DISPLAY POSTSCRIPT®

S Y S T E M

X WINDOW SYSTEM PROGRAMMER'S SUPPLEMENT to the Client Library Reference Manual

ADOBE SYSTEMS

I N C O R P O R A T E D

X Window System Programmer's Supplement to the Client Library Reference Manual

January 23, 1990

Written by Amy Davidson.

1 ABOUT THIS MANUAL

This manual contains information about the Client Library interface to the Display PostScript® system implemented as an extension to the X Window System™. We sometimes refer to this extension as DPS/X. DPS/X is the application programmer's means of displaying text and graphics on a screen using the PostScript® language.

1.1 DOCUMENTATION

The system-independent interface for DPS/X is documented in Adobe's *Client Library Reference Manual*. Only extensions to the interface are discussed here. The *dpsXclient.h* header file includes both system-independent and X system-specific procedures.

Before reading this manual, you should be familiar with the contents of the manuals listed below. This manual also assumes familiarity with the X Window System.

If you're new to the PostScript language, you should first read the following manuals:

- *PostScript Language Reference Manual*
- *PostScript Language Tutorial and Cookbook*
- *PostScript Language Program Design*

Once you're acquainted with the PostScript language, read the following manuals:

- *PostScript Language Extensions for the Display PostScript System*
- *pswrap Reference Manual*
- *Client Library Reference Manual*
- *PostScript Language Color Extensions*

1.2 WHAT THIS MANUAL CONTAINS

Section 2 briefly introduces the Display PostScript system extension to the X Window System.

Section 3 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; and user object indices.

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

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

Section 6 describes the X-specific data and procedures found in the *dpsXclient.h* header file.

Section 7 describes the X-specific PostScript operators provided for the Display PostScript extension to X.

Appendix A lists changes to the manual since the previous version.

Appendix B provides a workaround for cases where X lower-level software does not permit the normal Client Library error-handling mechanisms.

1.3 TYPOGRAPHICAL CONVENTIONS

The typographical conventions used in this manual are as follows:

Item	*Example of Typographical Style*
file	*dpsXclient.h*
variable, typedef, code fragment	‘cid’, ‘drawable’, ‘ctxt’, ‘x’, ‘y’, ‘DPSContextRec’, ‘XStandardColormap’, ‘enableMask = PSFROZENMASK | PSZOMBIEMASK;’
procedure	*XDPSCreateSimpleContext*
PostScript operator	**currentXgcdrawable**
new term or emphasis	‘‘*Protocol errors* are generated when....’’

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* 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 Section 3.3.1), event handling (see Section 4.8), and resource management (see Section 4.7).
- *The PostScript interpreter* allows an application to execute PostScript language code.
- *Wrapped procedures* allow PostScript language programs to be embedded in an application as C-callable procedures.

3 BASIC FACILITIES

The *Client Library Reference Manual* introduces the facilities needed to write a simple application program for the Display PostScript system. This section 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 DPS/X 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 DPS/X imaging operations, and an X 'GC' out of which certain fields are used by DPS/X. 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 DPS/X, a context (as described in the *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.

A '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’ is automatically created for use by the client (except for forked contexts; see Section 4.5). 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 Section 4.

