ULTRIX Worksystem Software

digital

Display PostScript® Client Library
Reference Manual

Order Number: AA–PAN8A–TE

DISPLAY POSTSCRIPT
SYSTEM

Client Library Reference Manual

Order No. AA–PAN8A–TE

ADOBE SYSTEMS
INCORPORATED

Client Library Reference Manual

Writer: Amy Davidson

October 25, 1989

This manual replaces the previous version dated October 7, 1988.

Contents

List of Figures

1 ABOUT THIS MANUAL

This manual provides the application programmer with descriptions of Client Library procedures and conventions; these constitute the programming interface to the DISPLAY POSTSCRIPT® system. The sections of the manual are listed below:

- Section 2 introduces the Client Library and provides a diagram of its relationship to the DISPLAY POSTSCRIPT system.
- Section 3 provides a brief overview of the Client Library; describes the phases of an application program interacting with the DISPLAY POSTSCRIPT system; introduces the C header files that represent the Client Library interface; and discusses the use of wrapped procedures.
- Section 4 describes the basic concepts an application programmer needs to know before writing a simple application for the DISPLAY POSTSCRIPT system.
- Section 5 discusses call-back procedures of various kinds, including text and error handlers.
- Section 6 contains advanced Client Library concepts including context chaining, encoding and translation, buffering, application/context synchronization, and forked contexts.
- Section 7 provides programming tips and summarizes notes and warnings.
- Section 8 lists and documents an application program that illustrates how to communicate with the DISPLAY POSTSCRIPT system using the Client Library.
- Section 9 documents the basic Client Library data structures and procedures found in *dpsclient.h.*
- Section 10 describes the single-operator procedures that implement POSTSCRIPT® operators and lists the *dpsops.h* header file in which they are declared.
- Section 11 describes the *dpsfriends.h* header file and its support of C-callable procedures produced by the *pswrap* translator.
- Appendix A lists changes to the manual since the last revision.

- Appendix B provides an example error handler for the X Window System™ implementation of the DISPLAY POSTSCRIPT system.
- Appendix C describes how an application can recover from POSTSCRIPT language errors and provides an example of an exception handler.

For more information about the POSTSCRIPT language, see the *POSTSCRIPT Language Reference Manual* and *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System.* For more information about using the *pswrap* translator to embed POSTSCRIPT language code in C programs, see the *pswrap Reference Manual.*

1.1 SYSTEM-SPECIFIC DOCUMENTATION

The term "system specific" is used throughout this manual. It refers to areas of the Client Library implementation that are necessarily customized to fit a given machine and operating-system environment. The *Client Library Reference Manual* describes those aspects of the Client Library that are common to all DISPLAY POSTSCRIPT system implementations.

You will find notes and comments in this manual alerting you to system-specific issues. For more information about these system-specific aspects of your Client Library implementation, see the documentation provided by your DISPLAY POSTSCRIPT system vendor.

1.2 TYPOGRAPHICAL CONVENTIONS

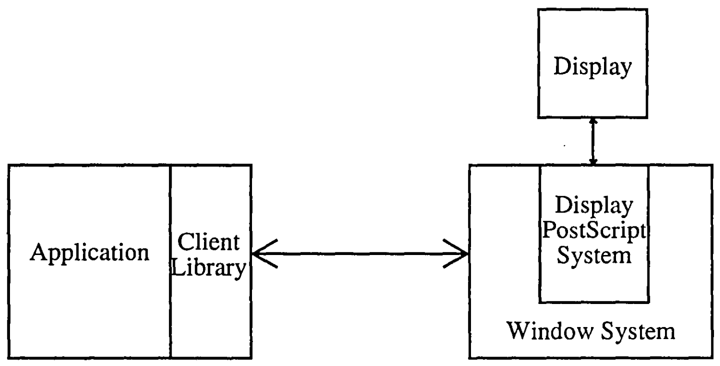
The typographical conventions used in this manual are as follows:

Item	*Example of Typographical Style*
file	*dpsclient.h*
variable, typedef, code fragment	'ctxt', 'DPSContextRec', 'DPSrectstroke(ctxt, 0.0, 0.0, 10.0, 20.0)'
procedure	*DPSSetContext*
POSTSCRIPT operator	**rectfill**
new term	"A *wrapped procedure* (*wrap* for short) consists of ...."

2 ABOUT THE CLIENT LIBRARY

The Client Library is the application programmer's link to the DISPLAY POSTSCRIPT system, which makes the imaging power of the POSTSCRIPT interpreter available for online displays as well as for printing. An application program can display text and images on the user's screen by calling Client Library procedures. These procedures are written with a C language interface. They generate POSTSCRIPT language code and send it to the POSTSCRIPT interpreter for execution, as shown in Figure 1.

Figure 1 *The Client Library Link to the DISPLAY POSTSCRIPT System*



Application programmers can customize and optimize their applications by writing POSTSCRIPT language programs. The *pswrap* translator, described in the *pswrap Reference Manual*, produces application-defined POSTSCRIPT language programs with C-callable interfaces.

Note: In this manual, the terms ''input'' and ''output'' apply to the execution context in the POSTSCRIPT interpreter, not to the application. An application ''sends input'' to a context and ''receives output'' from a context. This usage prevents the ambiguity that would otherwise exist, since input with respect to the context is output with respect to the application, and vice versa.

3 OVERVIEW OF THE CLIENT LIBRARY

The Client Library is a collection of procedures that provide an application program with access to the POSTSCRIPT interpreter. The Client Library includes procedures for creating, communicating with, and destroying POSTSCRIPT execution contexts. A context consists of all the information (or "state") needed by the POSTSCRIPT interpreter to execute a POSTSCRIPT language program. In the Client Library interface, each context is represented by a 'DPSContextRec' data structure pointed to by a 'DPSContext' handle. POSTSCRIPT execution contexts are described in *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System.*

To the application programmer, it appears that Client Library procedures directly produce graphical output on the display. In fact, these procedures generate POSTSCRIPT language statements and transmit them to the POSTSCRIPT interpreter for execution; the POSTSCRIPT interpreter then produces graphical output that is displayed by device-specific procedures in the DISPLAY POSTSCRIPT system. In this way, the Client Library makes the full power of the POSTSCRIPT interpreter and imaging model available to a C language program.

The recommended way of sending POSTSCRIPT language code to the interpreter is to call wrapped procedures generated by the *pswrap* translator; these procedures are described in Section 3.3. For simple operations, an application program can send POSTSCRIPT language fragments to the interpreter by calling single-operator procedures — each one the equivalent of a single POSTSCRIPT operator — as described in Section 10. It is also possible for an application program to send POSTSCRIPT language as ASCII text, as if to a laser printer with a POSTSCRIPT interpreter; this technique can be used for development and debugging.

3.1 PHASES OF AN APPLICATION

Here is how a typical application program, written in C, uses the Client Library in the different phases of its operation:

Initialization. First, the application establishes communication with the DISPLAY POSTSCRIPT system. Then it calls Client Library procedures to create a context for executing POSTSCRIPT language programs. It also performs other window-system-specific initialization. Some higher-level facilities, such as toolkits, do all of this initialization automatically.

Execution. Once an application is initialized, it displays text and graphics by sending POSTSCRIPT language programs to the interpreter. These programs may be of any complexity from a single-operator procedure to a program that previews a full-color illustration. The Client Library sends the programs to the POSTSCRIPT interpreter and handles the results received from the interpreter.

Termination. When the application is ready to terminate, it calls Client Library procedures to destroy its contexts, free their resources, and end the communication session.

