Digital UNIX

Developing Applications for the Display PostScript System

Order Number: AA-Q15WB-TE

March 1996

Product Version: Digital UNIX Version 4.0 or higher

This manual introduces the Display PostScript system extension of Digital's Worksystem Software and describes how to develop applications that use this extension.

Digital Equipment Corporation
Maynard, Massachusetts

Contents

Index

Examples

Figures

Tables

About This Manual

This manual introduces the Display PostScript system extension of Digital's Worksystem Software (WS). This manual describes the WS-specific concepts, tasks, and facts that programmers must know to write Display PostScript applications for WS.

This manual supplements Display PostScript system documentation written by Adobe Systems, Inc.

Audience

This manual is intended for experienced Worksystem application programmers who are familiar with C language programming; it assumes that the reader is familiar with the PostScript language.

Organization

This manual comprises six chapters:

Chapter 1	Introduces the Display PostScript system and lists the capabilities it adds to WS.
Chapter 2	Describes the main components that make up the Display PostScript system and summarizes key concepts.
Chapter 3	Explains how to start writing applications for the Display PostScript system and presents a simple example program.
Chapter 4	Presents advanced concepts and tasks.
Chapter 5	Describes the WS-specific header file of the Display PostScript system Client Library and describes each WS-specific Client Library routine.
Chapter 6	Describes X-specific operators provided by WS.

Related Documents

The following books, published by Addison-Wesley Publishing Company, Inc., help you understand the PostScript language:

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

The printed version of the Digital UNIX documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General users	G	Blue
System and network administrators	S	Red
Programmers	P	Purple
Device driver writers	D	Orange
Reference page users	R	Green

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the Digital UNIX documentation set.

Reader's Comments

Digital welcomes any comments and suggestions you have on this and other Digital UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-881-0120 Attn: UEG Publications, ZK03-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on line in the following location:

  `/usr/doc/readers_comment.txt`

- Mail:

  Digital Equipment Corporation
  UEG Publications Manager
  ZK03-3/Y32
  110 Spit Brook Road

Nashua, NH 03062-9987

A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Digital UNIX that you are using.
- If known, the type of processor that is running the Digital UNIX software.

The Digital UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate Digital technical support office. Information provided with the software media explains how to send problem reports to Digital.

Conventions

The following typographical conventions are used in this manual:

. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.
`symbol`	In text and examples, all directory names, file names, routine names, PostScript operator names, and code samples appear in this typeface.

Introduction to the Display PostScript System 1

To display or print graphics, an application must have an **imaging model**, a set of rules for describing pictures and text. One of the most popular imaging models is that of the **PostScript page-description language**, from Adobe Systems, Inc. Originally developed for hardcopy output devices, such as laser printers, the PostScript language imaging model has been adapted for bitmap displays through Adobe's Display PostScript system.

Digital's implements the imaging models of the X Window System and the Display PostScript system. WS applications can mix X and PostScript language imaging calls, even within a single window, using a single network connection to an X server. This manual introduces the Display PostScript system and shows how to develop WS applications that use it.

1.1 Overview of the Display PostScript System

The Display PostScript system is software that extends the PostScript imaging model to bitmap display systems. With the Display PostScript system, you can design and write applications in a general-purpose language like C, yet describe their images and text using the device-independent PostScript imaging model.

1.2 PostScript Language Imaging Capabilities

You are probably familiar with the capabilities of X imaging. The following capabilities are found in PostScript language imaging but not in X imaging:

- Coordinate system that can be moved, rotated, and scaled
- Bezier curves
- Device-independent color model with dithered (approximated) colors
- Text that can be scaled and rotated
- Image operators for scanned images
  (scaling, rotating, transformations, gray-scale manipulation)

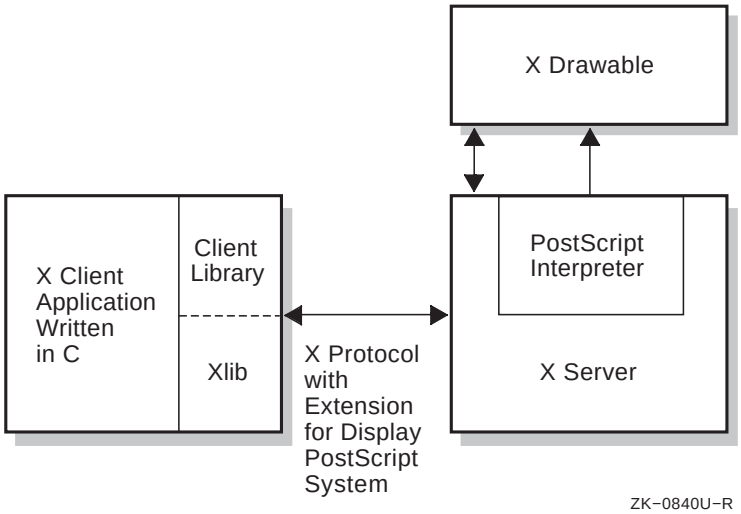
1.3 Display PostScript System in WS

The Display PostScript system is a system-independent client/server architecture that can be implemented on a variety of windowing systems. In this architecture, the **server** consists mainly of a PostScript interpreter, which executes PostScript language code that displays images on a user's screen. The **client** is an application that communicates with the server through a set of routines known as the Client Library.

WS implements the Display PostScript system as an extension to the X Window System, on which WS is based. The Display PostScript system server is an extension to the X server; the Client Library is an extension to Xlib. The Display PostScript system extension of WS lets a C language application display images in an X window by calling functions that send PostScript language code.

Figure 1-1 shows the WS implementation of the Display PostScript system. (For brevity, this manual often refers to this implementation as **XDPS**.) For more information about how WS implements the Display PostScript system, see Chapter 2.

Figure 1-1: Display PostScript System as Implemented in WS



To understand and use the Display PostScript system in WS, you must be familiar with these subjects:

- The operating system
- The C programming language
- WS programming
- The PostScript language
- The system-independent aspects of the Display PostScript system
- The WS-specific aspects of the Display PostScript system

This manual describes mainly the WS-specific aspects of the Display PostScript system.

Components and Concepts 2

Even for WS programmers who are familiar with the PostScript language, the Display PostScript system for WS introduces new concepts. For instance, some familiar terms such as ‘‘client,’’ ‘‘context,’’ and ‘‘state’’ take on new meanings.

This chapter summarizes components and concepts of the Display PostScript system. Some of these topics are **system-independent**; others are **system-specific**. In this manual, the term **system-independent** refers to components and concepts found in all implementations of the Display PostScript system. **System-specific** refers to components found in only some implementations of the Display PostScript system and whose exact names and capabilities vary among implementations.

The Display PostScript system for WS is the **system**being described in this manual, so **WS-specific** and **system-specific** mean the same thing here. Note that some WS-specific components are also **X-specific**: they exist only in X-based implementations of the Display PostScript system.

This chapter emphasizes mainly WS-specific concepts and components.

2.1 Components

The Display PostScript system consists of three main components:

- PostScript interpreter
- Client Library
- The `pswrap` translation program

In WS, the PostScript interpreter resides on the X server; the Client Library is linked with the X client. The client and server can reside on the same workstation or on different workstations connected by a network.

2.1.1 PostScript Interpreter

In WS, the PostScript interpreter is an X server extension that executes PostScript language code sent from applications. The interpreter implements the full PostScript language, including operators for color and display. You can imagine the PostScript interpreter as a PostScript printer. Unlike a printer, however, the interpreter can concurrently execute several jobs.

2.1.2 Client Library

The Client Library is the set of C language routines through which applications communicate with the PostScript interpreter. The Client Library routines communicate with the PostScript interpreter by calling Xlib routines and low-level Display PostScript system routines implemented as extensions to Xlib. Note that, although there is currently no toolkit interface to Display PostScript system itself, applications that use the system can use toolkit interfaces to X as usual.

Note

Except where noted, the term **application** means a WS application program that uses the Display PostScript system.

The Client Library routines and data structures that make up the application programming interface to the Display PostScript system are defined in six header files. Only one of these six files is X-specific: `dpsXclient.h`. (For more information about `dpsXclient.h`, see Chapter 5.)

2.1.3 The Translation Program: pswrap

The `pswrap` translator is a program that converts procedures written in the PostScript language into routines that can be called from applications written in C. The converted routines are called wrapped procedures, or **wraps**. In WS, `pswrap` is installed in the directory `/usr/bin`.

A special set of ready-to-call wraps is included in the Client Library; most of these wraps send a single PostScript operator. These single-operator wrapped procedures are called **singleops**. (For more information on singleops, see Chapter 5.

2.2 Concepts

Before you can write an application that uses the Display PostScript system, you should understand a few essential concepts. The following section introduces those concepts.

2.2.1 Contexts

The term **context** is familiar to X programmers. In the Display PostScript system, however, a context is not an X Graphic Context (GC). Instead, a context is a destination to which an application sends PostScript language code. A PostScript context is either an **execution context** or a **text context**. Except where noted otherwise, the term **context** refers to a PostScript context; the X Graphic Context is referred to as the GC or as the X Graphic

Context. Also, except where noted, the term **context** includes both execution contexts and text contexts.

2.2.1.1 Execution Context

An execution context is a destination that executes PostScript language code sent from an application. In WS, that destination is the PostScript interpreter of the X server. Just as the interpreter is like a PostScript printer, an execution context is like a print job.

In WS, a PostScript execution context is usually associated with an X display, an X drawable, and a GC. The PostScript execution context uses only the following fields of the GC:

```
clip_mask
clip_x_origin
clip_y_origin
plane_mask
subwindow_mode
```

The Display PostScript system in WS treats the X drawable and GC as part of the PostScript **graphics state**, a data structure that defines how PostScript operators execute.

2.2.1.2 Text Context

A text context is a destination that does not execute the PostScript language input it receives from an application. For example, the destination might be a text file or a WS stream, such as `stdout`. The destination is specified in the text-handling routine that the application assigns when creating the text context.

Sending PostScript language input to a text context provides a way to get a printable copy of input that would otherwise be sent to an execution context. This capability is particularly useful in debugging applications.

Note

In this manual, except where noted otherwise, the term **input** means input to a context on the server, not to an application on the client. Conversely, **output** means output from a context.

2.2.2 Context Record and DPSContext Handle

All contexts reside on the server. However, on the client, each context is represented by a **context record**, whose data type is `DPSContextRec`. The `DPSContextRec` stores the attributes of the context, for instance, the pointer to its error-handling routine.

Applications do not access the `DPSContextRec` directly. Instead, when calling Client Library routines, applications explicitly or implicitly pass a pointer to the `DPSContextRec`. This pointer, or **handle**, is of type `DPSContext` and is known as the **DPSContext handle**.

2.2.3 Context Status

An execution context can be in any of several states. For example, a context might be ready to execute, or it might be waiting for PostScript language code from the application. An application can monitor the execution state of a context by requesting **context status events** from the server. A context status event is an X event whose integer value represents the execution state of the context: its **context status**. Each time the context status changes, the server generates a context status event.

