji systems print Kanji characters by calling CopyBits instead of calling standard text drawing routines. This means the comments in the Text Rotation family cannot be used with these fonts. Instead, use the Graphics Rotation comment family described later in this Note.

Back to top

Line Layout Control

Comments: LineLayoutOn, LineLayoutOff, ClientLineLayout

When drawing to a printing grafPort, the selected printer driver does a lot of work "behind the scenes" to try to maintain the infamous "What-You-See-Is-What-You-Get" (WYSIWYG) metaphor from the screen to the paper, and generally to make the printed output look as good as possible. Depending on the target device, the printer driver, and the configuration of fonts in the system, the font you draw text with may be scaled, smoothed, remapped, or even replaced by a font built into the printer. In nearly all cases where the device resolution of the printer is different from QuickDraw's "hard-coded" 72-dpi screen resolution, the width of text rendered on the printer is different from the text width on the screen. This is due to nonproportionally scaling bitmapped fonts, different character widths after font substitution, and rounding errors of fractional character widths on the screen. The difference in the width of a line of text is called the **line layout error**.

The printer driver is responsible for adjusting the word and character spacing in the printed output so that the two widths are identical. If it doesn't, apparently fully justified text on the screen may appear ragged on the paper, and certain lines of text may extend beyond the right border and be badly clipped. Many existing applications make this task really difficult for the printer drivers (don't blame them, though!). They position the words (or even characters) separately on a line, and the printer driver has to figure out how to collect the complete line before applying its line layout algorithm to distribute the difference of the text widths into word and character spacing. Given the uneven distribution of the character width differences, and the requirement of achieving good typographical quality in the printed output, it is unavoidable that the position and width of a word within a justified line differs slightly from what appears on the screen; only the length of the whole line is maintained.

In computing the required line layout adjustments, the LaserWriter driver proceeds as follows:

1. It collects text coming through the printing grafPort's `textProc` bottleneck, and heuristically puts it together into what it "believes" is a logically contiguous line of text. This includes text moved vertically away from the baseline, to take care of indices or exponents in the text. The process of accumulating text is stopped when the LaserWriter driver detects that the horizontal component of the pen position has changed since the previous text drawing instruction, or when picture comments like `TextBegin`, `TextEnd`, `StringBegin`, `StringEnd` are encountered.
2. It determines the width of the accumulated logical line of text, both on the screen and on the printer, and distributes the line layout error among the interword and intercharacter spacing of the printed output.

The `LineLayoutOff` picture comment disables only the second step (distribution of the line layout error); the heuristic algorithm of accumulating text into a logically contiguous piece is not affected. Otherwise, if the character widths of the printer font are different from those of the screen font, and if the text contains exponents or indices, the latter would often be misplaced.

The following code fragment shows a probably unexpected consequence of this behavior. We draw a line in two pieces three times. A vertical line shows the starting pen position of the second `DrawString` call. The second line is enclosed by `LineLayoutOff` and `LineLayoutOn` picture comments.

```
PROCEDURE ObserveLineLayout;

   CONST
      testString1 = 'Whatever you like, preferably ';
      testString2 = 'with spaces, long and short words';
      fontName = 'New York';
      fontSize = 14;
      x0 = 20; { starting point }
      y0 = 40;
      h  = 30; { line height }

   VAR
      familyID: Integer;
      w, y    : Integer;

   BEGIN
      GetFNum(fontName, familyID);
      TextFont(familyID);
      TextSize(fontSize);

      w := StringWidth(testString1);
      y := y0;
      MoveTo(x0 + w, y - h);
      Line(0, 4 * h);    { This is to estimate the difference. }

      { Draw the first line, in two pieces. }
      { This is the default line layout behavior of the LaserWriter driver. }
      MoveTo(x0, y);
      DrawString(testString1);
      MoveTo(x0 + w, y);
      DrawString(testString2);

      { Draw the second line, in the same way as above. }
      { Because of the LineLayoutOff picture comment, the unmodified widths }
      { of the printer font are used. }
      y := y + h;
      PicComment(LineLayoutOff, 0, NIL);

                                  { *** (1)  ***}
      MoveTo(x0, y);
      DrawString(testString1);
      MoveTo(x0 + w, y);
```

```
                                    { *** (2)  ***}
    DrawString(testString2);
    y := y + h;

    PicComment(LineLayoutOn, 0, NIL);
    { Back to the original behavior. }
    MoveTo(x0, y);
    DrawString(testString1);
    MoveTo(x0 + w, y);
    DrawString(testString2);

END;
```

And this is (approximately) the output of the `ObserveLineLayout` (with LaserWriter driver version 7.1.1, and the default setting "Font Substitution enabled"):

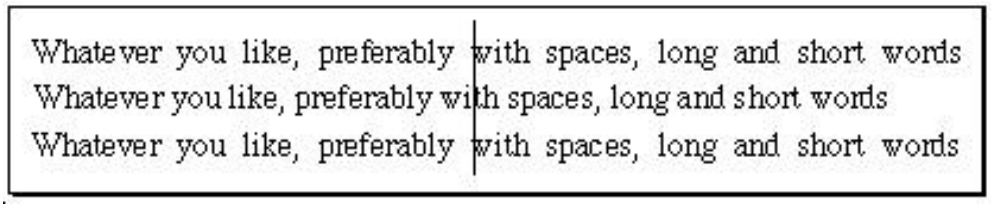

Figure 1. Effect of the `LineLayoutOff` comment

For most noticeable effects, we choose the bitmapped New York font, such that the LaserWriter driver substitutes PostScript Times (note that there are no line layout problems with TrueType fonts, unless the TrueType font has the same name and different character widths as a printer-resident PostScript font). The screen font New York is larger than the PostScript font Times, and in the first and third lines, the printer driver (after accumulating `testString1` and `testString2` into one logical line) distributes the line layout error (mainly) among the spaces between words. You may even notice that the starting point of `testString2` ("with ...") has been slightly moved to the left in the process. The width of the whole line, however, is the same as on the screen.