3.2.1 Using XDPSCreateSimpleContext

To create a context using the default colormap, call *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;
```

The *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 Section 7.

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 location, pass zero for ‘x’ and the height of the window in pixels for ‘y’. See the discussion of coordinate systems in Section 3.3.1.

The other arguments to *XDPSCreateSimpleContext* are described fully in the *Client Library Reference Manual*. To summarize: ‘textProc’ is a call-back procedure that handles text output from the context, ‘errorProc’ is a call-back 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 125 cells for the color cube, representing a 5x5x5 RGB cube. (The gray ramp typically uses the five grays that form the diagonal of the cube.) *XDPSCreateSimpleContext* checks the root window for the RGB_DEFAULT_MAP property. If the property exists, the color cells it specifies are used for the context's color cube. If the property does not exist, color cells are allocated and the property is defined. The allocated cells are typically treated as ‘‘read-only retained’’ so that other DPS/X clients may share the allocated colors. 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 is allocated from the default colormap.

3.2.2 Using XDPSCreateContext

To create a context with specific color information, call *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 *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 setting the colormap as an attribute of the window (using *XSetWindowColormap*).

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 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 (non-dithered) colors. If the application knows which colors it is going to 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 less. 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 **setrgbcolor**, 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.

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 allows the end user to specify only a few colors — the foreground and background colors of a performance meter, for example — should probably use *XDPSCreateContext* and set 'actual' to the number of colors that can be requested by the end user.

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.

3.3 EXECUTION

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

3.3.1 Coordinate Systems

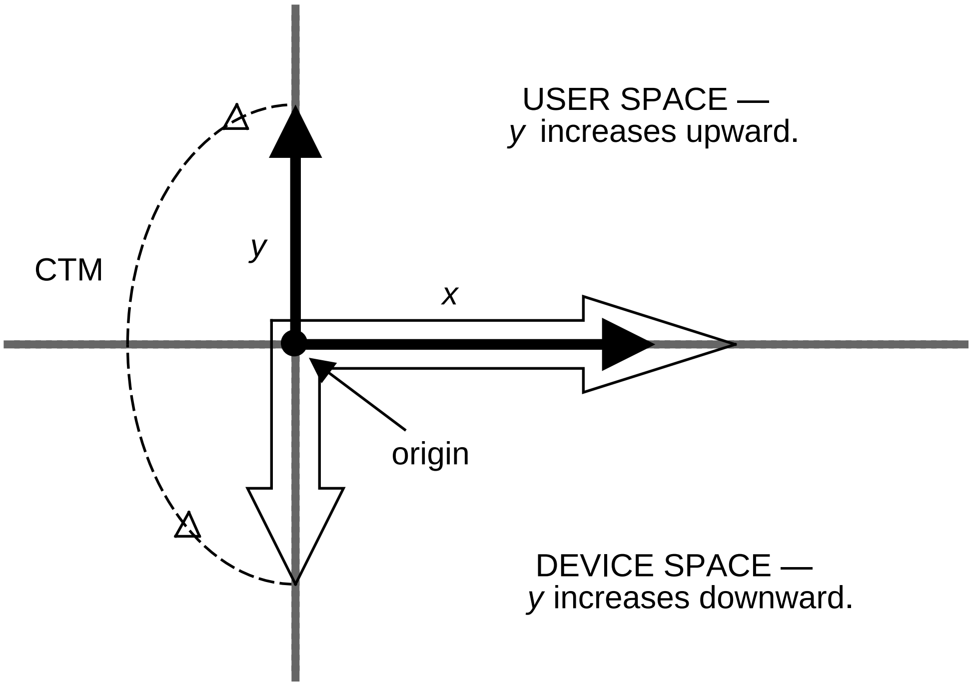
The application must use user space coordinates when communicating with the PostScript interpreter and X coordinates when communicating with other parts of the X Window System. Therefore coordinate conversions may be necessary. This section describes:

- 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.

The *PostScript Language Reference Manual* 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 DPS/X, 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 Extensions for the Display PostScript System*, pixel boundaries fall on integer coordinates in device space. A pixel is a half-open region, meaning that it includes half of 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.

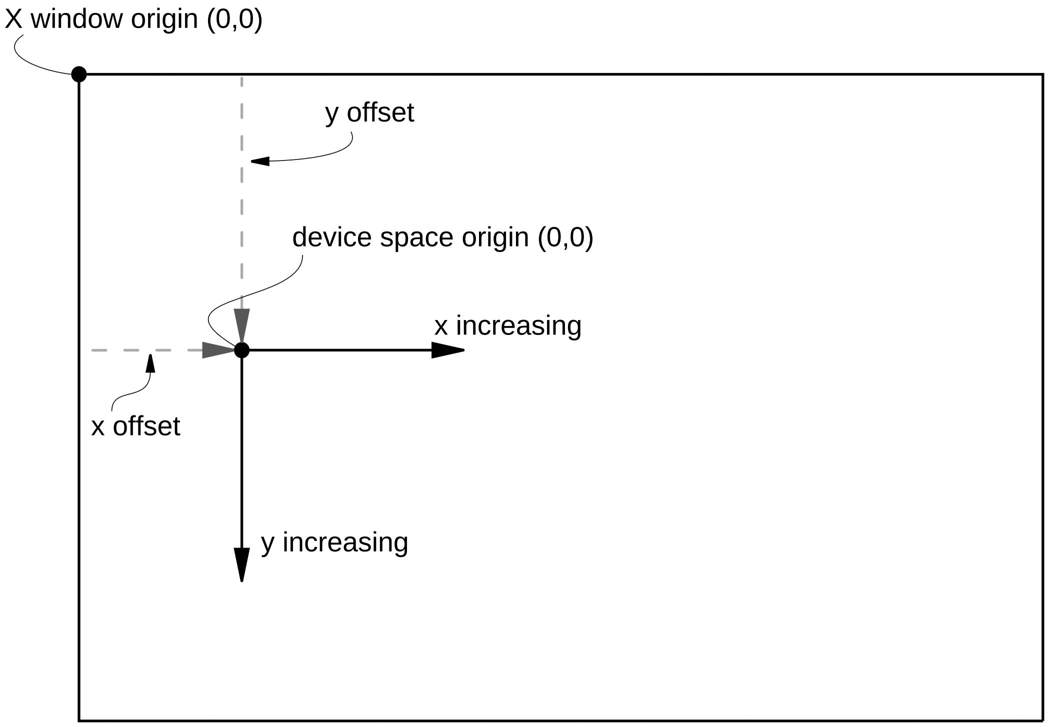
Similarly, to convert X coordinates to user space coordinates:

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 Section 5.3 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 Section 3.2); 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 will be 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, using *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.

See Section 5.3 for tips on how to efficiently convert X coordinates to user space coordinates and vice versa.

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 Section 3.3.4.

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 done 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. A series of mouse events would be handled 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 Section 3.3.1 for more information on coordinate systems.
- 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 Section 4.9 for more information on synchronization.
- X tends to be pixel and plane oriented; graphics 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 considerations 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 DPS/X, use the clipping specified by the X 'GC'.

Exposure events may be handled with a variety of strategies:

- Repaint all graphics for the window.
- Repaint all graphics through composite view clip.
- Repaint selected graphics through composite view clip.

Repainting the entire window is the simplest strategy to implement and is suitable for simple applications:

- 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.

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:

- Create a view clip (see *PostScript Language Extensions for the Display PostScript System*) by converting the coordinates of the list of exposure rectangles to user space coordinates and executing **rectviewclip** with this list.
- Then redisplay all of the text and graphics objects by executing the PostScript language programs that describe them. Only those areas within the 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 Extensions for the Display PostScript System*), 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 use for imaging with the PostScript language, the origin of the device space is also moved according to the bit gravity of the window; see Section 3.3.1 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.

Figure 3 shows the changes to the ‘x’ and ‘y’ offsets for each bit gravity type.

Figure 3 *How Bit Gravity Affects Offsets*

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

Bit Gravity	*x*	*y*
NorthWest	oldX	oldY
North	oldX + wc/2	oldY
NorthEast	oldX + wc	oldY
West	oldX	oldY + hc/2
Center	oldX + wc/2	oldY + wc/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’ offset, use **currentXoffset**.

3.3.5 User Object Indices

The Client Library provides a convenient and efficient way to refer to PostScript language objects. 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 integer (‘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 procedure *DPSNewUserObjectIndex* 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 neither be used as arithmetic values — for example, don’t add 1 to get the next available index — nor stored in a file or other long-term storage.

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

Once a user object has been defined, the application may use 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 Extensions for the Display PostScript System* and the *pswrap Reference Manual* for further discussion of user objects.

Note that it is the responsibility of the application and any run-time 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 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
/* A wrapped procedure that defines an icon dictionary. */

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 or 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
                                    %       and icon dictionary.
                                    %       Note: since we are going
```

```
                                %       to execute the object,
                                %       we can declare it as
                                %       userobject 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 procedure to create and display a new icon */

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();  /* client library routine */
    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

There are two classes of errors that can occur while using DPS/X: 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 DPS/X procedure or wrap. X-specific error codes are discussed in Section 6.1.1 on page 43. See the *Client Library Reference Manual* for a discussion of the standard Display PostScript system 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' did not contain a valid operator and therefore generated an **undefined** error. The error may not be received when *DPSPrintf* returns. It may not even be received 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 the section on synchronization in the *Client Library Reference Manual*.

Example: This code 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 folloed 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.