3.2 HEADER FILES

The Client Library procedures that an application can call are defined in C header files, also known as include files or interface files. There are four Client Library-defined header files and one or more system-specific header files. The Client Library interface represented by these header files may be extended in a given implementation, but the extensions are compatible with the definitions given in this manual.

- *dpsclient.h* provides support for managing contexts and sending POSTSCRIPT language programs to the interpreter. It supports applications as well as application toolkits. Always present.
- *dpsfriends.h* provides support for wrapped procedures created by *pswrap* as well as data representations, conversions, and other low-level support for context structures. Always present.
- *dpsops.h* provides the single-operator procedures that require an explicit context parameter. Optional; at least one

single-operator header file must be present; that is, *dpsops.h* or *psops.h* or both.

- *psops.h* provides the single-operator procedures that implicitly derive their context parameter from the current context. Optional; see *dpsops.h*.
- One or more *system-specific header files* provide support for context creation. These header files may also provide system-specific extensions to the Client Library, such as additional error codes.

3.3 WRAPPED PROCEDURES

The most efficient way for an application program to send POSTSCRIPT language to the interpreter is to use the *pswrap* translator to produce *wrapped procedures* — that is, POSTSCRIPT language programs that are callable as C procedures. A *wrapped procedure* (*wrap* for short) consists of a C language procedure declaration enclosing a POSTSCRIPT language body. There are several advantages to using wraps:

- Complex POSTSCRIPT programs can be invoked by a single procedure call, avoiding the overhead of a series of calls to single-operator procedures.
- You can insert C arguments into the POSTSCRIPT language code at runtime instead of having to push the C arguments onto the POSTSCRIPT operand stack in separate steps.
- Wrapped procedures can efficiently produce custom graphical output by combining operators and other elements of the POSTSCRIPT language in a variety of interesting ways.
- The POSTSCRIPT language code sent by a wrapped procedure is interpreted faster than ASCII text.

An application developer prepares a POSTSCRIPT language program for inclusion in the application by writing a wrap and passing it through the *pswrap* translator. The output of *pswrap* is a procedure written entirely in the C language. It contains the POSTSCRIPT language body as data. This body has been compiled into a *binary object sequence* (an efficient binary encoding), with placeholders left for arguments to be inserted at execution time. The translated wraps can then be compiled and linked into the application program.

When a wrapped procedure is called by the application, the procedure's arguments are substituted for the placeholders in the POSTSCRIPT language body of the wrap.

Example: A wrap that draws a black box could be defined as follows:

```
defineps PSWBlackBox(float x, y)
    gsave
     0 0 0 setrgbcolor
     x y 72 72 rectfill
    grestore
endps
```

pswrap produces a procedure that can be called from a C language program as follows (the values shown are merely examples):

```
PSWBlackBox(12.32, -56.78);
```

This procedure replaces the *x* and *y* operands of **rectfill**with the corresponding procedure arguments, producing executable POSTSCRIPT language code:

```
gsave
 0 0 0 setrgbcolor
 12.32 -56.78 72 72 rectfill
grestore
```

Any wrapped procedure works the same way as the above example: the arguments of the C language procedure must correspond in number and type to the operands expected by the POSTSCRIPT operator(s) in the body of the wrap. For instance, a procedure argument declared to be of type 'float' corresponds to a POSTSCRIPT real object; an argument of type 'char *' corresponds to a POSTSCRIPT string object; and so on.

The normal outcome of calling a wrapped procedure is the transmission of POSTSCRIPT language code to the interpreter for execution, normally resulting in display output. The Client Library may also provide means, on a system-specific basis, to divert transmission to another destination, such as a printer or a text file.

For more information about how wraps are defined and used, see the *pswrap Reference Manual.*

4 BASIC CLIENT LIBRARY FACILITIES

This section introduces the concepts needed to write a simple application program for the DISPLAY POSTSCRIPT system, including:

- Creating a context.
- Sending code and data to a context.
- Destroying a context.

The basic facilities provided by the Client Library to application programs are described in this section.

The Client Library procedures and data structures that are referred to in this introduction are documented in the following places:

Section 9.	Header file *dpsclient.h*. Provides general support for contexts; includes procedures that send POSTSCRIPT language programs for execution and receive results. General applications and application support software (that is, toolkits) make use of this header file.
Section 10.	Header files *dpsops.h* and *psops.h*. Declarations for single-operator procedures.
System-specific documentation.	Support for creating context records. An example of context creation is provided in Section 4.3.

4.1 CONTEXTS AND CONTEXT DATA STRUCTURES

An application creates, manages, and destroys one or more contexts. A typical application creates a single context in a single private VM (space). It then sends POSTSCRIPT language code to the context to display text, graphics, and scanned images on the screen.

The context is represented by a record of type 'DPSContextRec'; see Section 9.1 for the type definition. A handle to this record — a pointer of type 'DPSContext' — is passed explicitly or implicitly with every Client Library procedure call. In essence, to the application programmer, the 'DPSContext' handle *is* the context.

A context can be thought of as a destination to which POSTSCRIPT language code is sent. The destination is set when the context is created. In most cases, the code draws graphics in a window or specifies how a page will be printed. Other possible destinations include a file (for execution at a later time) or the standard output; multiple destinations are permitted. The execution by the interpreter of POSTSCRIPT language code sent to a context may be immediate or deferred, depending on which context creation procedure was called and on the setting of certain 'DPSContextRec' variables.

4.2 SYSTEM-SPECIFIC CONTEXT CREATION

The system-specific interface[1] contains, at minimum, procedures for creating the 'DPSContextRec' record for the given implementation of the Client Library. The system-specific interface also provides support for certain extensions to the Client Library interface, such as additional error codes.

Every context is associated with a system-specific object such as a window or a file. The context is created by calling a procedure in the system-specific interface. Once the context has been created, however, a set of standard Client Library operations may be applied to it; these operations, including context destruction, are defined in the standard header file *dpsclient.h.*

4.3 EXAMPLE OF CONTEXT CREATION

Context creation facilities are necessarily system specific. This is because they often need data objects that represent system-specific entities such as windows and files. However, most context creation facilities share a number of common attributes. In the text that follows, procedure parameters that are common to most systems are described in some detail, while system-specific parameters are listed without further discussion. The procedures

[1]In Adobe's sample X11/DPS extension implementation, the system-specific header file is *dpsXclient.h.*

described here were designed for the X Window System. They provide an example of an actual system implementation while at the same time demonstrating basic functions that all window systems must provide for context creation.

The creation of a 'DPSContextRec' data structure is usually part of application initialization. Contexts persist until they are destroyed; see *DPSDestroyContext* and *DPSDestroySpace* in Section 9.2.

```
/* EXAMPLE CONTEXT CREATION FOR THE X WINDOW SYSTEM */
DPSContext XDPSCreateSimpleContext(dpy, drawable, gc, x, y, textProc, errorProc, space)
  Display *dpy;
  Drawable drawable;
  GC gc;
  int x, y;
  DPSTextProc textProc;
  DPSErrorProc errorProc;
  DPSSpace space;

typedef void (*DPSTextProc)( /*
  DPSContext ctxt,
  char *buf,
  long unsigned int count */ );

typedef void (*DPSErrorProc)( /*
  DPSContext ctxt,
  DPSErrorCode errorCode,
  long unsigned int arg1, arg2 */ );
```

XDPSCreateSimpleContext is a system-specific procedure that creates an execution context in the POSTSCRIPT interpreter. The arguments 'dpy', 'gc', 'x', and 'y' have specific uses in the X Window System; discussion of these arguments is beyond the scope of this manual. The 'drawable' argument associates the 'DPSContextRec' data structure with a system-specific imaging object — in this case, an X drawable object, which could be a window or a pixmap. 'DPSTextProc' and 'DPSErrorProc' are standard procedures types declared in *dpsclient.h*; their type definitions are included here for ease of reading.