The second line, where the `LineLayoutOff` comment is active, demonstrates a dramatic counterexample to the popular belief that this picture comment is here to assure precise positioning of text. It seems the opposite is true, and the LaserWriter driver has deliberately ignored the `MoveTo(x0+w,y)` instruction! What we would have expected is this:

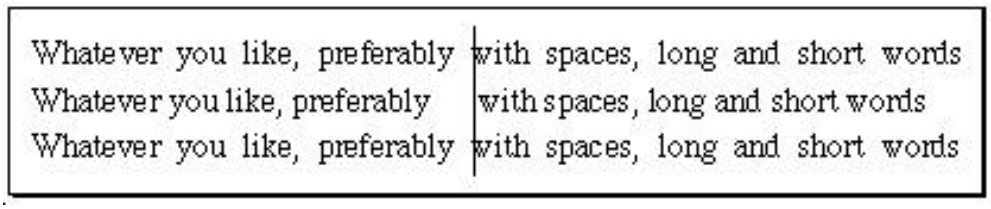

Figure 2. Desired result of the `LineLayoutOff` comment

The attentive reader already knows the explanation. As mentioned earlier, we must break the LaserWriter driver's heuristic line accumulation algorithm before drawing `testString2`. Short of adequate documentation, developers have found out that a `FlushGrafPortState` call right after the `MoveTo(x0+w,y)` instruction has the desired effect (see {*** (2) ***} in the code snippet given earlier). Unfortunately, it creates quite a lot of overhead in the pictures, and penalizes all printer drivers that don't need it. A better solution is to use the `StringBegin` and `StringEnd` picture comments at the markers {*** (1) ***} and {*** (2) ***} in the code shown earlier. This indicates that `testString1` is to be considered a logically independent text entity, and must not be put together with any other pieces of text coming through the `textProc` bottleneck. The overhead of these comments is much smaller, and they don't affect other printer drivers at all.

The `ClientLineLayout` picture comment, supported by the (PostScript) LaserWriter driver, has never been documented. Its effect is rather subtle and very specific to the PostScript LaserWriter driver. Basically, it allows the application to redefine the character that absorbs the major part of the line layout error (usually the space character),

and the percentages of the "major" and "minor" parts of the line layout error (usually 80 percent versus 20 percent). The "minor" part is distributed across intercharacter spacing.

Only very ambitious page layout applications might be interested in this functionality; but then, they should rather aim at a more general scheme of line layout control that does not rely upon this very driver-specific picture comment.

The `PicComment.p` interface (see the Appendix) describes the `TClientLLRecord` structure passed through the handle parameter to the picture comment. If you want, feel free to experiment with it; we recommend, however, that you do **not** use this picture comment in your application.

Caveats

- Some older printer drivers supporting the `LineLayoutOff` picture comment are unable to correctly obey a subsequent `LineLayoutOn` picture comment.
- Don't forget that if you use `LineLayoutOff`, the burden of "WYSIWYG" is now on your shoulders, and not the printer driver's.
- A previous version of this Note said that setting the Font Manager's `FractEnable` global to `TRUE` implied the effect of the `LineLayoutOff` picture comment. As it turned out, the statement was based on observations with a specific (older) version of the LaserWriter driver, and is not true in general. The setting of `FractEnable` does have some more or less subtle effects on the line layout algorithm, however; and this is quite plausible. Similarly, the results of combining the picture comments `LineLayoutOff` and `LineLayoutOn` with calls to `SpaceExtra` (*Inside Macintosh* Volume I, page 172) or `CharExtra` (*Inside Macintosh* Volume V, page 77) are sometimes unpredictable, depending on the particular printer driver.

And Finally the Good News

Given that the effect of the `LineLayoutOff` and `LineLayoutOn` comments does not require any changes in your printing code, you don't have to worry whether or not a particular driver supports them. They are useful mainly when you're sure you want no external assistance in computing word and character spacing for full justification, or when you need precise control over the horizontal placement of words and characters (such as in forms or tabulated text) and understand how to achieve this.

Back to top

String Delimitation

Comments: `StringBegin`, `StringEnd`

These comments allow applications to specify the logical beginning and end of a string, possibly drawn with multiple calls to a QuickDraw text drawing routine (such as `DrawChar`). If this was their only raison d'être, they would have no relationship with the PostScript LaserWriter driver. But, as already let out in the preceding section on line layout, they are important to notify the printer driver that it should consider the text coming through the `textProc` bottleneck between `StringBegin` and `StringEnd` as an independent entity. Otherwise, the driver might continue to perform its heuristic accumulation of text drawing instructions for the same line, and defeat your text positioning intentions. Indeed, both `StringBegin` and `StringEnd` trigger the generation of PostScript instructions for drawing the text that has been accumulated in a line layout buffer, and reinitialize the internal variables for line layout computations. In other words, you need these picture comments to turn the LaserWriter driver's line layout behavior completely off.

Back to top

Polygon Comment Family

Comments: `PolyBegin`, `PolyEnd`, `PolyClose`, `PolySmooth`, `PolyIgnore`

PostScript has the built-in capability of drawing cubic Bézier curve sections (see the *PostScript Language Reference Manual,* Second Edition, page 393). This is convenient for "smoothing" of polygons. The polygon-related picture comments have been provided to give applications easy access to this PostScript feature, with provision for including a QuickDraw approximation of the curve.

