
# Display PostScript System

*Adobe Systems Incorporated*

## Client Library Supplement for X

15 April 1993

Adobe Systems Incorporated

Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/

# Contents

# List of Figures

# List of Tables

# List of Examples

# Client Library Supplement for X

## 1 About this Manual

*Client Library Supplement for X* contains information about the Client Library interface to the Display PostScript system implemented as an extension to the X Window System. The Display PostScript extension is the application programmer's means of displaying text and graphics on a screen using the PostScript language.

The system-independent interface for Display PostScript is documented in *Client Library Reference Manual*. Only extensions to the interface are discussed in *Client Library Supplement for X*. The header file <*DPS/dpsXclient.h*> includes both system-independent and X system-specific procedures.



## 1.1 What this Manual Contains

Section 2, "About the Display PostScript Extension to X," briefly introduces the Display PostScript system extension to the X Window System.

Section 3, "Basic Facilities," introduces concepts that will enable you to write a simple application, including connecting to the X server; creating and terminating a context; differences in coordinate systems; issues of rendering in X versus PostScript language; clipping, repainting, and resizing; error codes; user object indices; and status events.

Section 4, "Additional Facilities," describes advanced concepts that not all applications need, including client and server identifiers, encodings, synchronization, shared resources, and multiple servers.

Section 5, "Programming Tips," contains tips for the application programmer on files, fonts, coordinate conversions, and other issues that require special attention.

Section 6, "X-Specific Data and Procedures," describes the X-specific data and procedures found in the <*DPS/dpsXclient.h*> header file.

Section 7, "X-Specific Custom PostScript Operators," describes the X-specific PostScript operators provided for the Display PostScript extension to X.

# 2 About the Display PostScript Extension to X

In order to understand the relationship of the Display PostScript system to the development of X applications, you should be familiar with the following concepts:

- *The PostScript imaging model*, which allows the application developer to express graphical displays at a higher level of abstraction than is possible with Xlib. This improves device independence and portability. The integration of the imaging model with X requires consideration of several issues, including coordinate system conversions (see “Coordinate Systems” in 3.3, “Execution”), event handling (see 3.4, “Status Events”), and resource management (see 4.7, “Sharing Resources”).

- *The PostScript interpreter*, which allows an application to execute PostScript language code.

- *Wrapped procedures*, which allow PostScript language programs to be embedded in an application as C-callable procedures.

# 3 Basic Facilities

*Client Library Reference Manual* introduces the facilities needed to write a simple application program for the Display PostScript system. This manual discusses Display PostScript system issues of particular concern in the X Window System environment, in the following categories:

- Initialization
- Creating a context
- Execution of PostScript language code
- Termination

## 3.1 Initialization

Before performing any Display PostScript operations, the application must establish a connection to the X server. You can connect to the server by using Xlib's **XOpenDisplay** routine or a standard toolkit's initialization process. Regardless of how the connection is established, an X *Display* record will be defined for the connection. Subsequent Display PostScript system operations will use this *Display* record to identify the server. Once the *Display* record is obtained, the application must create a *drawable* (window or pixmap) for Display PostScript imaging operations, and an X *GC* out of which certain fields are used by Display PostScript. There are a number of facilities in Xlib for creating new windows and *GC*s, such as **XCreateSimpleWindow** and **XCreateGC**.

## 3.2 Creating a Context

In Display PostScript, a context (as described in *Client Library Reference Manual*) is a resource in the server that represents all of the execution state needed by the PostScript interpreter to run PostScript language programs.

*DPSContextRec* is a data structure on the client side that represents all of the state needed by the Client Library to communicate with a context. A pointer of type *DPSContext* is a handle to this data structure. When the application creates a context in the interpreter, a *DPSContextRec* structure is automatically created for use by the client (except for forked contexts; see 4.5, "Forked Contexts"). The *DPSContextRec* contains pointers to procedures that implement all of the basic operations that a context can perform.

There are two procedures that create both a context in the server and a *DPSContextRec* for the client. The first, **XDPSCreateSimpleContext**, uses the default colormap, and is adequate for most applications. The second, **XDPSCreateContext**, is a more general function that allows you to specify

colormap information. Other procedures for creating just the *DPSContextRec*—for contexts that already exist in the server—are covered in 4, “Additional Facilities.”

### 3.2.1 Using XDPSCreateSimpleContext

To create a context using the default colormap, call **XDPSCreateSimpleContext**:

**XDPSCreateSimpleContext**

```
DPSContext XDPSCreateSimpleContext(dpy, drawable, gc, x, y,
    textProc, errorProc, space)

  Display *dpy;
  Drawable drawable;
  GC gc;
  int x;
  int y;
  DPSTextProc textProc;
  DPSErrorProc errorProc;
  DPSSpace space;
```

*Client Library Reference Manual* contains a general discussion of **XDPSCreateSimpleContext**, but does not discuss the details that are relevant to X. These details are covered here.

A context is created on the specified *Display* and is associated with a *Drawable* and *GC* on that *Display*. The context uses the following fields in the *GC* to render text and graphics on the *Drawable*:

- plane_mask
- subwindow_mode
- clip_x_origin
- clip_y_origin
- clip_mask

If the *Drawable* or *GC* is not specified (that is, passed as *None*), the context will execute programs correctly but will not render any text or graphics (it renders to the null device). A valid *Drawable* and *GC* may be associated with such a context at a later time using the **setXgcdrawable** operator, documented in 7, “X-Specific Custom PostScript Operators.”

The arguments *x* and *y* are offsets that specify where the device space origin is relative to the window origin. To place the device space origin (and thus the user space origin) in the standard lower-left corner, pass zero for *x* and the height of the window in pixels for *y*. See the discussion of coordinate systems in 3.3, “Execution.”

The other arguments to **XDPSCreateSimpleContext** are described fully in *Client Library Reference Manual*. To summarize: *textProc* is a callback procedure that handles text output from the context, *errorProc* is a callback procedure that handles errors reported by the context, and *space* is the private VM that the context uses for storage. If the space is passed as *NULL*, a new space is created.

If all of the arguments are valid and the context is successfully created in the server, a *DPSContext* handle is returned. Otherwise, *NULL* is returned.

**XDPSCreateSimpleContext** uses the default colormap. A device-specific number of grays is reserved in the default colormap, which represents a gray ramp. If the device supports color, an RGB color cube is also reserved. If a requested RGB color is found in the color cube or gray ramp, the associated pixel value is used. Otherwise, the color is approximated by dithering pixel values from the colormap to give the best possible rendering of the color.

**XDPSCreateSimpleContext** may allocate a substantial number of cells in the default colormap. For example, a typical allocation for an 8-plane PseudoColor device is 64 cells for the color cube, representing a 4x4x4 RGB cube. The gray ramp typically uses nine cells. **XDPSCreateSimpleContext** checks the root window for the *RGB_DEFAULT_MAP* and *RGB_GRAY_RAMP* properties. If the properties exists, the color cells they specify are used for the context’s color cube and gray ramp. If the properties do not exist, color cells are allocated and the properties are defined. The allocated cells are typically treated as “read-only retained” so that other Display PostScript clients may share the allocated colors.

The Display PostScript system uses entries from the default X colormap to display colors and grey values. You can configure this usage. Giving the Display PostScript system more colormap entries improves the quality of its rendering, but leaves fewer entries available to other applications since the default colormap is shared.

Resources in your *.Xdefaults* file control the colormap usage. Each resource entry should be of the form

DPSColorCube.*visualType.depth.color: size*

where

*visualType* is one of GrayScale, PseudoColor, or DirectColor.

*depth* is 1, 2, 4, 8, 12, or 24 and should be the largest depth equal to or less than the default depth.

*color* is one of the strings “reds”, “greens”, “blues”, or “grays”.

*size* is the number of values of that color to allocate .

These resources are not used for the static visual types *StaticGray*, *StaticColor*, or *TrueColor*.

Specifying 0 for reds directs the Client Library to use only a gray ramp. This specification is particularly useful for gray-scale systems that incorrectly use *PseudoColor* as the default visual.

For example, to configure a 5x5x4 color cube and a 17-element gray ramp for an 8-bit *PseudoColor* screen, specify these resources:

```
DPSColorCube.PseudoColor.8.reds: 5
DPSColorCube.PseudoColor.8.greens: 5
DPSColorCube.PseudoColor.8.blues: 4
DPSColorCube.PseudoColor.8.grays: 17
```

These resources use 117 colormap entries, 100 for the color cube and 17 for the gray ramp. For the best rendering results, specify an odd number for the gray ramp.

Resources that are not specified take these default values:

```
DPSColorCube.GrayScale.4.grays: 9
DPSColorCube.GrayScale.8.grays: 17

DPSColorCube.PseudoColor.4.reds: 2
DPSColorCube.PseudoColor.4.greens: 2
DPSColorCube.PseudoColor.4.blues: 2
DPSColorCube.PseudoColor.4.grays: 2
DPSColorCube.PseudoColor.8.reds: 4
DPSColorCube.PseudoColor.8.greens: 4
DPSColorCube.PseudoColor.8.blues: 4
DPSColorCube.PseudoColor.8.grays: 9
DPSColorCube.PseudoColor.12.reds: 6
DPSColorCube.PseudoColor.12.greens: 6
DPSColorCube.PseudoColor.12.blues: 5
DPSColorCube.PseudoColor.12.grays: 17

DPSColorCube.DirectColor.12.reds: 6
DPSColorCube.DirectColor.12.greens: 6
DPSColorCube.DirectColor.12.blues: 6
DPSColorCube.DirectColor.12.grays: 6
DPSColorCube.DirectColor.24.reds: 7
```

```
DPSColorCube.DirectColor.24.greens: 7
DPSColorCube.DirectColor.24.blues: 7
DPSColorCube.DirectColor.24.grays: 7
```

If none of the above defaults apply to the display, the Client Library uses no color cube and a 2-element gray ramp; that is, black and white.

The advantage of using the color allocation facilities provided by **XDPSCreateSimpleContext** is that the application has available a wide range of colors (many more than the number of cells), each with a reasonable rendering, without having to provide for the possibility that colormap allocations may fail. The disadvantage is that a large number of color cells may be allocated from the default colormap.

### 3.2.2 Using XDPSCreateContext

To create a context with specific color information, call **XDPSCreateContext**:

**XDPSCreateContext**

```
DPSContext XDPSCreateContext(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;
```

The *dpy*, *drawable*, *gc*, *x*, *y*, *textProc*, *errorProc*, and *space* arguments for **XDPSCreateContext** are the same as for **XDPSCreateSimpleContext**. The *eventmask* is currently not implemented and should be passed as zero.

The *grayramp* and *ccube* arguments are pointers to *XStandardColormap* data structures (defined in the *<X11/Xutil.h>* header file). An *XStandardColormap* specifies a colormap, a base pixel value, and multipliers and limits for red (or gray), green, and blue ramps. A valid gray ramp is required; *ccube* is optional (may be passed as *NULL*). If a color cube is present and is specified by *ccube*, *grayramp* may use pixel values in the color cube in order to conserve colormap entries. The X colormap resource specified in the *ccube* and *grayramp* arguments must be identical. The application must ensure that the specified colormap is installed—for example, by using **XSetWindowColormap** to set the colormap as an attribute of the window.

The application provides a colormap with a uniform distribution of colors. The colormap must provide a uniform distribution of grays (colors where red, green, and blue are equal in intensity), which is described by *grayramp*. However, the *grayramp* may be as simple as two levels: black and white. The colormap may also contain a uniform distribution of RGB colors arranged as a color cube, which is described by *ccube*. See X Window System reference documents for details about the *XStandardColormap* data structure.