Although the server generates context status events, it does not automatically send them. To receive context status events, an application must explicitly set the **context status mask**, a data structure associated with each execution context. (For more information about the context status mask, see the description of the Client Library routine `XDPSSetStatusMask` in Chapter 5.)

2.2.4 Current Context

A typical application creates only one context. For this reason, the Display PostScript system lets an application specify one context as the **current context**. The current context is the default context for Client Library routines that take an implicit context argument.

2.2.5 Space

On the server, each execution context has virtual memory (VM) known as a **space**. In addition to the space of each execution context, there is shared VM, which is shared among all execution contexts of a server.

If an application creates multiple contexts, it can make them share a single space, thereby simplifying communication among them.

2.2.6 Identifiers

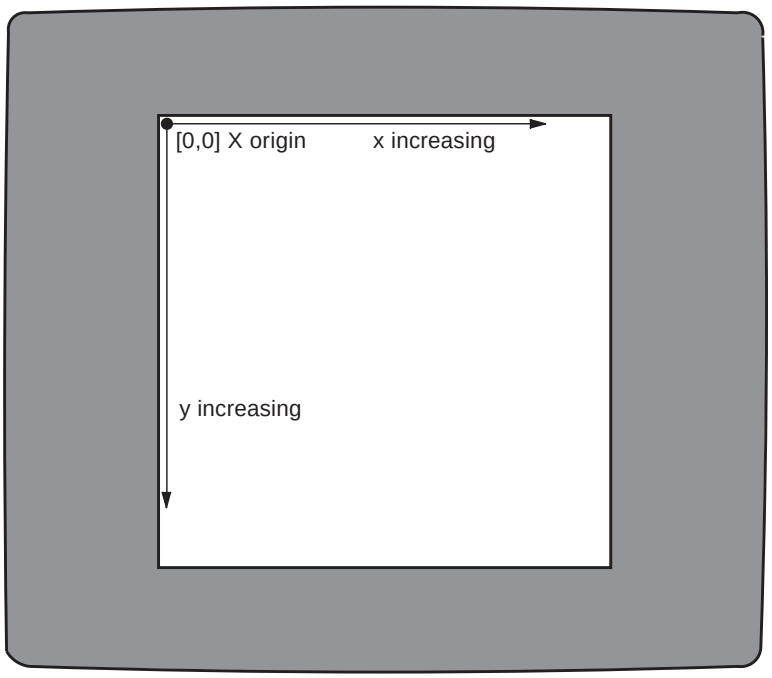
In WS, execution contexts and spaces are associated with X resources on the server. For this reason, execution contexts and spaces have, in addition to their PostScript language ID, an X resource ID (XID). Application programmers, however, seldom need to reference these XIDs.

2.2.7 Coordinate Systems

The Display PostScript system and X both use a coordinate system for imaging, but the coordinate system used by the Display PostScript system differs from that used by X. This section briefly explains both coordinate systems and explains how they interact in WS.

Each X window has a coordinate system whose origin is always the upper left corner. From this **X origin**, *x* increases from left to right; *y* increases from top to bottom, as shown in Figure 2-1.

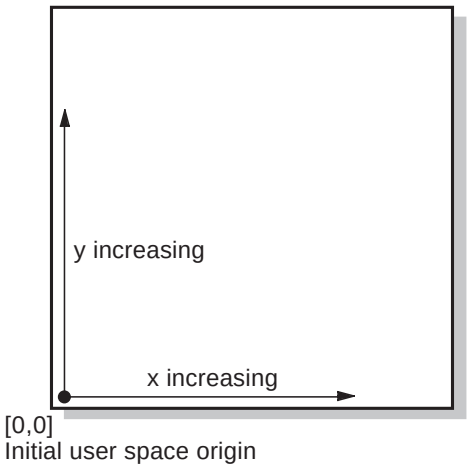
Figure 2-1: X Coordinate System



The origin used by the Display PostScript system is called the **user space origin**. Unlike the X origin, the user space origin can be specified.

From the initial user space origin, *x* increases from left to right (as in X), but *y* increases from bottom to top, as shown in Figure 2-2.

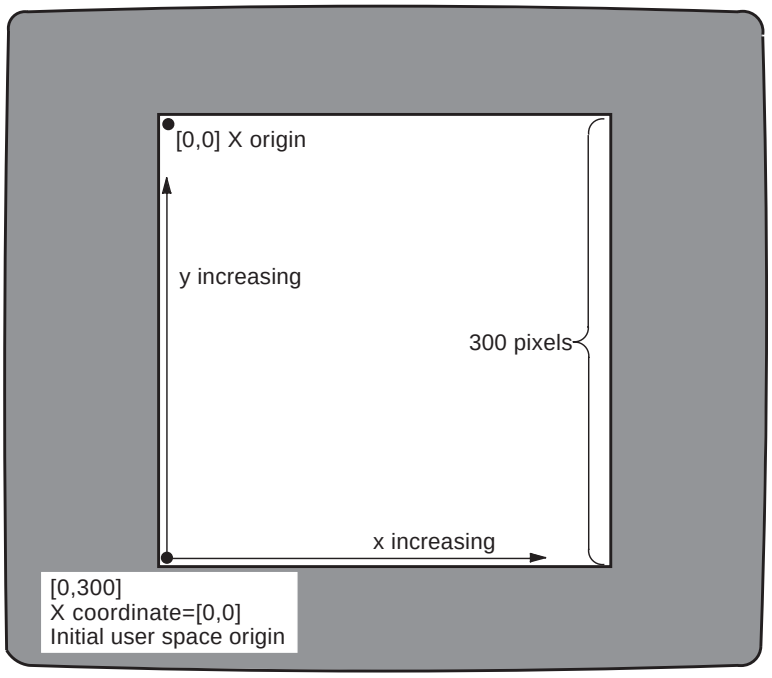
Figure 2-2: User Space Coordinate System Used by the PostScript Language



In WS, the initial user space origin is offset from the X origin. That is, applications specify the initial user space origin as a point in the X coordinate system, as shown in Figure 2-3.

In this figure, an application has created a window measuring 300 x 300 pixels. The application has specified the X coordinates `[0,300]` (the window's lower left corner) as the initial user space origin. Thus, the window's lower left corner becomes the origin `[0,0]` of the user space coordinate system.

Figure 2-3: Initial User Space Origin Offset from X Origin



When an X window is resized, its user space origin moves according to the bit gravity of the window. (For more information on how resizing a window affects its user space origin, see Section 4.5.)

Getting Started 3

This chapter describes the steps you follow to develop a typical application for the Display Postscript system and explains the steps that such an application performs. The chapter then presents a sample application.

Before reading this chapter, be sure you understand the following components and concepts, covered in Chapter 2; if you understand these, you are ready to start:

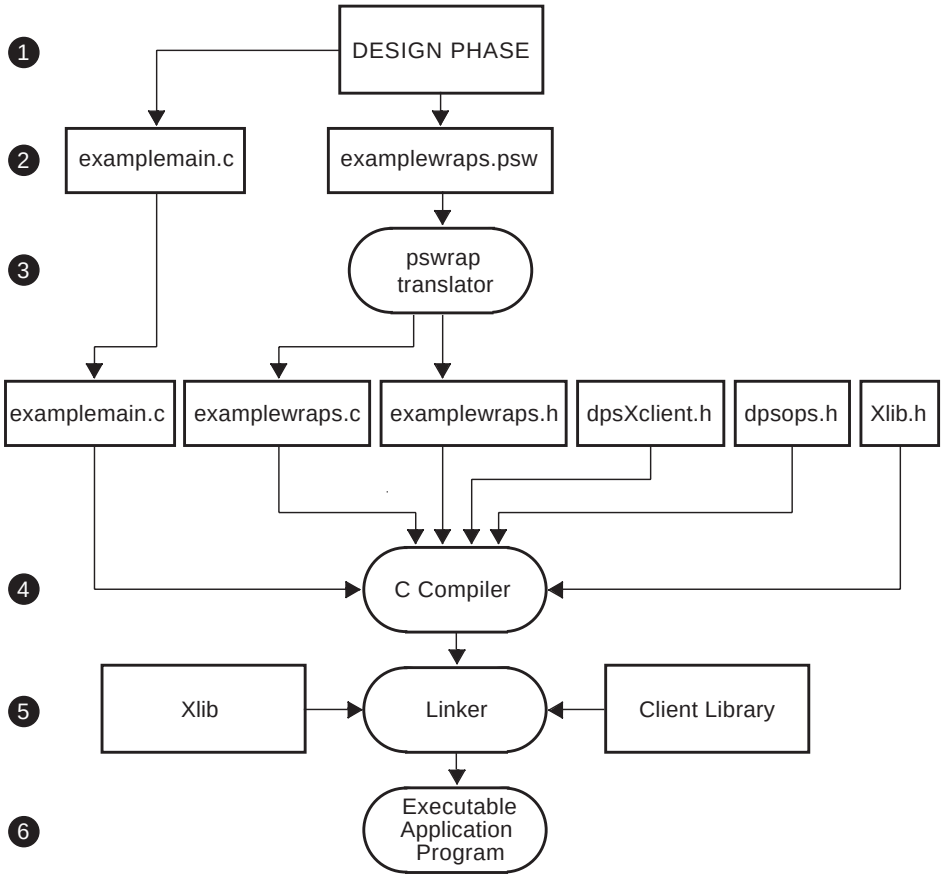
- PostScript interpreter
- Client Library
- The `pswrap` translation program
- PostScript context

3.1 Developing a Typical Application

To develop a typical application, you follow six main steps, as shown in Figure 3-1. (Steps 3 through 5 take much less time than the others.)

1. Design the application.
2. Write the main C-language module and any custom PostScript language procedures that the application calls.
3. Convert the custom PostScript language procedures into C-callable routines by running the `pswrap` translation program.
4. Compile the C-language code with:
   - The output files from `pswrap`
   - The X header files
   - The header file `dpsXclient.h` and any optional XDPS header files, like `dpsops.h`
5. Link the resulting object file with the X libraries and with the Client Library.
6. Run and debug the executable application.

Figure 3-1: Developing a Typical Application



Step 1: Design the Application

In WS, Display PostScript system applications are written in C and send PostScript language code to a context, usually an X server. To design an application you must make several decisions; For example, you must decide:

- Whether to code mostly in C or mostly in the PostScript language
- Whether to create one PostScript context or several
- Whether to send PostScript language code by custom wraps, by singleops, as text, or by a combination of these methods

For a typical simple application, the following design decisions are usually best:

- Code mostly in C; use the PostScript language for imaging-related tasks only.
- Create only one PostScript context.
- Send lengthy PostScript language segments as custom wraps; send single PostScript language statements as singleops.

A complete discussion of application design is beyond the scope of this book. To help you see and understand how design decisions affect XDPS applications, the WS distribution kit includes source files for several sample applications. (For more information about these sample applications, see Section 3.5.)

Step 2: Write Your C Code and PostScript Language Code

After you have designed your application, you write the C-language code and the PostScript language procedures that your application sends.

It is also possible to write applications that read PostScript language code from the user's keyboard or from a file. For a sample program of this type, see the program `DPStest`. By default, the source files for `DPStest` are installed in the directory `/usr/examples/dps/dpstest`. (For instructions on running the program, see Section 3.5.)

