This appendix describes the picture comments predefined by Apple Computer, Inc., for its PostScript printers and several of its QuickDraw printers (including the LaserWriter SC, ImageWriter LQ, and StyleWriter printers). This appendix introduces you to the use of picture comments for printing with features that are unavailable with QuickDraw alone.

For most applications, sending QuickDraw's picture-drawing routines to the printer driver is sufficient: the driver either uses QuickDraw or converts QuickDraw routines to PostScript code. See the chapter "Printing Manager" in this book for information about QuickDraw-based printing. For some applications, such as page-layout programs, QuickDraw-based printing may not be sufficient; such applications may rely on printer drivers—such as PostScript printer drivers—to provide features that are not available, or are difficult to achieve, using QuickDraw.

For PostScript printers, one solution is for your application to send PostScript code directly to the printer driver, but this approach requires you to know the PostScript language as well as QuickDraw. If your application requires features (such as rotated text and dashed lines) that are unavailable with QuickDraw, you may instead want to use picture comments to take advantage of these features on capable printers. Created with the QuickDraw procedure `PicComment`, picture comments are data or commands for special processing by output devices such as printer drivers. The `PicComment` procedure is introduced in the chapter "Pictures" in this book and is expanded upon in this appendix.

IMPORTANT
The picture comments supported by Apple printer drivers are described on page B-7. However, it is impossible to determine which picture comments are supported by the current printer driver. ▲

About Picture Comments

Within the drawing code sent to a printer driver after your application uses the `PrOpenPage` procedure, your application can specify picture comments by using the QuickDraw `PicComment` procedure. The `PicComment` procedure allows your application to pass data or commands directly to an output device.

```
PROCEDURE PicComment (kind: Integer; dataSize: Integer;
                      dataHandle: Handle);
```

The `kind` parameter specifies the kind of picture comment, and the `dataSize` parameter specifies the size of the data referred to by the `dataHandle` parameter. (For some picture comments, the values passed in the `dataSize` and `dataHandle` parameters should be 0 and `NIL`, respectively.)

You typically use a picture comment to give your application and an output device additional control over the rendering of images. A number of picture comments have been given special definitions by various printer drivers. When a printer driver encounters one of these comments, it interprets the comment as an appropriate drawing operation. A PostScript printer driver, for example, may convert a picture comment into PostScript code.

By including picture comments in your code that draws into a printing graphics port, your application can rotate text and graphics, smooth polygons, draw hairlines, create dashed lines, and pass PostScript code directly to the printer driver. (For information about the PostScript language, see the *PostScript Language Reference Manual*, second edition, published by Addison-Wesley.)

Picture comments were initially designed to allow applications to share data in the form of QuickDraw pictures (as described in the chapter "Pictures" in this book). With the advent of the PostScript LaserWriter printer, the use of picture comments was extended to allow applications to more easily take advantage of various PostScript features unavailable with QuickDraw.

However, you do not need to create a QuickDraw picture to use picture comments for printing. When your application calls the Printing Manager procedure `PrOpenPage`, the printer driver collects your drawing operations after they are handled by the low-level drawing routines contained in the `QDProcs` record for the printing graphics port. As explained in the chapter "QuickDraw Drawing" in this book, the default low-level procedure specified by QuickDraw in the `commentProc` field of the `QDProcs` record is the `StdComment` procedure, which simply ignores picture comments. However, a printer driver can replace the `StdComment` procedure with its own routine for handling picture comments.

▲ **WARNING**
As described in the chapter "Pictures" in this book, do not call the `OpenCPicture` or `OpenPicture` function between calls to `PrOpenPage` and `PrClosePage`. ▲

When you use the `PicComment` procedure after calling `PrOpenPage` and before calling `PrClosePage`, the printer driver either ignores the picture comment passed to `PicComment` or collects the results of its drawing operations, depending on whether the printer driver has installed its own low-level drawing routine that handles the picture comment.

Although the `PicComment` procedure is available on all Macintosh computers, the availability of the drawing operations that you can implement with picture comments depends on the driver for the current printer. The inability to determine which picture comments are supported by the current printer driver means that if you use picture comments to perform drawing operations not supported by QuickDraw, you must also provide for printing on QuickDraw-only printers.

This requires your application to maintain separate code branches: for example, one that takes advantage of the picture comment handling of a PostScript printer driver, and another for a printer driver that supports only QuickDraw. Furthermore, you must hide the code that takes advantage of PostScript printer drivers from QuickDraw-based drivers, and you must hide from PostScript drivers the code that uses QuickDraw-based approximations of these drawing operations. Your application's printed output will necessarily differ depending on the driver for the current printer.

Table B-1 lists picture comments defined for various printer drivers produced by Apple and used by third-party producers of various other printer drivers. For each picture comment, this table shows the name of the picture comment that you specify in the `kind` parameter of the `PicComment` procedure, the value represented by the name, the value for the `dataSize` parameter, and the value for the `dataHandle` parameter. (Be sure to dispose of the memory you allocate for any handle you pass in the `dataHandle` parameter.) Keep in mind that it is impossible to determine which picture comments are supported by the driver of the current printer.

Table B-1 Names, values, and data sizes for picture comments

Name	Value	Data size	Data handle	Description
Text picture comments				
`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	`TCenterRec`	Offset to center of rotation for text
`LineLayoutOff`	155	0	`NIL`	Turn printer driver's line layout off
`LineLayoutOn`	156	0	`NIL`	Turn printer driver's line layout on
`ClientLineLayout`	157	16	`TClientLLRec`	Customize line layout error distribution

continued

Table B-1 Names, values, and data sizes for picture comments (continued)

Name	Value	Data size	Data handle	Description
Graphics picture comments				
`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	`TPolyVerbRec`	Close, fill, frame
`PolyClose`	165	0	`NIL`	Smooth the curve between endpoints
`RotateBegin`	200	8	`TRotationRec`	Begin rotated port
`RotateEnd`	201	0	`NIL`	End rotation
`RotateCenter`	202	8	`TCenterRec`	Offset to center of rotation
Line-drawing picture comments				
`DashedLine`	180	Size of a `TDashedLineRec` record	`TDashedLineRec`	Draw following lines as dashed
`DashedStop`	181	0	`NIL`	End dashed lines
`SetLineWidth`	182	4	`TLineWidthHdl`	Set fractional line widths
PostScript picture comments				
`PostScriptBegin`	190	0	`NIL`	Set driver state to PostScript
`PostScriptEnd`	191	0	`NIL`	Restore QuickDraw state
`PostScriptHandle`	192	Length of PostScript data	`Handle`	PostScript data referenced in handle
`PostScriptFile`	193	Length of PostScript data	`Handle`	Filename referenced in handle
`TextIsPostScript`	194	0	`NIL`	QuickDraw text is sent as PostScript
`ResourcePS`	195	8	Resource type, resource ID, index	PostScript data in a resource file
`PSBeginNoSave`	196	0	`NIL`	Set driver state to PostScript

Table B-1 Names, values, and data sizes for picture comments (continued)

Name	Value	Data size	Data handle	Description
Forms-printing picture comments				
`FormsPrinting`	210	0	`NIL`	Don't clear print buffer after each page
`EndFormsPrinting`	211	0	`NIL`	End forms printing after `PrClosePage`
ColorSync picture comments				
`CMBeginProfile`	220	0	`NIL`	Begin ColorSync profile
`CMEndProfile`	221	0	`NIL`	End ColorSync profile
`CMEnableMatching`	222	0	`NIL`	Begin ColorSync color matching
`CMDisableMatching`	223	0	`NIL`	End ColorSync color matching

All PostScript LaserWriter drivers support the picture comments listed in Table B-1.

Some third-party QuickDraw printer drivers support the `TextBegin`, `TextCenter`, and `TextEnd` picture comments.

The QuickDraw LaserWriter SC driver supports the `LineLayoutOff`, `LineLayoutOn`, and `SetLineWidth` picture comments.

The QuickDraw ImageWriter LQ driver and versions prior to 7.2 of the QuickDraw StyleWriter driver support the `LineLayoutOff` and `LineLayoutOn` picture comments.

The QuickDraw Personal LaserWriter LS driver and versions later than 7.2 of the QuickDraw StyleWriter driver support no picture comments at all.

The `SetGrayLevel` picture comment is now obsolete. The `PostScriptFile`, `TextIsPostScript`, `FormsPrinting`, `EndFormsPrinting`, `ClientLineLayout`, `PSBeginNoSave`, and `ResourcePS` picture comments have limited use and are no longer recommended.

See *Inside Macintosh: Advanced Color Imaging* for information about the picture comments used by the ColorSync Utilities.

Maintaining Device Independence

Whenever printing, you should use both QuickDraw and non-QuickDraw representations of an image, so that the current printer driver can render the best possible picture. If you send an image described with picture comments to a QuickDraw printer driver that does not support those picture comments, the driver ignores the comments and subsequently does not print your image; if you send only a QuickDraw image to a printer driver that supports picture comments, the driver may not render its best possible image.

Printer drivers that support `TextBegin`, `TextCenter`, and `TextEnd` are expected to ignore calls to the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures that fall between the `TextBegin` and `TextEnd` picture comments. Between the `TextBegin` and `TextEnd` picture comments, you can use `CopyBits` to draw a bitmap representation of rotated text on QuickDraw printers; this bitmap is not used if the `TextBegin` and `TextEnd` picture comments are supported, but it is used if `TextBegin` and `TextEnd` are not supported. This is illustrated in Listing B-4 on page B-21.