'space' identifies the private POSTSCRIPT VM in which the new context executes. If 'space' is 'NULL', a new space is created for the context; otherwise, it will share the specified space with contexts previously created in the space. A simple application that creates one space and one context can pass 'NULL' for the 'space' argument. See the *POSTSCRIPT Language Reference Manual* for a definition of VM. See Section 4.6 for more information about spaces.

'textProc' and 'errorProc' point to customizable facilities for handling text and errors sent by the interpreter. Passing 'NULL' for these arguments is allowed but means that text and errors are ignored. For simple applications, it is sufficient to specify the system-specific default text procedure (*DPSDefaultTextBackstop* in the X Window System implementation) and *DPSDefaultErrorProc*. Use *DPSGetCurrentTextBackstop* to get the current default text procedure. See Section 5 for more information on text handlers and error handlers.

XDPSCreateSimpleContext creates a context for which the POSTSCRIPT interpreter is the destination of code and data sent to the context. It is sometimes useful to send the code and data elsewhere, such as to a file, to a terminal (UNIX® *stdout*), or to a printer; see *DPSCreateTextContext*.

```
DPSContext DPSCreateTextContext(textProc, errorProc)
  DPSTextProc textProc;
  DPSErrorProc errorProc;
```

DPSCreateTextContext creates a context whose input is converted to ASCII encoding (text that is easily transmitted and easily read by humans); see Section 6.2. The ASCII-encoded text is passed to the 'textProc' procedure rather than to the POSTSCRIPT interpreter. Since the application provides the implementation of the 'textProc' procedure, it determines where the ASCII text goes from there. The text can be sent to a file, to a terminal, or perhaps to a printer's communication port.

The 'errorProc' associated with a context handles errors that arise when a wrap or Client Library procedure is called with that context. Call the 'errorProc' to handle an error only when an appropriate error code is defined. See the discussion of text and error handlers in Section 5.

4.4 THE CURRENT CONTEXT

The current context is the one that was specified by the last call to *DPSSetContext*. If the application has only one context, *DPSSetContext* may be called once when the application is initialized. If the application manages more than one context, it must explicitly set the current context when necessary.

Many Client Library procedures do not require the application to specify a context; they assume the current context. This is true of all of the single-operator procedures defined in the *psops.h* header file as well as any wrapped procedures that were defined to use the current context implicitly.

An application can find out which is the current context by calling *DPSGetCurrentContext*.

4.5 SENDING CODE AND DATA TO A CONTEXT

Once the context has been created, the application can send POSTSCRIPT language code to it by calling procedures such as:

- Wraps (custom wrapped procedures developed for the application).
- Single-operator procedures defined in *dpsops.h* and *psops.h*.
- *DPSPrintf*, *DPSWritePostScript*, and *DPSWriteData* — Client Library procedures provided for writing to a context.

A wrapped procedure is a POSTSCRIPT language program encoded as a binary object sequence; binary object sequences are described in Section 11.4 and in *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System*. The creation of wrapped procedures is discussed in the *pswrap Reference Manual*. Once the POSTSCRIPT language program has been embedded in the body of a wrap by using the *pswrap* translator, it can be called like any other C procedure.

Example 1: Consider a wrap that draws a small colored circle around the point where the mouse was clicked, given an RGB color and the *x,y* coordinate returned by a mouse-click event. The exact POSTSCRIPT language implementation is left as an exercise for the reader, but the C declaration of the wrap might look like this:

```
extern void PSWDrawSmallCircle(/*
   DPSContext ctxt; int x, y; float r, g, b */);
```

An application might call this procedure as part of the code that handles mouse clicks. Suppose the struct 'event' contains the *x,y* coordinate. To draw a bright green circle around the spot, call the wrapped procedure with the following arguments:

```
PSWDrawSmallCircle(ctxt, event.x, event.y, 0.0, 1.0, 0.0);
```

Example 2: If a wrap returns values, the procedure that calls it must pass pointers to the variables into which the values will be stored. Consider a wrap that, given a font name, tells whether the font is in the **SharedFontDirectory**. Define the wrap like this:

```
defineps PSWFontLoaded(
   DPSContext ctxt; char *fontName | boolean *found)
```

The corresponding C declaration is:

```
extern void PSWFontLoaded(/*
   DPSContext ctxt; char *fontName; int *found */);
```

Note that booleans are of C type 'int'. Call the wrapped procedure by providing a pointer to a variable of type 'int':

```
int fontFound;

PSWFontLoaded(ctxt, "Courier", &fontFound);
```

Wraps are the most efficient way to specify any POSTSCRIPT language program as a C-callable procedure.

Example 3: Occasionally, a very small POSTSCRIPT language program — on the order of one operator — is needed. This is a case where a single-operator procedure is appropriate. For example, to get the current gray level, simply provide a pointer to a float and call the single-operator procedure equivalent of the POSTSCRIPT **currentgray** operator:

```
float gray;

DPScurrentgray(ctxt, &gray);
```

See Section 10.3 for a complete listing of single-operator procedure declarations.

Example 4: *DPSPrintf* is one of the Client Library facilities provided for writing POSTSCRIPT language code directly to a context.

DPSPrintf is similar to the Standard C Library routine *printf*. It formats arguments into ASCII text and writes this text to the context. Small POSTSCRIPT language programs or text data may be sent in this way. Here is an example that sends formatted text to the **show** operator to represent an author's byline:

```
struct {
    int x, y;            /* location on page for byline */
    char *titleString;   /* title of document */
    char *authorsName;   /* name of author */
    } byline;

DPSPrintf(ctxt, "%d %d moveto (%s by %s) show\n",
        byline.x,
        byline.y,
        byline.titleString,
        byline.authorsName);
```

The *x,y* coordinate is formatted in place of the two '%d' field specifiers, the title replaces the first '%s', followed by ''by'' followed by the author's name in place of the second '%s'.

Warning: When using *DPSPrintf*, it's important to leave some whitespace (newline with '\n', or just a space) at the very end of the format string if the string ends with an operator. POSTSCRIPT language code written to a context appears as a continuous stream. Thus, consecutive calls to *DPSPrintf* will appear as if all the text were sent at once. For example, suppose the following calls were made:

```
DPSPrintf(ctxt, "gsave");
DPSPrintf(ctxt, "stroke");
DPSPrintf(ctxt, "grestore");
```

The context will receive a single string 'gsavestrokegrestore', with all the operators run together. Of course, this effect may be useful for constructing a long string that isn't a part of a program. But when sending operators to be executed, don't forget to put whitespace at the end of each format string; for example:

```
DPSPrintf(ctxt, "gsave\n");
```

Example 5: The *DPSWritePostScript* procedure is a facility provided for writing POSTSCRIPT language of any encoding to a context. If *DPSChangeEncoding* is provided by the system-specific interface, *DPSWritePostScript* can be used to convert a binary-encoded POSTSCRIPT language program into another binary form (for instance, binary object sequences to binary-encoded tokens) or into ASCII text. Code destined for immediate execution by the interpreter should be sent as binary object sequences. Code that's intended to be read by a human should be sent as ASCII text. See Section 6.2 for a discussion of language encodings.

Warning: Although PostScript language of any encoding may be written to a context, unexpected results can occur when intermixing code of different encodings. This is particularly important when ASCII encoding is mixed with binary encoding. (See *PostScript Language Extensions for the Display PostScript System* for a discussion of encodings.)