The argument *actual* can be used to conserve colormap entries as well as to display pure (nondithered) colors. If the application has been informed which colors it will use, or if the number of colors to be used is relatively few (fewer than the default allocation that **XDPSCreateSimpleContext** would use for the device), the *actual* argument can be used. *actual* is a hint about the number of colors the context is going to request. It is considered a hint because the server cannot guarantee that the specified number of colors will be available. The server will reserve the number of cells specified by *actual* or the number of cells available in the specified colormap, whichever is smaller. As the context makes color requests, colormap entries are defined on a "first come, first served" basis. For example, suppose *actual* is given the value 3 and there are at least three cells available. The first time the context executes the **setrgbcolor** operator, the requested color will be stored in the colormap, leaving two more cells reserved by *actual*. When the context executes **setrgbcolor** for a different color, the second cell reserved by *actual* is used, and so on. The colors requested by the PostScript language program executed by the context will be rendered without dithering.

*Note: Supporting actual is an optional part of a Display PostScript system. Some implementations ignore actual, so portable applications should not count on its effects.*

Consider the characteristics of your application when deciding whether to use **XDPSCreateSimpleContext**, with its default allocation of colors, or **XDPSCreateContext**, with *actual*. An application may allow the end user to define a variety of colors. Such an application—a graphics editor, for example—could use **XDPSCreateSimpleContext**.

On the other hand, an application that uses only a few colors—the foreground and background colors of a performance meter, for example—could use **XDPSCreateContext**, specify a small color cube, and set *actual* to the number of colors used. Since *actual* is just a hint, the small cube is necessary as a fallback strategy; it ensures that the application will display correctly regardless of the environment.

If all the arguments are valid and the context is successfully created in the server, a *DPSContext* handle is returned. Otherwise, *NULL* is returned.

## 3.3 Execution

This section discusses the following Display PostScript issues: coordinate systems, rendering, clipping, repainting, resizing a window, user object indices, and errors.

### 3.3.1 Coordinate Systems

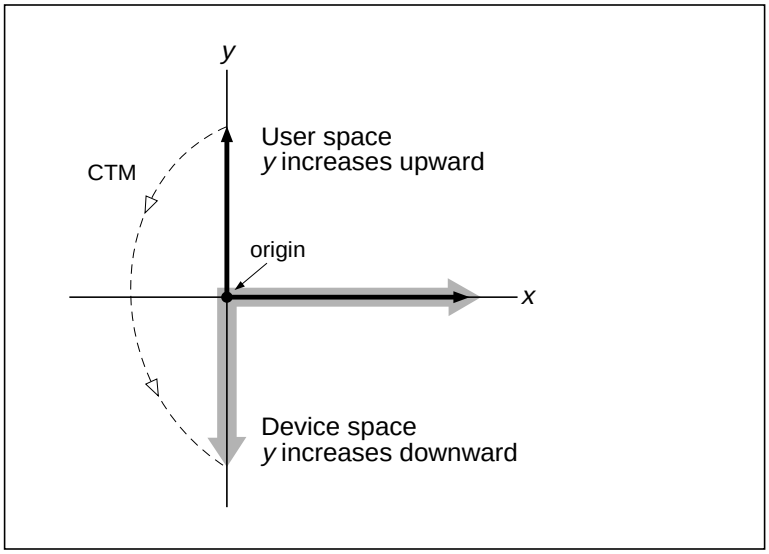
The application must specify user space coordinates when communicating with the PostScript interpreter and X coordinates when communicating with other parts of the X Window System. Therefore coordinate conversions may be necessary. This section explains:

- How to specify the device space origin for the window at context creation time
- How to convert user space coordinates to X coordinates
- How to convert X coordinates to user space coordinates

*PostScript Language Reference Manual, Second Edition*, describes the coordinate system used by the PostScript imaging model. To summarize: coordinates are specified in a user-defined space and are automatically converted to the output device space. The default user space unit is 1/72 of an inch. The default origin is in the lower left corner of the page, with *x* increasing to the right and *y* increasing to the top (upwards).

Figure 1 shows a linear transformation from user space to device space by means of the current transformation matrix (CTM). Note that this transformation is one-way only.

**Figure 1** *User space and device space*



In PostScript language terminology, the window is the output device. In Display PostScript, the window is treated as a page, with the conventional location of the origin in the lower left corner. The device space is equivalent to the X coordinate system for the window, except for the following:

- The device space origin is offset from the window origin.
- Device space is a real-number space, whereas the X coordinate system is an integer space.

As described in *PostScript Language Reference Manual, Second Edition*, pixel boundaries fall on integer coordinates in device space. A pixel is a half-open region, meaning that it includes half its boundary points. For any point (*x, y*) in device space, let *i* = floor(*x*) and *j* = floor(*y*), where *x* and *y* are real numbers and *i* and *j* are integers. The pixel that contains this point is the one identified as (*i, j*), which is equivalent to the X coordinate for that pixel.

To convert user space coordinates to X coordinates:

1. Convert the user space coordinates to device space coordinates by computing a linear transformation using the current transformation matrix (CTM).
2. Compute the X coordinates by applying an additional translation to the device space coordinates derived in Step 1 to account for the offset of the device space origin from the window origin.

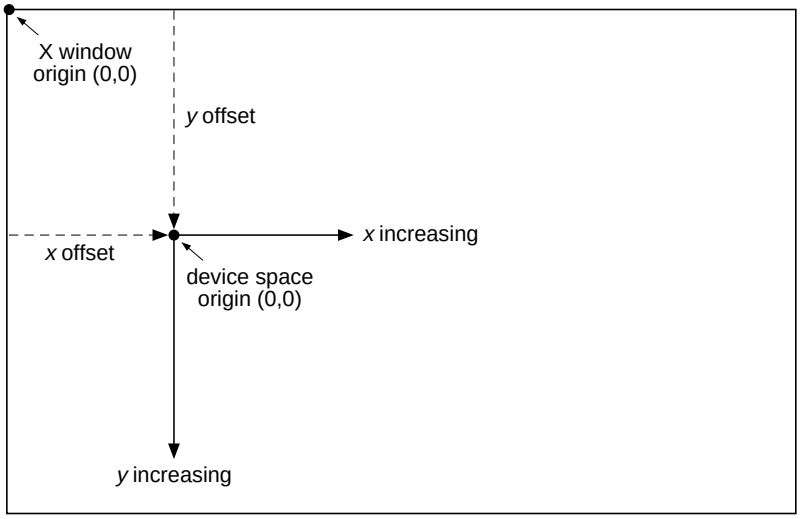
Similarly, to convert X coordinates to user space coordinates:

1. Translate the X coordinates to device space coordinates by applying the offset of the device space origin to the X coordinates.
2. Convert the device space coordinates to user space coordinates by using the inverse of the current transformation matrix.

See 5.5, “Coordinate Conversions,” for examples of coordinate conversions.

Figure 2 illustrates how the device space origin is located in the window as an offset from the window origin. The *x* and *y* offset values are established at context creation time (see 3.2, “Creating a Context”); they can be changed by X-specific PostScript operators such as **setXoffset**.

**Figure 2** *Window origin and device space origin*




The device origin is offset in order to support the method of scrolling that involves copying areas of the window (as opposed to shifting a child window under an ancestor). You can put the device space origin anywhere in the window. Then, as you scroll the contents of the window, you can offset the origin from its original position to make coordinate conversions easier. The default location for the device space origin is in the lower left corner of the window.

Coordinate conversions are required under the following conditions:

- If you use the PostScript imaging model to render graphics using coordinates received from X events, the X coordinates must first be converted into user space coordinates. For instance, if you allow the user to select a line of text in a text editor, coordinate conversions are required.

- If X rendering is to be done in the same window as PostScript language rendering, it may be necessary to convert user space coordinates to X coordinates—for example, call **XCopyArea** to move a graphical object that was rendered by the PostScript interpreter.

Coordinate conversions are not required under the following conditions:

- If you use the PostScript imaging model for output only (rendering text and graphics without user interaction in the display area), no coordinate conversions are required. Simply express coordinates in user space.

  For example, assuming the default user space, the letter *A* shown at coordinate *(x*=72, *y*=72) will appear upright 1 inch to the right and 1 inch above the bottom left corner of the window.

- If the only rendering you do in response to X events is with X primitives, you don't have to perform coordinate conversions unless you are altering pixels that were rendered by the PostScript interpreter.

Resizing the window may have an effect on the device space origin, and thus the offsets to that origin, depending upon the bit gravity of the window. See the section titled "Resizing the Window" on page PG-96.

### 3.3.2 Mixing Display PostScript and X Rendering

X drawing requests and PostScript language code can be sent to the same drawable. For example, X primitives such as **XCopyArea** can be used to move, copy, and change pixels that have been painted with PostScript language programs.

Interactive feedback, such as selection highlighting and control points, can be accomplished with X drawing requests. For example, control points on a graphics object in a graphics editor application can be displayed with X primitives as follows:

- Copy the pixels that were painted by a PostScript language program to a pixmap with several **XCopyArea** calls. These pixels will temporarily be obscured by the control points, so they must be preserved.

- Call **XFillRectangle** to paint the control points, which may be grabbed and stretched, rotated, moved, and so on.

Now suppose a control point is moved. Handle a series of subsequent mouse events as follows:

- Copy the pixels underlying the control point back from the pixmap, effectively erasing the control point at the original location.

- Compute the new position of the control point from the mouse event.

- Copy the pixels at the new location to the pixmap. Call **XFillRectangle** to display the control point at the new location.

Here are some considerations to keep in mind when mixing X and Display PostScript system imaging:

- Their coordinate systems are different. See the section titled "Coordinate Systems" on page PG-91 for more information.

- PostScript language programs run asynchronously with respect to other X requests. A PostScript language rendering request is not guaranteed to be complete before a subsequent X request is executed, unless synchronized. See 4.8, "Synchronization," for more information.

- X tends to be pixel and plane oriented; graphic operations that manipulate pixels and planes are necessarily device dependent. The PostScript imaging model deals with abstract graphical representations (paths) and abstract colors. The PostScript interpreter tries to give the best rendering possible for the device. If device independence is important for your application, use X primitives sparingly, preserving device independence as much as possible.

### 3.3.3 Clipping and Repainting

Text and graphics rendered with the PostScript interpreter are subject to all of the X clipping rules as well as the clipping defined by the PostScript imaging model.

The default clipping region is the window. When clipping other than to the default, the following recommendations apply:

- If you're drawing with PostScript language code only, use the clipping mechanism provided by the PostScript imaging model. This is sufficient for nearly all applications.

- If you're also using X primitives and want to clip them as well as draw using PostScript language code, use the clipping specified by the X GC.

Expose events may be handled with a variety of strategies:

- Repainting all graphics for the window

- Repainting all graphics through composite view clip

- Repainting selected graphics through composite view clip

Repainting the entire window is the simplest strategy to implement and is suitable for simple applications. To do so:

- Ignore exposure events with counts greater than zero.

- For exposure events with counts equal to zero, clear the window and then redisplay all of the text and graphics objects by executing the PostScript language programs that describe them.

Though simple to implement, this strategy makes the window flash or flicker every time it is repainted, which can be distracting to the end user.

A somewhat more sophisticated strategy involves making a list of the rectangles specified in a series of exposure events until a zero count is detected, as follows:

- Create a view clip (see *PostScript Language Reference Manual, Second Edition*) by converting the coordinates of the list of exposure rectangles to user space coordinates and executing **rectviewclip** with this list.

- Then redisplay all the text and graphics objects by executing the PostScript language programs that describe them. Only those areas within the view clip will actually be repainted.

This strategy reduces annoying window flicker, but may do more work than is necessary since programs describing graphics objects that are completely clipped are executed anyway.

The most sophisticated technique, perhaps the optimal strategy, is similar to the one just described:

- Use a list of rectangles from the exposure events to create a view clip.

- Then, instead of running all of the PostScript language programs, redraw only those graphics objects whose bounding boxes intersect the view clip.

This strategy requires that the application keep track of the bounding boxes and locations of each graphical object, but this task is usually necessary anyway, particularly for interactive applications that allow selection and manipulation of objects. User paths are handy for this purpose (see *PostScript Language Reference Manual, Second Edition*), since they are compact data structures that contain their own bounding box information. The list of rectangles obtained from the exposure events can be enumerated and intersected with the bounding box of each user path. Bounding box intersection may still result in some code being executed unnecessarily, but it is a good compromise between time spent deciding which graphical objects to redraw and time spent drawing the objects.