When your application draws polygons on a PostScript printer, you can use `PolyBegin`, `PolySmooth`, and `PolyEnd` picture comments to draw smoothed polygons; QuickDraw printer drivers ignore these comments. To make a PostScript printer driver ignore your QuickDraw representation of the polygons, you can use the `PolyIgnore` picture comment, as illustrated in Listing B-6 on page B-27.

A technique for maintaining two sets of drawing codes, described in "Rotating Graphics" beginning on page B-29 and "Drawing Dashed Lines" beginning on page B-33, makes use of a "magic pen" visible only to PostScript drivers. Graphics comments for drawing dashed lines and for rotating graphics require the use of the `PenMode` procedure to set the pattern mode to a value of 23. Normally this value is undefined, but it is handled specially by PostScript printer drivers (all QuickDraw drivers ignore it). Your application can use this pattern mode to draw objects in a picture, and if the picture is printed on a QuickDraw printer, these objects are not visible.

To maintain device independence when you send routines to a PostScript printer driver, you can "hide" QuickDraw routines between the `PostScriptBegin` and `PostScriptEnd` picture comments. The `PostScriptBegin` comment is recognized only by PostScript printer drivers. When a PostScript driver receives the `PostScriptBegin` comment, it tells the PostScript printer to save the current state of the printer and to disable all low-level standard QuickDraw drawing procedures. Thus, the QuickDraw representation of the graphic is ignored by PostScript printer drivers.

Table B-2 lists the QuickDraw low-level procedures and the affected high-level drawing routines that are disabled by the `PostScriptBegin` picture comment.

Table B-2 Low-level QuickDraw routines disabled by the `PostScriptBegin` comment

Low-level routine	Examples of affected high-level QuickDraw routines
`StdText`	QuickDraw text-drawing routines (as described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*)
`StdLine`	`MoveTo`, `Move`, `LineTo`, `Line`
`StdRect`	`FrameRect`, `PaintRect`, `FillRect`, `EraseRect`, `InvertRect`
`StdRRect`	`FrameRoundRect`, `PaintRoundRect`, `FillRoundRect`, `EraseRoundRect`, `InvertRoundRect`
`StdOval`	`FrameOval`, `PaintOval`, `FillOval`, `EraseOval`, `InvertOval`
`StdArc`	`FrameArc`, `PaintArc`, `FillArc`, `EraseArc`, `InvertArc`
`StdPoly`	`FramePoly`, `PaintPoly`, `FillPoly`, `ErasePoly`, `InvertPoly`
`StdRgn`	`FrameRgn`, `PaintRgn`
`StdBits`	`CopyBits`, `CopyMask`, `CopyDeepMask`

To mark the end of a sequence of hidden QuickDraw drawing routines and to reenable QuickDraw drawing routines, you can use the picture comment `PostScriptEnd`. The `PostScriptEnd` comment is recognized only by PostScript printer drivers. When a PostScript driver receives the `PostScriptEnd` comment, it tells the PostScript printer driver to restore the previous state of the printer driver and to enable QuickDraw drawing operations.

For a LaserWriter PostScript printer driver, QuickDraw routines that draw text, lines, and shapes and copy bitmaps or pixel maps have no effect when placed between the `PostScriptBegin` and `PostScriptEnd` picture comments. Instead, the driver expects to receive imaging instructions in subsequent picture comments. On the other hand, a QuickDraw printer driver ignores the `PostScriptBegin` and `PostScriptEnd` picture comments.

Only PostScript printer drivers should support the `DashedLine`, `DashedStop`, `RotateBegin`, `RotateCenter`, and `RotateEnd` picture comments. Therefore, you can use the `PostScriptBegin` and `PostScriptEnd` picture comments to hide your QuickDraw implementations of these comments from the printer driver. Listing B-7 on page B-31 illustrates how to use `PostScriptBegin` and `PostScriptEnd` when rotating graphics on PostScript printers; Listing B-9 on page B-34 illustrates how to use `PostScriptBegin` and `PostScriptEnd` when drawing dashed lines on PostScript printers.

Synchronizing QuickDraw and PostScript Printer Drivers

QuickDraw instructions such as those generated by the `Move`, `MoveTo`, `PenPat`, and `PenSize` routines change the state of the current graphics port without going through the standard low-level routines pointed to in the `QDProcs` record for the current graphics port. A printer driver takes these changes into account only at the time it executes an actual drawing instruction. The printer driver uses the routines specified in the `QDProcs` record at execution time and responds only to those instructions handled by the routines in the `QDProcs` record. Therefore, you should flush the state of the printing graphics port explicitly by calling any routine that goes through the `QDProcs.lineProc` field, as shown in Listing B-1, before inserting code using picture comments for a PostScript driver. The use of the application-defined routine `MyFlushGrafPortState` shown here is further illustrated in Listing B-8 on page B-32.

Listing B-1 Synchronizing QuickDraw and the PostScript driver

```
PROCEDURE MyFlushGrafPortState;
VAR
   penInfo: PenState;
BEGIN
   GetPenState(penInfo);   {save pen size}
   PenSize(0,0);           {make it invisible}
   MoveTo(-3200,-3200);    {move the pen way off the page in }
                           { case the printer driver draws a dot }
                           { even with a pen size of (0,0)}
   Line(0,0);              {go through QDProcs.lineProc}
                           {next, restore pen size}
   PenSize(penInfo.pnSize.h, penInfo.pnSize.v);
END;
```

A PostScript printer driver separates the PostScript code generated for text-drawing instructions (which usually involves font queries and, sometimes, font downloading) from the picture comments intended for PostScript devices. In certain cases, this results in apparently nonsequential execution of drawing instructions and may affect clipping regions or have side effects on the drawing operations you include in picture comments. To synchronize the sequence of QuickDraw routines with the generation of PostScript code, you need to flush the buffer maintained by the PostScript driver. You can do this by using the `PostScriptBegin` picture comment followed immediately by the `PostScriptEnd` picture comment. This causes all PostScript code, generated either by the application or by the printer driver, to be sent to the printer. Listing B-2 shows an application-defined procedure that does this. The use of the application-defined routine `MyFlushPostScriptState` shown here is further illustrated in Listing B-4 on page B-21.

Listing B-2 Flushing the buffer for a PostScript printer driver

```
PROCEDURE MyFlushPostScriptState;
BEGIN
   PicComment(PostScriptBegin, 0, NIL);
   PicComment(PostScriptEnd, 0, NIL);
END;
```

Using Text Picture Comments

The text picture comments listed in Table B-1 on page B-5 allow you to disable the printer driver's line layout capabilities (as described in the next section), construct lines of text out of disparate strings (as described in "Delimiting Strings" on page B-16), and rotate text on the page (as described in "Rotating Text" on page B-17).

For information on drawing text, see *Inside Macintosh: Text*.

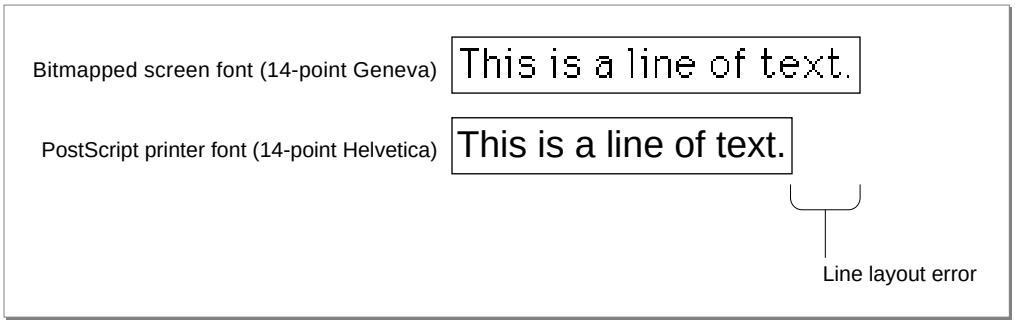
Disabling and Reenabling Line Layout

When your application draws text into a printing graphics port, the printer driver may do a lot of extra work depending on the current printer; the printer driver may have to scale and smooth fonts, remap characters, and substitute one font used onscreen for another that exists on the printer (this last action is called ***font substitution***).

After it selects the appropriate font, the printer driver matches the width of the printed line with the width of the screen line. If the driver has to perform font substitution, the two lines may be very different. For example, if your application draws a document with the Geneva bitmapped font (instead of the Geneva TrueType font), a PostScript printer

driver could substitute the Helvetica® font for Geneva in the PostScript code it generates. Since Helvetica is a different font, it has different metrics. A rather exaggerated example of the effects of font substitution can be found in Figure B-1.

Figure B-1 The line layout error between a bitmapped font and a PostScript font



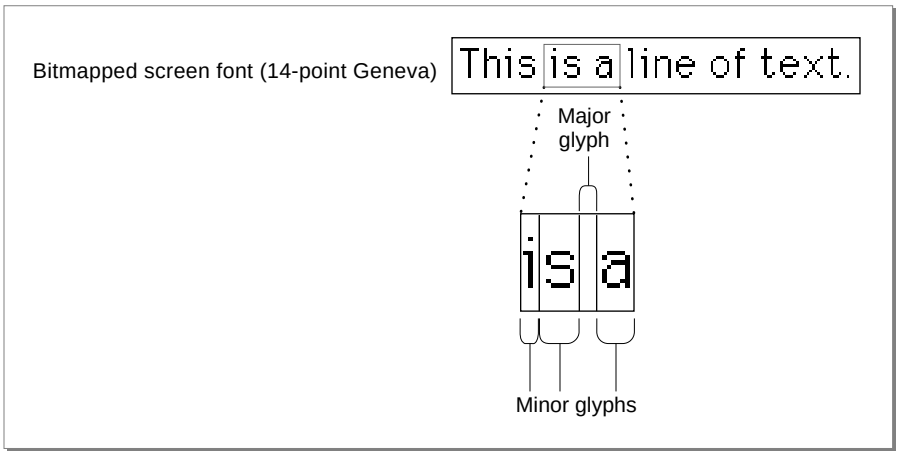
For the typical user, the appearance of Helvetica on the printed page is not that much different from the appearance of Geneva on the screen. However, the width of the lines using the two fonts is different; this difference is called the ***line layout error.*** The line of text using the bitmapped screen font is much wider than the line of text using the PostScript printer font. (Depending on the font used in the document or substituted on the printer, you might also run into cases where the screen width is narrower than the printed width.)