The following code, which looks correct, may fail with a syntax error in the interpreter, depending on the contents of the buffer:

```
while (/* more buffers to send */) {
    count = GetBuffer(file, buffer);
    DPSWritePostScript(ctxt, buffer, count);
    MyWrap(ctxt);
}
```

GetBuffer reads a PostScript language program in the ASCII encoding from a file. The call to *MyWrap* generates a binary object sequence. If the program in the buffer passed to *DPSWritePostScript* is complete, with no partial tokens, *MyWrap* works correctly. Imagine, however, that the end of the buffer contains a partial token, 'mov', and the next buffer starts with 'eto'. In this instance, the binary object sequence representing *MyWrap* will be inserted immediately after the partial token, resulting in a syntax error.

This warning applies to all procedures that send code or data to a context, including the Client Library procedures *DPSPrintf*, *DPSWritePostScript*, *DPSWriteData*, and *DPSWaitContext*.

Example 6: To send any type of data to a context (such as hexadecimal image data), or to avoid the automatic conversion behavior built into *DPSWritePostScript*, use *DPSWriteData*. See Section 9.2 for details on *DPSWritePostScript* and *DPSWriteData*.

The following example reads hexadecimal image data line by line from a file and sends the data to a context:

```
while (!feof(fp)) {
   fgets(buf, BUFSIZE, fp);
   DPSWriteData(ctxt, buf, strlen(buf));
}
```

4.6 SPACES

A context is created in a space. The space is either shared with a previously created context or is created when a new context is created. Multiple contexts in the same space share all data; careful coordination is required to ensure that they don't interfere with each other. Contexts in different spaces can operate more or less independently and still share data by using *shared VM*. See the discussion of VM and spaces in the *POSTSCRIPT Language Reference Manual*.

Destroying a space automatically destroys all of the contexts within it. *DPSDestroySpace* calls *DPSDestroyContext* for each context in the space.

The parameters that define a space are contained in a record of type 'DPSSpaceRec'.

4.7 INTERRUPTS

An application may need to interrupt a POSTSCRIPT language program running in the POSTSCRIPT interpreter. Call *DPSInterruptContext* for this purpose. (Note that although this procedure returns immediately, an indeterminate amount of time may pass before execution is actually interrupted.)

An interrupt request causes the context to execute an **interrupt** error. Since the implementation of the **interrupt** error can be changed by the application, the exact results of requesting an interrupt cannot be defined here. The default behavior is that the **stop** operator will execute. For a discussion of the **interrupt** error, see the *POSTSCRIPT Language Reference Manual*; for a discussion of error handling in the Client Library, see Section 5.4.

4.8 DESTROYING CONTEXTS

An application should destroy all the contexts it creates by calling *DPSDestroyContext* or *DPSDestroySpace* when they are no longer needed. Destroying a context does not destroy the space it occupies, but destroying a space destroys all of its contexts; see Section 4.6.

If an application terminates abnormally, the POSTSCRIPT interpreter detects that the application has terminated and destroys any spaces and contexts that the application had created.

5 HANDLING OUTPUT FROM THE CONTEXT

Output is information returned from the POSTSCRIPT interpreter to the application. In the DISPLAY POSTSCRIPT system, three kinds of output are possible:

- Output parameters (results) from wrapped procedures.
- ASCII text written by the context (for example, by the **print** operator).
- Errors.

Each kind of output is handled by a separate mechanism in the Client Library. The handling of results is discussed in Section 11. The handling of text and errors is discussed in the remainder of this section.

Note: You may not get text and error output when you expect it.

For example, a wrap that generates text to be sent back to the application (for instance, with the **print** operator) may return before the application actually receives the text. Unless the application and the interpreter are synchronized (see Section 6.4), the text may not appear until some other Client Library procedure or wrap is called. This is due to delays in the communication channel or delays in scheduling execution of the context in the POSTSCRIPT interpreter.

These kinds of delays are a particularly important consideration for handling errors, since the notification of the error may be received by the application long after the code that caused the error was sent.

Keep these issues in mind while reading the remainder of Section 5.

5.1 CALL-BACK PROCEDURES

The application programmer must specify call-back procedures to handle text and errors. A *call-back procedure* is code provided by an application and called by a system function.

A *text handler* is a call-back procedure that handles text output from the context. It is specified in the 'textProc' field of the 'DPSContextRec'. A system-specific default text handler may be provided; in the DISPLAY POSTSCRIPT system extension for the X Window System, the default text handler is *DPSDefaultTextBackstop*.

An *error handler* is a call-back procedure that handles errors arising when the context is passed as a parameter to any Client Library procedure or wrap. It is specified in the 'errorProc' field of the 'DPSContextRec'. *DPSDefaultErrorProc* is the default error handler provided with every Client Library implementation.

Text and error handlers are associated with a given context when the context is created, but the *DPSSetTextProc* and *DPSSetErrorProc* procedures, described in Section 9.2, give the application the flexibility to change these handlers at any time.

Using a call-back procedure reverses the normal flow of control, which is as follows:

- An application that is active calls the system to provide services; for example, to get memory or open a file.
- The application then gives up control until the system has provided the service.
- The system procedure returns control to the application, passing it the result of the service that was requested.

In the case of call-back procedures, the application wants a custom service provided at a time when it is not in control. It does this as follows:

- The application notifies the system, often but not necessarily at initialization time, of the address of the call-back procedure to be invoked when the system recognizes a certain condition, say, an error condition.
- When the error is raised, the system gets control.
- The system passes control to the error handler specified by the application — thus ''calling back'' the application.
- The error handler does processing on behalf of the application.

- When the error handler completes, it returns not to the application but to the system.

In the DISPLAY POSTSCRIPT system, the text and error handlers in the Client Library interface are designed to be used this way.

Note: Client Library procedures and wraps should not be called from within a call-back procedure. This restriction protects the application against unintended recursion.

5.2 TEXT HANDLERS

A context generates text output with operators such as **print**, **writestring**, and ==. The application handles this text output with a text handler, which is specified in the 'textProc' field of the 'DPSContextRec'. The text handler is passed a buffer of text and a count of the number of characters in the buffer; what is done with this buffer is up to the application. The text handler may be called several times to handle large amounts of text. Note that the Client Library just gets buffers; it doesn't provide any logical structure for the text and it doesn't indicate (or know) where the text ends.

The text handler may be called as a side effect of calling a wrap, a single-operator procedure, or a Client Library procedure that takes a context. You can't predict when the text handler for a context will be called unless the application is synchronized (see Section 6.4).

5.3 EXAMPLE TEXT HANDLER

Consider an application that normally displays a log window to which it appends plain text or error messages received from the interpreter. The handlers for this window were associated with the context when it was created. Occasionally, the application calls a wrapped procedure that generates a block of text intended for a file. Before calling the text-generating procedure, the application must install a temporary text handler for its output. The temporary text handler stores the text it receives in a file instead of in the log window. When the text-generating procedure completes, the application restores the original text handler.