Schematically, the polygon comments are used as follows:

PolyBeginComment; { Put the PostScript driver into "polygon mode." }

ClipRect(zeroRect); { Hide the following from QuickDraw. }

PolyClose *Comment;* { Optionally, if "closed" smoothing desired. }

PolySmoothComment; { Tell the driver to draw a Bézier curve. }

DrawPolygon; { Invisible for QuickDraw; PostScript output = curve. }

PolyIgnoreComment; { The driver will ignore the following line drawing. }

SetClip(origClipRgn); { Make it visible for QuickDraw. }

DrawQDPolygon; { Usually, a QuickDraw approximation of the curve. }

PolyEndComment; { PostScript driver resumes standard mode. }

A piece of sample code is sometimes worth more than one or two pictures; below, you'll find both. For clarity and completeness of the exposition, we provide the coordinate definition of the polygons through arrays of `Point`s, initialized in a preliminary `DefineVertices` procedure. You can enclose the `PolygonDemo` procedure between `OpenPicture` and `ClosePicture` calls to create a picture containing both QuickDraw and PostScript representations (see Figures 3 and 4), or you can use it as is when a printing page is open.

```
USES PicComments;
{ See Appendix of this Note for the definition of the TPolyRec structure. }

   CONST
      kN = 4; { number of vertices for PostScript }
      kM = 6; { number of vertices for QuickDraw approximation }

   TYPE
      PointArray = array[0..0] of Point;  { range checking OFF }
      PointArrayPtr = ^PointArray;

PROCEDURE DefineVertices(VAR p,q: PointArrayPtr);

   CONST
      cx = 280;
      cy = 280;
      r0 = 200;

   BEGIN
   { The array p^ contains the control points for the BÉzier curve: }
      SetPt(p^[0],cx + r0,cy);
      SetPt(p^[1],cx,cy + r0);
      SetPt(p^[2],cx - r0,cy);
      SetPt(p^[3],cx,cy - r0);
      p^[4] := p^[0];
   { q^ contains the points for a crude polygon approximation of the curve: }
      q^[0] := p^[0];
      SetPt(q^[1],cx,cy + round(0.7 * (p^[1].v - cy)));
      SetPt(q^[2],(p^[1].h + p^[2].h) DIV 2,(p^[1].v + p^[2].v) DIV 2);
      SetPt(q^[3],cx + round(0.8 * (p^[2].h - cx)),cy);
      SetPt(q^[4],q^[2].h,cy + cy - q^[2].v);
      SetPt(q^[5],q^[1].h,cy + cy - q^[1].v);
      q^[6] := q^[0];
   END;

PROCEDURE PolygonDemo;

   VAR
      p,q: PointArrayPtr;
```

```
      aPolyVerbH: TPolyVerbHdl;
      i: Integer;
      clipRgn, polyRgn: RgnHandle;
      zeroRect: Rect;

   BEGIN
      p := PointArrayPtr(NewPtr(SizeOf(Point) * (kN + 1)));
      q := PointArrayPtr(NewPtr(SizeOf(Point) * (kM + 1)));
      IF (p = NIL) OR (q = NIL) THEN DebugStr('NewPtr failed');
      DefineVertices(p,q);

      PenNormal;              { First show the standard QuickDraw polygon. }
      MoveTo(p^[0].h,p^[0].v);
      FOR i := 1 TO kN DO LineTo(p^[i].h,p^[i].v);

      PenSize(2,2);                { Now show the same polygon "smoothed." }
      PenPat(gray);
      { First, the PostScript representation, clipped off from QuickDraw: }
      aPolyVerbH:= TPolyVerbHdl(NewHandle(SizeOf(TPolyVerbRec)));
      IF aPolyVerbH<> NIL THEN
         WITH aPolyRecH^^ DO BEGIN        { *** See comment 1, below. *** }
            fPolyFrame := TRUE;
            fPolyFill  := FALSE;
            fPolyClose := FALSE;     { Compare with the result for TRUE ! }
            f3 := FALSE;
            f4 := FALSE;
            f5 := FALSE;
            f6 := FALSE;
            f7 := FALSE;
         END;
      MoveTo(p^[0].h,p^[0].v);            { *** See comment 2, below. *** }
      PicComment(PolyBegin,0,NIL);
   {  PicComment(PolyClose,0,NIL);  <<< Only if fPolyClose = TRUE, above! }
      PicComment(PolySmooth,SizeOf(TPolyVerbRec),Handle(aPolyVerbH));
      clipRgn := NewRgn;
      GetClip(clipRgn);
      ClipRect(zeroRect);
      FOR i := 1 TO kN DO LineTo(p^[i].h,p^[i].v);

      { Next, the -crude- QuickDraw approximation of the smoothed polygon, }
      { invisible for PostScript because of PolyIgnore: }
      SetClip(clipRgn);
      PicComment(PolyIgnore,0,NIL);
      polyRgn := NewRgn;                  { *** See comment 3, below. *** }
      OpenRgn;
      MoveTo(q^[0].h,q^[0].v);
      FOR i := 1 TO kM DO LineTo(q^[i].h,q^[i].v);
      CloseRgn(polyRgn);
      FrameRgn(polyRgn);          { Or FillRgn, if fPolyFill above is TRUE. }
      PicComment(PolyEnd,0,NIL);

      DisposHandle(Handle(aPolyVerbH));
      DisposeRgn(polyRgn);
      DisposPtr(Ptr(p));
      DisposPtr(Ptr(q));
   END;
```