Step 3: Convert Your PostScript Language Procedures

If you have written any PostScript language procedures for your application, you should convert them to wraps, that is, to routines that can be called from your C-language code. To convert the PostScript language procedures, you process them with the `pswrap` translation program.

For each PostScript language input file, `pswrap` can produce two output files: a C-callable procedure and an associated header file.

Steps 4 and 5: Compile and Link

After you have converted your PostScript language procedures to C-callable routines, you compile and link your source files. That is, you compile your main C-language file with:

- The output files from `pswrap`
- The X header files
- The `dpsXclient.h` header file and, optionally, other Client Library header files

You link your application with the Client Library and with the X libraries. (For instructions on compiling and linking XDPS applications, see Section 3.4.)

Step 6: Run and Debug Your Application

You are now ready to run and debug your application.

3.2 Basic Application Requirements

All applications send PostScript language statements to a context. Typically, the context is an execution context – in XDPS, the PostScript interpreter of an X server. Most XDPS applications perform three main steps:

1. Initialization
2. Communication
3. Termination

Step 1: Initialization

Typically, to initialize an XDPS application, you perform three steps:

1. Establish communication with an X server, create a window, and create a GC.
2. Create a PostScript execution context by calling an X-specific Client Library routine such as `XDPSCreateSimpleContext`. (For more information on creating contexts, see the descriptions of `XDPSCreateSimpleContext` and `XDPSCreateContext` in Chapter 5.)
3. Perform any additional X-specific initialization, such as mapping the window.

Step 2: Communication

After initializing, most XDPS applications call custom wraps, singleops, or other Client Library routines to send text and PostScript language statements to a context. For example, to send information to a context, an application might either call a custom wrap or call one of two Client Library routines: `DPSWritePostScript` (for PostScript language statements) or `DPSWriteData` (for data).

To process text or errors from a context, the Client Library calls the text-handling routine or error-handling routine that the application assigned when creating the context. The Client Library defines a default text-handling routine (`DPSDefaultTextBackstop`) and a default error-handling routine (`DPSDefaultErrorProc`). Although these routines are called default routines, to use them you must specify them explicitly when creating a context. (For more information on the default routines, see their descriptions in Chapter 5.)

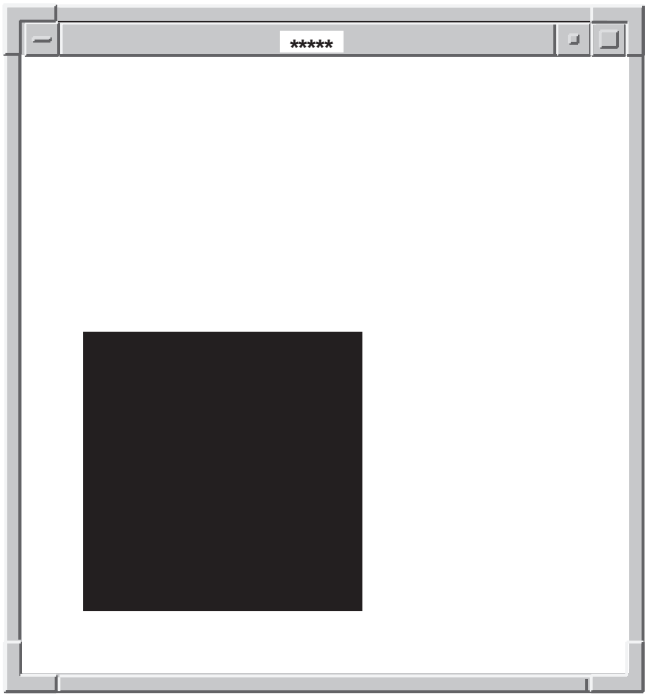
Step 3: Termination

Terminating a typical XDPS application is like terminating any other typical X application. When you terminate an application, the X Window System destroys the application's contexts, their spaces, and any other X resources belonging to the application.

3.3 Sample Application: examplemain

This section presents `examplemain`, a simple program that shows the fundamentals of XDPS programming. The program uses the Display PostScript system to paint a shaded square in a window of the user's screen, as shown in Figure 3-2.

Figure 3-2: Output of the examplemain Program



The `examplemain` program uses the Xlib interface to X, calls a custom wrap to pick the shade of gray for painting, and calls a Client Library single-operator procedure to do the actual painting.

3.3.1 What the Sample Application Does

The sample application `examplemain` performs the following operations:

1. Connects the client to an X server with `XOpenDisplay.`
2. Creates a window with `XCreateSimpleWindow.`
3. Selects X event types Expose and ButtonPress with `XSelectInput.`
4. Creates a Display PostScript execution context with `XDPSCreateSimpleContext`, using the default text handler, the default error handler, and the default GC.
5. Displays the window with `XMapWindow.`
6. Chooses the shade of gray for painting, with a custom wrapped PostScript language procedure named `ChooseGray.`
7. Sets the shade of gray with `DPSsetgray`, a singleop from the Client Library.
8. Paints a gray square with the singleop `DPSrectfill` each time an Expose event is received, and exits when a ButtonPress event is received.
9. Destroys the context and space with `DPSDestroySpace`, then closes the display connection and exits.

Unlike a more complete application, `examplemain` does not handle resizing of the X window. (For information about window resizing in XDPS applications, see Section 4.5.)

3.3.2 The Main Code

Example 3-1 is a complete listing of `examplemain.c`, the main C language file of the sample application.

Example 3-1: Sample Application: examplemain

```
/*
 * examplemain.c -- Simple X application that uses the DPS
 * system to draw a shaded square in a window, then exits
 * when the user clicks the mouse.
 */

#include <stdio.h>

#include <Xlib.h>       /* Standard X Window C-lang library */
#include <dpsXclient.h> /* X interface to DPS Client Library */
#include <dpsops.h>     /* Declarations of singleops */

#include "examplewraps.h" /* Interface to wrapped PS lang code*/

main ()
```

Example 3-1: (continued)

```
{
    Display *dpy;       /* An X display */
    Window window;      /* A window of the X display */
    DPSContext context; /* A single PostScript context */
    float grayLevel;    /* The shade of gray for the square */
    XEvent event;       /* An X event */
    void TextOut();     /* Forward declaration */
    void FatalError();  /* Forward declaration */
    /*
     * Open a connection to the X display specified in the arg
     * to the XOpenDisplay routine. The NULL argument causes
     * XOpenDisplay to open a connection to the display specified
     * by the DISPLAY variable of the user's environment.
     */
    dpy = XOpenDisplay(NULL);
    /*
     * If unable to open the display, return an error message and
     * exit immediately.
     */
    if (dpy == NULL)
        FatalError("Can't open display.0);
    /*
     * Create a window on the X display. When mapped, the
     * window will be 10 pixels from the left edge and 20 pixels
     * from the upper edge. The window will be 800 pixels high
     * by 800 pixels wide, with a black border 1 pixel wide
     * and a white background.
     */
    window = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
                            10, 20, 800, 800, 1,
                            BlackPixel(dpy, DefaultScreen(dpy)),
                            WhitePixel(dpy, DefaultScreen(dpy)));
    /*
     * Select the X event types that the window accepts from
     * the X server. The window accepts Expose events and
     * ButtonPress events.
     */
    XSelectInput(dpy, window, ExposureMask | ButtonPressMask);
    /*
     * Create a PostScript execution context to draw in the window.
     * The origin of the context's coordinate grid is the point
     * (0, 800) of the window. The origin is therefore the bottom
     * left corner of the window (the typical origin for a
     * PostScript context).
     */
    context = XDPSCreateSimpleContext(dpy, window,
                            DefaultGC(dpy, DefaultScreen(dpy)),
                            0, 800,
                            TextOut, DPSDefaultErrorProc, NULL);
    /*
     * If unable to create the context, return an error message
     * and exit immediately.
     */
```

Example 3-1: (continued)

```
if (context == NULL)
    FatalError("DPS refused to create a context.0);
/*
 * Map the window--that is, make it appear on the display.
 * The window will appear only after the window manager of
 * the X server is free to process the mapping request.
 * When the window appears, the context receives an Expose
 * event as notification.
 */
XMapWindow(dpy, window);
/*
 * Generate a random number that corresponds to the shade
 * of gray (the graylevel) to be used when painting.
 * To generate this number, call the ChooseGray routine,
 * which is exported from the examplewraps.c file.
 * ChooseGray sends wrapped PostScript language code to
 * the context, which then executes the code.
 */
ChooseGray(context, &grayLevel);
/*
 * Set the current graylevel to the shade of gray chosen by
 * ChooseGray. Setting the graylevel does not cause any
 * painting; so you can set the graylevel even if the window
 * has not yet appeared.
 */
DPSsetgray(context, grayLevel);
/*
 * Wait for events from the X server; process each one
 * received. For each Expose event, paint the same gray square
 * in the same place on the display. To do this, call the
 * DPSrectfill routine, a single-operator wrapped procedure
 * declared in dpsops.h, a DPS Client Library header file.
 * The bottom left corner of the square is 100 points above
 * the origin and 100 points to the right of it. Each side of
 * the square is 300 points.
 *
 * When a ButtonPress event is received, exit the
 * event-processing loop.
 */
for (;;) {
    XNextEvent(dpy, &event);
    if (event.type == Expose) {
        DPSrectfill(context, 100.0, 100.0, 300.0, 300.0);
    } else if (event.type == ButtonPress) {
        break;
    }
}
/*
 * Exit in an orderly manner. First, destroy the context by
 * destroying its space (its memory). Next, destroy
 * the window. Finally, close the connection to the X display.
 */
DPSDestroySpace(DPSSpaceFromContext(context));
```

Example 3-1: (continued)

```
    XDestroyWindow(dpy, window);
    XCloseDisplay(dpy);
}
    /*
     * Output procedure for plain text messages from the context.
     * Output is sent directly to standard error.
     */
void TextOut(context, buffer, count)
  DPSContext context;
  char *buffer;
  unsigned count;
{
  fwrite(buffer, 1, count, stderr);
  fflush(stderr);
}
    /*
     * Error procedure. The application has encountered an error
     * from which it cannot recover, so exit immediately.
     */
void FatalError(msg)
  char *msg;
{
  fprintf(stderr, msg);
  exit(1);
}
```

3.3.3 Source File for Wrap

Example 3-2 is a complete listing of `examplewraps.psw`, the PostScript language source file for the wrapped procedure called by the sample application `examplemain`.

Processing `examplewraps.psw` with the `pswrap` translator produces two output files: `examplewraps.c` and `examplewraps.h`. These output files must then be compiled with `examplemain.c`.