Note
There are no line layout problems with TrueType fonts, unless one font has the same name as—but a different character width from—a printer-resident PostScript font. ◆

To distribute the layout error, a printer driver must effectively increase or decrease the width of each glyph in the line. A ***glyph*** is the distinct representation of a character in a form that a screen or printer can display. A glyph may represent one character (the lowercase *a*), more than one character (the *fi* ligature, two characters but one glyph), or a nonprinting character (the space character). When using Roman scripts, most lines of text contain some number of space character glyphs. Printer drivers take advantage of this fact and normally apply most of the layout error to space glyphs (known as the ***major glyphs***) and the rest of the error to the other glyphs in the string (known as the ***minor glyphs***).

In Figure B-2, the *i*, *s*, and *a* characters are examples of minor glyphs, where *s* and *a* are separated by the major glyph (the space character).

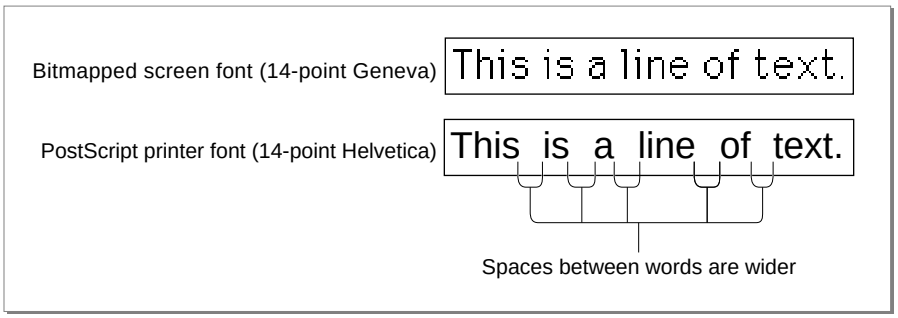
Figure B-2 Major and minor glyphs



The amount of error applied to the major glyph is known as the ***major error,*** and the amount applied to the other glyphs is the ***minor error.***

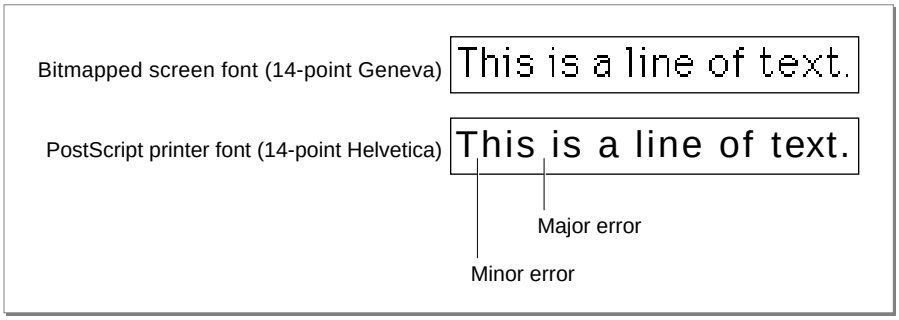
In Figure B-3, the printer driver corrects most of the difference between the line widths by expanding the width of the space glyphs in the string.

Figure B-3 Distributing layout error to the major glyphs



However, if the printer driver expands only the width of the spaces, the line has a strange appearance. To balance the changes made to the space glyphs, the driver's line layout routines increase the space between each glyph in the string by a small amount. After the line is laid out in this way, the printed string should be almost exactly as wide as the string that was displayed on the screen. As shown in Figure B-4, the space between the uppercase *T* and the lowercase *h* in the word *This* has been increased, but only slightly; most of the error has been applied to the spaces. By default, most drivers apply about 80 percent of the total line layout error to the major glyphs and the other 20 percent to the minor glyphs. When using a script system that does not use the space glyph to delimit words, the layout error is distributed evenly across all characters in the line.

Figure B-4 Distributing layout error among major and minor glyphs



A printer driver's line layout routines are device-dependent. Since different devices have different resident fonts, the layout error can be quite large. For this reason, you should not assume that if you have the correct output on one type of laser printer you will have the correct output on all devices or with all fonts.

Although the printer driver can compute the placement of a line of text on the page so that it closely approximates the placement of the line on the screen, there are times when adjusting the line of text by adding space can have an adverse effect on the line layout that your application has already done.

You can disable the line layout routines of the current printer driver and give your application more control over placement of the glyphs on the page by using the `LineLayoutOff` picture comment. You may want to use this picture comment if your application prints monospaced, tab-formatted text; draws notes or other music symbols using glyphs from a music font; or renders mathematical equations or formulas. For example, if your application displays musical notation, the notes should stay where your application placed them, because small shifts in position can cause the music to be misread.

The `LineLayoutOff` picture comment instructs the printer driver to make no adjustments to the text being sent. Your application is then responsible for identically matching the appearance of text displayed on the screen to the printer. If the current printer driver does not support these comments, it ignores them and places the text on the page as well as it can.

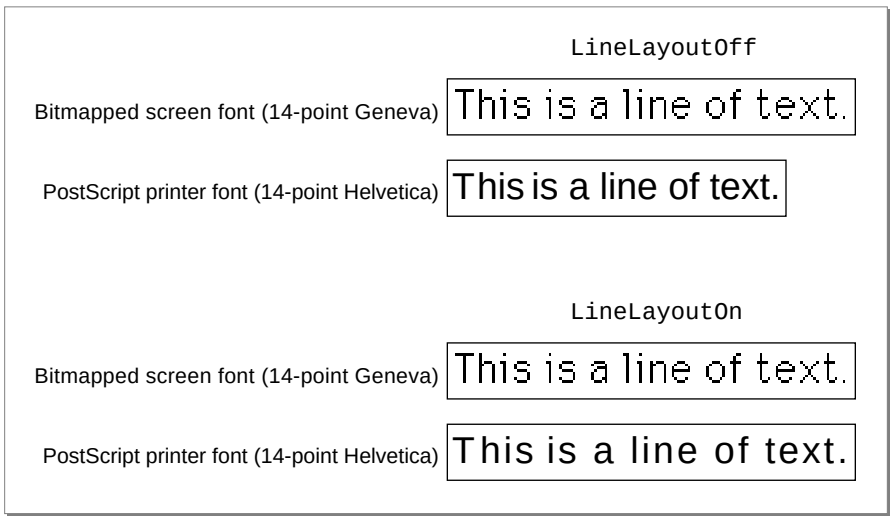
You can reenable the printer driver's line layout routines with the `LineLayoutOn` picture comment (however, some printer drivers support only the `LineLayoutOff` comment). Although general line layout is disabled, some small shifts in glyph position may still occur. These shifts are usually not a problem, but, if they are, you should use the `PrGeneral` procedure with the `getRslDataOp` and `setRslOp` opcodes (described in the chapter "Printing Manager" in this book) to draw text at the resolution of the current printer.

IMPORTANT

Setting the `FractEnable` global variable (described in the chapter "Font Manager" in *Inside Macintosh: Text*) to `TRUE` does not have precisely the same effect as using the `LineLayoutOff` picture comment. You should explicitly use the `LineLayoutOff` picture comment rather than the `SetFractEnable` procedure. ▲

Figure B-5 compares the results of an application using the `LineLayoutOff` picture comment and the `LineLayoutOn` picture comment. In the first example, the text is printed exactly as it is rendered on the printer, with a much smaller width. In the second example, the printer driver's line layout routines make the screen and printer lines the same length.

Figure B-5 Using the `LineLayoutOff` and `LineLayoutOn` picture comments



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

1. It collects text processed by the routine pointed to in the `textProc` field of the printing graphics port's `QDProcs` record, and assembles the text into a logically contiguous line. This includes text moved vertically away from the baseline to take care of diacritical marks or exponents in the text. The accumulation of text stops when the PostScript LaserWriter driver detects that the pen position has moved horizontally since the conclusion of the previous text-drawing instruction, or when the driver encounters picture comments such as `TextBegin`, `TextEnd`, `StringBegin`, and `StringEnd`.
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 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 diacritical marks or exponents, the diacritical marks and exponents would often be misplaced.

If you want precise control over the placement of different text strings within a line, you must override the heuristic line accumulation algorithm of the PostScript LaserWriter driver (described in the first step). A good way to override this algorithm is to use the `StringBegin` and `StringEnd` picture comments to mark individual strings as logically independent text entities; this prevents the PostScript LaserWriter driver from assembling the strings into one logically contiguous line of text. The `StringBegin` and `StringEnd` picture comments are described in the next section; Listing B-3 on page B-17 illustrates how to completely disable line layout by using the `LineLayoutOff` and `StringBegin` picture comments.

