Macintosh Technical Notes

#91: Optimizing for the LaserWriter—Picture Comments

See also: The Print Manager
QuickDraw
Technical Note #72—
Optimizing for the LaserWriter—Techniques
Technical Note #27—MacDraw Picture Comments
PostScript Language Reference Manual, Adobe Systems
PostScript Language Tutorial and Cookbook,
Adobe Systems
LaserWriter Reference Manual

Written by:	Ginger Jernigan	November 15, 1986
Modified by:	Ginger Jernigan	March 2, 1987
Updated:		March 1, 1988

This technical note is a continuation of Technical Note #72. This technical note discusses the picture comments that the LaserWriter driver recognizes.

This technical note has been modified to include corrected descriptions of the `SetLineWidth`, `PostScriptFile` and `ResourcePS` comments and to include some additional warnings.

The implementation of QuickDraw's `picComment` facility by the LaserWriter driver allows you to take advantage of features (like rotated text) which are available in PostScript but may not be available in QuickDraw.

Warning: Using PostScript-specific comments will make your code printer-dependent and may cause compatibility problems with non-PostScript devices, so don't use them unless you absolutely have to.

Some of the picture comments below are designed to be issued along with QuickDraw commands that simulate the commented commands on the Macintosh screen. When the comments are used, the accompanying QuickDraw comments are ignored. If you are designing a picture to be printed by the LaserWriter, the structure and use of these comments must be precise, otherwise nothing will print. If another printer driver (like the ImageWriter I/II driver) has not implemented these comments, the comments are ignored and the accompanying QuickDraw commands are used.

Below are the picture comments that the LaserWriter driver recognizes:

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	Turns LaserWriter line layout off
* LineLayoutOn	156	0	NIL	Turns LaserWriter line layout on
PolyBegin	160	0	NIL	Begin special polygon
PolyEnd	161	0	NIL	End special polygon
PolyIgnore	163	0	NIL	Ignore following poly data
PolySmooth	164	1	PolyVerb	Close, Fill, Frame
picPlyClo	165	0	NIL	Close the poly
* 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
** 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 only implemented in LaserWriter driver 3.0 or later.
** These comments are only implemented in LaserWriter driver 3.1 or later.
† These comments are not available when background printing is enabled.

Each of these comments are discussed below in six groups: Text, Polygons, Lines, PostScript, Rotation, and Forms. Code examples are given where appropriate. For other examples of how to use picture comments for printing please see the Print example program in the Software Supplement (currently available through APDA as “Macintosh Example Applications and Sources 1.0”).

Note: The examples used in the *LaserWriter Reference Manual* are incorrect. Please use the examples presented here instead.

Text

In order to support the What-You-See-Is-What-You-Get paradigm, the LaserWriter driver uses a line layout algorithm to assure that the placement of the line on the printer closely approximates the placement of the line on the screen. This means that the printer driver gets the width of the line from QuickDraw, then tells PostScript to place the text in exactly the same place with the same width.

The `TextBegin` comment allows the application to specify the layout and the orientation of the text that follows it by specifying the following information:

```
TTxtPicRec = PACKED RECORD
   tJus:  Byte;    {0,1,2,3,4 or greater => none, left, center, right, full
                   justification }
   tFlip: Byte;    {0,1,2 => none, horizontal, vertical coordinate flip }
   tRot:  INTEGER; {0..360 => clockwise rotation in degrees }
   tLine: Byte;    {1,2,3.. => single, 1-1/2, double.. spacing }
   tCmnt: Byte;    {Reserved }
END; { TTxtPicRec }
```

Left, right or center justification, specified by `tJust`, tells the driver to maintain only the left, right or center point, without recalculating the interword spacing. Full justification specifies that both endpoints be maintained and interword spacing be recalculated. This means that the driver makes sure that the specified points are maintained on the printer without caring whether the overall width has changed. Full justification means that the overall width of the line has been maintained. `tFlip` and `tRot` specify the orientation of the text, allowing the application to take advantage of the rotation features of PostScript. `tLine` specifies the interline spacing. When no `TextBegin` comment is used, the defaults are full justification, no rotation and single-spaced lines.