Example 3-2: Source File for Wrap Called by examplemain

```
/*
 * examplewraps.psw -- source file for wrapped PostScript
 * language procedure
 *
 * This is an example of PostScript language code to be converted
 * to Client Library calls by pswrap.
 *
 * This PostScript language routine, ChooseGray, generates a random
 * number that corresponds to the graylevel (shade of gray) to be
 * used when the Display PostScript system paints. Note that the
 * PostScript operator rand always generates the same sequence of
 * random numbers. So each time the program examplemain runs,
 * ChooseGray chooses the same graylevel.
```

Example 3-2: (continued)

```
 */

defineps ChooseGray (DPSContext ctx| float *result)
  rand              % Pick a random number between 0 and 2^31 - 1.
  2 31 exp          % 2^31
  div               % Random number between 0.0 and 1.0
  result            % Return result.
endps
```

3.3.4 Running examplemain

By default, all the program-specific files needed to compile, link, and run `examplemain` are installed in the `/usr/examples/dps/gray-square` directory of your system. For instructions on compiling and linking, see Section 3.4.

3.4 Building XDPS Applications

After you code an application, you build it by compiling and linking it. The following sections describe how to build an application, assuming that you are using the WS `make` utility. (For more information, see the `make`(1) reference page.)

Section 3.4.5 includes a complete makefile for the `examplemain` program presented in Section 3.3.2.

3.4.1 Including Header Files

Before building an XDPS application, make sure that the main source module includes the appropriate X header files and the WS-specific Client Library header file, `dpsXclient.h`.

The `dpsXclient.h` file is the only Client Library header file that all XDPS applications must include. It, in turn, includes all other Client Library header files, except `psops.h`, `dpsops.h`, and `dpsexcept.h`.

If your application calls singleops, you should also include `psops.h` or `dpsops.h`, or both, depending on which defines the singleops that your application calls. If your application uses the exception handling capability of the Display PostScript system, you must also include `dpsexcept.h`. (Not to be confused with error handling; exception handling is an advanced capability that few applications require.)

3.4.2 Compiling

You compile the main C-language module of your XDPS application with:

- The X header files—for example, `Xlib.h`
- The `dpsXclient.h` header file
- The `psops.h` and `dpsops.h` header files (if application calls singleops)
- The output files from `pswrap` (if application calls custom wraps)

The Display PostScript system header files (among them, `dpsXclient.h`, `psops.h`, and `dpsops.h`) are installed in the directory `/usr/include/DPS`. To automatically include these files at compilation, add the following statement to your makefile:

```
CFLAGS = -I/usr/include/DPS
```

The option `-I/usr/include/DPS` causes the Workstation Software C compiler to search for include files in `/usr/include/DPS`.

3.4.3 Linking

You link your XDPS application with the following libraries, in the order listed:

Library	Linker Option
Client Library	`-ldps`
Xlib extensions for Display PostScript system	`-lXext`
DECwindows toolkit library	`-ldwt`
Xlib library	`-lX11`
Workstation Station math library	`-lm`

3.4.4 Invoking pswrap from a Makefile

Your makefile can automatically convert PostScript language procedures to C-callable routines by running the `pswrap` translation program. For example, if the PostScript language procedures have file names ending in `.psw`, the following `make` statements convert the procedures automatically:

```
.SUFFIXES: $(.SUFFIXES) .psw .h
.psw.o: $*.psw
        ${PSWRAP} -o $*.c $*.psw
        $(CC) $(CFLAGS) -c $*.c
        rm $*.c

.psw.h: $*.psw
        ${PSWRAP} -h $*.h $*.psw > /dev/null
```

3.4.5 Sample Makefile

Example 3-3 shows a complete Makefile that builds the `examplemain` program presented earlier in this chapter.

Example 3-3: Makefile for examplemain

```
# @(#)Makefile 1.5 9/2/88

DESTDIR=
EXAMPLETOPDIR=${DESTDIR}/usr/examples/dps
EXAMPLESUBDIR=${EXAMPLETOPDIR}/gray-square

INSTALLLIST = Makefile examplemain.c *.psw

OBJS = examplemain.o examplewraps.o

PSWRAP= ${DESTDIR}/usr/bin/pswrap

.SUFFIXES: $(.SUFFIXES) .psw .h
.psw.o: $*.psw
 ${PSWRAP} -o $*.c $*.psw
 $(CC) $(CFLAGS) -c $*.c
 rm $*.c

.psw.h: $*.psw
 ${PSWRAP} -h $*.h $*.psw > /dev/null

.SUFFIXES: .uil .uid

CFLAGS = -g -I${DESTDIR}/usr/include/X11 \
 -I${DESTDIR}/usr/include/DPS \
 -I${DESTDIR}/usr/include -I.

LIBS = ${DESTDIR}/usr/lib/libdps.a \
 ${DESTDIR}/usr/lib/libXext.a \
 ${DESTDIR}/usr/lib/libdwt.a \
 ${DDIFROOT}/usr/lib/libddif.a \
 ${DESTDIR}/usr/lib/libX11.a \
 -lm

all: examplemain

examplemain: $(OBJS)
```

Example 3-3: (continued)

```
 $(CC) -o examplemain $(OBJS) $(LIBS)

examplemain.o: examplemain.c examplewraps.h

clean:
 rm -f *.o examplemain examplewraps.[ch] #* *~ core

clobber: clean
 -rm -f *

relink::
 rm -f examplemain

relink:: all
```

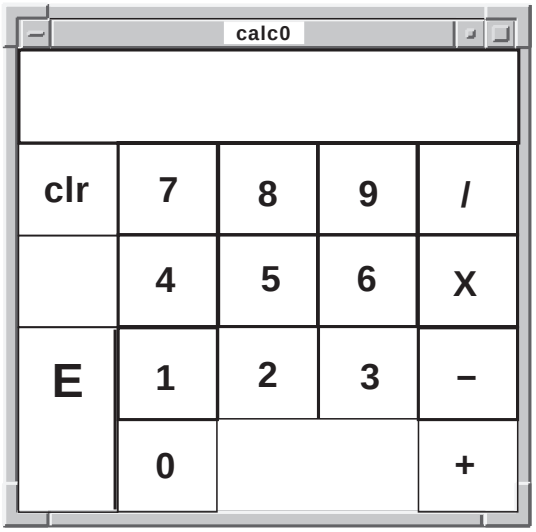
3.5 More Sample Applications

In addition to `examplemain`, the WS software includes source listings of several other sample XDPS applications.

3.5.1 Examples Contrasting Design Approaches

WS includes source listings and makefiles for four related sample programs: `calc0`, `calc1`, `calc2`, and `calc3`. Each of these sample programs is an implementation of the same application: a desktop calculator. Although all four programs present a similar user interface (shown in Figure 3-3), the source code of each program shows a different approach to XDPS application design.

Figure 3-3: Output of the Sample Calculator Programs



For the location of the sample calculator programs, see Table 3-1, which lists and describes the sample Display PostScript system applications included in WS.

Table 3-1: Online Sample Programs

Program Name	Description	Location
`calc0`	Calculator coded mainly in C, with one window and one context	`/usr/examples/dps/calc0`
`calc1`	Calculator coded mainly in the PostScript language, with one window and one context	`/usr/examples/dps/calc1`
`calc2`	Calculator coded mainly in C, with multiple windows and one context	`/usr/examples/dps/calc2`

Table 3-1: (continued)

Program Name	Description	Location
`calc3`	Calculator coded mainly in C, with multiple windows, multiple contexts, and intercontext communication	`/usr/examples/dps/calc3`
`DPStest`	Executes PostScript language statements entered from the keyboard	`/usr/examples/dps/dpstest`
`examplemain`	Displays a gray square generated from a custom wrap and a singleop	`/usr/examples/dps/gray-square`
`psclock`	An implementation of `xclock` that uses the Display PostScript system	`/usr/examples/dps/psclock`
`psdraw`	A graphic editor that paints PostScript language images; a complex sample application	`/usr/examples/dps/psdraw`
`pyro`	Displays fireworks generated from custom wraps	`/usr/examples/dps/pyro`

3.5.2 Running the Sample Applications

To run a sample application, you must first build it by following these steps:

1. Log on to your system and find the subdirectory where the sample application is stored.
2. Copy the entire contents of that subdirectory to a subdirectory in your account. (Note that the sample programs `calc0`, `calc1`, `calc2`, and `calc3` must be copied to sibling directories, that is, to subdirectories at the same level of the file system.)
3. Set your working directory to the subdirectory that received the copies in Step 2.
4. Invoke the WS `make` utility by entering the command `make` at the system prompt. The `make` utility compiles and links the program. (Note that for the sample application `psdraw`, you must enter `make install` instead of `make`. For information, see the `make`(1) reference page.

You can then run the program by entering its name at the system prompt. (For more information on building XDPS applications, see Section 3.4.)

3.6 Summary of Basic Tasks

Table 3-2 lists common XDPS programming tasks, shows the operators (in bold type), and Client Library routines for performing each task.

Table 3-2: Summary of Basic Tasks

Task	Associated Routines and Operators
Create an execution context	XDPSCreateSimpleContext or `XDPSCreateContext`
Create a text context	`XDPSCreateTextContext`
Use the default text handler	`DPSDefaultTextBackstop`
Use the default error handler	`DPSDefaultErrorBackstop`
Find the space of a context	`DPSSpaceFromContext`
Find the default user space origin	`currentXoffset`
Set the default user space origin	`setXoffset`
Find the GC of a context	`currentXgcdrawable`
Set the GC of a context	`setXgcdrawable`
Restart a context	`DPSResetContext`
Find the current drawable	`currentXgcdrawable`
Set the current drawable	`setXgcdrawable`

Convert between PostScript language IDs and XIDs	`XDPSXIDFromContext`
	`XDPSXIDFromSpace`
	`XDPSContextFromXID`
	`XDPSSpaceFromXID`

Destroy a space	`DPSDestroySpace`
Destroy a context	`DPSDestroyContext`

Advanced Concepts and Tasks 4

In Chapter 2 and Chapter 3 you learned the basic concepts and tasks you need to write simple applications using XDPS. To write more complex applications, however, you need the additional concepts and tasks described in this chapter.

4.1 PostScript Language Encoding

In XDPS, PostScript language code can be sent to a context in three **encodings**: as a binary object sequence, as binary-encoded tokens, or as ASCII text. Each PostScript context has two **encoding parameters**: `DPSProgramEncoding` and `DPSNameEncoding`.

XDPS uses default values for the encoding parameters, so application programmers can usually ignore encoding. Table 4-1 shows the default values for the encoding parameters.

Table 4-1: Default PostScript Language Encodings for XDPS

Context Type	Encoding Parameter	Default Value
execution	DPSProgramEncoding	Binary object sequence (`dps_binObjSeq`)
execution	DPSNameEncoding	User name index (`dps_indexed`)
text	DPSProgramEncoding	ASCII characters (`dps_ascii`)
text	DPSNameEncoding	User name string (`dps_string`)

XDPS lets you change the encoding parameters of a context to any of the three possible encodings. To change the encoding parameters, use the Client Library routine `DPSChangeEncoding`, described in Chapter 5.

4.2 Buffering and the Client Library

In most implementations of the Display PostScript system, the Client Library buffers its communications with the Display PostScript server. But in XDPS, the Client Library communicates with the server by way of Xlib, which buffers its own communication. To avoid duplicate buffering, the XDPS Client Library performs no internal buffering. Instead, all buffering of Client Library communication occurs in Xlib. As a result, the XDPS Client Library routine `DPSFlushContext` performs the same tasks as the Xlib procedure `XFlush`.