Delimiting Strings

You may want to draw a particular text string in pieces instead of a whole. For example, to draw kerned glyphs, you can draw the first part of the string—up to the point where kerning occurs—using the `DrawText` procedure, and you can then adjust the pen and draw the kerned glyph using the `DrawChar` procedure. (The `DrawText` and `DrawChar` procedures are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.) You can also draw a single string that contains different fonts, styles, or sizes—if you call `DrawText` each time the typeface or font style changes. To identify the beginning of a single string that will be drawn using multiple calls to a QuickDraw text-drawing routine, you can use the `StringBegin` picture comment. Use the `StringEnd` picture comment to mark its end.

You can use the `StringBegin` and `StringEnd` picture comments if your application needs complete control over glyph placement on a page. If your application uses text-editing boxes for individual strings, it can use these picture comments to treat each string as a separate piece of text and place all glyphs into one text-editing box.

Listing B-3 uses the `StringBegin` and `StringEnd` picture comments. Use the `LineLayoutOff` picture comment (described in the preceding section) in conjunction with the `StringBegin` comment to turn line layout completely off.

Listing B-3 Disabling line layout by using the `LineLayoutOff` and `StringBegin` picture comments

```
PROCEDURE MyStringReconDemo (x: XArray; y: Integer);
BEGIN
   PicComment(LineLayoutOff,0,NIL);
   PicComment(StringBegin,0,NIL);
   {position each character of the word 'Test' using }
   { MoveTo and DrawChar}
   MoveTo(x[1],y); DrawChar('T');
   MoveTo(x[2],y); DrawChar('e');
   MoveTo(x[3],y); DrawChar('s');
   MoveTo(x[4],y); DrawChar('t');
   {reenable the printer driver's line layout routines}
   PicComment(StringEnd,0,NIL);
   PicComment(LineLayoutOn,0,NIL);
END;
```


Rotating Text

You can use picture comments to rotate text on PostScript devices and on any QuickDraw-based drivers that support text rotation. (This is not the kind of rotation associated with landscape and portrait orientation of the printer paper as selected by the user through the style dialog box. This rotation occurs in reference to the current QuickDraw graphics port only.) The picture comments to rotate text are `TextBegin`, `TextCenter`, and `TextEnd`.

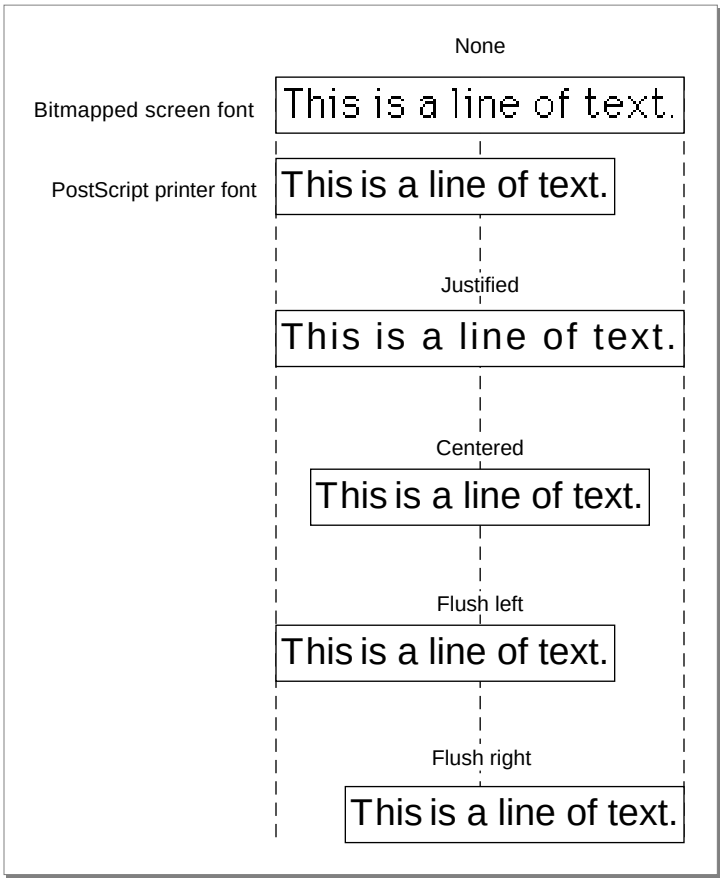
If you use picture comments to rotate text, you should also generate a device-independent representation, such as a bitmapped version of the text, to be used on QuickDraw devices that don't support these picture comments. Printer drivers that support `TextBegin`, `TextCenter`, and `TextEnd` are expected to ignore calls to the `CopyBits`, `CopyMask`, and `CopyDeepMask` procedures (as well as QuickDraw clipping regions) between the `TextBegin` and `TextEnd` picture comments. In this way, you can use `CopyBits` to draw a bitmap representation of rotated text on QuickDraw printers; the bitmap is not used if the `TextBegin` and `TextEnd` picture comments are supported, but it is used if `TextBegin` and `TextEnd` are not supported.

Some versions of 2-byte Kanji systems print Kanji glyphs by calling the `CopyBits` procedure instead of calling standard text-drawing routines. You cannot use the text rotation picture comments with these fonts. Instead, use the picture comments described in "Rotating Graphics" beginning on page B-29.

To use picture comments to rotate text, you begin by specifying the amount of rotation as a parameter to the `TextBegin` comment. Next, you pass the center of rotation in the `TextCenter` comment. The printer driver rotates any text drawn between the `TextCenter` and `TextEnd` comments.

The `TextBegin` picture comment allows your application to specify left, right, center, or full justification; horizontal or vertical flipping; and degrees of rotation. The possible types of alignment are shown in Figure B-6.

Figure B-6 Variations in text alignment



When you specify the `TextBegin` picture comment in the `kind` parameter of the `PicComment` procedure, you also specify a `TTxtPicHdl` handle (a handle to a `TTxtPicRec` record) in the `dataHandle` parameter. Here is how you should declare these as Pascal data types in your application:

```
TYPE
   TTxtPicHdl  = ^TTxtPicPtr;
   TTxtPicPtr  = ^TTxtPicRec;
   TTxtPicRec  =
   PACKED RECORD
      tJus:          Byte;    {justification of text}
      tFlip:         Byte;    {horizontal or vertical flipping}
      tAngle:        Integer; {0..360 degrees clockwise rotation }
                              { in integer format}
      tLine:         Byte;    {reserved}
      tCmnt:         Byte;    {reserved}
      tAngleFixed:   Fixed;   {0..360 degrees clockwise rotation }
                              { in fixed-number format}
   END;
```

You supply the `tJus` field with one of these constants to specify the alignment setting of the text:

```
   CONST
      tJusNone    = 0;  {no alignment}
      tJusLeft    = 1;  {flush left}
      tJusCenter  = 2;  {centered}
      tJusRight   = 3;  {flush right}
      tJusFull    = 4;  {full justification}
```

Setting the `tJus` field to left, right, or centered tells the printer driver to maintain only the left, right, or center point of the line (respectively), preventing the driver from recalculating the interword spacing. A value of `tJusFull` specifies that both endpoints of the line must be maintained, so the driver recalculates interword spacing instead of rejustifying text.

You supply the `tFlip` field with one of these constants to specify the horizontal or vertical flipping of text about the center point (which, in turn, is specified with the `TextCenter` picture comment):

```
   CONST
      tFlipNone         = 0; {no flip of text}
      tFlipHorizontal   = 1; {horizontal flip of text}
      tFlipVertical     = 2; {vertical flip of text}
```

You supply the `tAngle` field with an integer to specify the number of degrees by which the printer driver should rotate the text.

The `tLine` and `tCmnt` fields are reserved.

You supply the `tAngleFixed` field with a fixed-point number to specify the number of degrees by which the printer driver should rotate the text.

In a `TTxtPicRec` record, you can provide the degrees of rotation both as an integer (in the `tAngle` field) and as a fixed-point number (in the `tAngleFixed` field). You should always specify the rotation in both fields, even for drivers that support only integral rotation. The driver determines which field to use based on the size of the handle passed to `PicComment`. If you do not define the `tAngleFixed` field in the `TTxtPicRec` record, the printer driver automatically uses the `tAngle` field.

To rotate an object, a printer driver needs information concerning the center of rotation. Immediately after a `TextBegin` comment, the driver expects the `TextCenter` picture comment specifying the offset to the center of rotation for any text enclosed within the text picture comments. The driver stores this offset and adds it to the location of the first text-drawing routine after it receives the `TextCenter` picture comment. This allows you to send multiple runs of text to be rotated with different centers of rotation, while using only one set of `TextBegin` and `TextEnd` picture comments. The printer driver expects the string locations to be in the coordinate system of the current graphics port.

The printer driver rotates the entire graphics port to draw the text, so it can draw several strings with one `TextBegin` picture comment and one `TextCenter` picture comment. You should always include as much text as possible in a single `TextBegin` picture comment so that the driver makes the fewest number of rotations.

The printer driver can draw nontextual objects within the bounds of the text rotation comments, but it must restore the printing graphics port to its original state to draw the object, and then rotate the printing graphics port again to draw the next string of text. You must send another `TextCenter` comment before each new rotation.

When you specify the `TextCenter` (or `RotateCenter`) picture comment in the `kind` parameter of the `PicComment` procedure, you also supply in the `dataHandle` parameter a `TCenterHdl` handle, which is a handle to a `TCenterRec` record. You can use this record to specify the center of rotation for text or (as described in "Rotating Graphics" beginning on page B-29) for graphics. Here is how you should declare these as Pascal data types in your application:

```
TYPE
   TCenterHdl  = ^TCenterPtr;
   TCenterPtr  = ^TCenterRec;
   TCenterRec  =
   RECORD
      y:    Fixed;   {vertical offset from current pen location}
      x:    Fixed;   {horizontal offset from current pen location}
   END;
```

You use the `y` field to specify the vertical offset along the y-axis from the current pen location to the center of rotation.