String Reconstruction

The `StringBegin` and `StringEnd` comments are used to bracket short strings of text that are actually sections of an original long string. MacDraw, for instance, breaks long strings into shorter pieces to avoid stack overflow problems with QuickDraw in the 64K ROM. When these smaller strings are bracketed by `StringBegin` and `StringEnd`, the LaserWriter driver assumes that the enclosed strings are parts of one long string and will perform its line layout accordingly. Erasing or filling of background rectangles should take place before the `StringBegin` comment to avoid confusing the process of putting the smaller strings back together.

Text Rotation

In order to rotate a text object, PostScript needs to have information concerning the center of rotation. The `TextCenter` comment provides this information when a rotation is specified in the `TextBegin` comment. This comment contains the offset from the present pen location to the center of rotation. The offset is given as the y-component, then the x-component, which are declared as fixed-point numbers. This allows the center to be in the middle of a pixel. This comment should appear after the `TextBegin` comment and before the first following `StringBegin` comment.

The associated comment data looks like this:

```
TTxtCenter = RECORD
   y,x: Fixed;      {offset from current pen location to center of rotation}
END; { TTxtCenter }
```

Right after a `TextBegin` comment, the LaserWriter driver expects to see a `TextCenter` comment specifying the center of rotation for any text enclosed within the text comment calls. It will ignore all further `CopyBits` calls, and print all standard text calls in the rotation specified by the information in `TTxtPicRec`. The center of rotation is the offset from the beginning position of the first string following the `TextCenter` comment. The printer driver also expects the string locations to be in the coordinate system of the current QuickDraw port. The printer driver rotates the entire port to draw the text so it can draw several strings with one rotation comment and one center comment. It is good practice to enclose an entire paragraph or paragraphs of text in a single rotation comment so that the driver makes the fewest number of rotations.

The printer driver can draw non-textual objects within the bounds of the text rotation comments but it must unrotate to draw the object, then re-rotate to draw the next string of text. To do this the printer driver must receive another `TextCenter` comment before each new rotation. So, rotated text and unrotated objects can be drawn inter-mixed within one `TextBegin`/`TextEnd` comment pair, but performance is slowed.

Note that all bit maps and all clip regions are ignored during text rotation so that clip regions can be used to clip out the strings on printers that can’t take advantage of these comments. This has the unfortunate side effect of not allowing rotated text to be clipped.

Rotated text comments are not associated with landscape and portrait orientation of the printer paper as selected by the Page Setup dialog. These are rotations with reference to the current QuickDraw port only.

All of the above text comments are terminated by a `TextEnd` comment.

Turning Off Line Layout

If your application is using its own line layout algorithm (it uses its own character widths or does its own character or word placement), the printer driver doesn’t need to do it too. To turn off line layout, you can use the `LineLayoutOff` comment. `LineLayoutOn` turns it on again.

Turning on `FractEnable` for the 128K ROMs has the same effect as `LineLayoutOff`. When the driver detects that `FractEnable` has been turned on, line layout is not performed. The driver assumes that all text being printed is already spaced correctly for the LaserWriter and just sends it as is.

Polygons

The polygon comments are recognized by the LaserWriter driver because they are used by MacDraw as an alternate method of defining polygons.

The `PolyBegin` and `PolyEnd` comments bracket polygon line segments, giving an alternate way to specify a polygon. All `StdLine` calls between these two comments are part of the polygon. The endpoints of the lines are the vertices of the polygon.

The `picPlyClo` comment specifies that the current polygon should be closed. This comes immediately after `PolyBegin`, if at all. It is not sufficient to simply check for `begPt = endPt`, since MacDraw allows you to create a “closed” polygon that isn’t really closed. This comment is especially critical for smooth curves because it can make the difference between having a sharp corner or not in the curve.

These comments also work with the `StdPoly` call. If a `FillRgn` is encountered before the `PolyEnd` comment, then the polygon is filled. Unlike QuickDraw polygons, comment polygons do not require an initial `MoveTo` call within the scope of the polygon comment. The polygon will be drawn using the current pen location at the time the polygon comment is received. The pen must be set before the polygon comment is called.

Splines

A spline is a method used to determine the smallest number of points that define a curve. In MacDraw, splines are used as a method for smoothing polygons. The vertices of the underlying unsmoothed polygon are the control nodes for the quadratic B-spline curve which is drawn. PostScript has a direct facility for cubic B-splines and the LaserWriter translates the quadratic B-spline nodes it gets into the appropriate nodes for a cubic B-spline that will exactly emulate the original quadratic B-spline.

