New Technical Notes
Macintosh

®

Developer Support

Picture Comments—The Real Deal

Imaging M.IM.PictComments

Revised by: Scott “Zz” Zimmerman, Dave Hersey, Matt Deatherage, and Joseph Maurer July 1992
Written by: Ginger Jernigan November 1986

Changes since March 1988: 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.

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. Technical Note #181, “Every Picture [Comment] Tells Its Story, Don’t it?” documents how applications can define their own types of picture comments without conflicting with other applications’ definitions.

The `QDProcs` record also is the key to understanding how Macintosh printer drivers work. When the application calls `PrOpenPage` and draws into the printing port, 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 port, the printer driver gets a chance to capture 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 the standard QuickDraw calls, like rotated text, rotated graphics, dashed lines, fractional line widths, and special PostScript operations on 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. This is how the list of picture comments was created (see Table 1). They 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 pieces of original string
	StringEnd	153	0	NIL	End pieces of original string
	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	
	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 PrClosePage

† 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 (more or less private) picture comments, or support some of the above in order to provide additional capabilities. Thirdparty printer drivers might implement rotation of text and bitmaps even for printers without PostScript, for example.

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

Even the ImageWriter driver, which does not care about `LineLayoutOff`, `LineLayoutOn`, and `SetLineWidth`, reacts to picture comments:

BitMapThinningOff	1000	0	NIL	Turn off hiRes bitmap thinning
BitMapThinningOn	1001	0	NIL	Turn on hiRes bitmap thinning,

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

The point of all this is

It is impossible to determine which picture comments have been implemented by which printer driver.

In other words: Your application should never assume a particular picture comment is available in the current printer driver.

How can you then take advantage of the features provided?

Fortunately, there is a solution, at least for most of the picture comments directly related to PostScript. Your application must generate the code required to perform the operation with the picture comments and without, and the selected printer driver determines the correct representation to use.

It's important to always send **both** representations to **any** printer driver—future system software may allow spool files to be redirected to a printer other than the one chosen when you sent your picture comments. This is also why DTS does not like the idea of determining the type of printer driver by looking at the high byte of the `prStl .wDev` field of the print record. As documented in *Inside Macintosh* Volume II, page 152, the PostScript LaserWriter printer drivers have a `wDev` value of $03, and some applications use this information to send only the PostScript representation to the printer. Although the `wDev` field will continue to be supported for some time (for obvious compatibility reasons), it's usually preferable not to rely upon it.

In this Note, we first discuss the available techniques to include both a QuickDraw and a PostScript representation in a picture, or in the imaging instructions sent to a printing port. This leads us quite naturally to some general caveats. We then present two procedures to synchronize the QuickDraw and PostScript graphic states. This is sometimes required when

PostScript picture comments are intertwined with a sequence of QuickDraw calls. Finally, we go through the individual picture comments by subject, as suggested by Table 1, describe their effects, demonstrate how to use them correctly, and explain the potential problems with them.

Cohabitation of QuickDraw and PostScript

Device-Independent Pictures

We can think of the Printing Manager's `PrOpenPage` and `PrClosePage` calls as being equivalent 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. As recommended above, we want to create pictures that include both QuickDraw and optimized PostScript representations so that we obtain best results under all circumstances.

The two picture comments `PostScriptBegin` and `PostScriptEnd` clearly suggest that any imaging instructions in between are intended exclusively for PostScript printing devices (or, potentially, for printer drivers endeavoring to emulate some PostScript features). 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 port when enclosed by `PostScriptBegin` and `PostScriptEnd`. Obviously, this is precisely what we need to hide the QuickDraw representation from a PostScript printing device: The QuickDraw representation of the graphics will appear only on printers whose driver does not understand `PostScriptBegin` and `PostScriptEnd`.

So we are left with the opposite problem: how to prevent QuickDraw printers (or the screen) from displaying the effect of drawing instructions intended for a PostScript device exclusively. This can be tricky, and will be addressed on a case-by-case basis in the discussion of the various picture comment families.

Basically, there are two ways:

• Under certain circumstances (like between the `TextBegin` and `TextEnd` picture comments), the PostScript driver ignores the QuickDraw clip region. This helps for rotated text, which has to be communicated to the PostScript driver through a standard QuickDraw text drawing command like `DrawString`. Setting the clip region to an empty rectangle prevents the QuickDraw text from appearing.

• For drawing operations that involve the GrafPort's `pnMode`, 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 printer 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;

using it was considered a compatibility risk (and frowned upon) for some time. Given the current state of affairs, however, it’s the only viable way for certain picture comments to include both QuickDraw and PostScript representations in the same picture.

