DPS Features and Enhancements 2

This chapter provides information on the Display PostScript (DPS) extension to the Solaris X server. The following topics are briefly discussed:

- Overview information on the DPS system
- Solaris font enhancements to DPS
- DPS security issues
- DPS compositing operators

About DPS

The Display PostScript system displays graphical information on the computer screen with the same PostScript language imaging model that is a standard for printers and typesetters.[1] The PostScript language makes it possible for an X application to draw lines and curves with perfect precision, rotate and scale images, and manipulate type as a graphic object. In addition, X applications that use the Display PostScript system have access to the entire Adobe Type Library.

1. This section is based on Chapter 4 of *Programming the Display PostScript System with X* by Adobe Systems Incorporated (Addison-Wesley Publishing Company, Inc., 1993) and is used with the permission of the copyright holder.

2

Device and resolution independence are important benefits of PostScript printers and typesetters. The Display PostScript system extends these benefits to interactive displays. An application that takes advantage of the DPS system will work and appear the same on any display without modification to the application program.

How DPS Works

The DPS system has several components, including the PostScript interpreter, the Client Library, and the `pswrap` translator. The Client Library is the link between an application and the PostScript interpreter.

Each application that uses the DPS extension creates a *context*. A context can be thought of as a virtual PostScript printer that sends its output to a window or an offscreen pixmap. It has its own set of stacks, input/output facilities, and memory space. Separate contexts enable multiple applications to share the PostScript interpreter, which runs a single process in the server.

Although the DPS system supports multiple contexts for a single application, one context is usually sufficient for all drawing within an application. A single context can handle many drawing areas. There are exceptions, however, when it is preferable to use more than one context in a client. For example, a separate context might be used when importing Encapsulated PostScript (EPS) files. This simplifies error recovery if an included EPS file contains PostScript errors.

An application draws on the screen by making calls to Client Library procedures. These procedures generate PostScript language code that is sent to the PostScript interpreter for execution. In addition to the Client Library, the DPS system provides the `pswrap` translator. It takes PostScript language operators and produces a C-language procedure–called a wrap–that can then be called from an application program.

The PostScript interpreter handles the scheduling associated with executing contexts in time slices. The interpreter switches among contexts, giving multiple applications access to the interpreter. Each context has access to a private portion of PostScript virtual memory space (VM). An additional portion of VM, called *shared VM*, is shared among all contexts and holds system fonts and other shared resources. *Private VM* can hold fonts private to the context.

Figure 2-1 shows the components of DPS and their relationship to X.

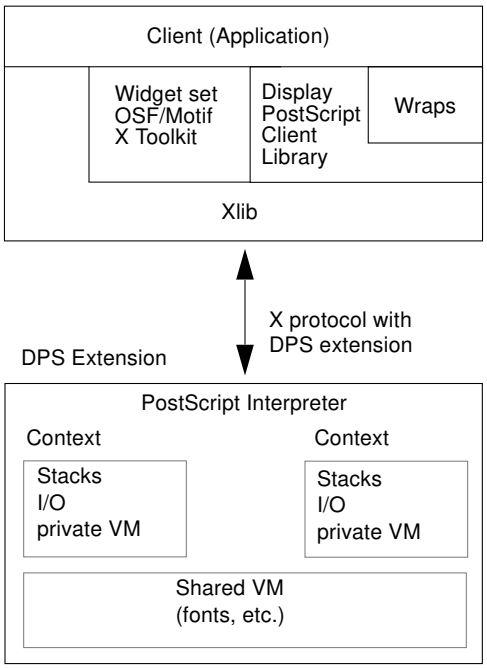


Figure 2-1 DPS Extension to X

An application interacts with the DPS system in the following manner:

1. The application creates a PostScript execution context and establishes a communication channel to the server.
2. The application sends Client Library procedures and wraps to the context and receives responses from it.
3. When the application exits, it destroys the context and closes the communications channel, freeing resources used during the session.

The structure of a context is the same across all DPS platforms. Creating and managing a context, however, can differ from one platform to another. The *Client Library Reference Manual* and *Client Library Supplement for X* contain information on contexts and the routines that manipulate them, and *Display PostScript Toolkit for X* contains utilities for Display PostScript developers.