### 3.3.4 Resizing the Window

When the window is resized, the X server moves the window bits according to the bit gravity of the window. If the window is being used for imaging with the PostScript language, the origin of the device space is also moved according to the bit gravity of the window; see the section titled “Coordinate Systems” on page PG-91” for a discussion of coordinate systems. The result of this automatic movement is that the *x* and *y* offsets that were specified when the context was created (or that were last changed with the **setXoffset** operator) are changed. The application may need to keep track of these changes.

Table 1 shows the changes to the *x* and *y* offsets for each bit gravity type.

**Table 1** *How bit gravity affects offsets*

| *Symbol* | *Meaning* |
|---|---|
| oldX | Original *x* offset |
| oldY | Original *y* offset |
| wc | Change in window size along the *x* axis (width) |
| hc | Change in window size along the *y* axis (height) |

| *Bit Gravity* | *New x offset* | *New y offset* |
|---|---|---|
| NorthWest | oldX | oldY |
| North | oldX + wc/2 | oldY |
| NorthEast | oldX + wc | oldY |
| West | oldX | oldY + hc/2 |
| Center | oldX + wc/2 | oldY + hc/2 |
| East | oldX + wc | oldY + hc/2 |
| SouthWest | oldX | oldY + hc |
| South | oldX + wc/2 | oldY + hc |
| SouthEast | oldX + wc | oldY + hc |
| ForgetGravity | *No change* | *No change—appears as if NorthWest* |
| Static | oldX + wc | oldY + hc |

To get the current *x* and *y* offsets, use **currentXoffset**.

### 3.3.5 User Object Indices

The Client Library provides a convenient and efficient way to refer to PostScript language objects. This section presents one set of utilities available for working with these objects. An alternate set of utilities is available in the Display PostScript Toolkit and documented in section 4 of the *Display PostScript Toolkit for X*.

Some types of composite or structured objects, such as dictionaries, gstates, and user paths, are not visible as data outside the PostScript interpreter; that is, they cannot be represented directly in any encoding of the language, not even in binary object sequence encoding. Instead, an application must refer to such objects by means of surrogate objects whose values can be encoded and communicated easily.

The surrogate objects provided by the Client Library are called user objects. A user object is simply an identifier (typically an integer of type *long int*) that represents an actual object (of any type) in the interpreter. To define a new user object, the application must first obtain a user object index from the Client Library. The **DPSNewUserObjectIndex** procedure returns a new user object index. The Client Library is the sole allocator of new user object indices in order to guarantee that indices are unique. User object indices are dynamic and should not be used as arithmetic values (for example, don't add 1 to get the next available index). Also, do not store user object indices in a file or other long-term storage.

After obtaining a user object index, the application must associate this index with an actual object. You first execute a PostScript language program to create the object, then execute the **defineuserobject** operator.

Once a user object has been defined, the application may call wrapped procedures to manipulate it. User objects may be passed as input arguments to a wrapped procedure.

User objects are typically employed under the following circumstances:

- *When graphical objects or other application objects are created dynamically*, such as the user path a graphics editor builds as the user draws an illustration.

- *When a user name should not be employed*. A user object is a convenient and efficient substitute for a dynamically defined user name, which must be passed to a wrap as a string.

See *PostScript Language Reference Manual, Second Edition*, and *pswrap Reference Manual* for further discussion of user objects.

Note that it is the responsibility of the application and any runtime facilities or support software (such as toolkits) to keep track of user object definitions. A user object must be defined before it is used. Unlike user name indices (which are defined automatically by the Client Library), user objects must be defined explicitly. To assist in keeping track of user object definitions, the last user object index assigned can be read from *DPSLastUserObjectIndex*, which should be treated as read-only.

In the following code example, a hypothetical toolkit implements a user interface that displays icons for files and programs. The user interface allows the end user to customize the label of the icon by changing the text and to specify the font of the label text. The icon is represented as a PostScript language dictionary.

**Example 1** *Implementing a user interface to display icons*

*Wrap definitions:*

```
defineps New_Icon(long iconIndex; int x,y; long progIndex;
         char *font, *text)
  % Input Arguments:
  % iconIndex
  %  user object index provided by application
  % x,y
  %  coordinates of lower left corner of icon
  % progIndex
  %  user object index which represents a PostScript
  %  language program for drawing the icon
  % font
  %  string to be used as a font name
  % text
  %  label for icon

 5 dict dup% Create the icon dict.
 iconIndex exch defineuserobject
         % Define the user object for the dict.

 begin    % Begin the icon dict.
 /icon_x x def% Assign x coordinate.
 /icon_y y def% Assign y coordinate.
 /icon_prog
  UserObjects progIndex get
         % Get and def icon drawing procedure
  def    % (assumes userdict is on dict stack)
 /icon_font /font def% Assign label font name.
 /icon_label (text) def% Assign label text.
 end          % End icon dictionary.
endps

/* a wrapped procedure to draw an arbitrary icon */

defineps Draw_Icon(userobject icon)
  %Input Arguments:
  %icon
  %  user object representing an icon dictionary.
  %  Note: since we are going to execute the object,
  %  we can declare it asuserobject to pswrap.

 icon begin% Gets and execs the user object
         % which is a dictionary, begins it.
         % Note that there is an implicit
         % execuserobject here since icon
         % was declared 'userobject'.

 gsave
 icon_x icon_y translate% Put origin at specified
              % coordinates.
```

```
  gsave
  icon_prog  % Draw icon.
  grestore

  1 setgray
  icon_font 10 selectfont% Scale and set icon label font.
  0 0 moveto
  icon_label show% Show label.
  grestore
  end

 endps
```

*C language code:*

```
 void MakeNewIcon(x, y, prog, label)
  int x, y;
  long prog;/* user object defined by application code */
  char *label;
 {
  /* get a new user object index */
  long icon = DPSNewUserObjectIndex( );
  char *defaultFontName = GetDefaultFontName( );

  /* Icon is a user object index: define icon user object */
  NewIcon(icon, x, y, prog, defaultFontName, label);
  /* Icon is now a user object: draw it */
  DrawIcon(icon);
  /* The following procedure call is not defined
   * in this example. It saves the user object created for
   * the new icon so that the application can use the user
   * object to refer to the icon. */
  SaveNewIconObject(icon);
 }
```

### 3.3.6 Errors and Error Codes

Two classes of errors can occur while using Display PostScript: protocol errors and context errors.

*Protocol errors* are generated when invalid requests are sent to the server. The result of receiving a protocol error is that lower-level facilities in Xlib handle the error and perhaps print a message, while the higher-level facilities simply return *NULL* or do nothing. The default protocol error handler prints an error message and causes the application to exit. The application can substitute its own error handler for protocol errors, but results are undefined if the handler returns rather than exiting. (Generally, an attempt to continue processing after a protocol error results in incorrect operation of procedures further up in the call stack.)

*Context errors* can arise whenever a *DPSContext* handle is passed to a Display PostScript procedure or wrap. X-specific error codes are discussed in “Extended Error Codes” in 6.1, “Data Structures.” See *Client Library Reference Manual* for a discussion of the standard Display PostScript error codes.

Because of various delays related to buffering and scheduling, a PostScript language error may be reported long after the C procedure responsible for the error has returned. Consider the following example:

```
DPSPrintf(ctxt, "%d %d %s\n", x, y, operatorName);
MyWrap1(ctxt);
MyWrap2(ctxt, &result);
```

Suppose the string pointed to by *operatorName* does not contain a valid operator and therefore generates an **undefined** error. The error may not be received when **DPSPrintf** returns. It may not be received even when **MyWrap1** returns. **MyWrap2** returns a result, thereby forcing synchronization, so any errors caused by the call to **DPSPrintf** or **MyWrap1** will finally be received.

If **MyWrap2** is called several statements after **MyWrap1**, it may be difficult to figure out where the error originated. However, you can determine where errors are likely to collect, such as places where the application and context will be forced into synchronization, and work backward from there. If you make a list of synchronization points in your code, say, A, B, C, D, and so on, an error received at point C must have been generated by code somewhere between B and C. This will help narrow down your debugging search.

A debugging alternative is to have the application check for an error by forcing synchronization. (The synchronization should be removed in the final version of the software because of its negative impact on performance.) For the details of implementing synchronization, see section 6.4 in *Client Library Reference Manual*.

The code in Example 2 has been simplified to make the principle clear; in an actual application, you would probably want to choose a less verbose means of including the debugging procedures. Every procedure call that sends PostScript language code is followed by a call to *DEBUG_SYNC*. If the macro *DEBUGGING* is *true*, *DEBUG_SYNC* will force the context to be synchronized; if there are any errors, they will be reported. If *DEBUGGING* is *false*, *DEBUG_SYNC* will do nothing. Note that although a call to *DEBUG_SYNC* after the call to **MyWrap2** would be harmless, it is not needed because **MyWrap2** returns a value and is therefore automatically synchronized.

**Example 2** *Debugging by forcing synchronization*

*C language code:*

```
#ifdef DEBUGGING
#define DEBUG_SYNC(c) DPSWaitContext((c))
#else
```

```
#define DEBUG_SYNC(c)
#endif
  ...
DPSPrintf(ctxt, "%d %d %s\n", x, y, operatorName);
DEBUG_SYNC(ctxt);
MyWrap1(ctxt);
DEBUG_SYNC(ctxt);
MyWrap2(ctxt, &result);
```

### 3.3.7 Termination

When an application exits normally, all resources allocated on its behalf, including contexts and spaces, are automatically freed. (This actually depends upon the “close-down mode” of the server.) This is the most typical and convenient method of releasing resources. However, any storage allocated in shared VM (such as fonts loaded by the application) remains allocated even after the application exits.

**DPSDestroyContext** and **DPSDestroySpace** are provided to allow an application to release these resources without exiting. This might be needed if, for example, the context and space must be destroyed and recreated from scratch to recover from a PostScript language error. These procedures are described in detail in *Client Library Reference Manual*. To summarize, **DPSDestroyContext** destroys the context resource in the server and the *DPSContextRec* in the client. **DPSDestroySpace** destroys the space resource in the server and the *DPSSpaceRec* in the client as well as all contexts within the space, including their *DPSContextRec* records.

Note that closing the display—with **XCloseDisplay**, for example—destroys all context and space resources associated with that display, but does not destroy the corresponding client data structures (*DPSContextRec* or *DPSSpaceRec*).

## 3.4 Status Events

At any given time, a context has a specific execution status. Status events are provided for low level monitoring of context status. Most simple applications won’t need this facility.

Status events can be used to perform the following tasks:

- Sending code, using flow control, from the application to a context.
- Controlling the suspension and resumption of execution.
- Synchronizing PostScript interpreter execution with X rendering requests.
- Monitoring a context to determine whether it is runaway, “wedged” (stuck), or zombie.

A status event is generated whenever a context changes from one state to another. Status events can be masked in the server so that uninteresting events are not sent to the client (see **XDPSSetStatusMask**). Furthermore, the application will not see any status events unless it registers a status event handler by calling **XDPSRegisterStatusProc**. The default is to have no status events enabled and no status event handler registered.

The procedure **XDPSGetContextStatus** returns the current status of a context (as a synchronous reply to a request, not as an asynchronous event). The status of a context may be one of the following states:

- *PSSTATUSERROR*. The context is in a state that is not described by the other four status values. For example, a context that has been created but has never been scheduled to execute will return *PSSTATUSERROR* to **XDPSGetContextStatus**. No asynchronous status event will have this value.

- *PSRUNNING*. The context has been running, has code to execute, or is capable of being run. Fine point: No context is running while the server processes requests or generates events, so this value really means that the context is runnable.

- *PSNEEDSINPUT*. The context is waiting for code to execute, a condition commonly known as being “blocked on input.”