General Caveats (for those who would like to stop reading now)

The picture comments `PostScriptFile` and `ResourcePS` do not work with background printing. They are considered obsolete and should not be used.

There is no good reason at all to use the `TextIsPostScript` picture comment. Use the `PostScriptHandle` comment instead.

The picture comments `ClientLineLayout`, `SetGrayLevel`, `FormsPrinting`, and `EndFormPrinting` are not recommended (for various reasons). It seems extremely unlikely that you'll encounter situations where you really need to use them.

Nesting of picture comments, or combining them beyond their primary intention, is not supported. This means, for example, that you can't use the polygon comments in conjunction with the rotation comments to draw a rotated polygon (instead, rotate the points of the polygon before drawing). Similarly, the `DashedLine` picture comment behaves very poorly with polygons (don't even try it).

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 bear the risk of not printing correctly with future versions of the PostScript LaserWriter driver. Also, be aware of compatibility problems within the PostScript world, and test your application with as many different PostScript devices as possible. In particular, watch out for printers with PostScript Level 1 and PostScript Level 2 interpreters, and "PostScript-compatible" printers (PostScript clones).

Flushing the Port State and Flushing PostScript

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 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 procedure. Pen size and pen location are   }
{ preserved so that there are no side effects.                           }

  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 clip 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.

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 that are likely to be used with PostScript printers (like desktop publishing and advanced drawing applications), but don't want to use PostScript explicitly. QuickDraw does not support rotated or flipped text, and you must provide a bitmap representation of the rotated/flipped text as a fallback solution in case the printer driver does not support these picture comments.

Let's look at sample code right away. Please, consult the Appendix "Pascal Interface for Picture Comments" for the definition of the structures used in the `TextBegin` and `TextCenter` comments.

```
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;
{ QDStringRotation provides a QuickDraw substitute for the PostScript feature. }
{ May contain any QuickDraw imaging, except picture comments. }
{ Left as an exercise for the reader ... }

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 printed to a PostScript device, the rotation is done by the PostScript   }
{ interpreter; if the printer driver does not recognize the PostScriptBegin   }
{ and PostScriptEnd picture comments, the external procedure QDStringRotation }
{ is used 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 }

     { This is for non-PostScript printers: }
     { ---------------------------------- }
     PicComment(PostScriptBegin,0,NIL);
     QDStringRotation(s, ctr, just, flip, rot);
     PicComment(PostScriptEnd,0,NIL);

     { The following is for PostScript printers only: }
     { ------------------------------------------- }
     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 counter-clockwise 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 }

     { Hide the following DrawString from QuickDraw }
     oldClip := NewRgn;
     GetClip(oldClip);
     SetRect(zeroRect,0,0,0,0);
     ClipRect(zeroRect);
     { The PostScript driver ignores clipping between TextBegin and TextEnd  }
     DrawString(s); { in the rotated PostScript environment }
     ClipRect(oldClip^^.rgnBBox);

     PicComment(TextEnd,0,NIL);
     { Set PostScript's 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 QuickDraw representation is hidden from PostScript by means of `PostScriptBegin` and `PostScriptEnd`, and the

PostScript representation is hidden from QuickDraw by setting the clip region to empty (ignored by the PostScript LaserWriter driver between `TextBegin` and `TextEnd`).

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 "mentally" 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.

• If you don't insert the QuickDraw representation (surrounded by the `PostScriptBegin`/ `PostScriptEnd` picture comments) **before** the section with `TextBegin - TextCenter - TextEnd`, the effect of ignoring clip regions might be propagated to preceding sections of your drawing instructions because of the internal buffering of generated PostScript code. In this case, you need to call the `FlushPostScriptState` procedure described earlier 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 (corresponding to the values `tJusLeft`, `tJusRight`, and `tJusCenter`), without recalculating the interword and intercharacter spacing. Using `tJusFull` justification specifies that the original length of the string (on the QuickDraw screen) must be maintained. This is important when rotating a fully justified block 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 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.

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) 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 “hardcoded” 72 dpi screen resolution, the width of text rendered on the printer is not the same as the text width on the screen. This is due to nonproportionally scaling bitmap 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.

There are situations, however, where the printer driver’s line layout algorithm has effects that do not meet the intentions of the application programmer. Music applications that draw notes or other music symbols using characters from a font (like Sonata) definitely don’t want to have the printer driver take care of line layout! Similarly, printing mathematical formulas and equations requires precise placement of each symbol rather than automatic line layout adjustment. Examples in less artistic environments include tables with columns of vertically aligned text entries (or formatted programming source code . . .); they better not be submitted to the printer driver’s zealous word shifting for line layout purposes.

The `LineLayoutOff` picture comment turns much of the printer driver’s line layout adjustment off. Text will be printed using the default character and word spacing as built into the font used by the printer, regardless of differences with the original font used on the screen. This allows the application to better control the placement of words and characters on a line—at least, in principle. Not so surprisingly, the `LineLayoutOn` picture comment is meant to reactivate the printer driver’s line layout algorithm.

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% versus 20%). The “minor” part is distributed across intercharacter spacing.

Only very ambitious page layout applications might be interested in this functionality; but usually, they prefer designing their own line layout scheme and generating their own PostScript code. It is conceivable that applications for non-Roman scripts with a word delimiter different from ASCII $20 might want to use this picture comment, but then again, 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. Feel free to experiment with it, by including the `ClientLineLayout` picture comment in the sample code below. Note, however, that we do not recommend that you use this picture comment in your application.

The following piece of code can be used to observe the line layout activity of a printer driver. For noticeable effects, choose fonts such that the printer driver will substitute printer fonts with different character widths (like New York), or bitmap fonts with character widths nonproportional to the point size (like bitmap Courier), and change the font sizes. A printer driver has no line layout problems with TrueType fonts, unless it has the same name (and different character widths) as a printer-resident PostScript font. You may also want to add instructions like `SetFractEnable(TRUE)` and compare the results for different versions of the LaserWriter driver (see "Caveats" below).