2

DPS Font Enhancements in the Solaris Server

The Solaris X server includes the following font enhancements to the DPS system:

- Support for F3 Latin and Asian fonts
- Support for obtaining prescaled bitmap font formats from X11 font code
- Type 1 fonts (`.pfa` and `.pfb`)

See Chapter 4, "Font Support," for more information.

DPS Libraries

Table 2-1 lists the DPS libraries. The `.so` and `.a` files that comprise these libraries are located in the `/usr/openwin/lib` and `/usr/openwin/lib/libp` directories. For information on these libraries, see *Programming the Display PostScript System with X* and *PostScript Language Reference Manual*.

Table 2-1 DPS Libraries

Library	Description
`libdps`	DPS Client library
`libdpstk`	DPS Toolkit library
`libpsres`	PostScript Language Resource Location library
`libdpstkXm`	DPS Motif Toolkit library

Adobe NX Agent Support

The context creation routines (`XDPSCreateSimpleContext` and `XDPSCreateContext`) in `libdps` attempt to contact the DPS NX agent if they are unable to connect to the DPS/X extension. The NX client must be started manually, usually during the boot or X startup process.

The Adobe DPS NX agent, which is available from Adobe Systems Inc., is a separate process from the X server and the DPS/X client. When connected to the DPS NX agent, the client's DPS calls are intercepted and converted into standard X Protocol requests. Thus, a DPS client can run on an X server that does not natively support the DPS extension.

2

DPS Security Issues

The Solaris environment provides, and in some cases exceeds, the X Consortium's X11R5 sample server security levels. In particular, DPS programmers should be aware of two DPS-specific security features: PostScript file operators' inability to access system files, and secure context creation. These features are described below.

System File Access

The PostScript language provides file operations that allow users to access system devices such as disk files. This presents a serious security problem. In the Solaris environment, you cannot—by default—use PostScript file operators to open or otherwise access a system file.

For applications, the client rather than the server should perform necessary file operations. Thus, the client does not need all the same access privileges that the server needs. If you want PostScript file operators to access system files, start the server with the `-dpsfileops` option (see the `Xsun(1)` man page). If you attempt to access system files without specifying `-dpsfileops`, you will get a PostScript `undefinedfilename` error. This issue is particularly important in the CDE or `xdm` environment, as the server process is owned by a super-user.

Secure Context Creation

DPS contexts normally have access to global data. This allows a context to look into the activities of another context. For example, one context could intercept a document that another context is imaging. This section describes how to create secure contexts in the Solaris environment.

Section 7.1.1, "Creating Contexts," in the *PostScript Language Reference Manual, Second Edition* describes three ways that contexts can share VM:

1. "Local and global VM are completely private to the context." This capability is new with Level 2, and a context created this way is called a *secure context*.
2. "Local VM is private to the context, but global VM is shared with some other context." This is the normal situation for contexts created with `XDPSCreateContext` and `XDPSCreateSimpleContext`.

3. "Local and global VM are shared with some other context." This is the situation for contexts created with `XDPSCreateContext` and `XDPSCreateSimpleContext` when the `space` parameter is not `NULL`.

To create a secure context, use `XDPSCreateSecureContext` as shown below:

```
XDPSCreateSecureContext DPSContext XDPSCreateSecureContext(dpy,
        drawable, gc, x, y, eventmask, grayramp, ccube, actual,
        textProc, errorProc, space)
    Display *dpy;
    Drawable drawable;
    GC gc;
    int x;
    int y;
    unsigned int eventmask;
    XStandardColormap *grayramp;
    XStandardColormap *ccube;
    int actual;
    DPSTextProc textProc;
    DPSErrorProc errorProc;
    DPSSpace space;
```

All parameters have the identical meaning to those in `XDPSCreateContext`, but the context being created has its own private global VM. If the `space` parameter is not `NULL`, it must identify a space created with a secure context. A space created with a secure context cannot be used for the creation of a nonsecure context. Specifying a nonsecure space with a secure context or a secure space with a nonsecure context generates an access error.

When DPS Encounters Internal Errors

DPS conducts consistency checks during execution. In the rare event that it encounters internal errors, DPS applications will not be able to connect to the server. If this happens, you must restart the Solaris environment. If a client tries to connect to a server with the DPS extension in this state, the following error message sometimes appears:

```
XError: 130
Request Major code 129 (Adobe-DPS_Extension)
```

How To Access Information From Adobe

The following information is readily available from Adobe's public access file server: source code examples, Adobe Metric Font (AMF) files, documentation, PostScript printer description (PPP) files, and press releases. You can obtain this information if you have access to the Internet or UUCP electronic mail.