- *PSFROZEN*. The execution of the context has been suspended by the **clientsync** operator. A frozen context may be killed with **DPSDestroyContext**, interrupted with **DPSInterruptContext**, or reactivated with **XDPSUnfreezeContext**.

- *PSZOMBIE*. The context is dead. The resource data allocated for the context still exists in the server, but the PostScript interpreter no longer recognizes the context.

Except for *PSSTATUSERROR*, these status events can be disabled (see below).

If an application requires information about one or more types of status events, a handler of type *XDPSStatusProc* must be defined. Two arguments will be passed to the callback procedure: the *DPSContext* handle for the context that generated the status event and a code specifying the status event type. The **XDPSRegisterStatusProc** procedure associates a status event handler with a particular *DPSContext*. Each context may have a different handler.

Once a status event handler is established for the context, the application should set the status event masks for the context by calling **XDPSSetStatusMask**. The symbols for the mask values are

- PSRUNNINGMASK

- PSNEEDSINPUTMASK

- PSZOMBIEMASK
- PSFROZENMASK

A mask is constructed by applying a logical OR of the mask values to the appropriate mask; for example,

```
enableMask = PSRUNNINGMASK | PSNEEDSINPUTMASK;
```

sets the bits that indicate interest in the *PSRUNNING* and *PSNEEDSINPUT* status event types. A 1-bit means interest in that type; a 0-bit means "no change" or "don't care."

The context can handle a given status event type in one of three ways:

- If the application wants to be notified of the event every time it occurs, the event should be enabled.
- If the application wants never to be notified of the event, the event should be disabled.
- If the application wants to be notified of only the next occurrence of the event, the event should be set to *next*.

*Caution:* *Because the Display PostScript extension executes asynchronously from the application, careful synchronization must take place when requesting the next occurrence of an event or future occurrences of the event. Without this synchronization, the event that the application is looking for may have already occurred and been discarded.*

The application defines the method of handling each status event type by setting bits in three masks: *enableMask*, *disableMask*, and *nextMask*.

Call **XDPSSetStatusMask** to set the masks. Note that a particular bit may be set in only one mask. Bits set in the *nextMask* enable the events of that type. When the context changes state, an event is generated. If its type is specified in the *nextMask*, the application is notified of the event and all subsequent events of that type are automatically disabled.

In Example 3, an application currently has *PSNEEDSINPUT* and *PSRUNNING* enabled and all other types disabled. It now asks to be notified of every transition to *PSFROZEN* and *PSZOMBIE* and only the next transition to *PSNEEDSINPUT*. The masks would be constructed as follows:

**Example 3** *Constructing masks*

*C language code:*

```
enableMask = PSFROZENMASK | PSZOMBIEMASK;
disableMask = PSRUNNINGMASK;
nextMask = PSNEEDSINPUTMASK;
```

```
XDPSSetStatusMask(ctxt, enableMask, disableMask,
nextMask);
```

Even though the previous setting for *PSNEEDSINPUT* was enabled, *PSNEEDSINPUT* need not be disabled in order to change the treatment of this event to “next only.”

See 4.8, “Synchronization,” for details on how the *PSFROZEN* status event can be used.

### 3.4.1 Event Dispatching

The Client Library is responsible for handling events from the Display PostScript X extension. In addition to the status events described in the previous section, the library handles output events that send wrap return values and PostScript language output back to the application.

The Client Library usually dispatches events from the Display PostScript extension in a wire-to-event converter (a procedure that Xlib calls to format event data from the X server). The events do not appear in the normal X event stream. This can cause problems with certain software libraries—for example, the R4 Xt library—that assume that **XNextEvent** will not block if its connection has data available to be read. If you use one of these libraries and the library calls **XNextEvent**, that call does not return until there is an actual X event to dispatch. Since the events the Display PostScript extension sends do not appear on the normal X event stream, your application may hang until the user does something to generate an event.

Further, event handlers that are invoked using this internal dispatching scheme (described in the previous paragraph) cannot call X or Display PostScript procedures, since Xlib is not reentrant at this level. In that case, the event handler must either queue a task to be done outside the handler or must set a flag. The resulting program logic is often complex.

An alternative to internal event dispatching is pass-through event dispatching. Here, the Client Library causes the events to appear in the normal X event stream. The application is then responsible for dispatching the events by calling **XDPSDispatchEvent**.

To change or query how the Client Library delivers events, an application can call **XDPSSetEventDelivery**. **XDPSSetEventDelivery** allows an application to choose between the default, internal event dispatching, and pass-through event dispatching.

Applications that use pass-through event delivery can call **XDPSIsDPSEvent**, **XDPSIsStatusEvent**, and **XDPSIsOutputEvent** to identify events from the Display PostScript extension. Alternatively, they can pass all events to **XDPSDispatchEvent** and let **XDPSDispatchEvent** identify the extension events.

Pass-through event dispatching is strongly recommended for the following types of applications:

- Applications that use the X Toolkit (Including OSF/Motif applications)
- Applications that handle status events
- Applications that handle text messages from the Display PostScript extension by displaying them in a window

When using pass-through event delivery, you *must* pass all output events to **XDPSDispatchEvent**. Status events may be passed to **XDPSDispatchEvent**, or they may be handled in place. **XDPSDispatchEvent** passes any status events to the status event handler for the event's context. If the application wants to handle events in place, it can call **XDPSIsStatusEvent**, which identifies an event as a status event and extracts the status information from it. The application can then process the information directly.

Applications that use pass-through event delivery must not use **XtAppProcessEvent** to handle X events; **XtAppProcessEvent** ignores the extension events. It is safe to call **XtAppProcessEvent** with a mask of *XtIMTimer* or *XtIMAlternateInput*, but it is unsafe to call it with a mask of *XtIMXEvent* or *XtIMAll*.

Always call **XDPSDispatchEvent** before calling **XtDispatchEvent**. In the MIT release of the X Window System, a bug in the implementation of **XtDispatchEvent** may cause a core dump when an extension event is passed. The Xt main loop for this case is shown in Example 4.

**Example 4** *Calling XtDispatchEvent*

*C language code:*

```
while (1) {
  XEvent event;
  XtAppNextEvent(app, &event);
  if (!XDPSDispatchEvent(&event) &&
      !XtDispatchEvent(&event)) {
   /* Handle undispatched event */
   }
}
```

The call

```
XDPSIsDPSEvent(&event)
```

is equivalent to

```
(XDPSIsStatusEvent(&event, NULL, NULL) ||
    XDPSIsOutputEvent (&event))
```

The call

```
XDPSDispatchEvent(&event)
```

is equivalent to

```
if (XDPSIsStatusEvent(&event, NULL, NULL)) {
 <Call status event handler>
 return True;
} else if (XDPSIsOutputEvent(&event)) {
 <call output event handler>
 return True;
} else return False;
```

### 3.4.2 Wrap Considerations

When an application calls a wrap that returns a value, the Client Library must wait for the results. During this wait, the Client Library dispatches any status and output events to the appropriate event handler as they arrive, using the current event dispatching mode.

If pass-through event dispatching is used, status event handlers and text procedures are allowed to call wraps that do not return values, Xlib procedures, and Display PostScript procedures other than **DPSWaitContext**. They are not allowed to call **DPSWaitContext** or wraps that return a value; if they do, a *dps_err_recursiveWait* error can occur.

If a status event handler or text procedure is invoked with internal event dispatching, it may not call wraps or any X or Client Library procedures.

If a *dps_err_recursiveWait* error occurs, wraps usually return incorrect values, and further errors may be triggered. Applications that handle their own errors should treat *dps_err_recursiveWait* as a fatal error.

*Note: To avoid occurrences of dps_err_recursiveWait errors, status event handlers and text procedures must not call* **DPSWaitContext** *or wraps that return values.*

# 4 Additional Facilities

This section describes advanced features of the Display PostScript extension to the X Window System.

## 4.1 Identifiers

Display PostScript defines two new server resource types: one for contexts and another for spaces. A context or space resource in the server is defined by an X resource ID (XID).

The client has its own representation of contexts and spaces. *DPSContext* is a handle (a pointer) to a *DPSContextRec* allocated in the client's memory. *DPSSpace* is a handle to a *DPSSpaceRec* allocated in the client's memory.

Applications need not use X resource IDs to refer to contexts or spaces. Instead, they can pass the appropriate handle to Client Library procedures.

However, if the resource ID of a context or space is required, there are routines available for translating back and forth between handles and IDs.

- **XDPSXIDFromContext** returns an X resource ID for a given *DPSContext* handle.
- **XDPSXIDFromSpace** returns an X resource ID for a given *DPSSpace* handle.
- **XDPSContextFromXID** returns a *DPSContext* handle, given its X resource ID.
- **XDPSSpaceFromXID** returns a *DPSSpace* handle, given its X resource ID.

The PostScript interpreter uses a unique integer, the *context identifier*, to identify a context. The context identifier is defined by the PostScript language and is completely independent of X resource IDs. The **currentcontext** operator returns the context identifier for the current PostScript context.

*Note: A context created by an existing context with the* **fork** *operator has no identity other than the context identifier returned by the* **fork** *operator; the forked context has neither an X resource ID nor a DPSContext handle. See section 4.5, "Forked Contexts" for more information.*

To get the *DPSContext* handle associated with a particular context identifier, call **XDPSFindContext**. If the client knows about the specified context, a valid *DPSContext* handle is returned; otherwise *NULL* is returned.

There is no direct translation between the PostScript context identifier and the X resource ID.

If a PostScript context terminates (either by request or as the result of an error), the resource allocated for it lingers in the server. The X resource ID for the context is still valid, but the context identifier is not. Such a context is called a zombie. See 4.2, “Zombie Contexts,” for more information.

## 4.2 Zombie Contexts

A context can die in a number of ways, most commonly as the result of a PostScript language error, such as operand stack underflow or use of an undefined name.

If a context is killed, or dies from an error, its server resource lingers. An X server resource that represents a terminated context is known as a *zombie context*. Requests made to a zombie context will fail. The resource associated with a zombie context can be freed with the **DPSDestroyContext** procedure. Alternatively, the resources will be freed when the *Display* is closed, typically at application exit.

Any request made to a zombie context will generate a status event of type *PSZOMBIE*. See section 3.4, “Status Events,” for more information.

## 4.3 Buffers

As discussed in *Client Library Reference Manual*, buffering is often used to enhance throughput. For the most part, an application need not be concerned with buffering of requests to a context or output from a context. However, facilities are provided to flush buffers if needed.

All Display PostScript requests sent to the server are buffered by Xlib, like any other X requests. **DPSFlushContext** will flush any code or data pending for a context, as well as any X requests that have been buffered. For portability and performance enhancement, use **DPSFlushContext** rather than **XFlush** if the application has sent code or data to a context since the last flush.

Streams created by the PostScript interpreter are buffered, including the input and output streams associated with a PostScript execution context. Buffers are automatically flushed as needed. The automatic flushing is usually sufficient. However, should the application need to flush output from a context, the **flush** operator can be used. Note that wrapped procedures that return results include a **flush** operator at the end of the wrap code.

## 4.4 Encodings

*Client Library Reference Manual* discusses the general concept of encodings and conversions. A wrapped procedure always generates a binary object sequence, which is passed to the context for further processing. Typically, the binary object sequence is simply passed to the lowest level of the Client Library to be packaged into a request, without any change to its contents. However, by setting the

encoding parameters of the *DPSContextRec* with the **DPSChangeEncoding** procedure, you can convert the binary object sequence to some other encoding before it is sent or written.