```
#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);
```

3.4 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 the *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').

4 ADDITIONAL FACILITIES

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

4.1 IDENTIFIERS

DPS/X 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 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, given a 'DPSContext' handle.
- *XDPSXIDFromSpace* returns an X resource ID, given a 'DPSSpace' handle.
- *XDPSContextFromXID* returns a 'DPSContext' handle, given an X resource ID.
- *XDPSSpaceFromXID* returns a 'DPSSpace' handle, given an 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 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 for more information on forked contexts.

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 Section 4.2 for a discussion of zombie contexts.

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 may 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 4.8 for more information about status events.

4.3 BUFFERS

As discussed in the *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 DPS/X requests sent to the server are buffered by Xlib, like any other X requests. *DPSFlushContext* (see the *Client Library Reference Manual*) 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 may be used. Note that wrapped procedures that return results include a **flush** operator at the end of the wrap code.

4.4 ENCODINGS

The *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, the binary object sequence can be converted to some other encoding before it is sent or written.

DPS/X supports the conversions shown in Figure 4:

Figure 4 *Encoding Conversions*

Conversion	*Description*
binary object sequence to ASCII	Makes a binary object sequence readable by humans. The output of wrapped procedures may be inspected and analyzed. Also useful for generating page descriptions to be printed. This is the default setting for text contexts. Execution contexts may also be made to convert binary object sequences to ASCII, but there is little purpose in doing this.
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 may 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 Section 4.7). 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 of the application’s process address space.
binary-encoded tokens to ASCII	Binary-encoded tokens read from an external data source such as a file can be converted to ASCII for human inspection, sent to an intepreter, 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.

Example 1: To cause a text context to generate binary-encoded tokens, call:

```
DPSChangeEncoding(textContext, dps_encodedTokens,
        textContext->nameEncoding);
```

Example 2: To cause an execution context to convert user name indices to user name strings, call:

```
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 Section 4.1). 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.