If you have access to the Internet, use the file transfer protocol (`ftp`) program to download files from the `ftp.mv.us.adobe.com` machine. Read the `README.first` file for information on the archived files. For details on obtaining information from Adobe by electronic mail, see the "Public Access File Server" section in the preface of *Programming the Display PostScript System with X*.

2

DPS Compositing Operators

Warning – The operators defined in this section are extensions to the Display PostScript language. They are not part of the standard DPS and thus are not available in all DPS implementations. An application that depends on these operators is not portable and cannot display on servers that do not support these operators.

Compositing is an OpenStep™ extension to the Display PostScript system. Compositing enables separately rendered images to be combined into a final image. It encompasses a wide range of imaging capabilities:

- It provides a means for simply copying an image as is from one place to another with PostScript.
- It allows two images to be added together so that both appear in the composite superimposed on each other.
- It defines a number of operations that take advantage of transparency in one or both images that are combined. When the images are composited, the transparency of one image can let parts of the other image show through.

Compositing can be used for copying within the same window, as during scrolling, or for taking an image rendered in one drawable and transferring it to another. In OpenStep applications, images are often stored in pixmaps and composited into windows as they are needed.

When images are partially transparent, they can be composited so that the transparent sections of one image determine what the viewer sees of the other. Each compositing operation uses transparency in a different way. In a typical operation, one image provides a background or foreground for the other. When parts of an image are transparent, it can be composited over an opaque background, which will show though transparent “holes” in the image on top. In other operations, transparent sections of one image can be used to “erase” matching sections of the images it is composited with. In most operations, the composite is calculated from the transparency of both images.

Compositing with transparency can achieve a variety of interesting visual effects. A partially transparent, uniformly gray area can be used like a pale wash to darken the image it is composited with. Patches of partially

transparent gray can add shadows to another image. Repeated compositing while slowly altering the transparency of two images can dissolve one into another. Or an animated figure can be composited over a fixed background.

Before images can be composited, they must be rendered. To take advantage of transparency when compositing, at least one of the images needs to be rendered with transparent paint.

The following PostScript program fragment shows the use of the compositing operators. The program creates two simple images and composites them. The first image, the destination, is a 0.8 gray triangle on a white backround; the second, the source, is a 0.6 gray triangle on a transparent background.

```
% Create the Destination triangle
    0.8 setgray
    100 100 moveto
    100 0 rlineto
    0 -100 rlineto
    fill

% Make the background of the source transparent
    0 setalpha
    0 0 100 100 rectfill

% Draw the Source triangle
    1 setalpha
    0.6 setgray
    0 0 moveto
    0 100 rlineto
    100 0 rlineto
    fill

% Compute the result
    0 0 100 100 null 100 0 Sover composite
```

The eighth operand to the composite operator, `Sover`, defines how the source and destination pixels are combined. In the example, the opaque parts of the source image are placed over the destination image. The resulting image looks like Figure 2-2 on page 26.

Figure 2-2 Compositing Operator Example Program

Operator Descriptions

This section describes the new DPS operators. The information is provided in the format used in the PostScript manuals *PostScript Language Reference Manual* and *Programming the Display PostScript System with X*.

setalpha *coverage* **setalpha**

Sets the *coverage* parameter in the current graphics state to *coverage*. *coverage* should be a number between 0 and 1, with 0 corresponding to transparent, 1 corresponding to opaque, and intermediate values corresponding to partial coverage. The default value is 1. This establishes how much background shows through for purposes of compositing. If the coverage value is less than 0, the coverage parameter is set to 0. If the value is greater than 1, the coverage parameter is set to 1.

The coverage value affects the color painted by PostScript marking operations. The current color is pre-multiplied by the alpha value before rendering. This multiplication occurs after the current color has been transformed to RGB space.

Errors **stackunderflow**, **typecheck**

See also **composite**, **currentalpha**

currentalpha - **currentalpha** *coverage*

Returns the coverage parameter of the current graphics state.