4.3 Accessing Files on the Server

To preserve security on servers, XDPS lets applications access only certain files stored on the server. Specifically, XDPS lets applications access only files stored in two directories referred to here as `tempdir` and `permdir`.

The `tempdir` directory is temporary: its contents are deleted each time the XDPS server is started or reset, such as when the user logs out. In contrast, `permdir` is a permanent directory: resetting and restarting does not affect its contents. Applications can both read from `tempdir` and write to it. Applications can only read from `permdir`; they cannot write to it.

To specify a file stored in `tempdir`, an application must prefix the filename with `%temp%`. To specify a file in `permdir`, an application must use the prefix `%perm%`. If a filename is preceded by neither `%temp%` nor `%perm%`, XDPS searches for the file first in `tempdir` and then in `permdir`. XDPS does not let applications access file names that include a slash (/), a bracket ([), or a colon (:).

By default, `tempdir` is the directory `/usr/lib/DPS/tempdir`; `permdir` is `/usr/lib/DPS/permdir`. You can, however, assign other directory names. To do so, specify those names in the XDPS server startup command.

4.4 Converting Coordinates

The X Window System and the PostScript language use different coordinate systems to specify points within the drawing area. As a result, XDPS applications sometimes need to convert user space coordinates (used by the PostScript language) into X coordinates, and vice versa.

4.4.1 Preparing to Convert Coordinates

Before converting coordinates, an application should create a context and perform the following steps:

1. Perform any user space transformations.
2. Get the current transformational matrix (CTM), its inverse, and the X coordinates of the current user space origin.
3. Store these values in the VM associated with the context.

The application can then perform coordinate conversions for the context.

To get the CTM, its inverse, and the X coordinates of the current user space origin, an application can call a custom wrap such as `PSWGetTransform`, whose `pswrap` source file is shown in Example 4-1.

Example 4-1: Wrap Returning CTM, Its Inverse, and Current User Space Origin

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

The following C language code calls `PSWGetTransform`:

```
DPSContext ctxt;
float ctm[6], invctm[6];
int xOffset, yOffset;
PSWGetTransform(ctxt, ctm, invctm, &xOffset, &yOffset);
```

4.4.2 X Coordinates to User Space Coordinates

To convert an X coordinate into a user space coordinate, an application can execute the following C language code:

```
#define A_COEFF 0
#define B_COEFF 1
#define C_COEFF 2
#define D_COEFF 3
#define TX_CONS 4
#define TY_CONS 5
int x,y; /* X coordinate */
float ux, uy; /* user space coordinate */

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

4.4.3 User Space Coordinates to X Coordinates

To convert a user space coordinate into an X coordinate, an application can execute the following C language code:

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

4.5 Resizing Windows

An application or user can resize the window in which XDPS paints. Resizing can affect two PostScript language settings, the clipping path and the user space origin, as described in the following sections.

4.5.1 Window Resizing and the Clipping Path

PostScript language painting occurs only within the area known as the **clipping path**. When initializing a context, XDPS sets the clipping path equal to the size of the window. If the window is resized, however, XDPS does not reset the clipping path. Instead, each time the window is resized, the application should execute the PostScript language operator `initclip`, which reinitializes the clipping path to match the window's new size. The application can then reexecute any code that performs further clipping.

4.5.2 Window Resizing and the User Space Origin

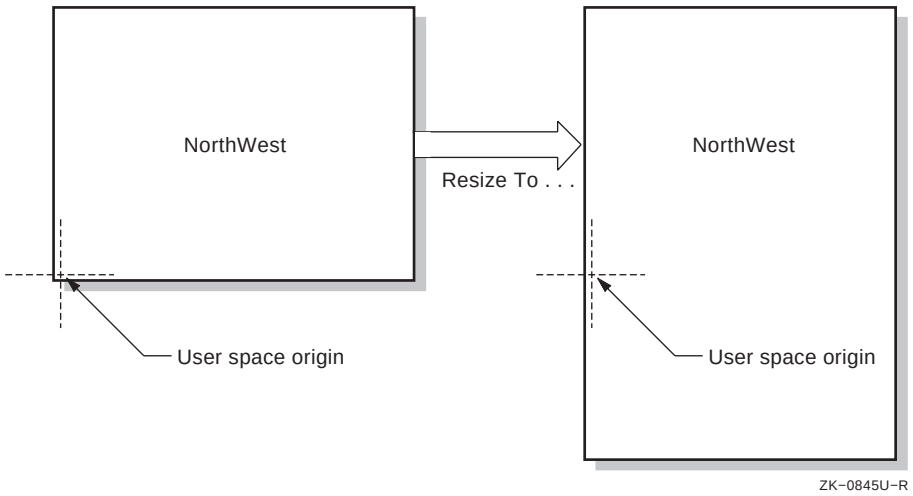
When an application resizes the window of a context, the user space origin moves according to the **bit gravity** of the window. Bit gravity is an X window attribute that governs how partial window contents are preserved when a window is resized. (Bit gravity is not to be confused with window gravity, an X attribute that does not affect the user space origin.) In X, specifying the bit gravity of a window is optional: the default value is `ForgetGravity`. XDPS treats `ForgetGravity` as `NorthWest` gravity.

Because a window's user space origin moves according to the window's bit gravity, resizing does not change the distance between the user space origin and any PostScript language images already displayed. Because this distance is unchanged, future PostScript language images align with those already displayed.

Compare Figure 4-1 and Figure 4-2. The left side of Figure 4-1 shows a window displaying the text ''NorthWest.'' As shown, the user space origin is the window's lower left corner, and the bit gravity is `NorthWest`.

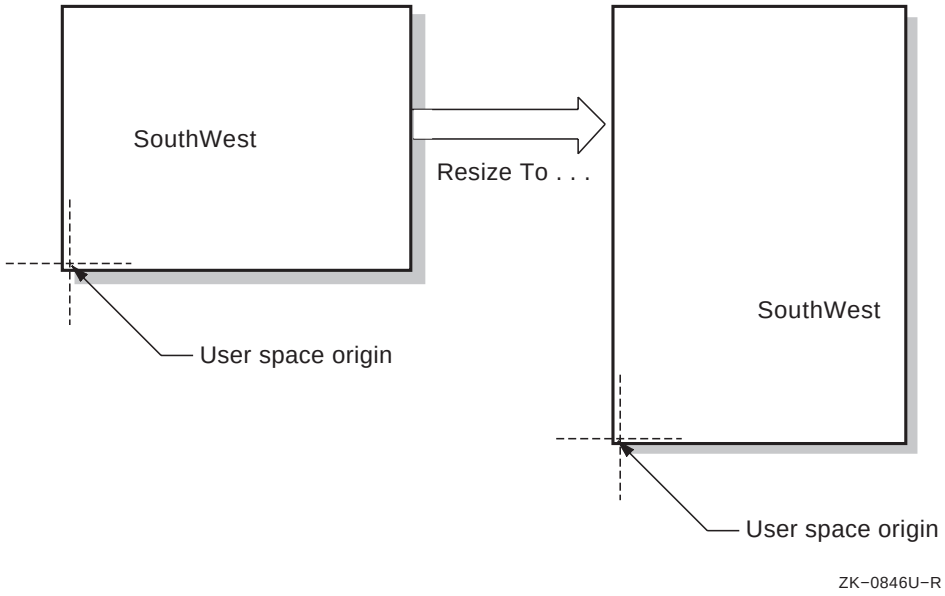
The right side of the figure shows the same window after resizing. Notice that the user space origin (and hence the displayed text) remains a constant distance from the window's upper left corner: its ''NorthWest'' corner.

Figure 4-1: Resizing a Window with NorthWest Bit Gravity



In Figure 4-2, the size of the window on the left and the position of its text are the same as in Figure 4-1. Also the same is the user space origin: the lower left corner. In Figure 4-2, however, the bit gravity is `SouthWest`. Therefore, when the window is resized, the user space origin and displayed text remain a constant distance from the window's lower left corner: its ''SouthWest'' corner.

Figure 4-2: Resizing a Window with SouthWest Bit Gravity



The user space origin is typically the lower left corner of the drawing space. For this reason, typical XDPS applications should explicitly set the bit gravity of windows to `SouthWest`.

4.6 Synchronizing the Display PostScript System and X

X imaging calls complete atomically. Therefore, XDPS applications need not take special precautions when issuing X imaging calls before PostScript language imaging calls. PostScript contexts, however, complete nonatomically and asynchronously within the X server. Thus, when an application issues X imaging calls immediately after issuing PostScript language calls, the X calls can sometimes execute before the PostScript language calls. That is, it is possible for X and the Display PostScript system to become unsynchronized.

Few applications need to synchronize the Display PostScript system and X explicitly. To do so, an application can call the Client Library routine `DPSWaitContext` before issuing the X imaging calls that follow PostScript language calls. `DPSWaitContext` forces the PostScript language calls to complete before the X calls. Note that `DPSWaitContext` causes a round trip to the server. Such trips impair performance, so call `DPSWaitContext` only when needed.

4.7 Synchronizing Client and Context

Applications, or clients, sometimes need to pause the execution of a context. Pausing a context lets an application take control when the PostScript interpreter reaches certain points within a PostScript language procedure.

To pause a context, an application sends the system-specific PostScript language operator `clientsync`. The `clientsync` operator causes a context to enter the FROZEN state. The context remains in that state until the application calls the Client Library routine `XDPSUnfreezeContext`. (For more information on `clientsync`, see its description in Chapter 6. For a description of `XDPSUnfreezeContext`, see Chapter 5.)

4.8 Sharing Contexts and Spaces

Although the XDPS Client Library lets applications share contexts and spaces, it does not coordinate the sharing. Instead, the applications themselves must coordinate any sharing of resources.

The sharing applications must avoid race conditions and deadlocks. In addition, if one application obtains the XID of a resource created by another, the application that obtained the XID must create records and handles to access the shared resource through the Client Library.

A context or space cannot be destroyed while shared. If such a resource is shared, the routines `DPSDestroyContext` and `DPSDestroySpace` destroy the client data structures created to access the shared resource but do not destroy the resource itself. After a resource is no longer shared, an application can destroy it by calling `DPSDestroyContext` or `DPSDestroySpace`.

4.9 Using Color

In XDPS, the Display PostScript system paints colors and gray shades on an X server. An X server can render only a finite number of exact colors and shades simultaneously; it represents each as a pixel value. In contrast, the PostScript language represents colors and shades not as pixel values but as ‘‘pure’’ colors and ‘‘pure’’ shades, without regard for whether the output device can render them exactly. As a result, to paint on an X display, a PostScript context must first find whether there is a pixel value that matches

the pure color or shade specified by the PostScript language.

4.9.1 Converting Colors and Shades into Pixel Values

To find the pixel value that matches a particular color or shade, a context searches the **color cube** or **gray ramp**. The color cube and gray ramp specify pixel values that correspond to a subset of all possible pure colors and shades.

The color cube defines a set of colormap cells whose values form a series of color ramps (progressive changes in color). Each axis of the color cube represents one of three hues: red, green, or blue (r/g/b); all displayed colors are composites of these hues. Values along the axes of the cube represent intensity of hue and increase from 0% to 100% of the displayed color. Note that the color cube is not a cube in the strict sense of the word: the axes need not have the same ''length,'' that is, the same number of values.

The gray ramp defines a set of colormap cells whose values form a single color ramp of gray shades. Values along the gray ramp represent comparative intensities of black and white. Along the ramp, the intensity of white increases from 0% to 100%.

If the color cube or gray ramp contains a pixel value that exactly matches the specified pure color or shade, the context uses the pixel value to paint the pure color or shade. Otherwise, the context approximates the color or shade by **dithering**, by painting a pattern of colors or gray shades from its color cube or gray ramp.

4.9.2 Defining a Color Cube and Gray Ramp

When creating a context, an application must allocate and define a color cube and gray ramp. If the application defines no color cube, the context renders colors by dithering from the gray ramp. If the application defines neither a color cube nor a gray ramp, the context cannot paint.

Typically, applications create contexts by calling `XDPSCreateSimpleContext`. This routine allocates and defines a color cube and gray ramp using the `XStandardColormap` structures `RGB_DEFAULT_MAP` and `RGB_GRAY_MAP`. If these structures do not exist, `XDPSCreateSimpleContext` allocates them. To allocate and define a different color cube and gray ramp, an application can use either of two methods:

- Create the context by calling `XDPSCreateContext`.
- Create the context by calling either `XDPSCreateSimpleContext` or `XDPSCreateContext`, then use the X-specific operator `setXgcdrawablecolor` to redefine the color cube and gray ramp.

To allocate and define a color cube and gray ramp, an application performs the following steps:

1. Call `XCreateColormap` to create a colormap. (This optional step is needed only if the application does not use the default colormap.)
2. Call `XAllocColorCells` to allocate the colormap cells needed to store the color cube and gray ramp.
3. Call `XStoreColors` to store a color for each pixel value in the color cube and gray ramp.
4. Call `XDPSCreateContext` to create a context and pass the `XStandardColormap` structures describing the color cube and gray ramp.

The following sections describe how XDPS uses the color cube and gray ramp, referring to the following elements of the color cube and gray ramp:

```
maxred      redmult
maxgreen    greenmult
maxblue     bluemult
maxgrays    graymult
firstgray   firstcolor
colormapid
```

These names are the same as those used for elements of the `colorinfo` array, which is accessed by the X-specific operators `setXgcdrawablecolor` and `currentXgcdrawablecolor`. (For more information, see the description of these operators in Chapter 6.)

4.9.2.1 Using the Color Cube

To render an exact color, XDPS searches the colormap for the pixel value matching the r/g/b value specified in the color cube. Conceptually, the color cube is three-dimensional; the colormap, however, is conceptually one-dimensional. Thus, to find the pixel value that matches an r/g/b value, XDPS uses the following formula:

PixelValue = *r* * *redmult* + *g* * *greenmult* + *b* * *bluemult* + *firstcolor*

In this formula, *r*, *b*, and *g* are integers. The integer *r* is in the range [*0*; *maxred*]; *g* is in the range [*0*; *maxgreen*]; and *b* is in the range [*0*; *maxblue*].

A color cube must start at pixel `firstcolor` in the X colormap `colormapid`. Along the red, green and blue axes of the cube, values should increase from zero to the maximum values for each axis. For example, one common color allocation is 3/3/2 (three reds, three greens, and two blues). This allocation results in the following maximum value for each hue:

maxred = 2
maxgreen = 2
maxblue = 1

In the `colorinfo` array, the elements *redmult*, *greenmult*, and *bluemult* are the scale factors that determine the spacing of the cube in the linear colormap. For the 3/3/2 color cube mentioned earlier, appropriate values might be:

redmult = 32
greenmult = 4
bluemult = 1

Note

In an empty color cube, *maxred*, *maxgreen*, and *maxblue* each equal -1, not zero.

4.9.2.2 Using the Gray Ramp

The gray ramp must start at pixel `firstgray` in XStandardColormap `colormapid`. To find the pixel value that matches a gray value, XDPS uses the following formula, where *gray* is an integer in the range [*0*; *maxgrays*]:

PixelValue = *gray* * *graymult* + *firstgray*

For example, suppose you want to define a 5-cell gray ramp whose values increase from 0% to 100% in steps of 20%. If the corresponding five colormap entries are contiguous, you can describe the map by setting *maxgray* to 4 and *graymult* to 1.

A gray ramp must consist of at least two cells: one for black, one for white. If the colormap is associated with the default visual type, you can use the following values to form a 2-cell gray ramp consisting of `BlackPixel` and `WhitePixel`:

maxgrays = 1
graymult = *WhitePixel*– *BlackPixel*
firstgray = *BlackPixel*

4.9.3 Rendering Colors Not in the Color Cube

By default, XDPS dithers to render any color not in the color cube. To render such an additional color exactly, an application must cause the X server to allocate a colormap cell for the additional color.

To control whether additional colors are rendered exactly or by dithering, an application can set the *actual* element of the `colorinfo` array. The *actual* element specifies the maximum number of additional colormap cells that the server attempts to allocate. Thus, it limits the number of additional colors that the server attempts to render exactly.

If *actual* is nonzero, the server attempts to allocate a colormap cell for each additional color until it has allocated *actual* cells. After *actual* cells have been allocated, the server renders any future additional colors by dithering. If *actual* equals zero, the server dithers to render all colors not found in the color cube.

To override the maximum set by *actual*, an application can use the X-specific operator `setrgbXactual`.

Note

XDPS does not limit the number of colormap cells that one context or one application can allocate.

4.9.4 The colorinfo Array and XStandardColormaps

The color cube and gray ramp are passed to `XDPSCreateContext` as `XStandardColormap` structures. Table 4-2 and Table 4-3 show how the entries in these `XStandardColormap` structures correspond to elements in the colorinfo array.

Table 4-2: Mapping Between colorinfo Array and XStandardColormap Storing Color Cube

colorinfo Element	XStandardColormap Element
maxred	red_max
redmult	red_mult
maxgreen	green_max
greenmult	green_mult
maxblue	blue_max
bluemult	blue_mult
firstcolor	base_pixel

Table 4-3: Mapping Between colorinfo Array and XStandardColormap Storing Gray Ramp

colorinfo Element	XStandardColormap Element
maxgrays	red_max
graymult	red_mult
firstgray	base_pixel
colormapid	colormap

Client Library Routines for WS 5

The Client Library is the set of C language routines by which XDPS applications access a server, that is, the PostScript interpreter of an X server. The Client Library includes routines that create, communicate with, and destroy PostScript contexts on the server.

Most Client Library routines are common to all windowing systems that implement the Display PostScript system. But for any particular windowing system, such as X, additional routines and data structures must be added to the Client Library.

This chapter describes WS-specific routines and data structures that have been added to the Client Library.

For the rest of this chapter, except where noted, the term ‘‘Client Library’’ refers to the Display PostScript system Client Library as implemented in WS.

The Client Library routines are defined in six C-language header files:

- `dpsclient.h`
- `dpsfriends.h`
- `dpsexcept.h`
- `dpsops.h`
- `psops.h`
- `dpsXclient.h`

The first five of these files are common to all implementations of the Display PostScript system. The sixth file, `dpsXclient.h`, is specific to XDPS and is described in the following section.

5.1 System-Specific Header File

The header file `dpsXclient.h` defines the system-specific Client Library routines and data structures of XDPS. Like the other Display PostScript system header files, `dpsXclient.h` is located in the directory `/usr/include/DPS`. The `dpsXclient.h` file is the only Client Library header file that all XDPS applications must include.

5.2 X-Specific Singleops

The Client Library includes a set of routines called singleops (single-operator wrapped procedures). Each singleop sends one or more operators to a context. For instance, the singelop `PSshowpage` sends operator `showpage`.

For each operator, the Client Library defines two singleops: one takes an implicit context argument (always the current context); the other takes an explicit context argument. For example, the Client Library contains the singleops `PSshowpage` and `DPSshowpage`. Although both singleops execute the operator `showpage`, `PSshowpage` takes an implicit context argument; `DPSshowpage` takes an explicit one.

Implicit-context singleops are defined in the header file `psops.h`; explicit-context singleops are defined in `dpsops.h`. If your application creates only one context, using implicit-context singleops can make coding easier.

The Client Library includes **X-specific singleops**. Each of these singleops sends an X-specific operator, for example, `setXgcdrawable`. Like other singleops, X-specific singleops are of two types: implicit-context and explicit-context. X-specific singleops that take an implicit context argument are defined in the file `pscustomops.h`, which is included by `psops.h`. X-specific singleops that take an explicit context are defined in `dpscustomops.h`, which is included by `dpsops.h`.

Example 5-1 shows the definitions of the X-specific singleops. Table 5-1 describes the arguments used in the definitions. For descriptions of the operators that the X-specific singleops send, see Chapter 6.

Example 5-1: Definitions of X-specific Singleops

```
extern void DPSclientsync( /* DPSContext ctxt; */ );

extern void DPScurrentXgcdrawable( /* DPSContext ctxt; int *gc, *d,
                                   *x, *y; */ );

extern void DPScurrentXgcdrawablecolor( /* DPSContext ctxt;
                                        int *gc, *d, *x, *y,
                                        colorInfo[12]; */ );

extern void DPScurrentXoffset( /* DPSContext ctxt; int *xOffset,
                               *yOffset; */ );

extern void DPSsetXgcdrawable( /* DPSContext ctxt; int gc, d, x, y; */ );

extern void DPSsetXgcdrawablecolor( /* DPSContext ctxt;
                                    int gc, d, x, y, colorInfo[12]; */ );

extern void DPSsetXoffset( /* DPSContext ctxt; short int x, y; */ );

extern void DPSsetXrgbactual( /* DPSContext ctxt; int r, g, b;
                              Boolean *success; */ );
```

Example 5-1: (continued)