Warning: 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 on more than one server at the same time. If this is done, the application must process events from each server (display) to which it is connected.

In order to support access to multiple servers, DPS/X 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.

Warning: 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 Extensions for the Display PostScript System*.

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 are 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 clients sharing the context, as specified by the status event mask (see Section 4.8).
- 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 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 applications won’t need this facility.

Status events can be used to perform the following tasks:

- Send code, using flow control, from the application to a context.
- Control the suspension and resumption of execution.
- Synchronize PostScript interpreter execution with X rendering requests.
- Monitor 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 would 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 ‘‘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 may be disabled (see below).

If an application is interested in one or more types of status events, a handler of type ‘XDPSStatusProc’ must be defined. Two arguments will be passed to the call-back 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 *never* wants 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*.

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.

Example: An application currently has ‘PSNEEDSINPUT’ and ‘PSRUNNING’ enabled and all other types disabled. It now wants to be notified of every transition to ‘PSFROZEN’ and ‘PSZOMBIE’ and only the next transition to ‘PSNEEDSINPUT’. The masks would be contructed as follows:

```
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 Section 4.9 for details on how the ‘PSFROZEN’ status event can be used.

4.9 SYNCHRONIZATION

As discussed in Section 3.3.2, 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 the *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.

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.

4.9.1 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 Systems 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.

4.9.2 Freezing

Freezing a context is appropriate if the application has insufficient knowledge of 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, as shown in the following example.

Example: The application has defined **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 Section 4.8 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 for to help you program applications that use the Display PostScript system extension to the X Window System.

5.1 DON'T USE XIFEVENT

Don't call *XIfEvent* in your application. 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 may not apply to future implementations of Xlib.

Warning: 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 *dpsXclient.h* header file when compiling DPS/X applications. This header file includes the required header files described in the *Client Library Reference Manual*, *dpsclient.h* and *dpsfriends.h*.

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

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

Include *dpsexcept.h* if your application uses exception handling as defined in the *Client Library Reference Manual*.

5.3 COORDINATE CONVERSIONS

The code examples in this section demonstrate an efficient method of doing coordinate conversions. (For an introduction to coordinate system issues, see Section 3.3.1.)

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 following wrapped procedure will return these values:

```
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:

```
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 following calculations:

```
#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 following calculations:

```
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 error. Incorrect coordinates may be computed in either direction.

5.4 FONTS

The **filenameforall** operator can be used to obtain a list of the fonts available to the server. See *PostScript Language Extensions for the Display PostScript System* 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. Like 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.

5.5 PORTABILITY ISSUES

Portability issues may arise under any of the following situations:

- Converting an existing X application to use the DPS/X extension.
- Porting a non-X window system application to use the DPS/X extension.
- Writing a portable application that uses the DPS/X extension.

A major factor in portability is device independence. The DPS/X extension enhances the device independence of X applications by providing flexibility with respect to color, resolution, and fonts.

5.5.1 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.

DPS/X color rendering is device independent. Here's how DPS/X 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 non-dithered 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 DPS/X.

- 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.

5.5.2 Resolution

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

In DPS/X, 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.

5.5.3 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, DPS/X generates it for you.
- The typeface can be rendered in any rotation or two-dimensional transformation.

6 X-SPECIFIC DATA AND PROCEDURES

This section describes the system-specific data types and procedures for DPS/X.

6.1 DATA STRUCTURES

Data structures defined in the *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 the *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).

6.1.2 Status Event Masks

The status event types supported in DPS/X are shown in Figure 5. 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.

Figure 5 *Status Events*

Status Event	*Mask Value*	*Status 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 4.8.

6.1.3 Types and Global Variables

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* and Section 3.3.5.

XDPSStatusProc

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

This is a procedure type for defining the call-back 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* and Section 6.1.2.

6.2 PROCEDURES

This section contains descriptions of the system-specific procedures in the *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. See the *Client Library Reference Manual* for definitions of 'DPSNameEncoding' and 'DPSProgramEncoding'. Supported conversions are described in Figure 4 on page 28.

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 DPS/X. 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 the *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 the *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 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 Section 3.3.5 for more information on user object indices.

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.

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, 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;
```

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 Section 3.2 for additional information on creating a context.

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 Section 3.2 for additional information on creating a context.

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 Sections 4.8 and 6.1.2.