Errors **None**

See also **composite, setalpha**

composite *srcx srcy width height srcgstate destx desty op* **composite**

Performs the compositing operation specified by *op* between pairs of pixels in two images, a source and a destination. The source pixels are in the drawable referred to by the *srcgstate* graphics state, and the destination pixels are in the drawable specified by the current graphics state. If *srcgstate* is `NULL`, the current graphics state is assumed.

The rectangle specified by *srcx*, *srcy*, *width*, and *height* defines the source image. The outline of the rectangle may cross pixel boundaries due to fractional coordinates, scaling, or rotated axes. The pixels included in the source are all those that the outline of the rectangle encloses or enters.

The destination image has the same size, shape, and orientation as the source; *destx* and *desty* give destination's location image compared to the source. Even if the two graphic states have different orientations, the images will not; composite will not rotate images.

Both images are clipped to the frame rectangles of the respective drawables. The destination image is further clipped to the clipping path of the current graphics state. The result of a composite operation replaces the destination image.

op specifies the compositing operation. The color of each destination image pixel (alpha value) after the operation, dst' (dstA'), is given by:

```
dst' = src * Fs(srcA, dstA, op) + dst * Fd(srcA, dstA, op)
dstA' = srcA * Fs(srcA, dstA, op) + dstA * Fs(srcA, dstA, op)
```

where `src` and `srcA` are the source color and alpha values, `dst` and `dstA` are the destination color and alpha values, and `Fs` and `Fd` are the functions given in Table 2-2 on page 28.

The choices for the composite *op* are given in Table 2-2. See Figure 2-3 on page 29 for the result of each operation.

Errors	**rangecheck, stackunderflow, typecheck**
See also	**compositerect, setalpha, setgray, sethsbcolor, setrgbcolor**

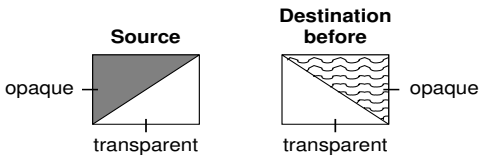
Table 2-2 Factors of the Compositing Equation

Op	Fs	Fd
Clear	0	0
Copy	1	0
Sover	1	1 - srcA
Sin	dstA	0
Sout	1 - dstA	0
Satop	dstA	1 - srcA
Dover	1 - dstA	1
Din	0	srcA
Dout	0	1 - srcA
Datop	1 - dstA	srcA
Xor	1 - dstA	1 - srcA
PlusD[1]	N/A	N/A
PlusL[2]	1	1

1. PlusD does not follow the general equation. The equation is dst'=(1-dst)+(1-src). If the result is less than 0 (black), then the result is 0.

2. For PlusL, the addition asturates. That is, if (src+dst) > white), the result is white.

Figure 2-3 on page 29 shows the result of the compositing operations.



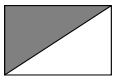
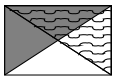
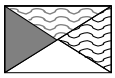
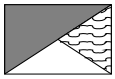
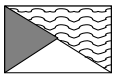
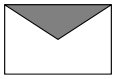
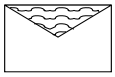
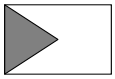
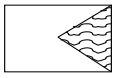
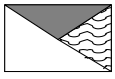
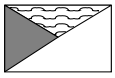
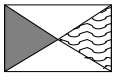
Operation	Destination after	
Copy		Source image.
Clear		Transparent.
PlusD		Sum of source and destination images, with color values approaching 0 as a limit.
PlusL		Sum of source and destination images, with color values approaching 1 as a limit.
Sover		Source image wherever source image is opaque, and destination image elsewhere.
Dover		Destination image wherever destination image is opaque, and source image elsewhere.
Sin		Source image wherever both images are opaque, and transparent elsewhere.
Din		Destination image wherever both images are opaque, and transparent elsewhere.
Sout		Source image wherever source image is opaque but destination image is transparent, and transparent elsewhere.
Dout		Destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.
Satop		Source image wherever both images are opaque, destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.
Datop		Destination image wherever both images are opaque, source image wherever source image is opaque but destination image is transparent, and transparent elsewhere.
Xor		Source image wherever source image is opaque but destination image is transparent, destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.