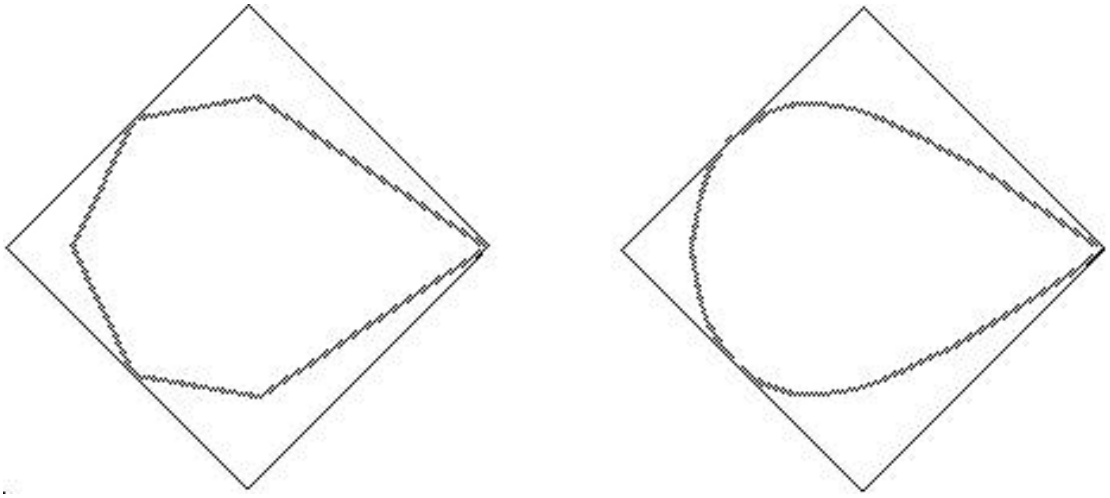

Figure 3. QuickDraw Output

Figure 4. PostScript printern printer output

Additional Comments and Explanations

1. The `fPolyFrame` and `fPolyFill` fields of the `TPolyRec` record are self-explanatory. The `fPolyClose` flag is redundant with the `PolyClose` picture comment, but is included for the convenience of the LaserWriter driver. It is often misunderstood. It does **not** mean the polygon is being closed automatically, such as with the PostScript `closepath` operator; instead, it affects the shape of the smooth curve. Figure 4 shows the result for `fPolyClose = FALSE;` the start and end point of the polygon is distinguished. In the case of `fPolyClose = TRUE`, all vertices of the polygon are treated in the same manner, and the resulting curve resembles a circle (in this case).
2. The anonymous fields `f3..f7` are reserved and should be set to zero (that is, `FALSE`).
3. The polygon is drawn at the current pen location when the `PolyBegin` comment is received.
4. In general (and in this example), you do **not** need to open a region, collect the line segments in the region, and draw the polygon through `FrameRgn`. It is demonstrated here only to prepare you for situations where you want to fill the polygon with a pattern. You **cannot** open a polygon and use `FillPoly`, because the PostScript driver "owns" the polygon concept at this point and captures--and ignores--all line drawing between the `PolyIgnore` and `PolyEnd` comment. Regions do not interfere with polygons, however, and can be used to paint or fill the polygonal shape.

Caveats

PostScript Level 1 has problems with very large polygons (more than about 1000 points). The workaround is to subdivide the large polygon into several smaller ones.

You cannot combine the polygon picture comments with other comments such as the rotation comments or the `DashedLine` comment. It's just another limitation--no comment.

Back to top

Dashed Lines

Comments: `DashedLine`, `DashedStop`

PostScript allows applications to draw precisely dashed lines with a given dash pattern in every direction (see the `setdash` operator, *PostScript Language Reference Manual,* Second Edition, page 500). The QuickDraw ersatz of setting the pen pattern appears to be awkward at best; the result depends very much on the direction of the line. Coding correctly dashed lines in QuickDraw is quite a hassle and rather clumsy. This is why the `DashedLine` and `DashedStop` picture comments have been provided for applications where dashed lines are important and used frequently. Applications can take advantage of these comments when printing to a PostScript printer.

The `DashedLine` comment tells the driver that the line drawing instructions following the comment should be dashed

according to the parameters in the TDashedLine structure (see the Appendix). These parameters closely correspond to the parameters passed to the PostScript setdash operator. Only the centered field of the TDashedLine structure is not currently supported by the LaserWriter driver. It should be set to 0 in case support for centering is added in the future.

Unlike the picture comments for text rotation or even polygon smoothing, the DashedLine picture comment should **not** be supported by a non-PostScript driver. The only way to include representations of dashed lines with and without usage of the DashedLine picture comment is to make the following assumption: If the DashedLine comment is supported, then the printer is a PostScript printer, and the PostScriptBegin/PostScriptEnd bracket may be used to hide the QuickDraw imaging from the printer. Remember that non-PostScript printer drivers must **not** ignore QuickDraw imaging within PostScriptBegin and PostScriptEnd!