XDPSRegisterStatusProc

```
void 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.

XDPSSetStatusMask

```
void XDPSSetStatusMask(ctxt, enableMask, disableMask, nextMask);
  DPSContext ctxt;
  unsigned long enableMask, disableMask, nextMask;
```

XDPSSetStatusMask sets the status mask for the context:

- '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 4.8 and 6.1.2.

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 POSTSCRIPT OPERATORS

This section describes the X-specific 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 4.9.2.

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**.

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 'colorinfo' array, described in Figure 6, has the following form:

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

Figure 6 *The 'colorinfo' Array*

Value	*Description*
'maxgrays'	Maximum number of gray values. Equivalent to 'red_max' field of 'XStandardColormap' for 'GrayScale' colormaps.
'graymult'	Scale factor to compute gray pixel. Equivalent to 'red_mult' field of 'XStandardColormap' for 'GrayScale' colormaps.
'firstgray'	First gray pixel value. Equivalent to 'base_pixel' field of 'XStandardColormap' for 'GrayScale' colormaps.
'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'.
'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.

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**.

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.

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 Section 3.2.2. **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

A CHANGES SINCE LAST PUBLICATION OF THIS DOCUMENT

Changes to the *X Window System Programmer's Supplement to the Client Library Reference Manual* from the document dated August 17, 1989, are noted in the paragraphs below.

An example error handler program for advanced error handling has been provided in Appendix B.

The discussion of the X colormap resource has been clarified, including discussions of the use of *XDPSCreateSimpleContext*, the 'actual' parameter in *XDPSCreateContext*, and the **setXrgbactual** operator.

The section on scan conversion has been removed. For information on this topic, please refer to *PostScript Language Extensions for the Display PostScript System*.

Numerous additional amplifications and corrections have been made.

B ADVANCED EXCEPTION HANDLING

This appendix contains an example error handler procedure that can be used when *DPSDefaultErrorProc*, the default error handler described in the ‘‘Example Error Handler’’ appendix of the *Client Library Reference Manual*, is not appropriate. The information in this appendix is not required by most programmers.

Note: In general, you can’t use exception handling with X because lower levels of software, such as Xlib, are not prepared to handle exceptions or to have control taken away from them. Under certain conditions, however, you can work around the limitations of the lower-level software. Note that workarounds may be implementation dependent. The example in this appendix is a workaround designed to be as general as possible.

Here is a brief review of how the default error handler works. The application installs the **resynchhandleerror** error procedure for the context and establishes an exception handler using the mechanism described in the *Client Library Reference Manual*. When a PostScript language error occurs, *DPSDefaultErrorProc* is invoked. It calls *RAISE* to raise an exception, allowing the application’s exception handler to intercept the error and attempt error recovery.

DPSDefaultErrorProc, while sufficient for most DPS/X applications, may not be suitable in cases where all procedures in the call stack must complete execution. The root of the problem is that *DPSDefaultErrorProc* returns control directly to the application’s error handler at the top of the calling stack, bypassing all of the procedures in between, including any Xlib procedures that were called. This can result in loss of Xlib state. Because the default error handler does not allow Xlib procedures to terminate gracefully, it is unsafe for the context’s client-side error procedure to raise exceptions when there are Xlib procedures on the call stack. In this case, a different error-handling mechanism must be used.

The example below shows an instance in which *DPSDefaultErrorProc* cannot safely be used.

```
DURING
  while (fgets(linebuf, LINEBUF_LEN, psFile) != NULL)
    DPSWritePostScript(c, linebuf, strlen(linebuf));
  DPSWaitContext(c);
HANDLER
  if (((DPSContext) Exception.Message == c) &&
      (Exception.Code == dps_err_ps))
    DPSResetContext(c);
END_HANDLER
```

In this example, the code between 'DURING' and 'HANDLER' attempts to read a PostScript language program from the *psFile* stream and pass the text of the program to a context for execution. It sends the entire program and waits until its execution is complete. If an error occurs, the 'HANDLER' clause is invoked. The handler attempts to reset the context and allow it to continue execution.

The error-handling strategy used in this example can fail if the context receives PostScript language code with an error in it. If Xlib processes output from the context while a previous request to the context remains incomplete because of a full X server request buffer, an X protocol error will result. The sequence of events that leads to this error is explained below.

If output from the context includes an error message, the Client Library calls the context's error procedure. Any error procedure that calls *RAISE* to raise exceptions, including *DPSDefaultErrorProc*, will cause all the Xlib stack frames to be unwound before returning control to the application's 'HANDLER' code. Since the Xlib procedure that was composing a request to the context was not allowed to complete, the application's X Display structure is left in an inconsistent state. When the application calls *DPSResetContext*, a ''reset context'' protocol request is sent to the context that reported the error, but the X server interprets this request as part of the data of the previous (incomplete) request. Subsequent messages from the application appear as garbage to the server, which rejects them as protocol errors.

B.1 DEFERRED ERROR HANDLING EXAMPLE

The following example employs a mechanism that buffers errors in a queue, thus deferring them so that the application can handle them synchronously, when it is safe and convenient to do so.

ERRDeferredErrorProc implements the part of the mechanism that buffers errors; the sample program specifies this procedure as the context's error procedure in the call to *XDPSCreateSimpleContext*. *ERRDeferredErrorProc* is called by the Client Library whenever an error is detected.

The sample program sends PostScript language code from an input file to a context for execution. The application's *handling* of errors queued by *ERRDeferredErrorProc* is separated in time from *recognition* of those errors by the Client Library; error handling is deferred until convenient to the application. After each *DPSWritePostScript* call in the 'while' loop, the application calls *ERRErrorsPending* to determine whether execution of any previously sent PostScript language code has resulted in an error. If *ERRErrorsPending* returns *true*, the application calls *ERRProcessErrors* to process the pending error. *ERRProcessErrors* does not dictate a particular way the application should handle errors. It simply provides a mechanism that allows the application to implement its own error-handling scheme by means of a call-back procedure that is called for each error dequeued. In this example, the call-back procedure (*ErrorCallbackProc*) calls *ERRPrintErrorMsg* to display a formatted error message on the standard output. *ErrorCallbackProc* then determines whether the error was 'dps_err_ps'. If so, control is returned to the application, which attempts to reset the context. If the error was not 'dps_err_ps', *ErrorCallbackProc* exits, causing the application to terminate abnormally.