Display PostScript supports the conversions shown in Table 2:

**Table 2** *Encoding conversions*

| *Conversion* | *Description* |
|---|---|
| Binary object sequence to ASCII | This conversion makes a binary object sequence readable by humans. It allows the output of wrapped procedures to be inspected and analyzed. It is also useful for generating page descriptions to be printed. This is the default setting for text contexts. Execution contexts can also be made to convert binary object sequences to ASCII, but there is little purpose in doing so. |
| Binary object sequence to binary-encoded tokens | Binary-encoded token encoding is the most compact encoding for the PostScript language. This conversion is useful for storing code permanently or for exchanging code with another application. Either a text context or an execution context can perform this conversion, but it is mainly used for text contexts. |
| Binary object sequence with user name indices to binary object sequence with user name strings | This conversion is necessary if the binary object sequence is going to be stored permanently (for example, on a file) or if the binary object sequence is to be used by another client or with a shared context (see 4.7, “Sharing Resources”). User name indices are created dynamically and are unique only within a single “instance” of the Client Library—for example, in the application’s process address space. In this case, user names must be represented by strings if they are to be used outside the application’s process address space. |

**Table 2** *Encoding conversions (Continued)*

| *Conversion* | *Description* |
|---|---|
| Binary-encoded tokens to ASCII | This conversion allows binary-encoded tokens read from an external data source such as a file to be converted to ASCII for human inspection, sent to an interpreter, or stored in a page description for printing. After the context's encoding has been set using **DPSChangeEncoding**, buffers of binary-encoded tokens can be read and passed to **DPSWritePostScript** for conversion. Either a text context or an execution context can perform this conversion, but it is used mainly for text contexts.<br><br>For example, the procedure call below causes a text context to generate binary-encoded tokens:<br><br>`DPSChangeEncoding(textContext, dps_encodedTokens, textContext->nameEncoding);`<br><br>The next example causes an execution context to convert user name indices to user name strings:<br><br>`DPSChangeEncoding(context, context->programEncoding, dps_strings);` |



## 4.5 Forked Contexts

The PostScript language allows an existing context to create another context by means of the **fork** operator. However, when a forked context is created, it has no *DPSContext* handle or X resource ID associated with it (see 4.1, "Identifiers"). This is fine if the application does not need to communicate with the forked context. A context that was forked to do some simple task in the background may terminate without generating any output. If the application does need to communicate with a forked context, both a *DPSContext* handle and an X resource ID must be created for the context.

To create a resource ID and *DPSContext* handle for a forked context, call **DPSContextFromContextID**:

```
DPSContext DPSContextFromContextID(ctxt, cid, textProc,
                    errorProc)
  DPSContext ctxt;
  ContextPSID cid;
  DPSTextProc textProc;
  DPSErrorProc errorProc;
```

*ctxt* specifies the context that created the forked context. In other words, *ctxt* is the context that executed the **fork** operator. *cid* is a *long int* that specifies the PostScript context identifier (not the X resource ID) of the forked context.

*textProc* and *errorProc* are the usual context output handlers. If *textProc* is *NULL*, the text handler from *ctxt* is used. If *errorProc* is *NULL*, the error handler from *ctxt* is used.

**DPSContextFromContextID** returns a *DPSContext* handle if *ctxt* and *cid* are valid, otherwise it returns *NULL*.

*Note: Implementation limitations should be kept in mind when using the* **fork** *operator. A context can consume a significant amount of memory. Furthermore, the total number of contexts that can be created in a server is relatively small—on the order of 50 to 100.*

*Caution: When using forked contexts, plan to use* **DPSContextFromContextID** *to hook up with them for debugging, even if the eventual use of the forked context does not require that the application communicate with it. If a forked context generates a PostScript language error but there is no resource ID or DPSContext handle associated with it, the application will never see the error.*

Contexts created by **fork** exist until they are killed or joined (using the **join** operator). A context terminated by the **detach** operator, however, goes away as soon as it finishes executing.

## 4.6 Multiple Servers

An application may create contexts simultaneously on several display devices, each with its own server, at the same time. In these cases, the application must process events from each server to which it is connected.

In order to support access to multiple servers, Display PostScript procedures take a pointer to *Display* records where appropriate.

## 4.7 Sharing Resources

Execution contexts and spaces can be identified by their X resource identifiers. These identifiers can be passed to other clients to enable sharing of resources.

*Caution: There is no support in the Client Library for maintaining the consistency of shared resources. In general, applications should not share resources because of the complexity of managing them.*

*If an application needs to share execution context information with other clients, the shared VM facility and the mutual exclusion operators provided by the PostScript language (***lock***,* **monitor***, and so on) may be adequate for that purpose. See PostScript Language Reference Manual, Second Edition.*

*If these facilities are not adequate, the procedures described in this section can be used.*

**XDPSContextFromSharedID** and **XDPSSpaceFromSharedID** are provided to allow a client to communicate with resources created by a different client.

For the most part, a *DPSContext* handle created for a shared resource can be used like any other handle. However, there are some restrictions. The following list, though not exhaustive, presents some of the issues related to sharing resources:

- User names in binary encodings of the PostScript language must be sent as strings. This is because the mapping of user name indices is not guaranteed to be unique across clients. The default *DPSNameEncoding* of the *DPSContextRec* created for a shared context is *dps_string*. It cannot be changed to *dps_indexed*.

- Output from the context, including wrap result values, text, and errors, is sent only to the context's original creator, not to any clients sharing the context. Status events, however, are sent to all clients sharing the context, as specified by the status event mask (see 3.4, "Status Events").

- When **DPSDestroyContext** or **DPSDestroySpace** is applied to a shared context or space, only the client-side data structures are destroyed. The execution context, the space, and the resources associated with these objects can be destroyed only by the creator.

- If the creator destroys resources, any reference to a destroyed resource will result in a protocol error, which is sent to the client sharing the resource.

It is up to the application that allows resource identifiers to be shared, and the clients sharing those resources, to cooperate and maintain consistency.

## 4.8 Synchronization

As discussed in "Mixing Display PostScript and X Rendering" in 3.3, "Execution," X rendering primitives and PostScript language execution may, in most cases, be intermixed freely. However, in some situations PostScript language execution needs to be synchronized with X.

See *Client Library Reference Manual* for a discussion of the general requirements for synchronization. To summarize, you can synchronize either by calling wraps that return results or by calling **DPSWaitContext**. Enforced synchronization is expensive and should be used only when absolutely necessary.

*Note: Synchronizing with the Display PostScript extension also synchronizes with the X server; there is no need to call* **XSync** *explicitly. The reverse is not true; calling* **XSync** *does not synchronize with the Display PostScript extension.*

Flushing, however, works both ways: flushing an X connection flushes all contexts on that connection, and flushing a context flushes the X connection of that context.

For example, suppose a previewer application displays a page of text and graphics that is represented by a PostScript language page description in a file. The user interface of the application may require the entire page to be imaged to a pixmap before it is realized on the physical display. The application reads the ASCII-encoded PostScript language code from the file and sends it to the server with the **DPSWritePostScript** procedure. The context executes the code as it is received, and renders to the pixmap.

If the file contains only one page, and the page description is simple, the application knows that the pixmap is complete when it has read to the end of the input file and called **DPSWaitContext**. It may now call **XCopyArea** to copy the pixmap to the application display window.

However, if the file contains more than one page, the application cannot know when the rendering to the pixmap is complete. If it calls **XCopyArea** too soon, the context may not have finished drawing. As a result, an incomplete image will be displayed on the screen.

There are two main strategies for handling situations such as the one described above: waiting and freezing. The first is applicable if the application has sufficient knowledge of the content of the PostScript language code to know where the beginning and the end are located. The second is used only if the application has no reliable knowledge of the code content.

### 4.8.1 Waiting

Causing the context to wait is appropriate when the PostScript language code to be executed has a known structure. This is true in either of the following circumstances:

- The application has complete control of the code to be executed. That is, it uses wrapped procedures, single-operator procedures, or dynamically generated code fragments such as user path descriptions. No code comes from external sources such as end-user input.

- The application reads external files with a known structure that can be parsed and understood, such as PostScript language page descriptions that are compliant with Adobe's Document Structuring Conventions.

Most applications that require synchronization fall into one of the two categories described above. In both cases, the application knows exactly how much PostScript language code needs to be sent for a complete display. In these cases, the application sends the code and then forces all code to be executed, either with **DPSWaitContext** or as a side effect of calling a wrap that returns a value. When either of these procedures returns, the application knows that all rendering is done and that other X requests can now be sent.

### 4.8.2 Freezing

Freezing a context is appropriate if the application cannot determine the completeness of the PostScript language code to be executed. This can happen if an end user is allowed to enter arbitrary PostScript language programs (for instance, in an interactive interpreter executive) or if an input file lacks a well-defined structure.

In this case, the input must contain an executable name that the application has defined. For example, the **showpage** operator terminates each page in a page description file. The application can take advantage of this by defining **showpage** to execute an operator that will notify the application that the page is done. The **clientsync** operator fulfils this function:

```
/old_showpage /showpage load def
/showpage {old_showpage clientsync} bind def
```

When **clientsync** is executed, the context is put into the *PSFROZEN* state, and a *PSFROZEN* event is generated. The application must have enabled the *PSFROZEN* event and registered a handler for that context; see 3.4, “Status Events” for more information on status events. The handler may then set a flag indicating that the image in the pixmap is complete. The next time the application goes around its main loop, it can test the flag and call **XCopyArea**.

A frozen context can still receive interrupts. **DPSInterruptContext** will interrupt a context whether it is frozen or not.

# 5 Programming Tips

This section contains tips to help you program applications that use the Display PostScript system extension to the X Window System.

## 5.1 Avoid XIfEvent

If your application uses internal event dispatching as described in section 3.4, it should not use **XIfEvent**. This routine will cause events that were generated and queued by an execution context to be processed repeatedly (once for each call to **XIfEvent**) without being dequeued. This may result in wrap results or text output being erroneously duplicated or may cause false status events to be reported. Use **XCheckIfEvent** instead.

This restriction does not apply to applications using pass-through event dispatching and may not apply to future implementations of Xlib.

*Caution: If your toolkit uses* **XIfEvent**, *you may see the erroneous effects described above even though your application does not use* **XIfEvent** *directly.*

## 5.2 Include Files

Include the <*DPS/dpsXclient.h*> header file when compiling Display PostScript applications. This header file includes the required header files (*dpsclient.h* and *dpsfriends.h*) described in sections 9 and 11 of *Client Library Reference Manual*.

Include <*DPS/dpsops.h*> if your application uses single-operator procedures with explicit contexts.

Include <*DPS/psops.h*> if your application uses single-operator procedures with implicit contexts.

Include <*DPS/dpsexcept.h*> if your application uses exception handling as defined in *Client Library Reference Manual*.

## 5.3 Use Pass-Through Event Dispatching

Use **XDPSSetEventDelivery** as described in section 3.4 to set pass-through event dispatching for your application's contexts. Pass-through event dispatching has many advantages:

- Your application can make X and Display PostScript calls in its text and status event handlers. For example, writing a text handler that displays the text in a window is easy. In contrast, internal event dispatching would require the text handler to queue up the display task and leave its execution to the main application.

- Your application avoids potential delays from toolkits that do not expect events to be dispatched internally.

- Your application can handle status events directly rather than having a status event handler that sets flags for the main application to test.

*Note: Here are two important things to remember when using pass-through event dispatching in X Toolkit applications:*

- Always call **XDPSDispatchEvent** before calling **XtDispatchEvent**.

- Never use **XtAppProcessEvent** to handle X events.

## 5.4 Be Careful With Exception Handling

The exception handling facilities described in Appendix B of *Client Library Reference Manual* can be used in X programs, but you must be very careful not to jump through any Xlib or X Toolkit procedures. The internal state of the libraries may become corrupted. Here are some examples of uses that *not* safe:

- Do not raise an exception in any X Toolkit callback procedure.

- Do not raise an exception in the predicate procedure to **XIfEvent** or any of the related event handling procedures.

- Do not raise an exception in an event handler or text procedure unless you are using pass-through event dispatching.

- Do not raise an exception in an error handler.

## 5.5 Coordinate Conversions

The code examples in this section demonstrate an efficient method of doing coordinate conversions. (For an introduction to coordinate system issues, see “Coordinate Systems” in 3.3, “Execution.”)