The `PolySmooth` comment specifies that the current polygon should be smoothed. This comment also contains data that provides a means of specifying which verbs to use on the smoothed polygon (bits 7 through 3 are not currently assigned):

```
TPolyVerb = PACKED RECORD
    f7, f6, f5, f4, f3, fPolyClose, fPolyFill, fPolyframe : Boolean;
END; { TPolyVerb }
```

Although the closing information is redundant with the `picPlyClo` comment, it is included for the convenience of the LaserWriter.

The LaserWriter uses the pen size at the time the `PolyBegin` comment is received to frame the smoothed polygon if framing is called for by the `TPolyVerb` information. When the `PolyIgnore` comment is received by the LaserWriter driver, all further `StdLine` calls are ignored until the `PolyEnd` comment is encountered. For polygons that are to be smoothed, set the initial pen width to zero after the `PolyBegin` comment so that the unsmoothed polygon will not be drawn by other printers not equipped to handle polygon comments. To fill the polygon, call `StdRgn` with the fill verb and the appropriate pattern set, as well as specifying fill in the `PolySmooth` comment.

Lines

The `DashedLine` and `DashedLineStop` comments are used to communicate PostScript information for drawing dashed lines.

The `DashedLine` comment contains the following additional data:

```
  TDashedLine = PACKED RECORD
     offset:   SignedByte;                {Offset as specified by PostScript}
     centered: SignedByte;                {Whether dashed line should be
                       centered to begin and end points}
     dashed:   Array[0..1] of SignedByte;  {1st byte is # bytes following}
  END; { TDashedLine }
```

The printer driver sets up the PostScript dashed line command, as defined on page 214 of Adobe's *PostScript Language Reference Manual*, using the parameters specified in the comment. You can specify that the dashed line be centered between the begin and end points of the lines by making the `centered` field nonzero.

The `SetLineWidth` comment allows you to set the pen width of all subsequent objects drawn. The additional data is a point. The vertical portion of the point is the numerator and the horizontal portion is the denominator of the scaling factor that the horizontal and vertical components of the pen are then multiplied by to obtain the new pen width. For example, if you have a pen size of 1,2 and in your line width comment you use 2 for the horizontal of the point and 7 for the vertical, the pen size will then be (7/2)*1 pixels wide and (7/2)*2 pixels high.

Below is an example of how to use the line comments:

```
PROCEDURE LineTest;
{This procedure shows how to do dashed lines and how to change the line width}
CONST
 DashedLine = 180;
 DashedStop = 181;
 SetLineWidth = 182;

TYPE
 DashedHdl = ^DashedPtr;
 DashedPtr = ^TDashedLine;
 TDashedLine = PACKED RECORD
   offset: SignedByte;
   Centered: SignedByte;
   dashed: Array[0..1] of SignedByte;   { the 0th element is the length }
 END; { TDashedLine }
 widhdl = ^widptr;
 widptr = ^widpt;
 widpt = Point;


VAR
 arect    : rect;
 Width    : Widhdl;
 dashedln : DashedHdl;
```

```
BEGIN {LineTest}
 Dashedln := dashedhdl(NewHandle(sizeof(tdashedline)));
 Dashedln^^.offset := 0;       { No offset}
 Dashedln^^.centered := 0;     { don't center}
 Dashedln^^.dashed[0] := 1;    { this is the length }
 Dashedln^^.dashed[1] := 8;    { this means 8 points on, 8 points off }

 Width := widhdl(NewHandle(sizeof(widpt)));
 Width^^.h := 2;                { denominator is 2}
 Width^^.v := 7;               { numerator is 7}

 myPic := OpenPicture(theWorld);
   SetPen(1,2);                { Set the pen size to 1 wide x 2 high }
   ClipRect(theWorld);
   MoveTo(20,20);
   DrawString('Do line test');
   PicComment(DashedLine,GetHandleSize(Handle(dashedln)),Handle(dashedln));
   PicComment(SetLineWidth,4,Handle(width));   {SetLineWidth}
   SetRect(arect,100,100,500,500);
   FrameRect(aRect);
   MoveTo(500,500);
   Lineto(100,100);
   PicComment(DashedStop,0,nil);              {DashedStop}
 ClosePicture;
 DisposHandle(handle(width));                {Clean up}
 DisposHandle(handle(dashedln));
 PrintThePicture;                            {print it please}
 KillPicture(MyPic);
END; {LineTest}
```

PostScript

The PostScript comments tell the printer driver that the application is going to be communicating with the LaserWriter directly using PostScript commands instead of QuickDraw. The driver sends the accompanying PostScript to the printer with no preprocessing and no error checking. The application can specify data in the comment handle itself or point to another file which contains text to send to the printer. When the application is finished sending PostScript, the `PostScriptEnd` comment tells the printer driver to resume normal QuickDraw mode.