```
/* This program creates a context and passes it PostScript code read from  a
   user-specified file. If a PostScript error occurs, a message is printed  and
   the program continues with the next specified file. Any other errors  result
   in abnormal termination.

   This program is a simplified example that illustrates the use of
   the deferred error-handling mechanism. A "real-world" application of this
   type would probably use a "clientsync / status event" type of synchronization
   rather than DPSWaitContext. */

#include        <stdio.h>
#include        "dpsclient.h"
#include        "dpsexcept.h"
#include        "errprocsample.h"
#ifdef XDPS
#include        "dpsXclient.h"
#endif XDPS

char resyncString[] = "resyncstart\n";
char initString[] = "clear cleardictstack\n";

/* Forward declarations */

void Error();
int ErrorCallbackProc();

main()
  {
#define LINEBUF_LEN 512
  char linebuf[LINEBUF_LEN];
  int len;
  DPSContext c;
  Display *dpy;
  FILE *psFile;

  dpy = XOpenDisplay(NULL);
  if (dpy == NULL)
    Error("Can't open display");

  c = XDPSCreateSimpleContext(dpy, None, None, 0, 0,
    DPSDefaultTextBackstop, ERRDeferredErrorProc, NULL);
  if (c == NULL)
    Error("Can't create DPS context");

  /* Set up context so it can recover after an error */
  DPSWritePostScript(c, resyncString, strlen(resyncString));

  while (1) {
    printf("File containing PostScript Code: ");
    scanf("%s", linebuf);
    if ((psFile = fopen(linebuf, "r")) == NULL)
```

```
      Error("Unable to open input file");
    DPSWritePostScript(c, initString, strlen(initString));
    while (fgets(linebuf, LINEBUF_LEN, psFile) != NULL) {
      len = strlen(linebuf);
      linebuf[len] = '\n'; linebuf[len+1] = '\0';
      DPSWritePostScript(c, linebuf, len + 1);
      if (ERRErrorsPending())
        break;
      }
    if (! ERRErrorsPending())
      /* Wait for context to complete if no errors yet */
      DPSWaitContext(c);
      /* Test for errors again -- they may have been queued by DPSWaitContext  */
    if (ERRErrorsPending()) {
      (void) ERRProcessErrors(ErrorCallbackProc,
                              (unsigned long) dps_err_ps);
      DPSResetContext(c);
      }
    }

  }

/* Print an error message and exit if the error was not
   the one expected */

int ErrorCallbackProc(err, expected)
  ERRQueueEntry *err;
  unsigned long expected;
  {
  ERRPrintErrorMsg(err);
  if ((DPSErrorCode) expected != err->errorCode)
    exit(2);
  return(0);
  }

void Error(msg)
  char  *msg;
  {
  printf("sample: %s\n", msg);
  exit(2);
  }
```

The procedures and data structures whose names start with ‘‘ERR’’ are part of the deferred error-handling package that is described below.

B.2 ERROR HANDLER INTERFACE

The header file described in this section, *errorproc.h*, defines the procedures and data structures that comprise a deferred error-handling mechanism.

A listing of the *errorproc.h* header file follows.