```
extern void PSclientsync();

extern void PScurrentXgcdrawable( /* int *gc, *d, *x, *y; */ );

extern void PScurrentXgcdrawablecolor( /* int *gc, *d, *x, *y,
                                    colorInfo[12]; */ );

extern void PScurrentXoffset( /* int *xOffset, *yOffset; */ );

extern void PSsetXgcdrawable( /* int gc, d, x, y; */ );

extern void PSsetXgcdrawablecolor( /* int gc, d, x, y,
                                 colorInfo[12]; */ );

extern void PSsetXoffset( /* int x, y; */ );

extern void PSsetXrgbactual( /* int r, g, b; Boolean *success; */ );
```

Table 5-1: Arguments Used by X-Specific Singleops

Name	Type	Description
`colorInfo[12]`	integer array	Stores color attributes of the context. The elements of this array are *`graymax`*, *`graymult`*, *`firstgray`*, *`redmax`*, *`redmult`*, *`greenmax`*, *`greenmult`*, *`bluemax`*, *`bluemult`*, *`firstcolor`*, *`colormapid`*, and *`numactual`*.
`d`	integer	The X resource ID of an X drawable. If *`d`* equals zero, all drawing operations are ignored.
`gc`	integer	The `GContext` resource ID for the X Graphic Context of *`drawable`*. If *`gc`* equals zero, all drawing operations are ignored. To obtain a value for *`gc`*, call the Xlib routine (`XGContextFromGC`), passing the Xlib data type `GC` of the current X Graphic Context as the argument.
`r`, *`g`*, *`b`*	integer	Levels for red, green, and blue, in the X color space [0..65535].
`success`	Boolean	When nonzero, shows that the singleop completed without a PostScript language error. When zero, shows that the singleop produced a PostScript language error on the server.

Table 5-1: (continued)

Name	Type	Description
x and *y*	integer	The horizontal and vertical coordinates (in X units) for the default user space origin of the current drawable. If *x* equals zero, and *y* equals the height of the drawable (in pixels), the default user space origin is at the lower left corner of the drawable. In the PostScript language, this is the typical location for the default user space origin.
xOffset and *yOffset*	integer	Same as *x y* ; see descriptions in this table.

5.3 Naming Conventions

Table 5-2 shows conventions used to name the WS-specific Client Library routines.

Table 5-2: Naming Conventions in the Client Library

Type of Routine	Naming Convention
System-specific Routine	DPS*Mnemonic_name*
X-specific Client Library routine	XDPS*Mnemonic_name*
Singleop with implicit context argument	PS*operator_name*
Singleop with explicit context argument	DPS*operator_name*

5.4 Client Library Routine Descriptions

This section describes the following system-specific Client Library routines:

```
DPSChangeEncoding
DPSContextFromContextID
DPSCreateTextContext
DPSDefaultTextBackstop
DPSNewUserObjectIndex
XDPSContextFromSharedID
XDPSContextFromXID
```

```
XDPSCreateContext
XDPSCreateSimpleContext
XDPSFindContext
XDPSRegisterStatusProc
XDPSSetStatusMask
XDPSSpaceFromSharedID
XDPSSpaceFromXID
XDPSUnfreezeContext
XDPSXIDFromContext
XDPSXIDFromSpace
```

In the following list, the routines are arranged alphabetically by name. Each description provides the C-language defintion of the routine, followed by text describing what the routine does and what its arguments represent.

DPSChangeEncoding

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

The DPSChangeEncoding routine sets the value of one or both encoding parameters of the context specified by *ctxt*. If the encoding parameters are set to values other than the default values, DPSWritePostScript, singleops, and custom wraps convert PostScript language code to the specified encoding before sending it to context *ctxt*.

For a list of the default encodings, see Section 4.1.

DPSContextFromContextID

```
DPSContext
DPSContextFromContextID(/*
    DPSContext ctxt;
    long int cid;
    DPSTextProc textProc;
    DPSErrorProc errorProc */);
```

The DPSContextFromContextID routine returns the DPSContext handle of the context whose PostScript language ID is *cid*. Context *cid* is one created when a preexistent context, *ctxt*, executed the PostScript operator fork. The arguments *textProc* and *errorProc* specify the two routines with which the calling client handles text and errors from the context *cid*.

If the calling client has no context record for context *cid*, DPSContextFromContextID creates one. The new context record uses the text handler and error handler passed in *textProc* and *errorProc*. If *textProc* or *errorProc* is NULL, the new context record uses the text handler and error handler of *ctxt*.

Except for the text handler, error handler, and chaining pointers, the

created context record inherits all its characteristics from *ctxt*.

DPSCreateTextContext

```
DPSContext
DPSCreateTextContext(/*
    DPSTextProc textProc;
    DPSErrorProc errorProc */);
```

The DPSCreateTextContext routine creates a context record and a DPSContext handle not associated with an execution context. When this DPSContext handle is passed as the argument to a Client Library routine, that routine converts all context input into ASCII text, then passes that text to the text-handling routine *textProc*. The routine specified by *errorProc* handles errors that result from improper context usage. (For example, one such error occurs if the context is invalid.)

Do not use the *errorProc* routine to handle errors that result from executing *textProc*. For example, if your *textProc* routine writes text to a file, do not use *errorProc* to handle file-related errors, such as those that occur when a file is write-protected.

DPSDefaultTextBackstop

```
void DPSDefaultTextBackstop
(/* DPSContext ctxt;
    char *buf;
    unsigned count */);
```

The DPSDefaultTextBackstop routine is a text-handling routine; it is the default text backstop installed by the Client Library. Because DPSDefaultTextBackstop is of type DPSTextProc, it can be specified as the text-handling routine (*textProc*) in context-creation routines, such as XDPSCreateSimpleContext. DPSDefaultTextBackstop writes text to WS stdout and flushes stdout.

DPSNewUserObjectIndex

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

The DPSNewUserObjectIndex routine returns a new user object index. All new user object indexes are allocated by the Client Library.

User object indexes are dynamic; do not compute with them or store them in long-term storage, such as in a file.

XDPSContextFromSharedID

```
DPSContext
XDPSContextFromSharedID(/*
    Display *dpy;
    PSContextID cid;
    DPSTextProc textProc;
    DPSErrorProc errorProc */);
```

The XDPSContextFromSharedID routine returns the DPSContext handle of an existing context, specified by PostScript language ID (*cid*) and X display (*dpy*). If the calling client has no such DPSContext, XDPSContextFromSharedID creates a DPSContext and the associated DPSContextRec.

The arguments *textProc* and *errorProc* specify the two routines with which the calling client handles text and errors from the specified context.

XDPSContextFromSharedID lets one client access a context created by another client, thereby letting multiple clients share a single context. When sending names to shared contexts, XDPSContextFromSharedID uses name string encoding.

XDPSContextFromXID

```
DPSContext
XDPSContextFromXID(/*
    Display *dpy;
    XID xid */);
```

The XDPSContextFromXID routine returns the DPSContext handle of an existing context, specified by X resource ID (*xid*) and X display (*dpy*).

XDPSCreateContext

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

The XDPSCreateContext routine creates an execution context and the associated DPSContextRec data structure. It returns a DPSContext handle.

Unlike XDPSCreateSimpleContext, XDPSCreateContext lets

you explicitly specify all characteristics of the context, including its colormap entries. But, unless your application uses color in an unusual way, you need not use XDPSCreateContext; use XDPSCreateSimpleContext instead.

When called, XDPSCreateContext checks whether the X server *dpy* supports a Display PostScript system extension. If not, the routine returns NULL; if so, it checks that the specified drawable and GC exist on the same screen. If they do not, the X server returns a BadMatch error. If they do, XDPSCreateContext creates a PostScript context having the characteristics specified in the arguments passed.

If the argument *drawable* or *GC* is NULL, the created context can receive and execute PostScript language input, but cannot paint images until the calling application specifies an X drawable and GC. (To specify these values, the application must send an X-specific operator, such as setXgcdrawable, described in Chapter 6.)

The following table describes the arguments of XDPSCreateContext:

dpy	An X display.
drawable	An X drawable on *display*.
GC	The X Graphic Context associated with *drawable*.
x and *y*	The horizontal and vertical coordinates (in X units) for the default user space origin of *drawable*. If *x* equals zero and *y* equals the height of *drawable* (in pixels), the default user space origin is at the lower left corner of *drawable*. In the PostScript language, this is the typical location for the default user space origin.
eventmask	Ignored; reserved for future use. Use zero as the value of this argument.
grayramp	(See *ccube*.)
ccube and *graymap*	*ccube* identifies a set of color cells defined as a series of color ramps; *grayramp* identifies a set of color cells defined as a gray ramp. The context uses *ccube* and *grayramp* to produce actual colors and dithered colors. If *ccube* equals NULL, colors are rendered in shades of gray only. If *grayramp* equals NULL, the context does not paint. The gray ramp must have at least two elements: one for black and one for white.

	The X client must allocate and define *ccube* and *grayramp* and must install the associated colormap. In general, if the client specifies a plane mask, *ccube* and *grayramp* should be within the planes selected by the plane mask, to ensure that the Display PostScript system interacts properly with the plane mask. (For more information, see Section 4.9.)
actual	Specifies whether the application prefers to paint with actual (not dithered) colors and, if so, specifies how many actual colors it needs. The *actual* argument is a hint to the X server: dithering and actual color allotment are governed by the X server, not by the application If *actual* equals zero, the application paints by dithering colors from *grayramp* and *ccube*. If *actual* is not zero, the application paints using a maximum of *actual* actual colors; all additional colors are dithered.
textProc	The routine that this context calls to handle text output.
errorProc	The routine that this context calls if it encounters an error condition.
space	The private VM in which this context executes. If *space* is NULL, a new space is created for the context; otherwise, the context shares the specified *space*.

XDPSCreateSimpleContext

```
DPSContext
XDPSCreateSimpleContext(/*
    Display *dpy;
    Drawable drawable;
    GC gc;
    int x,y;
    DPSTextProc textProc;
    DPSErrorProc errorProc;
    DPSSpace space */);
```

The XDPSCreateSimpleContext routine creates an execution context and the associated DPSContextRec data structure. It returns a DPSContext handle.

When called, XDPSCreateSimpleContext checks whether the X server *dpy* supports a Display PostScript system extension. If not, the routine returns NULL; if so, it checks that the specified drawable and GC exist on the same screen. If they do not, the X server returns a BadMatch error. If they do, XDPSCreateSimpleContext creates

a PostScript context having the characteristics specified in the arguments passed.

If the argument *drawable* or *GC* is NULL, the created context can receive and execute PostScript language input, but cannot paint images until the calling application specifies an X drawable and GC. (To specify these values, the application must send an X-specific operator, such as setXgcdrawable, described in Chapter 6.)

The following table describes the arguments of XDPSCreateSimpleContext:

dpy	An X display.
drawabl	An X drawable on *display*.
GC	The X Graphic Context associated with *drawable*.
x and *y*	The horizontal and vertical coordinates (in X units) for the default user space origin of *drawable*. If *x* equals zero and *y* equals the height of *drawable*, the default user space origin is at the lower left corner of *drawable*. In the PostScript language, this is the typical location for the default user space origin.
textProc *errorProc*	The routine that this context calls to handle text output. The routine that this context calls if it encounters an error condition.
space	The private VM in which this context executes. If *space* is NULL, a new space is created for the context; otherwise, the context shares the specified *space*.

Unlike the XDPSCreateContext routine, XDPSCreateSimpleContext does not let you explicitly specify the colormap of the created context, nor does it let you set characteristics of the colormap. Instead, the routine uses standard colormaps as described in the following paragraph.

XDPSCreateSimpleContext accesses the X server *dpy*, and finds out whether the standard colormaps RGB_DEFAULT_MAP and RGB_GRAY_MAP are defined. If they are defined, XDPSCreateSimpleContext uses them; otherwise, the routine defines them.

After these values are defined, any context that the application creates by calling XDPSCreateSimpleContext uses RGB_DEFAULT_MAP and RGB_GRAY_MAP. Note, however, that contexts created by calling XDPSCreateContext use the color cube and gray ramp specified in the call to that routine. For information on explicitly specifying the color characteristics of a context, see the description of

XDPSCreateContext in this chapter.)

XDPSFindContext

```
DPSContext
XDPSFindContext(/*
    Display *dpy;
    long int cid */);
```

The XDPSFindContext routine returns the DPSContext handle of the context whose ID is specified in *cid*.

The argument *cid* is the result returned by an operator such as currentcontext; *dpy* specifies the X display where the context is running.

XDPSRegisterStatusProc

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

void
XDPSRegisterStatusProc (/*
    DPSContext ctxt;
    XDPSStatusProc proc */);
```

The XDPSRegisterStatusProc routine specifies the routine that an application calls to handle status events (XDPSStatusEvent) from the context *ctxt*. That is, XDPSRegisterStatusProc registers, or associates, the XDPSStatusProc event-handling routine *proc* with the context *ctxt*.