But we still need a trick to hide the line drawing instructions within the DashedLine - DashedStop bracket from QuickDraw. Here comes the "magic pen mode" to our rescue:

```
PROCEDURE DashDemo;

   CONST
      magicPen = 23; { the infamous penMode ! }
      cx = 280;
      cy = 280;
      r0 = 200;

   VAR
      dashHdl: TDashedLineHdl;
      i: Integer;
      a, rad : Extended;

   BEGIN
      PenSize(2,2);
      { First the PostScript picture comment version.  }
      { The "magic pen mode" 23 makes the line drawing invisible for QuickDraw. }
      PenMode(magicPen);
      dashHdl := TDashedLineHdl(NewHandle(SizeOf(TDashedLineRec)));
      IF dashHdl <> NIL THEN
         WITH dashHdl^^ DO BEGIN
            offset := 4;       { just for fun }
            centered := 0;     { Currently ignored - set to 0. }
            intervals[0] := 2; { number of interval specs      }
            intervals[1] := 4; { This means 4 points on ...    }
            intervals[2] := 6; { ... and 6 points off.         }
            PicComment(DashedLine, SizeOf(TDashedLineRec), Handle(dashHdl));
         END;
      rad := 3.14159 / 180;    { Conversion degrees -> radians. }
      FOR i := 0 TO 9 DO BEGIN { Draw some dashed lines. }
         a := i * 20 * rad;
         MoveTo(cx, cy);
         Line(round(r0 * cos(a)), - round(r0 * sin(a)));
      END;
      PicComment(DashedStop, 0, NIL); { That's enough! }
      DisposHandle(Handle(dashHdl));
      PenMode(srcOr);  { No magic any more. }

      { Now, the QuickDraw version. The PostScript driver must ignore it, }
      { so we enclose it between PostScriptBegin and PostScriptEnd comments.}
      PicComment(PostScriptBegin, 0, NIL);
      PenSize(2,2);
      FOR i := 0 TO 9 DO BEGIN
         MoveTo(cx,cy);
         DashedQDLine(round(r0 * cos(i * 20 * rad)),
                     - round(r0 * sin(i * 20 * rad)), dashHdl);
      END;
      PicComment(PostScriptEnd,0,NIL);
```

```
    END;
```

By the way: The DashedQDLine procedure is intentionally missing. It's not precisely the subject of this Note, and thus, again, is left as a spare-time exercise for the reader.

Caveat

As mentioned earlier, the current version of the PostScript LaserWriter driver produces poor results when the DashedLine picture comment is applied to polygons. Just don't do it!

Back to top

Fractional Line Width

Comment: SetLineWidth

QuickDraw's design is based on a fixed 72-dpi resolution. Even when printing to a high-resolution device, the Printing Manager presents the printing grafPort, corresponding to the printable area of the page, in the integer-valued QuickDraw coordinate system with 72 dpi. Applications can use PrGeneral to image at higher device resolutions (see *Inside Macintosh* Volume V, page 410), but this is useful mainly for immediate printing. As a consequence, lines are usually always at least 1/72 inch wide, corresponding to the smallest pen size (1,1). For a 300-dpi device like the LaserWriter, this is disappointing.

The SetLineWidth comment allows an application to set the width of a line to any fractional value. A value of 1/4 approximately corresponds to a "hairline" on a 300 dpi LaserWriter. Curiously (but conveniently), a QuickDraw Point structure is passed in the PicComment's data handle, the vertical coordinate representing the denominator, and the horizontal coordinate the numerator of the fraction.

Unfortunately, it is not implemented in all high-resolution QuickDraw printers; and if it is (as in the LaserWriter SC), it works differently than in PostScript printer drivers. Moreover, there is no possibility to include alternative imaging instructions in case SetLineWidth is not supported. While this is not much of a loss for hairlines, it prevents us from using the comment for fractional widths > 1, where the alternative would be to include a PenSize call with rounded arguments. Another drawback may be that, allegedly, there are plotter drivers out there that abuse this comment to set the pen color--clearly an unpleasant situation.

The difference in the implementation of the SetLineWidth comment between the PostScript LaserWriter driver and the LaserWriter SC appears as soon as SetLineWidth is used for the second time. The PostScript driver keeps an internal line scaling factor; this factor is initialized to 1.0 when a job is started. Each number passed through SetLineWidth is multiplied by the current internal scaling factor to get the effective scaling factor for the pen size. The LaserWriter SC driver, on the other hand, replaces its current scaling factor for the pen size completely by the new value passed through SetLineWidth. In order to support both implementations, you must always use an additional SetLineWidth step in order to reset the PostScript driver line width to 1.0, before scaling to the new value.

Example

Let's say you have set the line width to 0.25, and want to replace it by a line width of 0.5. The following **two** SetLineWidth comments will have the desired effect on both PostScript (PS) and QuickDraw (QD) drivers that implement the SetLineWidth comment. You don't care about the temporary line width of 4.0 on the QuickDraw driver.

Current Line Width		Parameter Passed	New Line Width	
PS driver	QD driver	in SetLineWidth	PS Driver	QD Driver
0.25	**0.25**	4/1	1.0	4.0
1.0	4.0	1/2	**0.5**	**0.5**

The following sample code gives the expected results only on a PostScript LaserWriter and on QuickDraw printer drivers that have the SetLineWidth comment implemented.

```
PROCEDURE SetNewLineWidth(oldWidth,newWidth: TLineWidth);

   VAR
      tempWidthH: TLineWidthHdl;

   BEGIN
      tempWidthH := TLineWidthHdl(NewHandle(SizeOf(TLineWidth)));
      { If tempWidthH = NIL we are screwed anyway. }
      tempWidthH^^.v := oldWidth.h;
      tempWidthH^^.h := oldWidth.v;
      PicComment(SetLineWidth,SizeOf(TLineWidth),Handle(tempWidthH));
      tempWidthH^^ := newWidth;
      PicComment(SetLineWidth,SizeOf(TLineWidth),Handle(tempWidthH));
      DisposHandle(Handle(tempWidthH));
   END;


PROCEDURE LineWidthDemo;

   CONST
      y0 = 50;  { topleft of demo }
      x0 = 50;
      d0 = 440; { length of horizontal lines }
      e0 = 5;   { distance between lines }
      kN = 5;   { number of lines }
   VAR
      oldWidth,newWidth: TLineWidth;  { actually a "Point" }
      i,j,y: Integer;

   BEGIN
      PenNormal;
      y := y0;
      SetPt(oldWidth,1,1);            { initial linewidth = 1.0 }
      FOR i := 1 TO 5 DO BEGIN
         SetPt(newWidth,4,i);
                  { want to set it to i/4 = 0.25, 0.50, 0.75 ... }
         SetNewLineWidth(oldWidth,newWidth);
         MoveTo(x0, y);
         Line(d0, 0);
         y := y + e0;
         oldWidth := newWidth;
      END;
   END;
```