```
/* errorproc.h */

/* Structure containing all relevant information about a Client
   Library-generated error. This structure serves as a header
   for a potentially larger structure; some errors require
   additional information for optimal processing. In those cases,
   the 'arg1' element points to the additional information,
   which is appended to the entry header. See the Client Library
   Reference Manual for information on the structure of such
   additional information. */

typedef struct _t_ERRQueueEntry {
  struct _t_ERRQueueEntry *next;
  DPSContext     ctxt;
  DPSErrorCode   errorCode;
  long unsigned int arg1, arg2;
  } ERRQueueEntry;

/* Queue of deferred error entries */

extern ERRQueueEntry ERRQueueHead;

typedef int (*ERRCallBackProc)(/* ERRQueueEntry *error;
                                  unsigned long userArg */);

extern void ERRDeferredErrorProc(/* DPSContext ctxt; DPSErrorCode errorCode;
                                    long unsigned int arg1, arg2; */);

extern int ERRPrintErrorMsg(/* ERRQueueEntry *error; */);

extern int ERRProcessErrors(/* ERRCallBackProc proc; long int procArg; */);

#define ERRErrorsPending() (ERRQueueHead.next != NULL)
```

The header file is described in the paragraphs that follow.

'ERRQueueEntry' is a structure that contains information about errors of type 'DPSErrorCode'. This structure serves as a header for a potentially larger structure; some errors require additional information for optimal processing. In those cases, the 'arg1' element points to the additional information, which is appended to the entry header. See *DPSErrorProc* in the *Client Library Reference Manual* for information on the structure of such additional information.

'ERRQueueHead' is the head of a queue of deferred error entries of type 'ERRQueueEntry'.

'ERRErrorsPending' is a macro that tests whether any errors need processing. It yields *true* if *ERRProcessErrors* should be called.

ERRCallBackProc is the call-back procedure passed to *ERRProcessErrors*. The call-back procedure is passed an 'ERRQueueEntry' pointer and an optional argument supplied by the caller of *ERRProcessErrors*. This argument is uninterpreted by *ERRProcessErrors*. The call-back procedure returns a boolean indicating whether *ERRProcessErrors* is to continue processing pending error entries. If it returns *true*, processing continues.

ERRDeferredErrorProc is the 'DPSErrorProc' to be specified as the error handler for a context. Unlike *DPSDefaultErrorProc*, this procedure does not call *RAISE* to raise an exception. Instead, it encapsulates the relevant error information in an 'ERRQueueEntry' and puts this error structure on the queue of error entries waiting to be processed by *ERRProcessErrors*.

ERRPrintErrorMsg is the default 'ERRCallBackProc' called from *ERRProcessErrors*. It formats an error message from the information in the error queue entry passed to it. The error message is then passed to the application's text backstop procedure. *ERRPrintErrorMsg* always returns *true*, allowing *ERRProcessErrors* to continue to handle pending error entries.

ERRProcessErrors is called by the application when it is ready to handle any pending errors queued by *ERRDeferredErrorProc*. It removes as many pending error entries from the error queue as is allowed by the call-back procedure; the actual processing of each error entry is left to the call-back procedure passed as an argument to *ERRProcessErrors*. An argument to be passed to the call-back procedure is also provided, allowing the application to specify the disposition of an error without having to manage the error entry queue. If *ERRProcessErrors* is called with a NULL call-back procedure, *ERRPrintErrorMsg* is substituted. In other words, the default action is to print an error message. If the call-back procedure returns *false*, *ERRProcessErrors* returns immediately to the caller, potentially leaving unprocessed entries still on the error queue. *ERRProcessErrors* returns *true* if any errors were processed; it returns *false* if no error entries were found on the queue.

B.3 ERROR HANDLER IMPLEMENTATION

A sample implementation of the previously defined error-handling mechanism follows. The error handler procedure below is similar to the one provided in the ‘‘Example Error Handler’’ appendix of the *Client Library Reference Manual*, except that this one doesn’t call *RAISE*.

```
/* errprocsample.c */

#include        <stdio.h>
#include        <strings.h>
#include        <malloc.h>
#include        "dpsclient.h"
#include        "dpsexcept.h"
#include        "errprocsample.h"
#include        "dpsXclient.h"

/*      ===== PUBLIC VARIABLES =====     */

/* Queue of error entries is headed by a dummy entry that
   acts as an anchor */

ERRQueueEntry ERRQueueHead = { NULL };

/*      ===== PUBLIC PROCEDURES =====    */

void ERRDeferredErrorProc(ctxt, errorCode, arg1, arg2)
  DPSContext ctxt;
  DPSErrorCode errorCode;
  long unsigned int arg1, arg2;
  {
  ERRQueueEntry *entry, *e;
  int objLen = 0;

  /* Some error codes have extra data associated with them to
     help identify the problem. In each case, 'arg1' points to
     this extra data. Determine the byte length of the data
     (sometimes 'arg2' but not always). */
```

```
  switch (errorCode) {
    case dps_err_ps:
      objLen = ((DPSBinObj) arg1)->length;
      break;
    case dps_err_nameTooLong:
      objLen = arg2;
      break;
    case dps_err_resultTagCheck:
      objLen = arg2;
      break;
    case dps_err_resultTypeCheck:
      objLen = sizeof(DPSBinObjRec);
      break;
    default:;
    }