At initialization, and immediately after any user space transformation has been performed (for example, after **scale**, **rotate**, or **setmatrix**), the application should execute PostScript language code to get the CTM (current transformation matrix), the inverse of the CTM, and the current origin offset. The wrap shown in Example 5 will return these values:

**Example 5** *Getting CTM, inverse CTM, and current origin offset*

*Wrap definition:*

```
defineps PSWGetTransform(DPSContext ctxt | float ctm[6],
             invctm[6]; int *xOffset, *yOffset)
  matrix currentmatrix dup ctm
```

```
  matrix invertmatrix invctm
  currentXoffset yOffset xOffset
endps
```

Call the **PSWGetTransform** wrap as necessary, saving the return values in storage associated with the window:

**Example 6** *Calling PSWGetTransform*

*C language code:*

```
DPSContext ctxt;
float ctm[6], invctm[6];
int xOffset, yOffset;

PSWGetTransform(ctxt, ctm, invctm, &xOffset, &yOffset);
```

To convert an X coordinate into a user space coordinate, perform the calculations shown in Example 7.

**Example 7** *Converting an X coordinate to user space*

*C language code:*

```
#define A_COEFF 0
#define B_COEFF 1
#define C_COEFF 2
#define D_COEFF 3
#define TX_CONS 4
#define TY_CONS 5

int x, y;     /* X coordinate */
float ux, uy; /* user space coordinate */

x -= xOffset;
y -= yOffset;
ux = invctm[A_COEFF] * x + invctm[C_COEFF] * y +
                        invctm[TX_CONS];
uy = invctm[B_COEFF] * x + invctm[D_COEFF] * y +
                        invctm[TY_CONS];
```

To convert a user space coordinate into an X coordinate, perform the calculations shown in Example 8.

**Example 8** *Converting a user space coordinate to an X coordinate*

*C language code:*

```
x = ctm[A_COEFF] * ux + ctm[C_COEFF] * uy + ctm[TX_CONS] +
                        xOffset;

y = ctm[B_COEFF] * ux + ctm[D_COEFF] * uy + ctm[TY_CONS] +
                        yOffset;
```

The equations listed above have the following limitations:

- X coordinates must be positive. Otherwise, use the **floor** function to avoid the implicit truncation that happens when floating-point values are assigned to integers.
- Beware of round-off errors. Incorrect coordinates may be computed in either direction.

## 5.6 Fonts

The **filenameforall** operator can be used to obtain a list of the fonts available to the server. See *PostScript Language Reference Manual, Second Edition*, for a description of **filenameforall**. Use the pattern

```
(%font%*)
```

to generate a list of fonts. The font file names may be sent back as ASCII text and processed with a customized text handler, or they may be stored in an array and then accessed one at a time by calling a wrapped procedure.

Outline fonts are resources. As with any other resource, there's no guarantee that a given font will be present on any particular server. The application must be written to deal with a **findfont** or **selectfont** operator that fails because it can't find the font. It is possible to redefine **findfont** and **selectfont** so that they substitute some default font when the requested font is not available. Indeed, the default definition of **findfont** in a given environment may already do this.

## 5.7 Portability Issues

The Display PostScript extension enhances the portability of X applications by providing flexibility with respect to color, resolution, and fonts.

### 5.7.1 Color

Use PostScript operators such as **setrgbcolor** rather than X primitives to draw with color. The PostScript interpreter will provide the best rendering possible for the device. The Display PostScript system can produce a variety of halftone patterns representing gray values or colors, so that one color can be seen against

the background of another color even on a monochrome device. Contrast this with the rendering facilities of the X Window System, where a request for any color other than white on a monochrome device will give you black.

Display PostScript color rendering is device independent. Here's how Display PostScript handles color requests:

- On a monochrome device, you'll get a dithered (halftone) pattern of black and white pixels. For example, if you ask for red by specifying `1 0 0 setrgbcolor` you'll get some halftone gray pattern composed of black and white pixels; this pattern will be distinct from other "colors."

- On a grayscale device, you'll get a halftone pattern using gray levels; this offers greater distinction among "colors."

- On a color device (4-plane, 8-plane, and so on), you'll get the requested color if it's one of those predefined for the context; otherwise you'll get a dithered pattern of RGB pixels that approximates the color.

- If you've allocated solid colors beyond those predefined for the context, you'll get a nondithered color just as you would with X (subject to the same restrictions).

- A color request will never simply fail.

X Window System color rendering, on the other hand, is device dependent:

- On a monochrome device, a request for any color will give you black. There's no way to differentiate between "pink" and "olive green," as there is with PostScript language color rendering.

- On a color device, you'll get the color you requested only if there's space in the colormap or the device is a TrueColor device.

- A color request can fail, and there's no recourse except to try requesting a different color.

### 5.7.2 Resolution

The Display PostScript extension offers you device independence with respect to resolution.

In Display PostScript, positions and extents are specified with resolution-independent units such as points. An inch is always an inch. Window elements will always have the same absolute size, regardless of the device.

In the X Window System, positions and extents are specified in units of pixels. The size of a pixel depends on the device. One inch may be 75 pixels on one display and 100 pixels on another display. This causes strange distortions of size when creating windows on various display devices.

### 5.7.3 Fonts

In the X Window System, you can't rely on the availability of a given point size/typeface combination. If you request 9-point Helvetica*, for example, and that point size is not available, you must make another request.

The Display PostScript extension gives you added flexibility with respect to fonts:

- You can have any point size as long as the typeface is present. If you request a size that's not available, Display PostScript generates it for you.
- The typeface can be rendered in any rotation or two-dimensional transformation.

## 5.8 Using Custom Operators

After the execution of a **setXgcdrawable**, **setXgcdrawablecolor**, or **setXoffset** operator, the following graphics state parameters are left in an indeterminate state:

- The current transformation matrix
- The clipping path
- The transfer function

Each of the parameters will either keep its previous value or be restored to its initial value, but it is not always possible to predict which. To return the parameters to a known state, follow one of the following techniques:

- Reset the parameters to their initial values after executing **setXgcdrawable**, **setXgcdrawablecolor**, or **setXoffset** by executing the following PostScript language code:

**Example 9** *Resetting clipping path, transfer function, and CTM*

*PostScript language code:*

```
initmatrix
initclip
gsave initgraphics currenttransfer grestore settransfer
```

- Retain the previous values of the parameters by surrounding the use of **setXgcdrawable**, **setXgcdrawablecolor**, or **setXoffset** with the following PostScript language code:

**Example 10** *Retaining previous values of clipping path, transfer function, and CTM*

*PostScript language code:*

```
matrix currentmatrix
clippath
currenttransfer
%
% use of setXgcdrawable,
% setXgcdrawablecolor, or setXoffset
%
settransfer
initclip clip
setmatrix
```

Note that the second method changes the current path; maintaining both the current path and the clipping path is complex and rarely necessary. Retaining the graphics state parameter values may lead to unexpected results when the application switches among drawables on different screens or different visuals, and is not recommended in this case.

You may mix the two techniques for different graphics state parameters. For example, to reset the clipping path and transfer function but keep the current transformation matrix, execute the following PostScript language code:

**Example 11** *Resetting clipping path and transfer function while keeping CTM*

*PostScript language code:*

```
matrix currentmatrix
%
% use of setXgcdrawable, setXgcdrawablecolor, or setXoffset
%
setmatrix
initclip
gsave initgraphics currenttransfer grestore settransfer
```

If you know that one of the parameters has not been changed from its initial value, you can safely ignore that parameter. For example, if you do not change the transfer function, it will be left in its initial state after you execute **setXgcdrawable**, **setXgcdrawablecolor**, or **setXoffset**—either because the initial value has been reestablished, or because it has been inherited.

For performance reasons, you should execute **setXgcdrawable**, **setXgcdrawablecolor**, and **setXoffset** as infrequently as possible. It is more efficient to capture a particular graphics state configuration as a gstate object and

use **setgstate** to return to it than to use **setXgcdrawable**, **setXgcdrawablecolor**, or **setXoffset** to reestablish the configuration each time you need it. In addition, the graphics state indeterminacies described above do not occur when using gstate objects.

## 5.9 Changing Fields in Graphics Contexts

If you change any fields in a graphics context (GC) that is being used by an execution context, you must ensure correct synchronization with the extension by performing the following steps:

1. Call **DPSWaitContext** for each execution context that is using the GC. This guarantees that all PostScript language code that should execute with the old GC values has completed. You can omit this step if the contexts are already synchronized with the application.

2. Use Xlib calls to change the values in the GC.

3. Call **XFlushGC** for the GC. **XFlushGC** was added to Xlib in X11 Release 5 and is not available in libraries conforming to earlier releases. If necessary, you can define it in your program as shown in Example 12.

**Example 12** *Defining XFlushGC*

*C language code:*

```
#ifndef XlibSpecificationRelease/* New to X11/R5 */
#include <X11/Xlibint.h>

voidXFlushGC(dpy, gc)
    Display *dpy;
    GC gc;
{
    FlushGC(dpy, gc);
}
#endif /* XlibSpecificationRelease */
```

4. Further PostScript language code now executes with the new values of the GC.

# 6 X-Specific Data and Procedures

This section describes the system-specific data types and procedures for the Display PostScript extension to X.

## 6.1 Data Structures

Data structures defined in the <*DPS/dpsXclient.h*> header file are described below.

### 6.1.1 Extended Error Codes

The following error codes for the X Window System are in addition to those described under *DPSErrorCode* in *Client Library Reference Manual*:

| | |
|---|---|
| *dps_err_invalidAccess* | An attempt was made to receive output from a context created by another client. Contexts send their output only to the original creator. If the application tries to get output from a context created by another client—for example, by calling a wrap that returns a result—this error is reported. |
| *dps_err_encodingCheck* | An attempt was made to change name or program encoding to unacceptable values. This error can occur when changing name encoding for a context created by another client or a context created in a space that was created by another client. Such contexts must have string name encoding (*dps_strings*). |
| *dps_err_closedDisplay* | An attempt was made to send PostScript language code to a context whose display is closed. |
| *dps_err_deadContext* | An attempt was made to get output from a zombie context (a context that has died in the server but still has its X resources active). |
| *dps_err_recursiveWait* | An event handler called **DPSWaitContext** or a wrap that returns a value; see “Wrap Considerations” on page CLX-102 for more information |

### 6.1.2 Status Event Masks

The status event types supported in Display PostScript are shown in Table 3. The first column shows the status event type that is reported by the server. The second column shows the associated single-bit status mask values that can be combined with logical OR to set a context's status mask. The third column describes the status event.

**Table 3** *Status events*

| *Status Event* | *Mask ValueStatus* | *Description* |
|---|---|---|
| PSRUNNING | PSRUNNINGMASK | Context is runnable. |
| PSNEEDSINPUT | PSNEEDSINPUTMASK | Context needs input to continue running. |
| PSZOMBIE | PSZOMBIEMASK | Context is dead, but its X resources remain. |
| PSFROZEN | PSFROZENMASK | Context was frozen by PostScript language program. |
| PSSTATUSERROR | — | Could not reply to status request. |

For more information on status events, see section 3.4 on page CLX-102.



### 6.1.3 Types and Global Variables

**DPSEventDelivery**

```
typedef enum {

  dps_event_pass_through,
  dps_event_internal_dispatch,
  dps_event_query
} DPSEventDelivery;
```

*DPSEventDelivery* provides the possible options for **XDPSSetEventDelivery**.

This enumeration is not available in early versions of the Client Library.

**DPSLastUserObjectIndex**

```
long int DPSLastUserObjectIndex;
```

*DPSLastUserObjectIndex* is a global variable containing the last user object index assigned for this application. This variable should be treated as read-only. For more information about user object indices, see **DPSNewUserObjectIndex** on page CLX-128 and "User Object Indices" on page CLX-91.