***A Slight Imperfection**

- If you experiment with the above code and draw a whole series of hairlines, you will see (depending on the values of `e0` and `kN`) that certain lines appear thicker than they should be. This is due to rasterization effects in PostScript's scan conversion algorithm when the line width is close to the device pixel size. In many cases, the PostScript LaserWriter driver tries to compensate for this by rounding coordinates to the 300-dpi grid. If you include `SetLineWidth` (or, by the way, `DashedLine`) picture comments, however, this does not work. PostScript Level 2 addresses this problem by means of an optional *stroke adjustment* feature (see the *PostScript Language Reference Manual,* Second Edition, pages 322 and 515).

Back to top

Graphics Rotation

Comments: `RotateBegin`, `RotateCenter`, `RotateEnd`

Like the picture comments discussed earlier in this Note in the section "Text Rotation," the graphics rotation picture

comments provide a method of rotating QuickDraw objects on PostScript devices. Instead of having QuickDraw perform the rotation, the printer driver rotates the entire PostScript coordinate space so that **everything** drawn between RotateBegin and RotateEnd will be rotated on the printer itself. This includes text drawing! You specify the center of rotation with RotateCenter and the angle of the rotation, together possibly with horizontal or vertical flipping, through the TRotation record (see the interface definitions in the Appendix).

Unlike text rotation, you must insert the RotateCenter comment and pass the relative offset to the center of rotation **before** you use the RotateBegin picture comment. The point passed to RotateCenter specifies the offset from the anchor point of the first object drawn **after** RotateBegin to the desired center of rotation. Once you set up the rotation parameters with RotateCenter and RotateBegin, you can draw the graphics objects you want to rotate.

Bad news: In order to include a QuickDraw representation of the rotated objects in case the rotation comments are not supported, we have to assume (again) that **only PostScript** drivers implement these comments. The only way to hide the QuickDraw substitute from the driver is to surround it by PostScriptBegin and PostScriptEnd comments; and, similarly to the DashedLine comment, we need to use the "magic pen mode" (23) to hide the unrotated drawing between RotateBegin and RotateEnd from QuickDraw. The following sample demonstrates this:

```
PROCEDURE QDRotatedRect(r: Rect; ctr: Point; angle: Integer);
   BEGIN
      { An exercise again - this one is easy ...             }
      { Rotates the four points of the rectangle by "angle"  }
      { around the center obtained by adding the point "ctr" }
      { as offset to r.topLeft, and draws the rotated Rect.  }
   END;


PROCEDURE PSRotatedRect(r: Rect; offset: Point; angle: Integer);
{ Does the rectangle rotation for the PostScript LaserWriter driver. }
{ Uses the RotateCenter, RotateBegin and RotateEnd picture comments, }
{ and the "magic" pen mode 23 to hide the drawing from QuickDraw.    }

   CONST
      magicPen = 23;

   VAR
      rInfo: TRotationHdl;
      rCenter: TCenterHdl;
      oldPenMode: Integer;

   BEGIN
      rInfo := TRotationHdl(NewHandle(SizeOf(TRotationRec)));
      rCenter := TCenterHdl(NewHandle(SizeOf(TCenterRec)));
      IF (rInfo = NIL) OR (rCenter = NIL) THEN DebugStr('NewHandle failed');

      WITH rInfo^^ DO BEGIN
         rFlip := 0;
         rAngle := - angle;
         rAngleFixed := BitShift(LongInt(rAngle),16);
      END;

      WITH rCenter^^ DO BEGIN
         x := Long2Fix(offset.h);
         y := Long2Fix(offset.v);
      END;

      MoveTo(r.left,r.top);
      FlushGrafPortState;
      PicComment(RotateCenter,SizeOf(TCenterRec),Handle(rCenter));
      PicComment(RotateBegin,SizeOf(TRotationRec),Handle(rInfo));

      oldPenMode := thePort^.pnMode;
      PenMode(magicPen);
      FrameRect(r);
```

```
      PenMode(oldPenMode);

      PicComment(RotateEnd,0,NIL);

      DisposeHandle(Handle(rInfo));
      DisposeHandle(Handle(rCenter));
   END;

PROCEDURE RotateDemo;

   CONST
      angle = 30;

   VAR
      spinRect: Rect;
      delta: Point;

   BEGIN
      SetRect(spinRect,100,100,300,200);
      WITH spinRect DO SetPt(delta,(right - left) DIV 2,(bottom - top) DIV 2);

      PenSize(2,2);
      PenPat(ltGray);
      FrameRect(spinRect); { show the unrotated square }
      PenNormal;

      PSRotatedRect(spinRect,delta,angle);

 { QuickDraw equivalent of the rotated object, hidden from PostScript driver  }
 { because of PostScriptBegin and PostScriptEnd }

      PicComment(PostScriptBegin,0,NIL);
      QDRotatedRect(spinRect,delta,angle);
      PicComment(PostScriptEnd,0,NIL);

   END;
```

Back to top

PostScript Comments

Comments: PostScriptBegin, PSBeginNoSave, PostScriptEnd, PostScriptHandle

The PostScript comments tell the picture interpreter (usually the LaserWriter driver) that the application is going to communicate with the LaserWriter directly using PostScript code instead of QuickDraw. All QuickDraw drawing instructions between the PostScriptBegin and PostScriptEnd picture comments are ignored. The driver sends the PostScript text contained in the PostScriptHandle data to the printer with no preprocessing and no error checking. When the application is finished sending PostScript, the PostScriptEnd comment tells the printer driver to resume normal QuickDraw mode. The driver uses the PostScript save and restore operators to preserve the state of the PostScript interpreter across the section enclosed by PostScriptBegin and PostScriptEnd. Some applications do not want to restore the previous state of the PostScript interpreter after including their PostScript code; for these situations, the PSBeginNoSave comment is a replacement for PostScriptBegin that does not preserve the state. Clearly, this comment should be used with extreme caution.