```
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. }

     MoveTo(x0, y);           { Here is the default behavior. }
     DrawString(testString1);
     MoveTo(x0 + w, y);
     DrawString(testString2);
     y := y + h;

     PicComment(LineLayoutOff, 0, NIL);

{ ••• (1) - see below, under "String Delimitation" •••}
     MoveTo(x0, y);
     DrawString(testString1);
     MoveTo(x0 + w, y);
{ ••• (2) - see below, under "String Delimitation" •••}

     FlushGrafPortState;  {••• Try with and without! •••}

     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”). Each line is drawn in two pieces (`testString1` and `testString2`). Line layout is turned off before the second line, and turned on again for the third line.

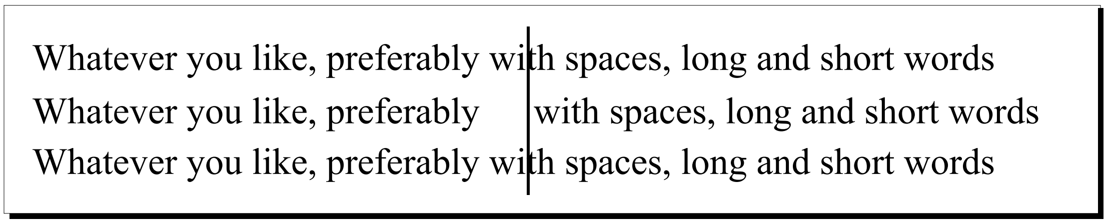
Whatever you like, preferably wi|th spaces, long and short words
Whatever you like, preferably |with spaces, long and short words
Whatever you like, preferably wi|th spaces, long and short words

Figure 1 With `FlushGrafPortState`

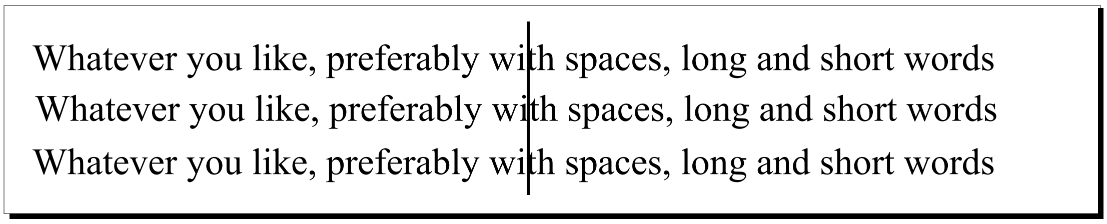
Whatever you like, preferably wi|th spaces, long and short words
Whatever you like, preferably wi|th spaces, long and short words
Whatever you like, preferably wi|th spaces, long and short words