Figure 2-3 Results of Compositing Operations

compositerect *destx desty width height op* **compositerect -**

In general, this operator is the same as the composite operator except that there is no real source image. The destination is in the current graphics state; *destx*, *desty*, *width*, and *height* describe the destination image in that graphics state's current coordinate system. The effect on the destination is as if there were a source image filled with the color and coverage specified by the graphics state's current color and coverage parameters. *op* has the same meaning as the *op* operand of the composite operator; however, one additional operation, **Highlight**, is allowed.

Highlight turns every white pixel in the destination rectangle to light gray and every light gray pixel to white, regardless of the pixel's coverage value. Light gray is defined as 2/3. Repeating the same operation reverses the effect. (On monochrome displays, Highlight inverts each pixel so that white becomes black, black becomes white.)

Note – The **Highlight** operation doesn't change the value of a pixel's coverage component. To ensure that the pixel's color and coverage combination remains valid, **Highlight** operations should be temporary and should be reversed before any further compositing.

For **compositerect**, the pixels included in the destination are those that the outline of the specified rectangle encloses or enters. The destination image is clipped to the frame rectangle and clipping path of the window in the current graphics state.

Errors **rangecheck, stackunderflow, typecheck**

See also **composite, setalpha, setgray, sethsbcolor, setrbgcolor**

dissolve *srcx srcy width height srcgstate destx desty delta* **dissolve -**

The effect of this operation is a blending of a source and a destination image. The first seven arguments choose source and destination pixels as they do for composite. The exact fraction of the blend is specified by *delta*, which is a floating-point number between 0.0 and 1.0. The resulting image is:

```
delta * source + (1-delta) * destination
```

If *srcgstate* is null, the current graphics state is assumed.

Errors **stackunderflow, typecheck**

See also **composite**

The values of the composite *op* are available for applications in the PostScript **systemdict.** The definitions are as follows:

/Clear 0 **def**

/Copy 1 **def**

/Sover 2 **def**

/Sin 3 **def**

/Sout 4 **def**

/Satop 5 **def**

/Dover 6 **def**

/Din 7 **def**

/Dout 8 **def**

/Datop 9 **def**

/Xor 10 **def**

/PlusD 11 **def**

/Highlight 12 **def**

/PlusL 13 **def**

Implementation Notes and Limitations

Partially Transparent Alpha

Alpha values that are not completely opaque (1) or completely transparent (0) should be used with caution. Compositing operations with partial transparency yield the highest image quality only when a large number of colors are available in the DPS color cube and gray ramp. That is, image

quality is best with a 24-bit TrueColor or 8-bit StaticGray visual, and image quality will be poor with an 8-bit PseudoColor visual. In addition, the performance of compositing operations is greatly reduced for partially transparent pixels due to the extra computation required in these cases.

Indexed Color Visuals

For best results with the **Highlight** *op*, the number of colors in the DPS context's gray ramp should be such that

```
fract(((float) numgrays - 1)* 2. / 3.) == 0
```

In other words, (numgrays = 4, 7, 6, 8, 16, ...). This ensures that the color 2/3 gray is not halftoned.

Given the limited number of colors usually available in the DPS color cube and gray ramp, images with alpha values that are not completely opaque (1) or completely transparent (0) should be avoided to obtain best image quality.

Compositing operations are only defined for pixels values that are in the gray ramp or color cube specified by the gstate. Compositing pixels with values outside the color cube and gray ramp may not yield expected results.

Monochrome Displays

The results of compositing operations for 1-bit drawables that have alpha values that are not equal to 0 or 1 is undefined.

The *op* **Highlight** inverts the color of the pixel on a 1-bit drawable.

Interaction with X Drawing Operations

Drawables that have been rendered to with non-opaque alpha have additional pixel storage associated with them, called the alpha channel. X Window system operations do not affect the alpha channel, with the following exceptions:

- When windows with alpha channel are exposed, if the window has an X background defined (background != None), when the background is painted, the alpha component of the exposed pixels is painted with alpha = 1.
- When a window is resized, the alpha channel storage is resized.

Destroying the Alpha Channel

The **erasepage** operator paints the current drawable of the graphics state with opaque white. Thus, the alpha values for all pixels in the drawable are equal to 1, and the alpha channel storage is destroyed.

Drawables with Unequal Depths

Compositing drawables with unequal depths is undefined.

2