**XDPSStatusProc**

```
typedef void (*XDPSStatusProc)(/*
  DPSContext ctxt,
  int code */);
```

This is a procedure type for defining the callback procedure that handles status events for the client. The procedure will be called with two parameters: the context it was registered with and the status code derived from the event. For more information about status events, see **XDPSRegisterStatusProc** on page CLX-134 and "Status Event Masks" on page CLX-124.

## 6.2 Procedures

This section contains descriptions of the system-specific procedures in the *<DPS/dpsXclient.h>* header file, listed alphabetically.

**DPSChangeEncoding**

```
void DPSChangeEncoding(ctxt, newProgEncoding,
 newNameEncoding)
  DPSContext ctxt;
  DPSProgramEncoding newProgEncoding;
  DPSNameEncoding newNameEncoding;
```

**DPSChangeEncoding** changes one or both of the context's encoding parameters. Supported conversions are described in Table 2 on page PG-110. See *Client Library Reference Manual* for definitions of *DPSNameEncoding* and *DPSProgramEncoding*.

**DPSContextFromContextID**

```
DPSContext DPSContextFromContextID(ctxt, cid, textProc,
    errorProc)
  DPSContext ctxt;
  ContextPSID cid;
  DPSTextProc textProc;
  DPSErrorProc errorProc;
```

**DPSContextFromContextID** creates a *DPSContextRec* and returns a *DPSContext* handle for a forked context; it returns *NULL* if it is unable to create these data structures.

The application must call this procedure before attempting to communicate with a forked context. **DPSContextFromContextID** creates the client-side data structures for the context and associates them with the server-side structures previously created by the **fork** operator. *cid* is the context identifier (of type *long int*) that is assigned to the forked context by the PostScript interpreter. *ctxt* is the

handle of the context that created the forked context; its *DPSContextRec* will be used as a model for the *DPSContextRec* of the forked context, as described below.

If a *DPSContextRec* has already been created for *cid*, its handle is returned by **DPSContextFromContextID**. Otherwise, a new context record is created according to the following rules:

- If supplied, the *textProc* and *errorProc* arguments are used for the forked context.
- If *textProc* or *errorProc* are *NULL*, the missing values are copied from the *DPSContextRec* of *ctxt*.
- The chaining pointers for the forked context are set to *NULL*.
- All other fields in the new *DPSContextRec* are copied from *ctxt*.

**DPSCreateTextContext**

```
DPSContext DPSCreateTextContext(textProc, errorProc)

  DPSTextProc textProc;
  DPSErrorProc errorProc;
```

**DPSCreateTextContext** creates a text context and returns its *DPSContext* handle. When this handle is passed as the argument to a Client Library procedure, all input to the context is passed to *textProc*. If the input is PostScript language in a binary encoding, the input is converted to ASCII encoding before being passed to *textProc*. *errorProc* is used to report any errors (such as *dps_err_nameTooLong*) resulting from converting binary encodings to ASCII encoding. *textProc* is responsible for dealing with errors resulting from handling the text, such as file system or I/O errors.

**DPSDefaultTextBackstop**

```
void DPSDefaultTextBackstop(ctxt, buf, count)

  DPSContext ctxt;
  char *buf;
  unsigned count;
```

**DPSDefaultTextBackstop** is the text backstop procedure automatically installed by Display PostScript. Since it is of type *DPSTextProc*, you may use it as your context *textProc*. The text backstop procedure writes text to *stdout* and flushes *stdout*.

**DPSDestroyContext**

```
void DPSDestroyContext(ctxt)

  DPSContext ctxt;
```

**DPSDestroyContext** is as defined in *Client Library Reference Manual*, except as it pertains to shared contexts.

Both the client and the server are affected by this procedure. On the client side, **DPSDestroyContext** destroys the *DPSContextRec*. On the server side, it destroys the PostScript execution context and the X resource associated with it. After a call to **DPSDestroyContext**, the *DPSContext* handle for *ctxt* is no longer valid.

If the context is a shared context (that is, a *DPSContextRec* allocated for a context created by another client), only the *DPSContextRec* is destroyed; the interpreter context and resource are unchanged.

For text contexts, **DPSDestroyContext** destroys the *DPSContextRec*.

**DPSDestroySpace**

```
void DPSDestroySpace(spc)

  DPSSpace spc;
```

**DPSDestroySpace** is as defined in *Client Library Reference Manual* except for shared spaces.

For spaces created by the client, this procedure destroys the space and the X resource associated with it. PostScript execution contexts that use this space are also destroyed, along with their X resources and *DPSContextRec* records. Finally, the *DPSSpaceRec* is destroyed.

If the space is a shared space (a *DPSSpaceRec* allocated by another client), the space and the X resource are not destroyed. Only the *DPSSpaceRec* is destroyed, along with any *DPSContextRec* records for contexts associated with this space. See section 4.7 on page CLX-112 for a discussion of shared resources.

If the client that created the space destroys it and there are other clients sharing it, the space is destroyed and the sharing clients will experience unpredictable results.

**DPSNewUserObjectIndex**

```
long int DPSNewUserObjectIndex( );
```

**DPSNewUserObjectIndex** returns a new user object index. The Client Library is the sole allocator of new user object indices. The application should not attempt to compute them from a previously obtained index. Because user object

indices are dynamic, they should not be used as numeric values for computation or saved in long-term storage such as a file. See “User Object Indices” on page CLX-97 for more information.

**XDPSContextFromSharedID**

```
DPSContext XDPSContextFromSharedID(dpy, cid, textProc,
    errorProc)

  Display *dpy;
  ContextPSID cid;
  DPSTextProc textProc;
  DPSErrorProc errorProc;
```

**XDPSContextFromSharedID** creates a *DPSContextRec* for a context that was created by another client.

*cid* specifies the context. (*cid* is the context identifier assigned by the PostScript interpreter, not the X resource ID.) *dpy* is the *Display* that both clients are connected to. *textProc* and *errorProc* are the context text and error handlers for the shared context. For information on sharing resources, see section 4.7 on page CLX-112.

**XDPSContextFromXID**

```
DPSContext XDPSContextFromXID(dpy, xid)

  Display *dpy;
  XID xid;
```

**XDPSContextFromXID** gets the context record for the given X resource ID on *dpy.* It returns *NULL* if *xid* is not valid.

**XDPSCreateContext**

```
DPSContext XDPSCreateContext(dpy, drawable, gc, x, y,
    eventmask,grayramp, ccube, actual,
    extProc, 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;
```

**XDPSCreateContext** creates a context with a customized colormap; it returns *NULL* if there is any error.

*dpy*, *drawable*, *gc*, *x*, *y*, *textProc*, *errorProc*, and *space* are the same as for **XDPSCreateSimpleContext**. *eventmask* is reserved for future extensions and should be passed as zero.

The colormap specified in *grayramp* and *ccube* must contain a range of uniformly distributed colors. *grayramp* specifies the factors needed to compute a pixel value for a particular gray level. *grayramp* is required. *ccube* specifies the factors needed to compute a pixel value for a particular RGB color. *ccube* is optional; if it is passed as *NULL*, rendering will be done in shades of gray. The colormap specified in *ccube* must be the same as the one specified in *grayramp*. *actual* specifies the upper limit of the number of additional RGB colors the application plans to request, beyond those specified in *ccube* and *grayramp*.

The following restrictions apply:

- *drawable* and *gc* must be on the same screen.
- *drawable* and *gc* must have the same depth *Visual*.
- If the *drawable* is a *Window*, any colormaps specified must have the same *Visual*.
- *grayramp* must be specified; *ccube* is optional; both must be valid.

See 3.2, “Creating a Context,” for additional information.

**XDPSCreateSimpleContext**

```
DPSContext XDPSCreateSimpleContext(dpy, drawable, gc, x, y,
    textProc, errorProc, space)

  Display *dpy;
  Drawable drawable;
  GC gc;
  int x;
  int y;
  DPSTextProc textProc;
  DPSErrorProc errorProc;
  DPSSpace space;
```

**XDPSCreateSimpleContext** creates a context with the default colormap; it returns *NULL* if there is any error.

The procedure creates a context associated with *dpy*, *drawable*, and *gc*.

*x* and *y* are offsets from the *drawable* origin to the PostScript device space origin in pixels.

*textProc* points to the procedure that will be called to handle text output from the context. *errorProc* points to the procedure that will be called to handle errors reported by the context. *space* determines the private VM of the new context. A *NULL* space causes a new one to be created.

The following restrictions apply:

- *drawable* and *gc* must be on the same screen.
- *drawable* and *gc* must have the same depth *Visual*.

See 3.2, “Creating a Context,” on page CLX-85 for additional information.

**XDPSDispatchEvent**

```
Bool XDPSDispatchEvent (event)

  XEvent *event;
```

**XDPSDispatchEvent** checks whether an event is a Display PostScript event and, if so, dispatches it to the appropriate status or output handler, as follows:

- If the event is not a Display PostScript event, **XDPSDispatchEvent** returns *False* and does nothing else.
- If the event is a Display PostScript event, **XDPSDispatchEvent** determines the context from the event, calls the context’s status or output handler, and returns *True*.

This procedure is not available in early versions of the Client Library.

**XDPSFindContext**

```
DPSContext XDPSFindContext(dpy, cid)

  Display *dpy;
  long int cid;
```

**XDPSFindContext** returns the *DPSContext* handle of a context given its context identifier, *cid*. It returns *NULL* if the context identifier is invalid.

**XDPSGetContextStatus**

```
int XDPSGetContextStatus(ctxt)

  DPSContext ctxt;
```

**XDPSGetContextStatus** returns the status of *ctxt*. This procedure does not alter the mask established for *ctxt* by **XDPSSetStatusMask**. For information on status events, see section 3.4 on page CLX-102 and section 6.1 on page CLX-124.

**XDPSGetDefaultColorMaps**

```
void XDPSGetDefaultColorMaps (dpy, screen, drawable,
    colorcube, grayramp)

  Display *dpy;
  Screen *screen;
  Drawable drawable;
  XStandardColormap *colorcube;
  XStandardColormap *grayramp;
```

**XDPSGetDefaultColorMaps** returns the colormaps used in creating a simple context. The display must be specified.

- If *screen* is *NULL* and *drawable* is *None*, the colormaps are retrieved for the default screen of the display.
- If *screen* is *NULL* and *drawable* is not *None*, the colormaps are retrieved for the drawable's screen.
- If *screen* is not *NULL*, the colormaps are retrieved for that screen.

Either *colorcube* or *grayramp* may be *NULL*, indicating that the colormap is not needed.

This procedure is not available in early versions of the Client Library.

**XDPSIsDPSEvent**

```
Bool XDPSIsDPSEvent (event)

  XEvent *event;
```

**XDPSIsDPSEvent** returns *True* if the event is a Display PostScript event and *False* otherwise.

This procedure is not available in early versions of the Client Library.

**XDPSIsOutputEvent**

```
Bool XDPSIsOutputEvent (event)

  XEvent *event;
```

**XDPSIsOutputEvent** returns *True* if *event* is a Display PostScript output event and *False* otherwise.

The contents of an output event are not defined. If **XDPSIsOutputEvent** returns *True*, the event must be passed to **XDPSDispatchEvent**. If the application does not pass the event to **XDPSDispatchEvent**, the results are undefined.

This procedure is not available in early versions of the Client Library.



**XDPSIsStatusEvent**

```
Bool XDPSIsStatusEvent (event, ctxt, status)

  XEvent *event;
  DPSContext *ctxt;
  int *status;
```

**XDPSIsStatusEvent** returns *True* if *event* is a Display PostScript status event and *False* otherwise. If the event is a status event, *ctxt* and *status* are set to that event's context and status. Either *ctxt* or *status* can be *NULL* if the information is not needed.

The contents of a status event is not defined; the returned context and status values are the only way to extract the information from the event.

This procedure is not available in early versions of the Client Library.

**XDPSRegisterStatusProc**

```
XDPSStatusProc XDPSRegisterStatusProc(ctxt, proc)

  DPSContext ctxt;
  XDPSStatusProc proc;
```

**XDPSRegisterStatusProc** registers a status event handler, *proc*, to be called when a status event is received by the client for the context specified by *ctxt*. The status event handler may be called by Xlib any time the client gets events or checks for events.