You use the `x` field to specify the horizontal offset along the x-axis from the current pen location to the center of rotation.

The application-defined routine `MyDrawXString`, shown in Listing B-4, rotates the strings by the degrees specified in the `rot` parameter. The rotation occurs around the current point, offset by the value passed in the `ctr` parameter. The strings are justified and flipped according to the `just` and `flip` parameters. If the printer driver supports the `TextBegin`, `TextCenter`, and `TextEnd` picture comments, the printer driver rotates the text at device resolution; otherwise, an application-defined procedure is called to generate a bitmap of the rotated and flipped text, using `CopyBits` to draw the text in the printing graphics port. The pen position is preserved. (Listing B-8 on page B-32 illustrates how to use the `TCenterRec` record to rotate graphics.)

Listing B-4 Displaying rotated text using picture comments

```
PROCEDURE MyDrawXString(s: Str255; ctr: Point;
                        just, flip: Integer; rot: Fixed);
VAR
   hT:         TTxtPicHdl;
   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)));
   WITH hT^^ DO
   BEGIN
      tJus := just;
      tFlip := flip;
      tAngle := - FixRound(rot); {counterclockwise}
      tLine := 0; {reserved}
      tCmnt := 0; {used internally by the printer driver}
      tAngleFixed := - rot;
   END;
   hC^^.y := Long2Fix(ctr.v);
   hC^^.x := Long2Fix(ctr.h);
   MyFlushPostScriptState; {see Listing B-2 on page B-11}
   PicComment(TextBegin,SizeOf(TTxtPicRec),Handle(hT));
   PicComment(TextCenter,SizeOf(TCenterRec),Handle(hC));
   {graphics state now has rotated/flipped coordinates}
```

```
   oldClip := NewRgn;
   GetClip(oldClip);
   SetRect(zeroRect,0,0,0,0);
   ClipRect(zeroRect);  {hides this DrawString from }
   DrawString(s);       { QuickDraw in the rotated }
                        { environment}
   ClipRect(oldClip^^.rgnBBox);
   {now the "fallback" bitmap representation}
   MyQDStringRotation(s, ctr, just, flip, rot);
   PicComment(TextEnd, 0, NIL);
   {set environment back to the original state}
   DisposeHandle(Handle(hT));
   DisposeHandle(Handle(hC));
   MoveTo(pt.h, pt.v);  {restore the pen position}
END;
```

Because the PostScript LaserWriter driver buffers generated PostScript code, and because the driver ignores clipping regions between the `TextBegin` and `TextEnd` picture comments, clipping regions for drawing instructions that precede `TextBegin` may be affected. Therefore, `MyDrawXString` uses the application-defined routine `MyFlushPostScriptState` (shown in Listing B-2 on page B-11) immediately before using the `TextBegin` picture comment.

Using Graphics Picture Comments

Graphics picture comments, listed in Table B-1 on page B-5, provide your application with the ability to render smoothed polygons (as described in the next section) and to rotate graphics (as described in "Rotating Graphics" on page B-29).

In general, you cannot use one set of graphics picture comments (for instance, the polygon-drawing picture comments) with another (graphics rotation comments). When using these two types of comments, you should simply rotate the points of the polygon before drawing.

The graphics comments for drawing dashed lines and for rotating graphics require the use of the `PenMode` procedure (described in the chapter "QuickDraw Drawing" in this book) to set the pattern mode to a value of 23. Normally this value is undefined, but it is handled specially by PostScript printer drivers, which treat it like the `srcCopy` Boolean transfer mode (described in the chapters "QuickDraw Drawing" and "Color QuickDraw"). All QuickDraw drivers ignore this pattern mode. Your application can use this pattern mode to draw objects in a picture and, if the picture is printed on a QuickDraw printer, these objects are not visible.

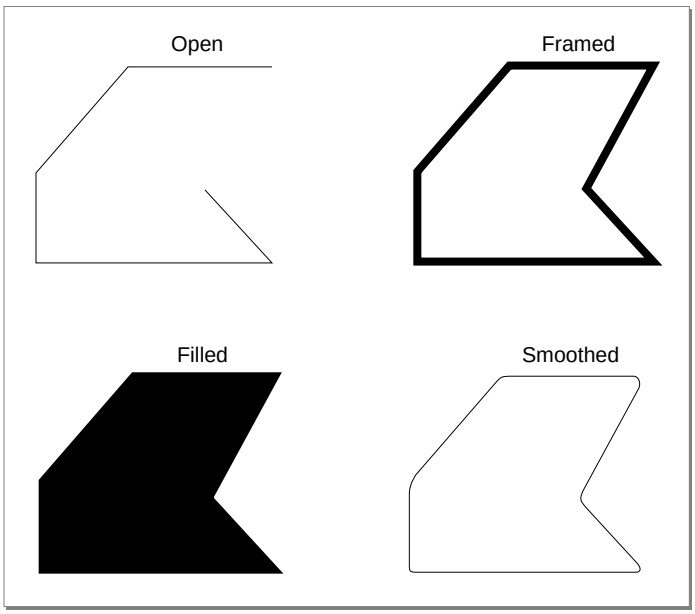
Drawing Polygons

By using picture comments, you can draw high-resolution polygons on PostScript printing devices. PostScript supports four types of polygons: open, framed, filled, and smoothed. (QuickDraw supports all of these types except smoothed.)

Type	Description
Open	A polygon whose endpoints do not join. This type of polygon cannot be filled.
Framed	A closed polygon that is not filled. Framed and filled polygons are exclusive to one another.
Filled	A closed polygon whose interior is entirely covered with a pattern.
Smoothed	A polygon (open, framed, or filled) whose edges have been rounded.

Figure B-7 shows these four types of polygons.

Figure B-7 Types of polygons



To draw polygons, perform the following steps:

1. Use the `PolyBegin` picture comment to alert the PostScript driver that you are drawing a polygon.
2. Optionally, you can use the `PolyClose` picture comment to use "closed" smoothing between the first and last vertices of the polygon.
3. Use the `PolySmooth` picture comment to tell the PostScript driver to draw a Bézier curve.
4. Use the `GetClip` procedure to save the current clipping region; then use the `ClipRect` procedure to hide your polygon's drawing commands from QuickDraw.
5. Draw your polygon. The PostScript driver renders it smoothly.
6. Use the `SetClip` procedure to restore the previous clipping region.
7. Use the `PolyIgnore` picture comment to make the printer driver ignore the line-drawing commands for your QuickDraw representation of the polygon.
8. Draw your QuickDraw representation of the polygon.
9. Use the `PolyEnd` picture comment.

The `PolyBegin` and `PolyEnd` picture comments surround the polygon description. Note that the printer driver draws the polygon at the location of the pen when it receives the `PolyBegin` picture comment, so you must set the pen's location before using the `PolyBegin` picture comment. For polygons that are smoothed, you must set the pen size to 0 after the `PolyBegin` picture comment to prevent the unsmoothed polygon from being drawn on printers that do not support the polygon comments.

All QuickDraw routines called between `PolyBegin` and `PolyEnd` that are processed by the low-level `StdLine` routine are part of the polygon—that is, the endpoints of each of the lines become vertices of the polygons.

You should use the `PolyClose`, `PolySmooth`, and `PolyIgnore` picture comments between the `PolyBegin` and `PolyEnd` picture comments.

The `PolyClose` comment specifies that the printer driver should treat all vertices of the polygon in the same manner; in particular, this affects the shape of the smooth curve between the polygon's first and last vertices, which might otherwise be distinguishable as separate points. The `PolyClose` comment, however, does not automatically close the polygon as the PostScript operator `closepath` does.

To render high-resolution B-splines when PostScript is available, use the `PolySmooth` picture comment, which directs the PostScript printer driver to interpret the polygon vertices as control nodes for a quadratic Bézier spline. PostScript has a direct facility for cubic B-splines, and the PostScript printer driver translates the quadratic B-spline nodes into the appropriate nodes for a cubic B-spline that will emulate the original quadratic. This allows you to use this PostScript feature without having to call PostScript routines directly.

Note
PostScript Level 1 has some problems with very large polygons that have more than 1500 points. For this reason, you may want to avoid doubling the points on large smoothed polygons, even though a greater number of points might aid in making the polygon smoother. ◆

When you use the `PolySmooth` picture comment, pass a `TPolyVerbHdl` handle, which is a handle to a `TPolyVerbRec` record, in the `dataHandle` parameter of the `PicComment` procedure. You use a `TPolyVerbRec` record to tell the printer driver to interpret the polygon vertices as control nodes for a quadratic Bézier spline. Here is how you should declare these as Pascal data structures in your application:

```
Type
   TPolyVerbHdl   = ^TPolyVerbPtr;
   TPolyVerbPtr   = ^TPolyVerbRec;
   TPolyVerbRec   =
   PACKED RECORD
      f7,f6,f5,f4,f3:   Boolean;    {reserved; set to 0}
      fPolyClose:       Boolean;    {TRUE is same as PolyClose }
                                    { picture comment}
      fPolyFill:     Boolean;       {TRUE means fill polygon}
      fPolyframe:    Boolean;       {TRUE means frame polygon}
   END;
```

The `f7`, `f6`, `f5`, `f4`, and `f3` fields are reserved bits; you should set them to 0.