An example of such an application, written for the X Window System, is shown below.

```
/* EXAMPLE TEXT HANDLER FOR AN X WINDOW SYSTEM APPLICATION */

/* wrapped procedure that generates text */

defineps WrapThatGeneratesText(DPSContext ctxt | boolean *done)
    % send a text representation of the contents of mydict
    mydict {== ==} forall
    % returning a value flushes output as a side-effect
    true done
endps

/* normal text proc appends to a log window */

void LogTextProc(ctxt, buf, count)
  DPSContext ctxt;
  char *buf;
  long unsigned int count;
{
/* ... code that appends text to a log window ... */
}

/* special text proc stores text to a file */

void StoreTextProc(ctxt, buf, count)
  DPSContext ctxt;
  char *buf;
  long unsigned int count;
{
    /* ... code that appends text to a file ... */
}

/* application initialization */

    ctxt = XDPSCreateSimpleContext(dpy, drawable, gc, x, y,
            LogTextProc, DPSDefaultErrorProc, NULL);

/* main loop for application */

    while (XPending(dpy)) > 0 {
        /* get an input event */
        XNextEvent(dpy, &event);
        /* react to event */
        switch (event.type) {
            /* any text that comes from processing EVENT_A or EVENT_B is logged */
            case EVENT_A: ...
            case EVENT_B: ...
            /* but EVENT_C means store the text in a file */
            case EVENT_C: {
```

```
            int done;
            DPSTextProc tmp = ctxt -> textProc;

            /* make sure interpreter is ready */
            DPSWaitContext(ctxt);
            /* temporarily install the other text proc */
            DPSSetTextProc(ctxt, StoreTextProc);
            /* call the wrapped procedure */
            WrapThatGeneratesText(ctxt, &done);
            /* since wrap returned a value, we know the interpreter is
               ready when we get here; restore original textProc */
            DPSSetTextProc(ctxt, tmp);
            /* close file by calling textProc with count = 0 */
            StoreTextProc(ctxt, NULL, 0);
            break;
        }
        /* ... */
        default:;
    }
}
```

5.4 ERROR HANDLERS

The 'errorProc' field in the 'DPSContextRec' contains the address of a call-back procedure for handling errors. The error call-back procedure is called when there is a POSTSCRIPT language error or when an error internal to the Client Library, such as use of an invalid context identifier, is encountered. The standard error codes are listed under *DPSErrorProc* in Section 9.2.

When the interpreter detects a POSTSCRIPT language error, it invokes the standard **handleerror** procedure to report the error, then forces the context to terminate. The error call-back procedure specified in the 'DPSContextRec' is called with the 'dps_err_ps' error code.

After a POSTSCRIPT language error, the context becomes invalid; further use of it will cause another error. See Section 5.5 for a discussion of error recovery issues. See Appendix B for an example of an error handler. See the Note on page 21 for a discussion of when error output is actually received.

5.5 ERROR RECOVERY REQUIREMENTS

For many applications, error recovery may not be considered an issue because an unanticipated POSTSCRIPT language error or Client Library error represents a bug in the program that will be fixed during development. However, since applications do sometimes go into production with undiscovered bugs, it is prudent to provide an error handler that allows the application to exit gracefully.

There are a small number of applications that require error recovery more sophisticated than simply exiting. If an application falls into one of the following categories, it is likely that some form of error recovery will be needed:

- Applications that read and execute POSTSCRIPT language programs generated by other sources (for example, a previewer application for POSTSCRIPT language documents generated by a word-processing program). Since the externally provided POSTSCRIPT language program may have errors, the application must provide some sort of error recovery.
- Applications that allow the user to enter POSTSCRIPT language programs. This category is a subset of the one above.
- Applications that generate POSTSCRIPT language programs dynamically in response to user requests (for example, a graphics art program that generates an arbitrarily long path description of a graphical object). Since there are system-specific resource limitations on the interpreter, such as memory and disk space, the application should be able to back away from an error caused by exhausting a resource, and perhaps attempt to acquire new or reclaim used resources.

Error recovery is complicated because both the Client Library and the context can be left in unknown states. For example, the operand stack may have unused objects on it.

In general, if an application needs to intercept and recover from POSTSCRIPT language errors, keep it simple. For some applications, the best strategy when an error occurs is either to destroy the space and construct a new one with a new context or to restart the application.

A given implementation of the Client Library may provide more sophisticated error recovery facilities; consult your system-specific documentation. Your system may provide the general-purpose exception-handling facilities described in Appendix C, which can be used in conjunction with *DPSDefaultErrorProc*.

5.6 BACKSTOP HANDLERS

Backstop handlers handle output when there is no other appropriate handler. The Client Library automatically installs backstop handlers.

To get a pointer to the current backstop text handler, call *DPSGetCurrentTextBackstop*. To install a new backstop text handler, call *DPSSetTextBackstop*. The text backstop may be used as a default text handler implementation. The exact definition of what the default text handler does is system specific. For instance, for UNIX systems, it writes the text to *stdout*.

To get a pointer to the current backstop error handler, call *DPSGetCurrentErrorBackstop*. To install a new backstop error handler, call *DPSSetErrorBackstop*. The backstop error handler processes errors internal to the Client Library, such as a lost server connection. These errors have no specific 'DPSContext' handle associated with them and therefore have no error handler.

6 ADDITIONAL CLIENT LIBRARY FACILITIES

The Client Library includes a number of utilities and support functions for applications. This section describes:

- Sending the same code and data to a group of contexts by chaining them.
- Encoding and translating POSTSCRIPT language code.
- Buffering and flushing the buffer.
- Synchronizing an application with a context.
- Communicating with a forked context.

6.1 CHAINED CONTEXTS

It is sometimes useful to send the same POSTSCRIPT language program to several contexts. This is accomplished most conveniently by chaining the contexts together and sending input to one context in the chain; for example, by calling a wrap with that context.

Two Client Library procedures are provided for managing context chaining:

- *DPSChainContext* links a context to a chain.
- *DPSUnchainContext* removes a child context from its parent's chain.

One context in the chain is specified as the parent context, the other as the child context. The child context is added to the parent's chain. Subsequently, any input sent to the parent is sent to its child, and the child of the child, and so on. A context can appear on only one chain. If the context is already a child on a chain, *DPSChainContext* returns a nonzero error code. However, you can chain a child to a context that already has a child.

Note: A parent context always passes its input to its child context. However, for a chain of more than two contexts, the order in which the contexts on the chain receive the input is not defined. Therefore an application should not rely on *DPSChainContext* to create a chain whose contexts process input in a particular order.

For chained contexts, output is handled differently from input, and text and errors are handled differently from results. If a context on a chain generates text or error output, the output is handled by that context only. Such output is not passed to its parent or its child. When a wrap that returns results is called, all of the contexts on the chain get the wrap code (the input), but only the context with which the wrap was called receives the results.

The best way to build a chain is to identify one context as the parent. Call *DPSChainContext* to make each additional context the child of that parent. For example, to chain contexts A, B, C, and D, choose A as the parent and make the following calls to *DPSChainContext*:

```
DPSChainContext(A,B);
DPSChainContext(A,C);
DPSChainContext(A,D);
```

Once the chain is built, send input only to the designated parent, A.

The most common use of chained contexts is in debugging. A log of POSTSCRIPT operators executed may be kept by a child context whose purpose is to convert POSTSCRIPT language programs to ASCII text and write the text to a file; this child is chained to a parent context that sends normal application requests to the interpreter. The parent's calls to wrapped procedures will then be logged in human-readable form as a debugging audit trail.

Chained contexts may also be used for duplicate displays. An application may want several windows, or even several different display screens, to show the same graphics without having to explicitly call the wrapped procedure in a loop for all of the contexts.

6.2 ENCODING AND TRANSLATION

POSTSCRIPT language code may be sent to a context in three ways:

- As a binary object sequence — typically for immediate execution on behalf of a context.
- As binary-encoded tokens — typically for deferred execution from a file.
- As ASCII text — typically for debugging, display, or deferred execution from a file.

POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System describes the encodings available in the POSTSCRIPT language.

Since the application and the POSTSCRIPT interpreter can be on different machines, the Client Library automatically ensures that the binary representation of numeric values, including byte order and floating-point format, are correctly interpreted.