Any Quickdraw drawing commands made by the application between the `PostScriptBegin` and `PostScriptEnd` comments will be ignored by PostScript printers. In order to use PostScript in a device independent way, you should always include two representations of your document. The first representation should be a series of Quickdraw drawing commands. The second representation of your document should be a series of PostScript commands, sent to the Printing Manager via picture comments. This way, when you are printing to a PostScript device, the picture comments will be executed, and the Quickdraw commands ignored. When printing to a non-PostScript device, the picture comments will be ignored, and the Quickdraw commands will be executed. This method allows you to use PostScript, without having to ask the device if it supports it. This allows your application to get the best results with any printer, without being device dependent.

Here are some guidelines you need to remember:

- The graphic state set up during QuickDraw calls is maintained and is not affected by PostScript calls made with these comments.

- The header has changed a number of parameters so sometimes you won't get the results you expect. You may want to take a look at the header listed in *The LaserWriter Reference Manual* available through APDA.

- The header changes the PostScript coordinate system so that the origin is at the top-left corner of the page instead of at the bottom-left corner. This is done so that the QuickDraw coordinates that are used don't have to be remapped into the standard PostScript coordinate system. If you don't allow for this, all drawing is printed upside down. Please see the *PostScript Language Reference Manual* for details about transformation matrices.

- Don't call `showpage`. This is done for you by the driver. If you do, you won't be able to switch back to QuickDraw mode and an additional page will be printed when you call `PrClosePage`.
- Don't call `exitserver`. You may get very strange results.
- Don't call `initgraphics`. Graphics states are already set up by the header.

- Don't do anything that you expect to live across jobs.

- You won't be able to interrogate the printer to get information back through the driver.

The PostScriptBegin comment sets the driver state to prepare for the generation of PostScript by the application by calling gsave to save the current state. PostScript is then sent to the printer by using comments 192 through 195. The QuickDraw state of the driver is then restored by the PostScriptEnd comment. All QuickDraw operations that occur outside of these comments are performed; no clipping occurs as with the text rotation comments.

PostScript From a Text Handle

When the PostScriptHandle comment is used, the handle PSData points to the PostScript commands which are sent. PSData is a generic handle that points to text, without a length byte. The text is terminated by a carriage return. This comment is terminated by a PostScriptEnd comment.

Note: Due to a bug in the 3.1 LaserWriter driver, PostScriptEnd will not restore the QuickDraw state after the use of a PostScriptHandle comment. The workaround is to only use this comment at the end of your drawing, after you have made all the QuickDraw calls you need. This problem is fixed in more recent versions of the driver.

Here's an example of how to use this comment:

```
PROCEDURE PostHdl;
{this procedure shows how to use PostScript from a text Handle}
CONST
 PostScriptBegin = 190;
 PostScriptEnd = 191;
 PostScriptHandle = 192;

VAR
 MyString  : Str255;
 tempstr   : String[1];
 MyHandle  : Handle;
 err       : OSErr;

BEGIN { PostHdl }
 MyString := '/Times-Roman findfont 12 scalefont setfont 230 600 moveto
              (Hello World) show';
 tempstr:=' ';
 tempstr[1] := chr(13); {has to be terminated by a carriage return }
 MyString := Concat(MyString, tempstr); { in order for it to execute}
 err := PtrToHand (Pointer(ord(@myString)+1), MyHandle, length(MyString));
 MyPic := OpenPicture(theWorld);
   ClipRect(theWorld);
   MoveTo(20,20);
   DrawString('PostScript from a Handle');
   PicComment(PostScriptBegin,0,nil);       {Begin PostScript}
   PicComment(PostScriptHandle,length(mystring),MyHandle);
   PicComment(PostScriptEnd,0,nil);         {PostScript End}
 ClosePicture;
 DisposHandle(MyHandle);                    {Clean up}
 PrintThePicture;                           {print it please}
 KillPicture(MyPic);
END; { PostHdl }
```

Defining PostScript as QuickDraw Text

All QuickDraw text following the `TextIsPostScript` comment is sent as PostScript. No error checking is performed. This comment is terminated by a `PostScriptEnd` comment.