Figure 2 Without `FlushGrafPortState`

The PostScript LaserWriter driver substitutes Times for New York. But New York (on the screen) is larger than Times, and the driver distributes the line layout error among the word spacings and, to a much lesser extent, among the intercharacter spacing. Note that the beginning of the `testString2` ("with spaces, ...") is slightly moved to the left in lines 1 and 3. The LaserWriter driver heuristically concludes that `testString1` and `testString2` belong to the same line, and applies the distribution of the line layout error to the concatenated string. It appears that the driver deliberately ignores the `MoveTo(x0+w,y)` instruction; figure 2 demonstrates this quite clearly in the second line, where the `LineLayoutOff` picture comment has been issued. `LineLayoutOff` does **not** disable the driver's attempt to collect the arguments of separate text drawing instructions into one logically coherent piece of text. This behavior is necessary to assure that indices or exponents in the text appear at the right horizontal position. Figure 1 shows that `FlushGrafPortState` after a `MoveTo` always guarantees precise text placement.

Caveats

• As demonstrated above, `LineLayoutOff` does **not** guarantee precise text placement, as you might expect. The driver just stops to preserve the text width as measured on the screen, and uses the default character widths of the printer font. In practice, this means that even when placing characters or words separately, the LaserWriter driver may shift them horizontally because it *supposes* that they are part of the same text run, and prints them as contiguous text. Flushing the GrafPort state after each `MoveTo` guarantees precise placement. Unfortunately, this creates a lot of overhead for all printer drivers, and affects their performance.

• 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.

• The previous version of this Note said that setting the Font Manager's `FractEnable` global to `TRUE` has the same effect as sending the `LineLayoutOff` picture comment. As it turned out, the statement was based on observations with a specific (older) version of the LaserWriter driver, under specific circumstances, 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. If the comment isn't recognized, the picture renderer will still be able to place your text as well as it would have without the comment. It's mainly useful 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, like in forms or tabulated text, and understand how to achieve this.

• Regarding the first "caveat" above, you must **not** necessarily call `FlushGrafPortState` if you want to turn `LineLayout` completely `OFF`, in placing characters or words separately on a line. There is a better solution: Read on!

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`). But even when there is only one `DrawString` between `StringBegin` and `StringEnd`, it is important to notify the printer driver that it should consider the string as an independent entity. Otherwise, it will continue to perform its heuristic accumulation of other 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.

If we insert

```
    PicComment(StringBegin,0,NIL);
```

in the ObserveLineLayout procedure above, where indicated by {••• (1) ... }, and

```
    PicComment(StringEnd,0,NIL);
```

where indicated by {••• (2) ... }, and remove the call to FlushGrafPortState, we will obtain the same output as in Figure 1. In other words, these picture comments are what you need to turn the LaserWriter driver's line layout behavior off, and completely so!

Of course, there is still a slight overhead tied to inserting the StringBegin and StringEnd picture comments, but it is much smaller than calling FlushGrafPortState, and it is restricted to where it belongs (that is, the PostScript LaserWriter driver), and does not affect all other printer drivers.

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 following line drawing is ignored by the PS driver. }
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 Points, 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 call 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 array of 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 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;
```

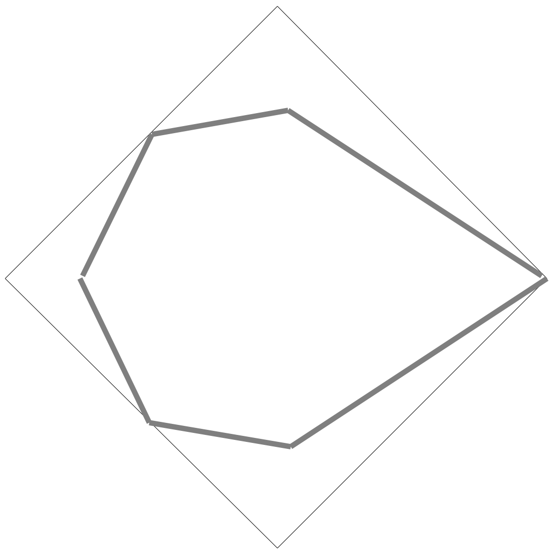
Figure 3 Output on QuickDraw Printer

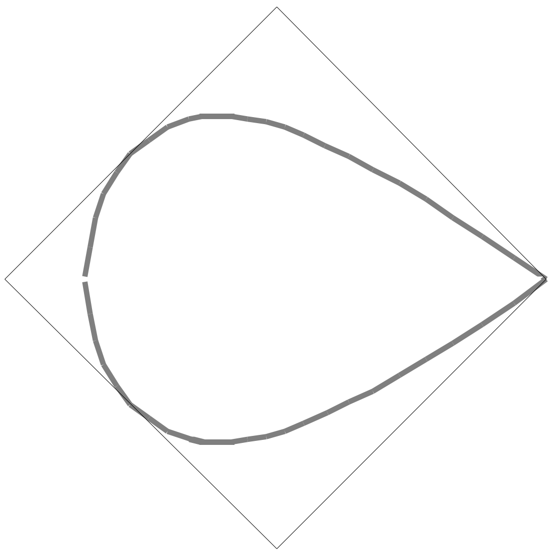
Figure 4 Output on PostScript Printer

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, like with the PostScript `closepath` operator; instead, it affects the shape of the smooth curve. Figure 4 above 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 will be 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 the situation 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.

Caveat (only one)

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.

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 upon 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 (compare. the Appendix "Interfaces for Picture Comments"). 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. For now, it should be set to `0` in case support for centering is added in the future.

The interesting question relating to this picture comment is again, how can both QuickDraw and PostScript representations be included in the same picture? We have already learned that the `PostScriptBegin` - `PostScriptEnd` bracket is perfect for hiding the QuickDraw imaging from being printed on a PostScript device. But we still need a trick to hide the line drawing instruction that produces the dashed lines in the PostScript output 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 }
             numIntvls := 2;    { number of interval specs }
             intervals[0] := 4;
             intervals[1] := 6; { this means 4 points on, 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;
```