The routine *proc* has two arguments: *ctxt* and *code*. The argument *ctxt* specifies the context with which *proc* is registered; *code* shows the status code of the event for which *proc* was called. The client can call *proc* at any time to process status events.

If an XDPSStatusProc routine is already registered with the context *ctxt*, XDPSRegisterStatusProc supersedes the existing registration with the value of *proc*.

XDPSSetStatusMask

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

The XDPSSetStatusMask routine sets the context status mask of the context specified in the argument *ctxt*. (For an explanation of context status and the context status mask, see Section 2.2.3.)

The argument *enableMask* specifies which kinds of context status

events the XDPS server sends to the calling application; *disableMask* specifies the kinds of context status events the server does not send. The argument *nextMask* causes the server to send only the next instance of each specified kind of context status event. The *enableMask*, *disableMask*, and *nextMask* arguments each represent one or more of the values listed in the following code extract:

```
#define PSRUNNINGMASK          0x0001
#define PSNEEDSINPUTMASK       0x0002
#define PSZOMBIEMASK           0x0004
#define PSFROZENMASK           0x0008
```

To assign more than one value to a single argument, perform a bitwise inclusive OR operation (|) on the values you wish to assign, as in the following example:

```
XDPSSetStatusMask(PSRUNNINGMASK | PSNEEDSINPUTMASK,0,0);
```

The following table describes the valid values for *enableMask*, *disableMask*, and *nextMask*:

PSFROZENMASK	Events that show the context is frozen
PSNEEDSINPUTMASK	Events that show the context needs input
PSRUNNINGMASK	Events that show the context is in the runnable state
PSZOMBIEMASK	Events that show the context is in the zombie state

Note that, if an application sends input to a context that is in the zombie state, the application receives a zombie status event, regardless of how the status mask is set.

XDPSSpaceFromSharedID

```
DPSSpace
XDPSSpaceFromSharedID(/*
    Display *dpy;
    SpaceXID sid */);
```

XDPSSpaceFromSharedID returns the DPSSpace handle of an existing private context space, specified by X resource ID (*sid*) and display (*dpy*). If the calling client has no such DPSSpace, XDPSSpaceFromSharedID creates the DPSSpace and associated DPSSpaceRec data structure.

XDPSSpaceFromSharedID lets a context created by one X client share the private space of a context created by another X client. When sending names to shared context whose private space is shared, XDPSSpaceFromSharedID uses ASCII encoding.

XDPSSpaceFromXID

```
DPSSpace
XDPSSpaceFromXID(/*
    Display *dpy;
    XID xid */);
```

XDPSSpaceFromXID returns the DPSSpace pointer of an existing private context space, specified by X resource ID (*sid*) and display (*dpy*).

XDPSUnfreezeContext

```
void
XDPSUnfreezeContext (/*
    DPSContext ctxt */);
```

XDPSUnfreezeContext causes the specified frozen context to resume executing. The argument *ctxt* is the ID of a context whose status is PSFROZEN.

XDPSXIDFromContext

```
XID
XDPSXIDFromContext(/*
    Display **Pdpy;
    DPSContext ctxt */);
```

XDPSXIDFromContext returns the X resource ID of the context whose DPSContext handle is *ctxt*. In addition, the routine returns the argument *Pdpy*, which points to the X Display structure associated with *ctxt*.

XDPSXIDFromSpace

```
XID
XDPSXIDFromSpace(/*
    Display **Pdpy;
    DPSSpace spc */);
```

XDPSXIDFromSpace returns the X resource ID of the context associated with the DPSSpace pointer *spc*. In addition, the routine returns the argument *Pdpy*, which points to the X Display structure associated with *spc*.

X-Specific Operators for WS 6

The Display PostScript system extends the PostScript language to include operators for generic window-related tasks; but for tasks that relate specifically to X, the window system of UWS, additional operators are needed. To fill this need, UWS extends the PostScript language to include X-specific operators.

This chapter describes the X-specific operators for UWS.

The Client Library defines single-operator procedures that execute the X-specific operators. For information on these procedures, see Chapter 5.

6.1 About the Operators

The operators described in the rest of this chapter are arranged alphabetically by operator name. Each description follows this format:

operand1 operandN **operator** *result1 … resultM*

> Text describing what the operator does
>
> EXAMPLE: (Optional)
>
> ```
> Sample PostScript language code showing how to use the
> operator
> ```
>
> ERRORS:
>
> **comma-separated list of errors this operator might execute**

Each operator description begins with a syntax summary. In it, *operand1* through *operandN* are the operands that the operator requires; *operand1* is the top element on the operand stack. A dash (−) in the operand position means the operator accepts no operands.

The operator pops the operands from the stack, and processes them. After executing, the operator pushes *result1* through *resultM* on the stack; *resultM* is the top element. A dash (−) in the result position means the operator returns no results.

Table 6-1 describes the values used as operands and results by the X-specific operators for WS. All operands are required.

Table 6-1: Operands and Results for X-Specific Operators

Name	Type	Description
colorinfo	integer array	Stores color attributes of the context. The 12 elements of *colorinfo* are graymax, graymult, firstgray, redmax, redmult, greenmax, greenmult, bluemax, bluemult, firstcolor, colormapid, and numactual. (For more information, see Section 4.9.4.)
drawable	integer	The X window ID or pixmap ID of an X drawable. If *drawable* equals zero, all drawing operations are ignored.
gc	integer	The GContext resource ID for the X Graphic Context of *drawable*. If *gc* equals zero, all drawing operations are ignored. To obtain a value for *gc*, call the Xlib routine XGContextFromGC, passing the Xlib data type GC of the current Graphic Context as the argument.
red, *green*, and *blue*	float	Three real numbers in the range 0.0 to 1.0 that, together, specify a color (as in the operator setrgbcolor).
success	integer	When nonzero, indicates that the operator completed without error.
x and *y*	integer	The horizontal and vertical coordinates (in X units) for the default user space origin of the current drawable. If *x* equals zero and *y* equals the height of the drawable, the default user space origin is at the lower left corner of the drawable. In the PostScript language, this is the typical location for the default origin.

Note that *drawable*, *gc*, *x*, and *y* are part of the PostScript graphics state, which can be saved and restored using the PostScript language operators gsave and grestore.

6.2 Operator Errors

Table 6-2 describes the errors for the X-specific operators.

Table 6-2: Errors for X-Specific Operators

Error	Probable Cause
`rangecheck`	Bad match: the drawable and GC do not have the same depth, or their visual does not match the colormap associated with the context.
`stackunderflow`	Too few operands on the operand stack.
`typecheck`	Invalid ID for drawable or for GC.
`undefined`	Context not associated with a display device.

6.3 Operator Descriptions

Following is an alphabetical list and description of the X-specific operators for WS. The format for these descriptions is explained in Section 6.1.

— **`clientsync`** —

The `clientsync` operator pauses the current context, sets the status of the context to FROZEN, and causes the X server to return a `PSFROZEN` status event. The context stays frozen until the application calls the Client Library routine `XDPSUnfreezeContext()`. Thus, `clientsync` synchronizes the application with the current context.

One possible use of `clientsync` is to display PostScript language output one page at a time by pausing the current context after each page, as in the following example. This example redefines the operator `showpage`, so that the operator first pauses the current context.

EXAMPLE:

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

ERRORS:

None

— **`currentXgcdrawable`** *`gc drawable x y`*

The `currentXgcdrawable` operator returns the X Graphic Context, drawable, and default user space origin of the current context.

Note that the results returned by `currentXgcdrawable` can be used as the operands of `setXgcdrawable`.

ERRORS:

undefined

— **currentXgcdrawablecolor** *gc drawable x y colorinfo*

The currentXgcdrawablecolor operator returns the GC, drawable, default user space origin, and color attributes of the current context.

Note that the results returned by currentXgcdrawablecolor can be used as the operands of setXgcdrawablecolor.

ERRORS:

undefined

— **currentXoffset** *x y*

The currentXoffset operator returns the default user space origin of the current context.

Note that the results returned by currentXoffset can be used as the operands of setXoffset.

ERRORS:

undefined

red green blue **setrgbXactual** *success*

The setrgbXactual operator allocates a new colormap entry to display the color specified by *red*, *green*, *blue*. If the allocation succeeds (if success is nonzero), future painting of this color uses the new colormap entry instead of dithering from the colorcube.

Note that setrgbXactual does not affect the graphics state. Thus, to paint with the specified color, you must first execute the operator setrgbcolor.

ERRORS:

stackunderflow, **undefined**, **typecheck**

gc drawable x y **setXgcdrawable** —

The setXgcdrawable operator sets the X Graphic Context, drawable, and default user space origin of the current context. The values supplied as operands supersede any existing values for these attributes. The setXgcdrawable operator causes all subsequent operations of the current context to occur in the specified X drawable, with the specified Graphic Context and default user space origin.

To make the effects of `setXgcdrawable` temporary, use it between the operators `gsave` and `grestore`.

ERRORS:

rangecheck, **stackunderflow**, **typecheck**, **undefined**

gc drawable x y colorinfo **`setXgcdrawablecolor`** —

The `setXgcdrawablecolor` operator sets the GC, drawable, default user space origin, and color attributes of the current context.

ERRORS:

rangecheck, **stackunderflow**, **typecheck**, **undefined**

x y **`setXoffset`** —

The `setXoffset` operator sets the default user space origin for the current context.

ERRORS:

stackunderflow, **undefined**

Index

D

E

F

G

H

I

L

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[a]	————	SSB Order Processing – NQO/V19 *or* U. S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

Digital UNIX
Developing Applications for the
Display PostScript System
AA-Q15WB-TE

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: `readers_comment@zk3.dec.com`
- Fax: (603) 881-0120, Attn: UEG Publications, ZKO3-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)				
Completeness (enough information)				
Clarity (easy to understand)				
Organization (structure of subject matter)				
Figures (useful)				
Examples (useful)				
Index (ability to find topic)				
Usability (ability to access information quickly)				

Please list errors you have found in this manual:

Page	Description

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using?

Name/Title Dept.

Company Date

Mailing Address

Email Phone

Do Not Cut or Tear − Fold Here and Tape

digital™

NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST−CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UEG PUBLICATIONS MANAGER
ZKO3−3/Y32
110 SPIT BROOK RD
NASHUA NH 03062−9987

Do Not Cut or Tear − Fold Here

Cut on
Dotted
Line