Here is an example:

```
PROCEDURE PostText;
{Shows how to use PostScript in strings in a QuickDraw picture}
CONST
 PostScriptBegin = 190;
 PostScriptEnd = 191;
 TextIsPostScript = 194;

BEGIN { PostTest }
 MyPic := OpenPicture(theWorld);
   ClipRect(theWorld);
   MoveTo(20,20);
   DrawString('TextIsPostScript Comment');
   PicComment(PostScriptBegin,0,nil);      {Begin PostScript}
   PicComment(TextIsPostScript,0,nil);     {following text is PostScript}
     DrawString('0 728 translate');         {move the origin and rotate the}
     DrawString('1 -1 scale');              {coordinate system}

     DrawString('newpath');
     DrawString('100 470 moveto');
     DrawString('500 470 lineto');
     DrawString('100 330 moveto');
     DrawString('500 330 lineto');
     DrawString('230 600 moveto');
     DrawString('230 200 lineto');
     DrawString('370 600 moveto');
     DrawString('370 200 lineto');
     DrawString('10 setlinewidth');
     DrawString('stroke');
     DrawString('/Times-Roman findfont 12 scalefont setfont');
     DrawString('230 600 moveto');
     DrawString('(Hello World) show');
   PicComment(PostScriptEnd,0,nil);        {PostScriptEnd}
 ClosePicture;
 PrintThePicture;                                {print it please}
 KillPicture(MyPic);
END; { PostText }
```

PostScript From a File

The `PostScriptFile` and `ResourcePS` comments allow you to send PostScript to the printer from a resource file. Before these comments are described there are some restrictions you need to follow:

- Don't ever copy a picture containing these comments to the clipboard. If it is pasted into another application and the specified file or resource is not available, printing will be aborted and the user won't know what went wrong. This could be very confusing to a user. If you want the PostScript information to be available when printed from another application, use one of the other comments and include the information in the picture.
- Don't keep the PostScript in a separate file from the actual data file. If the data file ever gets moved without the PostScript file, when the picture is printed the data file may not be found and the print job will be aborted, again without the user knowing what went wrong. Keeping the data and PostScript in the same file will forestall many headaches for you and the user.

Now, a description of the comments:

The `PostScriptFile` comment tells the driver to use the POST type resources contained in the file `FileNameString`. `FileNameString` is declared as a `Str255`.

When this comment is encountered, the driver calls `OpenResFile` using the file name specified in `FileNameString`. It then calls `GetResource('POST',theID);` repeatedly, where `theID` begins at 501 and is incremented by one for each `GetResource` call. If the driver gets a `ResNotFound` error, it closes the specified resource file. If the first byte of the resource is a 3, 4, or 5 then the remaining data is sent and the file is closed.

The format of the POST resource is as follows: The IDs of the resources start at 501 and are incremented by one for each resource. Each resource begins with a 2 byte data field containing the data type in the first byte and a zero in the second. The possible values for the first byte are:

0 ignore the rest of this resource (a comment)
1 data is ASCII text
2 data is binary and is first converted to ASCII before being sent
3 AppleTalk end of file. The rest of the data, if there is any, is interpreted as ASCII text and will be sent after the EOF.
4 open the data fork of the current resource file and send the ASCII text there
5 end of the resource file

The second byte of the field must always be zero. Resources should be kept small, around 2K. Text and binary should not be mixed in the same resource. Make sure you include either a space or a return at the end of each PostScript string to separate it from the following command.