Setting the `fPolyClose` field to 1 achieves the same result as the `PolyClose` picture comment. The `PolyClose` comment specifies that the printer driver should treat all vertices of the polygon in the same manner; in particular, this affects the shape of the smooth curve between the polygon's first and last vertices, which might otherwise be distinguishable as separate points. The `PolyClose` comment does not automatically close the polygon as the PostScript operator `closepath` does.

Set the `fPolyFill` field to 1 if you want the printer driver to fill the polygon, or set it to 0 if not.

Set the `fPolyFrame` field to 1 if you want the printer driver to frame the polygon, or set it to 0 if not.

In Listing B-5, the polygon coordinates are defined through arrays of points, initialized using an application-defined procedure, `MyDefineVertices`. The procedure `MyDefineVertices` specifies the points for two polygons. The array referenced through the parameter `p` defines the points used for the PostScript representation of the polygon. The array referenced through the parameter `q` defines the points used for the QuickDraw representation of the polygon.

Listing B-5 Creating polygons

```
PROCEDURE MyDefineVertices(VAR p,q: PointArrayPtr);
CONST
   cx = 280;   {x coordinate for center point}
   cy = 280;   {y coordinate for center point}
   r0 = 200;   {radius}
   kN = 4;     {number of vertices for PostScript}
   kM = 6;     {number of vertices for QuickDraw approximation}
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 QuickDraw 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;
```

Use the `PolyIgnore` comment before drawing your QuickDraw version of the polygon; between `PolyIgnore` and `PolyEnd`, drivers that support these two comments ignore all QuickDraw routines processed through the low-level procedure `StdLine`. You can enclose the application-defined procedure `MyPolygonDemo`, shown in Listing B-6, between `OpenPicture` and `ClosePicture` calls to create a picture containing both QuickDraw and PostScript representations of the polygon. Alternatively, you can call `MyPolygonDemo` when drawing directly into a printing graphics port.

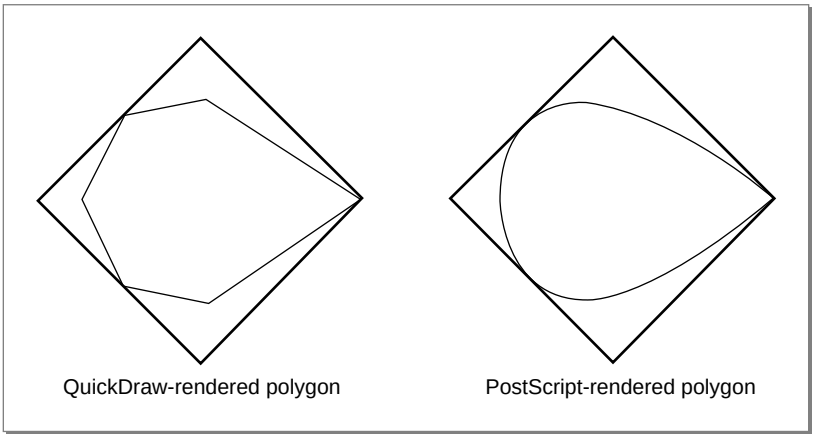
Listing B-6 Drawing polygons

```
PROCEDURE MyPolygonDemo;
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 DoErr(kMemError);
   MyDefineVertices(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 from QuickDraw}
   aPolyVerbH:=
      TPolyVerbHdl(NewHandle(SizeOf(TPolyVerbRec)));
   IF aPolyVerbH<> NIL THEN
      WITH aPolyRecH^^ DO
      BEGIN
         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);
      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 QuickDraw approximation of the smoothed }
      { polygon, invisible for PostScript because of PolyIgnore}
      SetClip(clipRgn);
      PicComment(PolyIgnore,0,NIL);
      polyRgn := NewRgn;
      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);
      DisposeHandle(Handle(aPolyVerbH));
      DisposeRgn(polyRgn);
      DisposePtr(Ptr(p));
      DisposePtr(Ptr(q));
END;
```

The two versions of the drawn polygon are shown in Figure B-8.

Figure B-8 QuickDraw and PostScript polygons



Note that you do not need to open a region, collect the line segments in the region, and draw the polygon through the `FrameRgn` procedure (described in the chapter "QuickDraw Drawing" in this book). This method is demonstrated in Listing B-6 only to prepare you for situations where you want to fill the polygon with a pattern. You cannot open a polygon and use the `FillPoly` procedure (also described in the chapter "QuickDraw Drawing" in this book), because the PostScript driver "owns" the polygon concept at this point and captures—and ignores—all line drawing between the `PolyIgnore` and `PolyEnd` comments. Regions do not interfere with polygons, however, and they can be used to paint or fill the polygonal shape.

Rotating Graphics

You can rotate QuickDraw objects on PostScript printers. The printer driver rotates the entire PostScript coordinate space before drawing the objects, which then appear rotated. All objects that you want to rotate must be contained between the `RotateBegin` and `RotateEnd` picture comments.

You specify the center of rotation with the `RotateCenter` picture comment. Unlike text rotation, where you pass the `TextBegin` picture comment first and then the `RotateCenter` picture comment, you must pass the offset (which is relative to the center of rotation) with the `RotateCenter` picture comment *before* you use the `RotateBegin` picture comment. When you specify the `RotateCenter` picture comment in the `kind` parameter of the `PicComment` procedure, you also supply in the `dataHandle` parameter a `TCenterHdl` handle, which is a handle to a `TCenterRec` record. You can use this record to specify the center of rotation for graphics or text. See "Rotating Text" beginning on page B-17 for a description of the fields of a `TCenterRec` record.

When you specify the `RotateBegin` picture comment in the `kind` parameter of the `PicComment` procedure, you also supply in the `dataHandle` parameter a `TRotationHdl` handle, which is a handle to a `TRotationRec` record. You use a `TRotationRec` record to specify the rotation of a graphic. Here's how you should declare these as Pascal data structures:

```
TYPE
   TRotationHdl = ^TRotationPtr;
   TRotationPtr = ^TRotationRec;
   TRotationRec =
   RECORD
      rFlip:         Integer; {horizontal/vertical flipping}
      rAngle:        Integer; {0..360 clockwise rotation in }
                              { integer format}
      rAngleFixed:   Fixed;   {0..360 clockwise rotation in }
                              { fixed-number format}
   END;
```

You use the `rFlip` field to specify whether to flip the graphic horizontally or vertically in addition to rotating it. Here are the possible values for this field:

Value	Description
0	No coordinate flip
1	Horizontal coordinate flip
2	Vertical coordinate flip

You supply the `rAngleFixed` field with a fixed-point number to specify the number of degrees by which the printer driver should rotate the graphic.

You can provide the degrees of rotation both as an integer (in the `rAngle` field) and as a fixed-point number (in the `rAngleFixed` field). You should always specify the rotation in both fields, even for drivers that support only integral rotation.

Once you set up the rotation with the `RotateCenter` and `RotateBegin` picture comments, you draw the graphics objects you want to rotate. Before drawing the objects, use the `PenMode` procedure to set the pattern mode to a value of 23, which represents a special pattern mode for PostScript printer drivers. You should draw the QuickDraw image, using the `CopyBits` procedure, inside its own pair of `PostScriptBegin` and `PostScriptEnd` comments so that the QuickDraw representation will not show up on PostScript devices. (You should also use the `PrGeneral` procedure with the `getRslDataOp` opcode, described in the chapter "Printing Manager" in this book, to determine and use the maximum printer resolution.)

In Listing B-7, the application-defined procedure `MyRotateDemo` rotates the same image for both QuickDraw and PostScript printers.

Listing B-7 Using picture comments to rotate graphics

```
PROCEDURE MyRotateDemo;
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;
   MyPSRotatedRect(spinRect,delta,angle);
   {QuickDraw equivalent of the rotated object, hidden from the PostScript }
   { driver because of PostScriptBegin and PostScriptEnd}
   PicComment(PostScriptBegin, 0, NIL);
   MyQDRotatedRect(spinRect, delta, angle);
   PicComment(PostScriptEnd, 0, NIL);
END;
```

The application-defined procedure `MyQDRotatedRect` rotates the four points of the rectangle by an angle around the center and draws the rotated rectangle. To include this QuickDraw representation of the rotated objects (in case the `RotateCenter` and `RotateBegin` picture comments are not supported), the code in Listing B-7 assumes that only PostScript drivers implement these comments. The only way to hide from the driver the application-defined procedure that provides a QuickDraw representation of the rotated objects is to surround it by `PostScriptBegin` and `PostScriptEnd` comments.

To hide from QuickDraw the graphics rotation for a PostScript printer, Listing B-8 uses pattern mode 23.

Listing B-8 Using the `RotateCenter`, `RotateBegin`, and `RotateEnd` picture comments

```
PROCEDURE MyPSRotatedRect(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" pattern 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);
   MyFlushGrafPortState;   {see Listing B-1 on page B-10}
   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;
```

Using Line-Drawing Picture Comments

Line-drawing picture comments, listed in Table B-1 on page B-5, provide your application with the ability to draw dashed lines (as described in the next section) and to display fractional line widths (as described in "Using Fractional Line Widths" on page B-35).

Drawing Dashed Lines

Your application may use dashed lines frequently, particularly if it is a spreadsheet or accounting application. You can use the `DashedLine` picture comment to draw dashed lines on capable printers without drawing each individual dash. You use the `DashedStop` picture comment to tell the printer driver when you are finished sending dashed line information.