But where is the `DashedQDLine` procedure? Well, that's another story. It's not precisely the subject of this Note, and thus, again, is left as a spare-time exercise for the reader. For the sake of testing, I used the following placeholder:

```
  PROCEDURE DashedQDLine(dx,dy: Integer; dashSpec: TDashedLineHdl);

    VAR
       oldPat: Pattern;

    BEGIN
       oldPat := thePort^.pnPat;
       PenPat(gray);
       Line(dx,dy);
       PenPat(oldPat);
    END;
```

Caveat

As mentioned earlier in the section "General Caveats," the current version of the PostScript LaserWriter driver produces poor results when the `DashedLine` picture comment is applied to polygons.

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 port, 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 mainly useful for immediate printing. As a consequence, lines are usually always at least one pixel 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, in particular to values less than a QuickDraw pixel width of 1/72 inch. 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 (like in the LaserWriter SC), it works differently than in PostScript printer drivers. You must be careful to support both implementations. The difference appears as soon as `SetLineWidth` is used for the second time.

The PostScript LaserWriter 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**

If the `SetLineWidth` picture comment is not implemented on the chosen printer driver, nothing happens to the pen size. Instead of hairlines, you’ll get the ordinary 1/72 inch thickness; and if you use the `SetLineWidth` comment to specify lines thicker than the standard one pixel width, they will still be drawn with the previous, nonscaled pen size if the comment is unsupported. There is no reasonable way out of this dilemma, other than to special case your printing code for a PostScript driver (which is not recommended), or to use `PrGeneral` and to image at device resolution, without using the `SetLineWidth` comment at all.

The following sample code currently gives the expected results only on a PostScript LaserWriter, and with 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).

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 picture interpreter (usually the LaserWriter 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 in 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.

In order to hide the unrotated QuickDraw objects between `RotateBegin` and `RotateEnd`, we'll use the "magic pen mode" (23) again. To complete the dual QuickDraw/PostScript representation, draw the rotated QuickDraw image with `CopyBits` (preferably at maximum printer resolution determined by `PrGeneral`) inside `PostScriptBegin` and `PostScriptEnd` comments so that the QuickDraw representation won't show up on PostScript devices. 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;
```

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 if the definition from above is still available.  }
            { This assumes that no font downloading has occurred.          }

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

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.

(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 has all text drawn through standard

QuickDraw text drawing calls (`DrawChar`, `DrawString`, `DrawText`, and anything else that eventually calls the `StdText` bottleneck) be interpreted as PostScript instructions. 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 MultiFinder or System 7, there are no guarantees the file will still be open when the Printing Manager needs it. Background printing makes this even more complicated, to the point where the comment is not supported when background printing is enabled. For this reason alone, you should write a small routine that loads the resources from the file 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.

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;

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

  { Values for the tFlip field of the TTxtPicRec record }
```

```
        tFlipNone = 0;
        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 works 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) }
                            numIntvls: SignedByte;  { Number of intervals }
                            intervals: ARRAY [0..5] { Array of dash intervals }
                                       OF SignedByte;
                         END;

        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;
```

Further Reference:

- *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.PictAndPrinting —
  Pictures and the Printing Manager
- develop Issue 3, “Meet PrGeneral” by Pete “Luke” Alexander