Here's an example:

```
PROCEDURE PostFile;
{This procedure shows how to use PostScript from a specified FILE}
CONST
  PostScriptBegin = 190;
  PostScriptFile = 193;
  PostScriptEnd = 191;

VAR
  MyString         : Str255;
  MyHandle         : Handle;
  err              : OSErr;

BEGIN  { PostFile }
  {You should never do this in a real program. This is only a test.}
  MyString := 'HardDisk:MPW:Print Examples:PSTestDoc';
  err := PtrToHand(pointer(MyString),MyHandle,length(MyString) + 1);
  MyPic := OpenPicture(theWorld);
  ClipRect(theWorld);
  MoveTo(20,20);
  DrawString('PostScriptFile Comment');
  PicComment(PostScriptBegin,0,nil); {Begin PostScript}
  PicComment(PostScriptFile,GetHandleSize(MyHandle),MyHandle);
  PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
  MoveTo(50,50);
  DrawString('PostScriptEnd has terminated');
  ClosePicture;
  DisposHandle(MyHandle); {Clean up}
  PrintthePicture; {print it please}
  KillPicture(MyPic);
END;   { PostFile }
```

Here are the resources:

```
type 'POST' {
  switch {
    case Comment:           /* this is a comment */
         key bitstring[8] = 0;
         fill byte;
         string;                                              case ASCII:
/* this is just ASCII text */
         key bitstring[8] = 1;
         fill byte;
         string;                                              case Bin:
/* this is binary */
         key bitstring[8] = 2;
         fill byte;
         string;

    case ATEOF:            /* this is an AppleTalk EOF */
         key bitstring[8] = 3;
         fill byte;
         string;
```

```
     case DataFork:         /* send the text in the data fork */
          key bitstring[8] = 4;
          fill byte;

     case EOF:              /* no more */
          key bitstring[8] = 5;
          fill byte;
     };
};

resource 'POST' (501) {
ASCII{"0 728 translate "}};

resource 'POST' (502) {
ASCII{"1 -1 scale "}};

resource 'POST' (503) {
ASCII{"newpath "}};

resource 'POST' (504) {
ASCII{"100 470 moveto "}};

resource 'POST' (505) {
ASCII{"500 470 lineto "}};

resource 'POST' (506) {
ASCII{"100 330 moveto "}};

resource 'POST' (507) {
ASCII{"500 330 lineto "}};

resource 'POST' (508) {
ASCII{"230 600 moveto "}};

resource 'POST' (509) {
ASCII{"230 200 lineto "}};

resource 'POST' (510) {
ASCII{"370 600 moveto "}};

resource 'POST' (511) {
ASCII{"370 200 lineto "}};

resource 'POST' (512) {
ASCII{"10 setlinewidth "}};

resource 'POST' (513) {
ASCII{"stroke "}};

resource 'POST' (514) {
ASCII{"/Times-Roman findfont 12 scalefont setfont "}};

resource 'POST' (515) {
ASCII{"230 600 moveto "}};

resource 'POST' (516) {
ASCII{"(Hello World) show "}};
```

```
/* It will stop reading and close the file after 517 */
  resource 'POST' (517) {
  EOF
  {}};

/* it never gets here */
  resource 'POST' (518) {
  DataFork
  {}};
```

When the `ResourcePS` comment is encountered, the LaserWriter driver sends the text contained in the specified resource as PostScript to the printer. The additional data is defined as

```
PSRsrc = RECORD
           PSType : ResType;
           PSID   : INTEGER;
           PSIndex: INTEGER;
         END;
```

The resource can be of type STR or STR#. If the Type is STR then the index should be 0. Otherwise an index should be given.

This comment is essentially the same as the PrintF control call to the driver. The imbedded command string it uses is `'^r^n'`, which basically tells the driver to send the string specified by the additional data, then send a newline. For more information about printer control calls see the *LaserWriter Reference Manual*.

Here's an example:

```
PROCEDURE PostRSRC;
{This procedure shows how to get PostScript from a resource FILE}
  CONST
    PostScriptBegin = 190;
    PostScriptEnd = 191;
    ResourcePS = 195;

  TYPE
    theRSRChdl = ^theRSRCptr;
    theRSRCptr = ^theRSRC;
    theRSRC = RECORD
       theType: ResType;
       theID: INTEGER;
       Index: INTEGER;
    END;

  VAR
    temp         : Rect;
    TheResource  : theRSRChdl;
    i,j          : INTEGER;
    myport       : GrafPtr;
    err          : INTEGER;
    atemp        : Boolean;
```

```
  BEGIN          { PostRSRC }
    TheResource := theRSRChdl(NewHandle(SizeOf(theRSRC)));
    TheResource^^.theID := 500;
    TheResource^^.Index := 0;
    TheResource^^.theType := 'STR ';
    HLock(Handle(TheResource));
    MyPic := OpenPicture(theWorld);
    DrawString('ResourcePS Comment');
    PicComment(PostScriptBegin,0,nil); {Begin PostScript}
    PicComment(ResourcePS,8,Handle(TheResource)); {Send postscript}
    PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
    ClosePicture;
    DisposHandle(Handle(TheResource)); {Clean up}
    PrintthePicture; {print it please}
    KillPicture(MyPic);
  END;           { PostRSRC }
```