6.2.1 Encoding POSTSCRIPT Language Code

On a system-specific basis, the Client Library supports a variety of conversions to and from the encodings and formats defined for the POSTSCRIPT language:

- Binary object sequence to binary object sequence. For expanding user name indices back to their printable names.
- Binary object sequence to ASCII encoding. For backward compatibility with printers, for interchange, and for debugging.
- Binary object sequence to binary-encoded tokens. For long-term storage.
- Binary-encoded tokens to ASCII. For backward compatibility and interchange.

'DPSProgramEncoding' defines the three encodings available to POSTSCRIPT language programs. 'DPSNameEncoding' defines the two possible encodings for user names in POSTSCRIPT language programs. See Section 11.6 for the type definitions.

6.2.2 Translation

Translation means the conversion of program encoding or name encoding from one form to another.

Any code sent to the context is converted according to the setting of the encoding fields. For a context that was created with the system-specific routine *DPSCreateTextContext*, code is automatically converted to ASCII encoding.

An application sometimes exchanges binary object sequences with another application. Since binary object sequences have user name indices by default, the sending application must provide name-mapping information to the receiving application; this information can be lengthy. Instead, some implementations allow the application to translate name indices back into user names by changing the 'nameEncoding' field to 'dps_strings'. In many implementations, *DPSChangeEncoding* performs this function.

6.3 BUFFERING

For optimal performance, programs and data sent to a context may be buffered by the Client Library. For the most part, the application programmer need not be concerned with this buffering. Flushing of the buffer happens automatically as required, such as just before waiting for input events.

However, in certain unusual situations, the application may explicitly flush a buffer (see example below). *DPSFlushContext* allows the application to force any buffered code or data to be sent to the context. Note that flushing does not guarantee that code is executed by the context, only that any buffered code is sent to the context. See Section 6.4 and *DPSWaitContext* for information on how to force code to be executed.

Unnecessary flushing is inefficient. It is unusual for the application to flush the buffer explicitly. Cases where the buffer might need to be flushed include the following:

- Nothing to send to the interpreter for a long time (for example, ''going to sleep'' or doing a long computation).

- Nothing expected from the interpreter for a long time. (Note that getting input automatically flushes the output buffers.)

The application may elect to flush buffers when client and server are separate processes and the execution of pending code is not critical.

6.4 SYNCHRONIZING APPLICATION AND CONTEXT

The POSTSCRIPT interpreter can run as a separate operating-system process (or task) from the application; it can even run on a separate machine. When the processes are separate, an application programmer must take into account the communication between the application and the POSTSCRIPT interpreter. This is important when time-critical actions must be performed based on the current appearance of the display. Also, errors arising from the execution of a wrapped procedure may be reported long after the procedure returns.

The application and the context are synchronized when all code sent to the context has been executed and it is waiting to execute more code. When the two are *not* synchronized, the status of code previously sent to the context is unknown to the application. Synchronization can be effected in two ways: as a side effect of calling wraps that return values, or explicitly, by calling the *DPSWaitContext* procedure.

A wrapped procedure that has no result values returns as soon as the wrap body is sent to the context. The data buffer is not necessarily flushed in this case. Sometimes, however, the application's next action depends on the completed execution of the wrap body by the POSTSCRIPT interpreter. The following example describes the kind of problem that can occur when the assumption is made that a wrap's code has been executed by the time it returns:

Example: An application calls a wrapped procedure to draw a large and complex picture into an offscreen buffer (such as an X11 pixmap). The wrapped procedure has no return value, so it returns immediately, although the context may not have finished executing the code. At this point, the application calls procedures to copy the screen buffer to a window for display. If the context has not finished drawing the picture into the buffer, only part of the image will be displayed on the screen. This is not what the application programmer had in mind.

Wrapped procedures that return results flush any code waiting to be sent to the context and then wait until all results have been received. Therefore they automatically synchronize the context with the application. The wrapped procedure will not return until the interpreter indicates that all results have been sent.[2] In this case, the application knows that the context is ready to execute more code as soon as the wrapped procedure returns.

The preceding discussion describes the side effect of calling a wrap that returns a value, but it is not always convenient, or indeed correct, to write wrapped procedures that return values. Forcing the application to wait for a return result for every wrap is inefficient and may degrade performance.

If an application has a few critical points where synchronization must occur, and a wrap that returns results is not needed, *DPSWaitContext* may be used to synchronize the application with the context. *DPSWaitContext* flushes any buffered code, and then waits until the context finishes executing all code that has been sent to it so far. This forces the context to finish before the application continues.

Like wraps that return results, *DPSWaitContext* should be used only when necessary. Performance may be degraded by excessive synchronization.

6.5 FORKED CONTEXTS

When the **fork** operator is executed in the POSTSCRIPT interpreter, a new execution context is created, but the application has no way to communicate with it. In order to communicate with a forked context, it must create a 'DPSContextRec' for it. For example, *DPSContextFromContextID* is an X Window System procedure that creates a 'DPSContextRec' for a forked context.

[2]But the wrapped procedure may return prematurely if an error occurs, depending on how the error handler works; see Section 5.4.

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

'ctxt' is the context that executed the **fork** operator.

'cid' is the integer value of the new context's identifier. 'NULL' is returned if 'cid' is invalid.

If 'textProc' or 'errorProc' are 'NULL', *DPSContextFromContextID* copies the corresponding procedure pointer from 'ctxt' to the new 'DPSContext'; otherwise the new context gets the specified 'textProc' and 'errorProc'.

All other fields of the new context are initialized with values from 'ctxt', including the space field.

7 PROGRAMMING TIPS

This section contains tips for avoiding mistakes commonly made by programmers using the Client Library interface. Some of the items listed here are brief summaries of **Notes** and **Warnings** emphasized elsewhere in this document. Section 7.1 contains some pointers on how to make the best use of the POSTSCRIPT language imaging model.

- Don't guess what the arguments to a single-operator procedure call are — look them up in the listing. See Section 10.
- Make sure that variables passed to wrapped procedures and single-operator procedures are of the correct C type. A common mistake is to pass a pointer to a 'short int' (only 16 bits wide) to a procedure that returns a boolean. A boolean is defined as an 'int', which can be 32 bits wide on some systems.
- Make sure that POSTSCRIPT language code is properly separated by whitespace when using *DPSPrintf.* Make sure that variables passed to *DPSPrintf* are of the right type. Passing type 'float' to a format string of '%d' will yield unpredictable results. See Section 4.5.
- There are two means of synchronizing the application with the context: either call *DPSWaitContext*, which causes the application to wait until the interpreter has executed all the code sent to the execution context, or call a wrap that returns a result, which causes synchronization as a side effect. If synchronization is not required, use a wrap that returns results only when results are needed. Unnecessary synchronization by either method will degrade performance. See Section 6.4.
- Use of *DPSFlushContext* is usually not necessary. See Section 6.3.
- Do not read from the file returned by the operator **currentfile** from within a wrap. In general, do not read directly from the context's standard input stream **%stdin** from within a wrap. Since a binary object sequence is a single token, the behavior of the code is different from what it would be in another encoding, such as ASCII. This will lead to unpredictable results. See the *pswrap Refer-*

ence Manual and *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System.*

- If the context is an execution context for a display, do not write POSTSCRIPT language programs, particularly in wraps, that depend on reading the end-of-file (EOF) indicator. Support for EOF on the communication channel is system specific, and should not be relied upon. However, POSTSCRIPT language programs that will be written to a file or spooled to a printer can make use of EOF indications.
- Be careful when sending intermixed encoding types to a context. In particular, it's best to avoid mixing ASCII encoding with binary encoding. See the warning on page 18 for an example; see also the following tip on *DPSWaitContext.*
- Before calling *DPSWaitContext*, make sure that code that has already been sent to the context is syntactically complete, such as a wrap or a correctly terminated POSTSCRIPT operator or composite object.
- Use of the **fork** operator requires understanding of a given system's support for handling errors from the forked context. A common error while developing multiple context applications is to fail to handle errors arising from forked contexts. See Section 5.4.
- To avoid unintended recursion, do not call Client Library procedures or wraps from within a call-back procedure.
- To avoid confusion about which context on a chain will handle output, don't send input to a context that's been made the child of another context; send input only to the parent. (This doesn't apply to text contexts, since they never get output.)
- Program wraps carefully. Copying the entire prologue from a POSTSCRIPT printer driver into a wrap without change is probably not going to result in efficient code.
- Avoid the temptation to do all of your programming in the POSTSCRIPT language. Because the POSTSCRIPT language is interpreted, not compiled, the application can generally do arithmetic computation and data manipulation such as sorting more efficiently in C. Reserve the POSTSCRIPT language for what it does best — displaying text and graphics.