*XDPSStatusProc* replaces the previously registered status event handler for the context, if any. *proc* handles only status events generated by *ctxt*; if the application has more than one context, **XDPSRegisterStatusProc** must be called separately for each context.

**XDPSRegisterStatusProc** returns the old status procedure when a new one is registered.

In early versions of the Client Library, this procedure returns *void*.

**XDPSSetEventDelivery**

```
DPSEventDelivery XDPSSetEventDelivery (dpy, newMode)

  Display *dpy;
  DPSEventDelivery newMode;
```

An application can call **XDPSSetEventDelivery** to change or query how the Client Library delivers events.

**XDPSSetEventDelivery** always returns the previous event delivery mode for the specified display.

- If *newMode* is *dps_event_query*, **XDPSSetEventDelivery** does nothing else.
- If *newMode* is *dps_event_internal_dispatch*, the Client Library dispatches events internally without passing them to the application. This is the default value.
- If *newMode* is *dps_event_pass_through*, the Client Library stops dispatching events internally and passes them through to the application as normal X events.

This procedure is not available in early versions of the Client Library.

**XDPSSetStatusMask**

```
void XDPSSetStatusMask(ctxt, enableMask, disableMask,
     nextMask)

  DPSContext ctxt;
  unsigned long enableMask, disableMask, nextMask;
```

**XDPSSetStatusMask** sets the status mask for the context, as follows:

- *enableMask* specifies status events for which continuing notification to the client is requested.
- *disableMask* specifies status events for which the client does not want to be notified.
- *nextMask* specifies status events for which the client wants to be notified of the next occurrence only. Setting *nextMask* is equivalent to setting *enableMask* for a status event and, after being notified of the next occurrence, setting *disableMask* for that event.

A given status event type may be set in only one of the three status masks. If an event is set in more than one mask, a protocol error (*Value*) is generated and the context is left unchanged. For more information on status events, see sections 3.4 and 6.1.



**XDPSSpaceFromSharedID**

```
DPSSpace XDPSSpaceFromSharedID(dpy, sxid)

  Display *dpy;
  SpaceXID sxid;
```

**XDPSSpaceFromSharedID** creates a *DPSSpaceRec* for the space identified by an X resource ID, *sxid*, that was created by another client. *dpy* is the *Display* that both clients are connected to. **XDPSSpaceFromSharedID** returns *NULL* if *sxid* is not valid.

**XDPSSpaceFromXID**

```
DPSSpace XDPSSpaceFromXID(dpy, xid)

  Display *dpy;
  XID xid;
```

**XDPSSpaceFromXID** gets the space record for the given X resource ID on *dpy*. It returns *NULL* if *xid* is not valid.

**XDPSUnfreezeContext**

```
void XDPSUnfreezeContext(ctxt)

  DPSContext ctxt;
```

**XDPSUnfreezeContext** notifies a context that is in the *PSFROZEN* state to resume execution. Attempting to unfreeze a context that is not frozen has no effect.

**XDPSXIDFromContext**

```
XID XDPSXIDFromContext(Pdpy, ctxt)

  Display **Pdpy;
  DPSContext ctxt;
```

**XDPSXIDFromContext** gets the X resource ID for the given context record and returns its *Display* in the location pointed to by *Pdpy*. *Pdpy* is set to *NULL* if *ctxt* is not a valid context.

**XDPSXIDFromSpace**

```
XID XDPSXIDFromSpace(Pdpy, spc)

  Display **Pdpy;
  DPSSpace spc;
```

**XDPSXIDFromSpace** gets the X resource ID for the given space record and returns its *Display* in the location pointed to by *Pdpy*. *Pdpy* is set to *NULL* if *spc* is not a valid space.

# 7 X-Specific Custom PostScript Operators

This section describes the custom PostScript operators for the Display PostScript system extension to the X Window System. The operators are organized alphabetically by operator name. Each operator description is presented in the following format:

**operator** ***operand***$_1$ ***operand***$_2$ *...* ***operand***$_n$ ***operator result***$_1$ *...* ***result***$_m$

Detailed explanation of the operator.

**Errors** **A list of the errors that this operator might execute.**

At the head of an operator description, ***operand***$_1$ through ***operand***$_n$ are the operands that the operator requires, with ***operand***$_n$ being the topmost element on the operand stack. The operator pops these objects from the operand stack and consumes them. After executing, the operator leaves the objects ***result***$_1$ through ***result***$_m$ on the stack, with ***result***$_m$ being the topmost element.

The notation '–' in the operand position indicates that the operator expects no operands; a '–' in the result position indicates that the operator returns no results.

Error conditions include the following:

| | |
|---|---|
| rangecheck | Invalid match: either the *drawable* and *gc* have different depths or they don't have a *Visual* that matches the colormap associated with the context. |
| stackunderflow | Not enough operands on the operand stack. |
| typecheck | Invalid X resource ID. |
| undefined | The device associated with the context is not a display device. |

**clientsync** – **clientsync** –

The **clientsync** operator synchronizes the application with the current context. **clientsync** notifies the current context to stop executing, sets the context status to *FROZEN*, and causes a *PSFROZEN* status event to be generated. To resume execution, call the **XDPSUnfreezeContext** procedure.

For an example of the use of **clientsync**, see section 4.8.2 on page CLX-115.

**currentXgcdrawable** – **currentXgcdrawable** *gc drawable x y*

The **currentXgcdrawable** operator returns the X *gc*, *drawable*, and offset from the origin of the *drawable* to the device space origin for the current context. Results returned by this operator can be input to **setXgcdrawable**. The returned *gc* is a *GContext* identifier, not a *GC* pointer.

**Errors: undefined**

**currentXgcdrawablecolor** – **currentXgcdrawablecolor** *gc drawable x y colorinfo*

The **currentXgcdrawablecolor** operator is similar to the **currentXgcdrawable** operator, except that it also returns an array of 12 integers describing the color cube, gray ramp, and other color variables used for the context. The returned *gc* is a *GContext* identifier, not a *GC* pointer. The *colorinfo* array, described in Table 4, has the form shown in Example 13.

**Example 13** *Form of colorinfo array*

```
[maxgrays graymult firstgray maxred redmult maxgreen
 greenmult maxblue bluemult firstcolor colormap actual]
```

**Table 4** *Description of colorinfo array values*

| *Value* | *Description* |
|---|---|
| *maxgrays* | Maximum number of gray values. Equivalent to *red_max* field of *XStandardColormap* for the gray ramp. |
| *graymult* | Scale factor to compute gray pixel. Equivalent to *red_mult* field of *XStandardColormap* for the gray ramp. |
| *firstgray* | First gray pixel value. Equivalent to *base_pixel* field of *XStandardColormap* for the gray ramp. |
| *maxred* | Maximum number of red values. Equivalent to *red_max* field of *XStandardColormap*. |
| *redmult* | Scale factor to compute color pixel. Equivalent to *red_mult* field of *XStandardColormap*. |
| *maxgreen* | Maximum number of green values. Equivalent to *green_max* field of *XStandardColormap*. |
| *greenmult* | Scale factor to compute color pixel. Equivalent to *green_mult* field of *XStandardColormap*. |
| *maxblue* | Maximum number of blue values. Equivalent to *blue_max* field of *XStandardColormap*. |
| *bluemult* | Scale factor to compute color pixel. Equivalent to *blue_mult* field of *XStandardColormap*. |

**Table 4** *Description of colorinfo array values (Continued)*

| *Value* | *Description* |
|---|---|
| *firstcolor* | First color pixel value. Equivalent to *base_pixel* field of *XStandardColormap*. |
| *colormap* | The colormap that these pixel values are allocated in. |
| *actual* | The upper limit of additional RGB colors, as in the *actual* argument to **XDPSCreateContext**. |

**Errors: undefined**

**currentXoffset** – **currentXoffset** *x y*

The **currentXoffset** operator returns the *x* and *y* coordinates representing the offset from the origin of the *drawable* to the device space origin for the current context. This operator returns a subset of the variables returned by **currentXgcdrawable**. Its result values can be input to **setXoffset**.

**Errors: undefined**

**setXgcdrawable** *gc drawable x y* **setXgcdrawable** –

The **setXgcdrawable** operator sets the X *gc*, *drawable*, and offset from the origin of the *drawable* to the device space origin for the current context. The specified values override any existing values.

The *gc* operand is a *GContext* identifier, not a *GC* pointer. Use **XGContextFromGC** to extract a *GContext* from a *GC*.

To temporarily change the values specified for **setXgcdrawable**, execute **gsave** before the operator and follow it with **grestore**.

**Errors: rangecheck, stackunderflow, typecheck, undefined**

**setXgcdrawablecolor** *gc drawable x y colorinfo* **setXgcdrawablecolor** –

The **setXgcdrawablecolor** operator changes *gc*, *drawable*, *offset*, and *colorinfo* for the context. The *colorinfo* argument is described under **currentXgcdrawablecolor**.

The *gc* operand is a *GContext* identifier, not a *GC* pointer. Use **XGContextFromGC** to extract a *GContext* from a *GC*.

To temporarily change the values specified for **setXgcdrawablecolor**, execute **gsave** before the operator and follow it with **grestore**.

**Errors: rangecheck, stackunderflow, typecheck, undefined**

**setXoffset** *x y* **setXoffset** –

The **setXoffset** operator sets the default origin for the user space of the current context. This operator is a subset of **setXgcdrawable**.

**Errors: stackunderflow, undefined**

**setXrgbactual** *red green blue* **setXrgbactual** *bool*

The **setXrgbactual** operator attempts to allocate a new entry in the context's colormap. It takes three floating-point numbers between 0.0 and 1.0 to specify the RGB color, as with **setrgbcolor**. The operator returns *true* if the color was successfully allocated in the colormap; it returns *false* if the color cannot be allocated or if an error occurs. If the operator returns *true*, future requests for the specified color will be rendered using the allocated colormap entry.

Executing **setXrgbactual** is a way of ensuring that the color you request is actually allocated, not dithered. Colors specified by **setXrgbactual** do not count against the number of *actual* colors that are allocated automatically; see "Using XDPSCreateContext" in 3.2, "Creating a Context." **setXrgbactual** may be called even if the context was created with *actual* set to zero.

**setXrgbactual** does not change the graphics state in any way; to paint with the specified color, execute **setrgbcolor**.

**Errors: stackunderflow, typecheck, undefined**

## 7.1 Single-Operator Procedures

*Client Library Reference Manual* explains and lists a number of single-operator procedures in section 9, "Single-Operator Procedures." The X Window System implementation of the Display PostScript system provides some additional procedures for the X-specific PostScript operators.

The procedure declarations listed below can be found in <*DPS/dpsops.h*>. <*DPS/psops.h*> contains the analogous definitions without the *ctxt* argument.

*Note: Some early releases of the Display PostScript system did not include these operators.*

**Example 14** *Procedure declarations for X-specific PostScript operators*

*C language code:*

```
extern void DPSclientsync( /* DPSContext ctxt; */ );
extern void DPScurrentXgcdrawable( /* DPSContext ctxt;
        int *gc, *draw, *x, *y; */ );
extern void DPScurrentXgcdrawablecolor( /* DPSContext ctxt;
        int *gc, *draw, *x, *y, colorInfo[ ]; */ );
extern void DPScurrentXoffset( /* DPSContext ctxt;
        int *x, *y; */ );
```

```
extern void DPSsetXgcdrawable( /* DPSContext ctxt;
        int gc, draw, x, y; */ );
extern void DPSsetXgcdrawablecolor( /* DPSContext ctxt;
        int gc, draw, x, y, colorInfo[ ]; */ );
extern void DPSsetXoffset( /* DPSContext ctxt;
        int x, y; */ );
extern void DPSsetXrgbactual( /* DPSContext ctxt;
        float r, g, b; int *success; */ );
```