Some state information may be stored in global variables, so nesting PostScriptBegin (or PSBeginNoSave) and PostScriptEnd comments is not allowed.

The PostScriptHandle comment gives developers direct access to PostScript from applications. Instead of having the LaserWriter driver convert QuickDraw calls into the corresponding PostScript code, the application can generate its own PostScript, and transmit it to the printer or include it in a picture through the data handle of the PicComment procedure. The handle contains pure ASCII text; the valid length of the data is specified in the PicComment's size

parameter. Don't forget to terminate the PostScript text at least with a space character, or better with a carriage return (ASCII $0D), so that it is separated from the following PostScript instructions (either yours, or the printer driver's).

You must still use PostScriptBegin (or PSBeginNoSave) and PostScriptEnd around PostScriptHandle comments or the LaserWriter driver will not properly save and restore the PostScript drawing environment.

As with all picture comments, the handle you pass belongs to you and you must dispose of it when you're finished with it.

```
PROCEDURE PostScriptLine(s: Str255);
{ A utility procedure to transmit a string of PostScript code through }
{ the PostScriptHandle picture comment to the PostScript printer.     }
{ It should be called only between PostScriptBegin and PostScriptEnd  }
{ picture comments. }

   VAR
      h: Handle;

   BEGIN
      h := NewHandle(256);
      IF h = NIL THEN DebugStr('NewHandle failed');
      BlockMove(@s[1],h^, Length(s));
      PicComment(PostScriptHandle, Length(s), h);
      h^^ := 13;
      PicComment(PostScriptHandle, 1, h); { add a carriage return }
      DisposeHandle(h);
   END;


PROCEDURE PostScriptComments;

   BEGIN
      { First, the simple example: }
      PicComment(PostScriptBegin,0,NIL);
      PostScriptLine('100 100 moveto 0 100 rlineto 100 0 rlineto ');
      PostScriptLine('0 -100 rlineto -100 0 rlineto');
      PostScriptLine('stroke');
      MoveTo(30,30);
      DrawString('This text does not appear on PostScript devices');
      PicComment(PostScriptEnd,0,NIL);

      { Now, a new PostScript definition you want to keep in the     }
      { userdict. If you used PostScriptBegin, the definition would  }
      { be lost when PostScriptEnd is encountered, because the state }
      { previous to the PostScriptBegin comment would be restored.   }
      PicComment(PSBeginNoSave,0,NIL);
      PostScriptLine('userdict begin');
      PostScriptLine('/myFrameRect {');
      PostScriptLine('250 250 moveto 0 100 rlineto');
      PostScriptLine('200 0 rlineto 0 -100 rlineto -200 0 rlineto ');
      PostScriptLine('stroke } def');
      PostScriptLine('end');
      PicComment(PostScriptEnd,0,NIL);

      { Let's test to see if the definition from above is still avail-}
      { able. This assumes that no font downloading has occurred.     }

      PicComment(PostScriptBegin,0,NIL);
      PostScriptLine('//userdict /myFrameRect get exec ');
      PicComment(PostScriptEnd,0,NIL);
   END;
```

Caveat

If you choose to use PostScript directly in your pictures, be very careful not to make assumptions about Apple's "md" dictionary (essentially the contents of the former LaserPrep file). Otherwise, your pictures will not print correctly with future versions of the PostScript LaserWriter driver. Also, be aware of compatibility problems within the PostScript world, and watch out for printers with PostScript Level 1 and PostScript Level 2 interpreters, and "PostScript-compatible" printers (PostScript clones).

Back to top

FormsPrinting Picture Comments

Comments: FormsPrinting, EndFormsPrinting

The FormsPrinting comment tells the PostScript LaserWriter driver not to clear its page buffer after printing a page. EndFormsPrinting turns this mode off. When the page is completed, the application must erase the areas that need to be updated and draw the new information. The graphics that make up the form are drawn only once per page, which may improve performance. Currently, you need to write special printing code for the PostScript LaserWriter driver if you want to use this comment.

Back to top

(More or Less) Obsolete PostScript Picture Comments

Comments: SetGrayLevel, TextIsPostScript, ResourcePS, PostScriptFile

The SetGrayLevel picture comment was designed to provide access to the PostScript setgray operator while still drawing with QuickDraw in black-and-white mode. In practice, this turned out to be not so useful, however. For most drawing operations, the printer driver sets the gray level to match the foreground color currently stored in the printing grafPort, and the effect of the SetGrayLevel comment is often unpredictable. If direct access to the PostScript setgray operator seems nevertheless desirable, it is easy to include the instruction in a PostScriptHandle comment.

The TextIsPostScript picture comment takes all the text coming through standard QuickDraw text drawing calls (DrawChar, DrawString, DrawText, and anything else that eventually calls the StdText bottleneck), and interprets it as a PostScript program. There is no good reason to use this picture comment, but there is one important reason not to use it: Printer drivers that do not deal with the TextIsPostScript comment will print the PostScript text instead of interpreting it! If you need to transmit pure PostScript code directly to a printer that understands it, use the PostScriptHandle comment, and include a QuickDraw representation for all other printer drivers.