Here's the resource:

```
resource 'STR ' (500)
{"0 728 translate 1 -1 scale newpath 100 470 moveto 500 470 lineto 100 330
moveto 500 330 lineto 230 600 moveto 230 200 lineto 370 600 moveto 370 200
lineto 10 setlinewidth stroke /Times-Roman findfont 12 scalefont setfont 230
600 moveto (Hello World) show"
};
```

Rotation

The concept of rotation doesn't apply to text alone. PostScript can rotate any object. The rotation comments work exactly like text rotation except that all objects drawn between the two comments are drawn in the rotated coordinate system specified by the center of rotation comment, not just text. Also, no clipping of `CopyBits` calls occurs. These comments only work on the 3.1 and newer LaserWriter drivers.

The `RotateBegin` comment tells the driver that the following objects will be drawn in a rotated plane. This comment contains the following data structure:

```
Rotation = RECORD
   Flip:  INTEGER; {0,1,2 => none, horizontal, vertical coordinate flip }
   Angle: INTEGER; {0..360 => clockwise rotation in degrees }
END; { Rotation }
```

When you are finished, the `RotateEnd` comment returns the coordinate system to normal, terminating the rotation.

The relative center of rotation is specified by the `RotateCenter` comment in exactly the same manner as the `TextCenter` comments. The difference, however, is that this comment **must** appear **before** the `RotateBegin` comment. The data structure of the accompanying handle is exactly like that for the `TextCenter` comment.

Here's an example of how to use rotation comments:

```
PROCEDURE Test;
{This procedure shows how to do rotations}
CONST
 RotateBegin = 200;
 RotateEnd = 201;
 RotateCenter = 202;

TYPE
   rothdl = ^rotptr;
   rotptr = ^trot;
   trot = RECORD
     flip : INTEGER;
     Angle : INTEGER;
   END; { trot }
   centhdl = ^centptr;
   centptr = ^cent;
   Cent = PACKED RECORD
            yInt: INTEGER;
            yFrac: INTEGER;
            xInt: INTEGER;
            xFrac: INTEGER;
   END; { Cent }

VAR
 arect    : Rect;
 rotation : rothdl;
 center   : centhdl;
```

```
BEGIN { Test }
 rotation := rothdl(NewHandle(sizeof(trot)));
 rotation^^.flip := 0;                              {no flip}
 rotation^^.angle := 15;       {15 degree rotation}

 center := centhdl(NewHandle(sizeof(cent)));
 center^^.xInt := 50;                               {center at 50,50}
 center^^.yInt := 50;
 center^^.xFrac := 0;                               {no fractional part}
 center^^.yFrac := 0;

 myPic := OpenPicture(theWorld);
   ClipRect(theWorld);
   MoveTo(20,20);
   DrawString('Begin Rotation');

   {set the center of Rotation}
   PicComment(RotateCenter,GetHandleSize(Handle(center)),Handle(center));
   {Begin Rotation}
  PicComment(RotateBegin,GetHandleSize(Handle(rotation)),Handle(rotation));
   SetRect(arect,100,100,500,500);
   FrameRect(aRect);
   MoveTo(500,500);
   Lineto(100,100);
   PicComment(RotateEnd,0,nil);              {RotateEnd}
 ClosePicture;
 DisposHandle(handle(rotation));            {Clean up}
 DisposHandle(handle(center));
 PrintThePicture;                            {print it please}
 KillPicture(MyPic);
END; { Test }
```

Forms

The two form printing comments allow you to prepare a template to use for printing. When the `FormsBegin` comment is used, the LaserWriter's buffer is not cleared after `PrClosePage`. This allows you to download a form then change it for each subsequent page, inserting the information you want. `FormsEnd` allows the buffer to be cleared at the next `PrClosePage`.