When you use the `DashedLine` comment, the printer driver draws the indicated lines or rectangles. You should pass a handle to a `TDashedLineRec` record in the `dataHandle` parameter of the `PicComment` procedure. You use a `TDashedLineRec` record to specify how the dashed line should look. Here is how you should declare these as Pascal data structures:

```
TYPE
   TDashedLineHdl = ^TDashedLinePtr;
   TDashedLinePtr = ^TDashedLineRec;
   TDashedLineRec =
   PACKED RECORD
      offset:     SignedByte; {offset}
      centered:   SignedByte; {reserved; set to 0}
      intervals:  ARRAY[0..0] OF SignedByte;
                              {points for drawing and not }
                              { drawing dashes}
   END;
```

Use the `offset` field to specify an offset as with the PostScript `setdash` operator.

The `centered` field is reserved and should be set to 0. Your application must center the dashed lines.

In the `intervals` field, specify an array of dash intervals describing the number of points drawn for a dash and the number of points not drawn between them.

You must provide both a QuickDraw and a picture comment version of the dashed line. The code in Listing B-9 uses the `PostScriptBegin` and `PostScriptEnd` picture comments to hide QuickDraw code from PostScript, and it uses pattern mode 23 to render PostScript drawing invisible in QuickDraw.

Listing B-9 Using the `DashedLine` picture comment

```
PROCEDURE DashDemo;
CONST
   magicPen = 23;
   cx = 280;      {center along x-axis}
   cy = 280;      {center along y-axis}
   r0 = 200;      {radius}
VAR
   dashHdl:    TDashedLineHdl;
   i:          Integer;
   a, rad:     Extended;
BEGIN
   PenSize(2,2);
   {First the PostScript picture comment version. Pattern mode }
   { 23 makes the line drawing invisible to 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!}
   DisposeHandle(Handle(dashHdl));
   PenMode(srcOr);  {no magic any more}
   {Now, the QuickDraw version. The PostScript driver must }
   { ignore it, so enclose it between PostScriptBegin and }
   { PostScriptEnd comments.}
   PicComment(PostScriptBegin, 0, NIL);
   PenSize(2,2);
```

```
   FOR i := 0 TO 9 DO
   BEGIN
      MoveTo(cx,cy);
      MyDashedQDLine(round(r0 * cos(i * 20 * rad)),
                     - round(r0 * sin(i * 20 * rad)), dashHdl);
   END;
   PicComment(PostScriptEnd, 0, NIL);
END;
```

Using Fractional Line Widths

Your application may need lines as thin as possible or thinner than the screen can display, especially if it is a desktop publishing, spreadsheet, or design application. You can draw ***hairlines*** (lines that are less than 1/72 of an inch wide) with printer drivers that support the `SetLineWidth` picture comment. Your application passes the printer driver a scaling factor (such as 1/4) that the driver applies to the pen size when rendering the picture.

QuickDraw and the PostScript language define 1 point to be 1/72 of an inch, so there are exactly 72 points per inch on the Macintosh screen. The resolution of a PostScript device such as the 300-dpi LaserWriter printer is about four times that of the screen, so the driver can render lines that are approximately 1/4 of a point thick, which is about 1/288 of an inch.

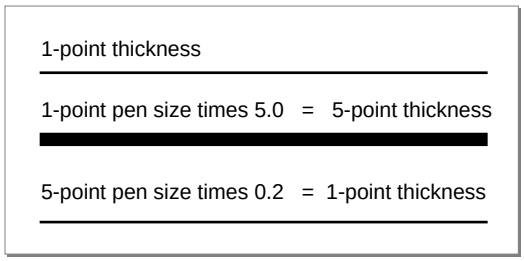
When you specify the `SetLineWidth` picture comment in the `kind` parameter of the `PicComment` procedure, you also specify a `TLineWidthHdl` handle (a handle to a data structure of type `TLineWidth`) in the `dataHandle` parameter. The `TLineWidth` data structure is defined by the `Point` data type. Here is how you should declare these as Pascal data types in your application:

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

Use the vertical coordinate of the point as the numerator and the horizontal coordinate as the denominator of the scaling factor: the driver multiplies the horizontal and vertical components of the pen by the scaling factor to obtain the new pen width. For example, if you have a pen size of (1,2) and your `SetLineWidth` picture comment uses 2 for the horizontal and 7 for the vertical, the pen size will then be $(7/2) \times 1$ pixel wide and $(7/2) \times 2$ pixels high.

In Figure B-9, the original pen size is 1 point. The first scaling factor is 5.0 or (5,1), which gives the pen a width of 5 points. The second scaling factor, applied to the new pen width, is 0.2 or (1,5), which gives the pen a width of 1 point again.

Figure B-9 Changing the pen width using the `SetLineWidth` picture comment



The `SetLineWidth` picture comment is implemented by all PostScript LaserWriter printer drivers and by some QuickDraw printer drivers. However, not all QuickDraw printer drivers support `SetLineWidth`, and there is no backup solution for cases where it is not supported. Among QuickDraw printer drivers that do support `SetLineWidth`, some drivers emulate PostScript printer drivers, while others—such as the QuickDraw LaserWriter SC driver—implement `SetLineWidth` differently.

The difference between the implementations of the `SetLineWidth` comment by the PostScript LaserWriter driver and the QuickDraw LaserWriter SC driver is apparent as soon as `SetLineWidth` is used a second time. The PostScript driver keeps an internal line-scaling factor, which 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 by the new value passed through `SetLineWidth`.

To support both implementations, you must always use an additional `SetLineWidth` picture comment to reset the PostScript driver line width to 1.0 before scaling to a new value width, as illustrated by the following lines of code:

```
PicComment(SetLineWidth, SizeOf(TLineWidth), Handle(1/oldLineWidth));
PicComment(SetLineWidth, SizeOf(TLineWidth), Handle(newLineWidth));
```

For example, suppose your application set the line width to 0.25, and now it needs a line width of 0.5. The following two `SetLineWidth` comments have the desired effect on all PostScript and QuickDraw drivers that implement the `SetLineWidth` comment.

Current line width, PS driver	Current line width, QD driver	Value passed along with `SetLineWidth`	New line width, PS driver	New line width, QD driver
0.25	0.25	4/1	1.0	4.0
1.0	4.0	1/2	0.5	0.5

The sample code in Listing B-10 gives the expected results on PostScript LaserWriter and QuickDraw printer drivers that implement the `SetLineWidth` comment.

Listing B-10 Using the `SetLineWidth` picture comment

```
PROCEDURE MySetNewLineWidth(oldWidth,newWidth: TLineWidth);
VAR
   tempWidthH: TLineWidthHdl;
BEGIN
   tempWidthH := TLineWidthHdl(NewHandle(SizeOf(TLineWidth)));
   tempWidthH^^.v := oldWidth.h;
   tempWidthH^^.h := oldWidth.v;
   PicComment(SetLineWidth, SizeOf(TLineWidth), Handle(tempWidthH));
   tempWidthH^^ := newWidth;
   PicComment(SetLineWidth, SizeOf(TLineWidth), Handle(tempWidthH));
   DisposeHandle(Handle(tempWidthH));
END;

PROCEDURE MyLineWidthDemo;
CONST
   y0 = 50;    {top left of demo}
   x0 = 50;
   d0 = 440;   {length of horizontal lines}
   e0 = 5;     {distance between lines}
   kN = 5;     {number of lines}
VAR
   oldWidth,newWidth:   TLineWidth;
   i,j,y:               Integer;
BEGIN
   PenNormal;
   y := y0;
   SetPt(oldWidth,1,1);            {initial line width = 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;
```

Using PostScript Picture Comments

You can access the PostScript language directly using the `PostScriptHandle` picture comment, and so bypass QuickDraw entirely. When you send PostScript code directly to the printer driver, it sends your code directly to the printer with no preprocessing and no error checking.

Note
These picture comments affect the state of the PostScript drawing environment and can have such effects as printing blank pages. Also, many PostScript printer drivers do not use the same version of PostScript and produce different outputs with the same commands; you should test your code on as many PostScript printers as possible. In all cases, use the PostScript picture comments with extreme caution. ◆

Calling PostScript Routines Directly

Your application can tell the printer driver to disable all QuickDraw drawing routines by using the `PostScriptBegin` picture comment. The driver uses the PostScript `save` and `restore` operators to preserve the state of the PostScript interpreter. When the driver receives the `PostScriptEnd` picture comment, it reenables QuickDraw drawing routines.

You send PostScript code to the driver via the `PostScriptHandle` picture comment by including a handle to the PostScript code in the `dataHandle` parameter of the `PicComment` procedure. The driver performs no preprocessing or error checking on this code. The handle contains text with no length byte or word; use the `dataSize` parameter to convey the length of the PostScript code. (As with all picture comments, the handle you pass belongs to you, and you must dispose of it when you're finished with it.) You indicate the end of the PostScript commands with a carriage return (ASCII $0D). You must use `PostScriptBegin` and `PostScriptEnd` around any `PostScriptHandle` comments; otherwise, the PostScript driver will not properly save and restore the PostScript drawing environment.

Listing B-11 gives an example of an application-defined procedure called `DoPostScriptLine`. The procedure is used to transmit a string of PostScript code through the `PostScriptHandle` picture comment to the PostScript printer driver. `DoPostScriptLine` should be called only between `PostScriptBegin` and `PostScriptEnd` picture comments, as shown in the application-defined procedure `DoPostScriptComments`.