7.1 USING THE IMAGING MODEL

The device-independent and resolution-independent imaging model defined by the POSTSCRIPT language is described in the *POSTSCRIPT Language Reference Manual.* For general advice on how to use the POSTSCRIPT language efficiently and detailed advice on how to write page descriptions, see *POSTSCRIPT Language Program Design.* Although that book is primarily concerned with printer applications, much of its information on the imaging model can be applied to writing applications for the DISPLAY POSTSCRIPT system. A thorough understanding of the imaging model is essential to writing efficient DISPLAY POSTSCRIPT system applications.

The imaging model helps make your application device and resolution independent. Device independence ensures that your application will work and look as you intended on any display or print media. Resolution independence lets you use the power of the POSTSCRIPT language to do scaling, rotation, and transformation of your graphical display without loss of quality. Use of the imaging model will automatically give you the best possible rendering for any device.

Design your application with the imaging model in mind. Consider issues like converting coordinate systems, representing paths and graphics states with data structures, rendering colors and patterns, setting text, and accessing fonts (to name just a few).

A few specific tips are listed below:

- Coordinates sent to the POSTSCRIPT interpreter should be in the user coordinate system (user space). Although it may be more convenient to express coordinates in the window coordinate system, this makes your code resolution dependent. Your application will run more efficiently if you compute the coordinate conversions to and from user space in C code, rather than letting the interpreter do it.
- Think in terms of color. Avoid programming to the lowest common denominator (low-resolution monochrome). The imaging model will always give the best rendering possible for a device, so use colors even if you expect that your application may be run on monochrome or gray-scale

devices. Avoid using **setgray** unless you really want black, white, or a gray level. Use **setrgbcolor** for all other cases. The imaging model will use a gray level or halftone pattern if the device does not support color, so objects of different colors will be distinguishable from one another.

- Don't use **setlinewidth** with a width of zero to get thin lines. On high-resolution devices, the lines will be practically invisible. Use fractions of 1 to get lines narrower than 1, such as 0.3 or 0.25.

8 EXAMPLE APPLICATION PROGRAM

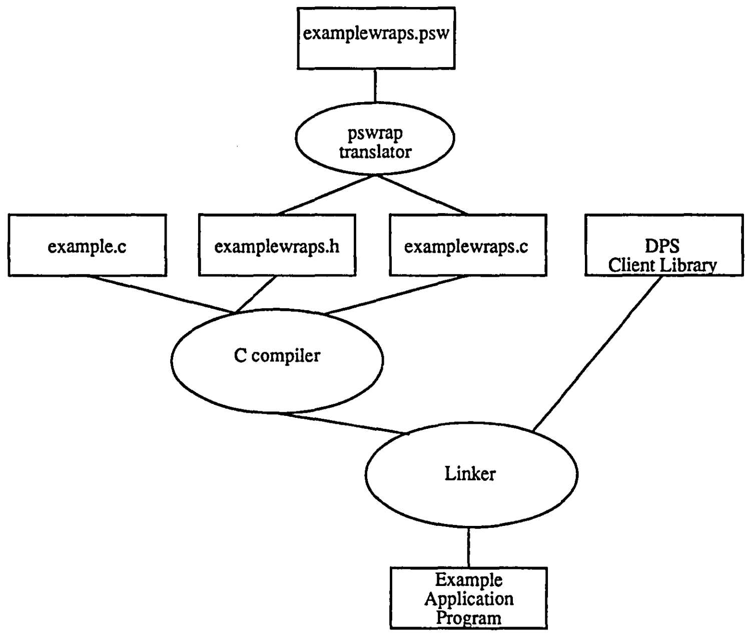
This section provides a simple example of how to use the DISPLAY POSTSCRIPT system through the Client Library. The example:

- Establishes communication with an X11 server.
- Creates a window and a context.
- Draws an ochre rectangle in the window.
- Waits for a mouse-button click.
- Terminates when the button is pressed.

To use the POSTSCRIPT imaging model, an application must describe its graphical operations in the POSTSCRIPT language. Therefore an application using the DISPLAY POSTSCRIPT system is a combination of C code and POSTSCRIPT language code.

The *pswrap* program generates a C code file and a C header file that defines the interface to the procedures in the code file. The application source code and the *pswrap* output file are compiled and linked together with the program libraries of the Client Library to form the executable application program. Figure 2 illustrates the complete process.

Figure 2 *Creating an Application*



8.1 EXAMPLE C CODE

The following code is used in conjunction with the wrap in the next section. See the description that follows.

```
/*
  example.c - simple X Window System application. Uses Display Postscript
  to draw an ochre box and uses X primitives to wait for a mouse click before
  terminating.
*/

#include <stdio.h>           /* Standard C library I/O routines */
#include <string.h>          /* Standard C library string routines */
#include <X11/X.h>             /* X definitions */
#include <X11/Xlib.h>          /* Interface to X library */
#include <X11/Intrinsic.h>    /* X toolkit definitions */
#include "psops.h"           /* Interface to PostScript single-op wraps */
```

```
#include "dpsXclient.h"         /* Interface to the DPS Client Library */
#include "examplewraps.h"       /* Interface to user-defined "wrap" procedures */

/* Window geometry definitions */
#define XWINDOW_X_ORIGIN       100
#define XWINDOW_Y_ORIGIN       100
#define XWINDOW_WIDTH         500
#define XWINDOW_HEIGHT        500

void main(argc, argv)
  int   argc;
  char  *argv[];
{
  Display *dpy;      /* X display structure */
  int screen;    /* screen on display */
  DPSContext ctxt;    /* DPS drawing context */
  DPSContext txtCtxt;    /* DPS text context for debugging */
  Window xWindow;    /* window where drawing occurs */
  int blackPixel, whitePixel;
  int debug = { FALSE };
  GC gc;
  XSetWindowAttributes attributes;
  unsigned long mask;
  DPSSpace space;
  float x, y, width, height;

  /* Connect to the window server by opening the display. Most of command
     line is parsed by XtOpenDisplay, leaving any options not recognized by
     the X toolkit: look for local -debug switch */

  XtToolkitInitialize();
  dpy = XtOpenDisplay(NULL, (String) NULL, "example", "example",
    (XrmOptionDescRec *) NULL, 0, &argc, argv);
  screen = DefaultScreen(dpy);
  if (argc == 2)
    if (strcmp(argv[1], "-debug") == 0)
      debug = TRUE;
    else {
      printf("Usage: example [-display xx:0] [-sync] [-debug]\n");
      exit(1);
      }

  /* Create a window to draw in: register interest in mouse button events. */

  blackPixel = BlackPixel (dpy, screen);
  whitePixel = WhitePixel (dpy, screen);
  attributes.background_pixel = whitePixel;
  attributes.border_pixel = blackPixel;
  attributes.bit_gravity = SouthWestGravity;
  attributes.event_mask = ButtonPressMask | ButtonReleaseMask;
  mask = CWBackPixel | CWBorderPixel | CWBitGravity | CWEventMask;
```

```
    xWindow = XCreateWindow(dpy, DefaultRootWindow(dpy),
      XWINDOW_X_ORIGIN, XWINDOW_Y_ORIGIN, XWINDOW_WIDTH, XWINDOW_HEIGHT,
      1, CopyFromParent, InputOutput, CopyFromParent, mask, &attributes);

    XMapWindow(dpy, xWindow);

    gc = XCreateGC(dpy, RootWindow(dpy, screen), 0, NULL);
    XSetForeground(dpy, gc, blackPixel);
    XSetBackground(dpy, gc, whitePixel);

    /* Create a DPS context to draw in the window we just created. If the
       user has asked for debugging, create a text context chained to the
       'drawing' context. */

    ctxt = XDPSCreateSimpleContext(dpy, xWindow, gc, 0, XWINDOW_HEIGHT,
      DPSDefaultTextBackstop, DPSDefaultErrorProc, NULL);
    if (ctxt == NULL) {
      fprintf(stderr, "Error attempting to create DPS context\n");
      exit(1);
      }

    DPSSetContext(ctxt);

    if (debug) {
      txtCtxt = DPSCreateTextContext(DPSDefaultTextBackstop, DPSDefaultErrorProc);
      DPSChainContext(ctxt, txtCtxt);
      }

    /* Get transformed dimensions, paint an ochre rectangle in middle
       of window. */

    PSitransform(
      (float) XWINDOW_WIDTH,
      (float) -XWINDOW_HEIGHT,
      &width,
      &height);

    x = width / 4.0;
    y = height / 4.0;

    PSWDrawBox(0.77, 0.58, 0.02, x, y, width / 2.0, height / 2.0);

    /* Wait for a mouse click on any button then terminate */

    while (NextEvent() != ButtonPress);
    while (NextEvent() != ButtonRelease);

    space = DPSSpaceFromContext(ctxt);
    DPSDestroySpace(space);
    exit(0);

  } /* main */

  int NextEvent()
```

```
{
  XEvent          event;

  XtNextEvent(&event);
  return(event.type);
}
```