The ResourcePS picture comment loads PostScript code from a specified resource. The resource file is expected to be open at the time that the ResourcePS comment is used. Under background printing, there are no guarantees the file will still be open when the Printing Manager needs it. For this reason alone, you should forget about this comment. If you want to keep PostScript instructions in a resource, it is easy to write a small routine that loads the resources and sends their contents using the PostScriptHandle comment described earlier in this Note.

PostScriptFile has the same problems as ResourcePS described above. Basically, the Printing Manager cannot guarantee that the file will be available when it's needed.

Back to top

Appendix: Pascal Interface for Picture Comments

(File PicComments.p)

```
CONST
   TextBegin = 150;
   TextEnd = 151;
   StringBegin = 152;
   StringEnd = 153;
   TextCenter = 154;
```

```
   LineLayoutOff = 155;
   LineLayoutOn = 156;
   ClientLineLayout = 157;
   PolyBegin = 160;
   PolyEnd = 161;
   PolyIgnore = 163;
   PolySmooth = 164;
   PolyClose = 165;
   DashedLine = 180;
   DashedStop = 181;
   SetLineWidth = 182;
   PostScriptBegin = 190;
   PostScriptEnd = 191;
   PostScriptHandle = 192;
   PostScriptFile = 193;
   TextIsPostScript = 194;
   ResourcePS = 195;
   PSBeginNoSave = 196;
   SetGrayLevel = 197;
   RotateBegin = 200;
   RotateEnd = 201;
   RotateCenter = 202;
   FormsPrinting = 210;
   EndFormsPrinting = 211;

   tJusNone = 0; { values for the tJus field of the TTxtPicRec record }
   tJusLeft = 1;
   tJusCenter = 2;
   tJusRight = 3;
   tJusFull = 4;

   tFlipNone = 0;         { values for the tFlip field of the TTxtPicRec record }
   tFlipHorizontal = 1;
   tFlipVertical = 2;

TYPE
   TTxtPicHdl = ^TTxtPicPtr;
   TTxtPicPtr = ^TTxtPicRec;
   TTxtPicRec = PACKED RECORD
                   tJus  : Byte;
                   tFlip : Byte;
                   tAngle: Integer;     { clockwise rotation in degrees 0..360 }
                   tLine : Byte;        { Unused/Ignored }
                   tCmnt : Byte;        { reserved }
                   tAngleFixed: Fixed;  { same as "tAngle" in Fixed precision  }
                END; { TTxtPicRec }

   TRotationHdl = ^TRotationPtr;
   TRotationPtr = ^TRotation;
   TRotationRec = RECORD
                    rFlip: Integer;
                    rAngle: Integer;    { clockwise rotation in degrees 0..360 }
                    rAngleFixed: Fixed; { same as "rAngle" in Fixed precision  }
                  END; { TRotationRec }

   TCenterHdl = ^TCenterPtr;
   TCenterPtr = ^TCenter;
   TCenterRec = RECORD  {offset from current pen location to center of rotation}
                   y: Fixed;
                   x: Fixed;
                END; { TCenterRec }


   TPolyVerbHdl = ^TPolyVerbPtr;
```

```
   TPolyVerbPtr = ^TPolyVerbRec;
   TPolyVerbRec = PACKED RECORD
                     f7,f6,f5,f4, f3,     { reserved }
                     fPolyClose,          { TRUE = smoothing across endpoint.}
                     fPolyFill,           { TRUE = Polygon should be filled. }
                     fPolyFrame: BOOLEAN; { TRUE = Polygon should be framed. }
                  END;

   TDashedLineHdl = ^TDashedLinePtr;
   TDashedLinePtr = ^TDashedLineRec;
   TDashedLineRec = PACKED RECORD
                     offset   : SignedByte;  { offset into pattern for first dash }
                     centered : SignedByte;  { (Ignored)                           }
                     intervals: ARRAY [0..5] { Array of dash intervals }
                                OF SignedByte; { intervals[0] = number }
                    END;                     { of interval specs.    }

   TLineWidthHdl = ^TLineWidthPtr;
   TLineWidthPtr = ^TLineWidth;
   TLineWidth    = Point;  { v = numerator, h = denominator. }

   TClientLLHdl = ^TClientLLPtr;  { used in the ClientLineLayout picture comment }
   TClientLLPtr = ^TClientLLRec;
   TClientLLRec = RECORD
                     chCount : Integer;  { Apply for so many characters. }
                     major   : Fixed;    { percentage of line layout error to be }
                                         { distributed among space characters.   }
                     spcChar : Integer;  { code of character that is to absorb   }
                                         { the "major" line layout error         }
                     minor   : Fixed;    { percentage of intercharacter distrib. }
                     ulLength: Fixed;    { underline length.                     }
                  END;
```

References

PostScript Language Reference Manual , Adobe Systems Inc.

Inside Macintosh , Volumes II, V, and VI

LaserWriter Reference Manual , Addison-Wesley

Macintosh Technical Note M.IM.AppPictComments -- Every Picture [Comment] Tells Its Story, Don't It?

Macintosh Technical Note M.IM.gifAndPrinting -- Pictures and the Printing Manager

develop Issue 3, "Meet PrGeneral" by Pete "Luke" Alexander

Back to top

Downloadables

Acrobat version of this Note (K). Download

Technical Notes by Date | Number | Technology | Title
Developer Documentation | Technical Q&As | Development Kits | Sample Code