Listing B-11 Sending PostScript code directly to the printer

```
PROCEDURE DoPostScriptLine(s: Str255);
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 DoPostScriptComments;
BEGIN
   {first, the simple example}
   PicComment(PostScriptBegin,0,NIL);
   DoPostScriptLine('100 100 moveto 0 100 rlineto 100 0 rlineto ');
   DoPostScriptLine('0 -100 rlineto -100 0 rlineto');
   DoPostScriptLine('stroke');
   MoveTo(30,30);
   DrawString('This text does not appear on PostScript printers.');
   PicComment(PostScriptEnd,0,NIL);
END;
```


Optimizing PostScript Printing

Although your printing code should be device-independent, you can optimize it for a PostScript printer. However, you cannot be sure that the current printer is a PostScript printer, so you may need to create two versions of the same drawing code: one for a PostScript printer and one for a QuickDraw printer, as described previously in this appendix.

For printing to a PostScript printer, you'll need to observe the following limitations:

- Regions aren't supported; try to simulate them with polygons or bitmaps.
- Clipping regions should be limited to rectangles. PostScript clips nonsquare patterns to squares.
- The `Invert` data type, part of the QuickDraw `GrafVerb` data type, is not supported by the PostScript LaserWriter printer driver.
- The PostScript LaserWriter driver does not support all Boolean transfer modes. It supports the `srcCopy`, `srcOr`, `srcBic`, `notSrcCopy`, and `notSrcBic` modes for

bitmaps and text. For all other objects drawn with QuickDraw, the PostScript LaserWriter driver supports only the `srcCopy` mode.

- There can be a small difference in glyph widths between fonts rendered on the screen and on the printer. Only the endpoints of text strings are the same.
- Only PostScript Level 2 supports color patterns that use colors other than red, green, blue, cyan, yellow, magenta, white, and black.
- The printer may print some large patterns at half size or smaller sizes, depending on its resolution.
- Polygons and smoothed polygons that result in the creation of paths larger than the limit of the PostScript printer (typically 1500 or 3000, depending on the version of PostScript) result in a PostScript error.

Although the PostScript LaserWriter printer is relatively fast, there are some techniques an application can use to ensure its maximum performance.

- Printing patterns takes time, because the bitmap for the pattern has to be built. The black-and-white patterns, and some of the gray patterns, have been optimized to use the PostScript grayscales.
- Use the `TextBegin` picture comment for text alignment. In the cases of flush left, flush right, or centered alignment, only the left, right, or center points are accurate, respectively; in the case of fully justified text, both the left and right endpoints are accurate.
- If you want to position each glyph independently, use the `LineLayoutOff` and `StringBegin` picture comments. If you are trying to position glyphs and the driver is trying to position glyphs too, there is conflict, and printing takes much longer than necessary.

For more information on the PostScript language, see the *PostScript Language Reference Manual*, second edition, available from Addison-Wesley.

Picture Comments to Avoid

The `SetGrayLevel` picture comment is now obsolete. The `PostScriptFile`, `TextIsPostScript`, `FormsPrinting`, `EndFormsPrinting`, `ClientLineLayout`, `PSBeginNoSave`, and `ResourcePS` picture comments have limited use and are not recommended. This section describes the shortcomings of these picture comments.

The `SetGrayLevel` picture comment was designed to provide access to the PostScript `setgray` operator while drawing with QuickDraw in black-and-white mode. For most drawing operations, however, the printer driver sets the gray level to match the foreground color for the printing graphics port, and the effect of the `SetGrayLevel` picture comment is often unpredictable. If direct access to the PostScript `setgray` operator seems desirable, it is preferable to send the instruction with the `PostScriptHandle` picture comment.

The `TextIsPostScript` picture comment interprets all the text manipulated with QuickDraw text-drawing routines (namely, `DrawChar`, `DrawString`, `DrawText`, and

anything else that calls the `StdText` low-level procedure) as PostScript code. 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 support the `TextIsPostScript` picture comment will print the PostScript text instead of interpreting it. If you need to transmit 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 resource file. The resource file is expected to be open at the time that you use `ResourcePS`. Under background printing, there are no guarantees the resource file will still be open when the Printing Manager needs it. If you want to keep PostScript code in a resource file, it is easy to write a routine that loads the resources and sends their contents using the `PostScriptHandle` picture comment.

The `PostScriptFile` picture comment loads PostScript code from a file; as with the `ResourcePS` comment, there are no guarantees the file will be open when the Printing Manager needs it during background printing. If you want to keep PostScript code in a file, it is easy to write a routine that loads the file and its contents using the `PostScriptHandle` picture comment.

As with the `PostScriptBegin` picture comment, the `PSBeginNoSave` picture comment allows applications to change the state of a PostScript printer driver. Some applications do not want to restore the previous state of the PostScript interpreter after sending PostScript code; the `PSBeginNoSave` comment was intended for situations where applications do not want to preserve the printer state. However, the `PSBeginNoSave` picture comment allows applications to interfere with the LaserWriter 8.0 printer driver, and the driver, by calling the PostScript operator `grestore`, can interfere with the application. The use of `PSBeginNoSave` can lead to incorrect clipping, incorrect colors, and PostScript language errors and should therefore be avoided.

By default, most drivers apply about 80 percent of the total line layout error to the major glyphs (the space character) and the other 20 percent to the minor glyphs (all other glyphs). (When using a script system that does not use the space glyph to delimit words, the layout error is distributed evenly across all characters in the font.) The `ClientLineLayout` picture comment allows applications to redefine the major glyph, and the percentages of the line layout error assigned to the major and minor glyphs. The `ClientLineLayout` picture comment is rather subtle and very specific to the PostScript LaserWriter driver. Only very ambitious page layout applications might be interested in this functionality, however; their designers should instead aim at a more general scheme of line layout control that does not rely upon this very driver-specific picture comment.

Intended for printing forms on PostScript LaserWriter printers, the `FormsPrinting` picture comment directs the PostScript LaserWriter driver not to clear its page buffer after printing a page. The `EndFormsPrinting` picture comment directs the PostScript LaserWriter driver to clear its page buffer after printing a page. When a page is completed, applications 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. However, you need to write a separate printing loop for the PostScript LaserWriter driver if you want to use this comment.

Including Constants and Data Types for Picture Comments

For the picture comments described in this appendix, neither QuickDraw nor the Printing Manager includes constant definitions or data type declarations; instead, you must include these in your own build files. Listed here are the constants and data types for picture comments that have been predefined for printer drivers from Apple Computer, Inc.

```
{PicComments.p}
CONST
   {values for picture comments}
   TextBegin         = 150;
   TextEnd           = 151;
   StringBegin       = 152;
   StringEnd         = 153;
   TextCenter        = 154;
   LineLayoutOff     = 155;
   LineLayoutOn      = 156;
   ClientLineLayout  = 157;   {considered to be of limited usefulness}
   PolyBegin         = 160;
   PolyEnd           = 161;
   PolyIgnore        = 163;
   PolySmooth        = 164;
   PolyClose         = 165;
   DashedLine        = 180;
   DashedStop        = 181;
   SetLineWidth      = 182;
   PostScriptBegin   = 190;
   PostScriptEnd     = 191;
   PostScriptHandle  = 192;
   PostScriptFile    = 193;   {considered to be of limited usefulness}
   TextIsPostScript  = 194;   {considered to be of limited usefulness}
   ResourcePS        = 195;   {considered to be of limited usefulness}
   PSBeginNoSave     = 196;   {dangerous to use with LaserWriter 8.0}
   SetGrayLevel      = 197;   {this comment now obsolete}
   RotateBegin       = 200;
   RotateEnd         = 201;
   RotateCenter      = 202;
   {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;       {justification for line layout of text}
      tFlip:         Byte;       {horizontal or vertical flipping}
      tAngle:        Integer;    {0..360 degrees clockwise rotation }
                                 { in integer format}
      tLine:         Byte;       {reserved}
      tCmnt:         Byte;       {reserved}
      tAngleFixed:   Byte;       {0..360 degrees clockwise rotation in }
                                 { fixed-number format}
   END;

   TRotationHdl = ^TRotationPtr;
   TRotationPtr = ^TRotationRec;
   TRotationRec = RECORD
      rFlip:         Integer;    {horizontal/vertical flipping}
      rAngle:        Integer;    {0..360 degrees clockwise rotation }
                                 { in integer format}
      rAngleFixed:   Fixed;      {0..360 degrees clockwise rotation in }
                                 { fixed-number format}
   END;

   TCenterHdl = ^TCenterPtr;
   TCenterPtr = ^TCenterRec;
   TCenterRec = RECORD
      y:    Fixed;   {vertical offset from current pen location}
      x:    Fixed;   {horizontal offset from current pen location}
   END;

   TPolyVerbHdl = ^TPolyVerbPtr;
   TPolyVerbPtr = ^TPolyVerbRec;
```

```
TPolyVerbRec = PACKED RECORD
   f7, f6, f5, f4, f3:  Boolean; {reserved; set to 0}
   fPolyClose:          Boolean; {TRUE is same as PolyClose }
                                 { picture comment}
   fPolyFill:           Boolean; {TRUE means fill polygon}
   fPolyFrame:          Boolean; {TRUE means frame polygon}
END;

TDashedLineHdl = ^TDashedLinePtr;
TDashedLinePtr = ^TDashedLineRec;
TDashedLineRec = PACKED RECORD
   offset:     SignedByte;    {offset into pattern for first dash}
   centered:   SignedByte;    {reserved; set to 0}
   intervals:  ARRAY[0..5] OF SignedByte;
                              {points for drawing and not drawing dashes}

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