  /* Allocate a queue entry large enough to hold all the normal
     error stuff plus any auxilary data associated with the
     error.  Fill in the generic entries. If extra data exists,
     copy it and make the 'arg1' element in the entry header
     point to the newly copied data. */

  entry = (ERRQueueEntry *) malloc(sizeof(ERRQueueEntry) + objLen);
  if (entry == NULL)
    exit(2);

  entry->ctxt = ctxt;
  entry->errorCode = errorCode;
  entry->arg2 = arg2;
  if (objLen > 0) {
    char *to = (char *) entry + sizeof(ERRQueueEntry);
    bcopy((char *) arg1, to, objLen);
    arg1 = (long unsigned int) to;
    }
  entry->arg1 = arg1;

  /* Enqueue the new entry */

  for (e = &ERRQueueHead; e->next != NULL; e = e->next);
  e->next = entry;
  entry->next = NULL;

  } /* ERRDeferredErrorProc */

int ERRPrintErrorMsg(error)
  ERRQueueEntry *error;
  {
  DPSContext ctxt = error->ctxt;
  DPSErrorCode errorCode = error->errorCode;
  long unsigned int arg1 = error->arg1;
  long unsigned int arg2 = error->arg2;
```

```
char m[100], str1[100], str2[100];
char *prefix = "%%[ Error: ";
char *suffix = " ]%%\n";

DPSTextProc textProc = DPSGetCurrentTextBackstop();

if (!textProc)
  return(1);

switch (errorCode) {
  case dps_err_ps: {
    char *buf = (char *)arg1;
    DPSBinObj ary = (DPSBinObj) (buf+DPS_HEADER_SIZE);
    DPSBinObj elements;
    char *error, *errorName;
    int errorCount, errorNameCount;

    Assert((ary->attributedType & 0x7f) == DPS_ARRAY);
    Assert(ary->length == 4);

    elements = (DPSBinObj)(((char *) ary) + ary->val.arrayVal);

    errorName = (char *)(((char *) ary) + elements[1].val.nameVal);
    errorNameCount = elements[1].length;
    (void) strncpy(str1, errorName, errorNameCount);
    str1[errorNameCount] = '\0';

    error = (char *)(((char *) ary) + elements[2].val.nameVal);
    errorCount = elements[2].length;
    (void) strncpy(str2, error, errorCount);
    str2[errorCount] = '\0';

    (void) sprintf(m, "%s; OffendingCommand: %s", str1, str2);
    break;
    }
  case dps_err_nameTooLong:
    (void) strncpy(str1, (char *) arg1, (int) arg2);
    str1[arg2] = '\0';
    (void) sprintf(m, "User name too long; Name: %s", str1);
    break;
  case dps_err_invalidContext:
    (void) sprintf(m, "Invalid context: 0x%lx", arg1);
    break;
  case dps_err_resultTagCheck: {
    unsigned char tag = *((unsigned char *) arg1+1);
    (void) sprintf(m, "Unexpected wrap result tag: %d", tag);
    break;
    }
  case dps_err_resultTypeCheck: {
    unsigned char tag = *((unsigned char *) arg1+1);
    (void) sprintf(m, "Unexpected wrap result type; tag: %d",  tag);
    break;
```

```
      }
    case dps_err_invalidAccess:
      (void) sprintf(m, "Invalid context access.");
      break;
    case dps_err_encodingCheck:
      (void) sprintf(m, "Invalid name/program encoding: %d/%d.",
                     (int) arg1, (int) arg2);
      break;
    case dps_err_closedDisplay:
      (void) sprintf(m, "Broken display connection %d.", (int) arg1);
      break;
    case dps_err_deadContext:
      (void) sprintf(m, "Dead context 0x%lx.", arg1);
      break;
    default:
      (void) sprintf(m, "Unknown error code: %d, context: %lx, arg 1, 2: %lx %lx",
                     errorCode, ctxt, arg1, arg2);
    }

  (*textProc)(ctxt, prefix, strlen(prefix));
  (*textProc)(ctxt, m, strlen(m));
  (*textProc)(ctxt, suffix, strlen(suffix));
  return(1);

  } /* ERRPrintErrorMsg */

int ERRProcessErrors(proc, procArg)
  int (*proc)();
  long int procArg;
  {
  ERRQueueEntry *error;
  int foundError = 0;

  if (proc == NULL)
    proc = ERRPrintErrorMsg;

  error = ERRQueueHead.next;
  while (error) {
    int cont;
    foundError = 1;
    ERRQueueHead.next = error->next;
    cont = (*proc)(error, procArg);
    free((char *) error);
    if (!cont)
      break;
    error = ERRQueueHead.next;
    }

  return(foundError);
  } /* ERRProcessErrors */
```

Index

Contents

List of Figures