8.2 EXAMPLE WRAP

This wrap provides the POSTSCRIPT language routine used by the example application. It is shown as *examplewraps.psw* in Figure 2 on page 40.

```
/* wrap for example application */

defineps PSWDrawBox(float r, g, b, x, y, width, height)
   gsave
   r g b setrgbcolor
   x y width height rectfill
   grestore
endps
```

8.3 DESCRIPTION OF THE EXAMPLE APPLICATION

The example application demonstrates the use of Client Library functions and custom wraps in the X11 environment. The application is simple: it draws a rectangle in the middle of a window, waits for a mouse button click in the window, and terminates.

The program starts by initializing the toolkit and connecting to the display device. Command-line options can include all options recognized by the X Intrinsics resource manager plus a local '–debug' option, which demonstrates the use of a chained text context for debugging.

The program creates a window that will contain the drawing produced by the POSTSCRIPT operators. The window's attributes are set to indicate interest in mouse button events in that window.

The program creates a context with 'xWindow' as its 'drawable'. The system-specific default handlers *DPSDefaultTextBackstop* and *DPSDefaultErrorProc* are specified in the *XDPSCreateSimpleContext* call. These handlers are adequate for this application.

If the '–debug' option was selected, the program creates a context that converts binary-encoded POSTSCRIPT language programs into readable text. The text is passed to 'PrintProc'. This context is then chained to the drawing context. The result is that any code sent to the drawing context will be also sent to the text context and displayed on *stdout*. This is a common technique for debugging wrapped procedures.

Now that the application is completely initialized, POSTSCRIPT language code can be executed to draw a rectangle into the window. This is done by using both a single-operator procedure and a customized wrapped procedure.

The single-operator procedure *PSitransform* determines the bounds of the window in terms of POSTSCRIPT user space; this allows the program to scale the size of the rectangle appropriately.

The wrap procedure *PSWDrawBox* takes red, green, and blue levels to specify the color of the rectangle. It also takes *x,y* coordinates for the bottom left corner of the rectangle, and it takes the rectangle's width and height. Simple arithmetic computation is most efficiently done in C code by the application, rather than in POSTSCRIPT language code by the interpreter.

PSWDrawBox is called to draw a colored square. If the display supports color, you'll see a square painted in ochre (a dark shade of orange). The values 0.77 for red, 0.58 for green, and 0.02 for blue approximate the color ochre. If the display supports only gray scale or monochrome, you'll see a square painted in some shade of gray.

The program now waits for events. Since the only events registered in this window are mouse-button events, events such as window movement and resizing are not directed to the application. When a button-press event is followed by a button-release event, the program destroys the space used by the drawing context. This destroys the context and its chained text context as well. The program then terminates normally.

9 THE *DPSCLIENT.H* HEADER FILE

This section documents the *dpsclient.h* procedures that constitute the core of the Client Library. They are system independent.

9.1 *DPSCLIENT.H* DATA STRUCTURES

This section documents:

- The standard context record.
- The standard error codes.

The context record, 'DPSContextRec', is shared by the application and the POSTSCRIPT interpreter. Except for its 'priv' field, this data structure should not be altered directly. The *dpsclient.h* header file provides procedures to alter it.

When calling Client Library procedures, refer to the context record by its handle, 'DPSContext'.

DPSContext

/* handle for context record */

See 'DPSContextRec'.

DPSContextRec

```
typedef struct _t_DPSContextRec {
  char *priv;
  DPSSpace space;
  DPSProgramEncoding programEncoding;
  DPSNameEncoding nameEncoding;
  DPSProcs procs;
  void (*textProc)();
  void (*errorProc)();
  DPSResults resultTable;
  unsigned int resultTableLength;
  struct _t_DPSContextRec *chainParent, *chainChild;
  } DPSContextRec, *DPSContext;
```

defines the data structure pointed to by 'DPSContext'.

Note: This record is used by *dpsclient.h* procedures but is actually defined in the *dpsfriends.h* header file.

'priv' is provided for use by application code. It is initialized to 'NULL' and is not touched thereafter by the Client Library implementation.

Warning: Although it is possible to read all the fields of the 'DPSContextRec' record directly, they should not be modified directly except for 'priv'. Data structures internal to the Client Library depend on the values in these fields and must be notified when they change. Call the procedures provided for this purpose, such as *DPSSetTextProc.*

'space' identifies the space in which the context executes.

'programEncoding' and 'nameEncoding' describe the encoding of the POSTSCRIPT language that is sent to the interpreter. The values in these fields are established when the context is created. Whether or not the encoding fields can be changed after creation is system specific.

'procs' points to a 'struct' containing procedures that implement the basic context operations, including writing, flushing, interrupting, and so on.

The Client Library implementation calls the 'textProc' and 'errorProc' procedures to handle interpreter-generated ASCII text and errors.

'resultTableLength' and 'resultTable' define the number, type, and location of results expected by a wrap. They are set up by the wrap procedure before any values are returned; see *DPSSetResultTable* in Section 11.7.

'chainParent' and 'chainChild' are used for chaining contexts. 'chainChild' is a pointer to the context that automatically receives code and data sent to the context represented by this 'DPSContextRec'. 'chainParent' is a pointer to the context that automatically sends code and data to the context represented by this 'DPSContextRec'. See the discussion of chained contexts in Section 6.1 for more information.

DPSErrorCode

```
typedef int DPSErrorCode;
```

defines the type of error code used by the Client Library. Here are the standard error codes:

- 'dps_err_ps' identifies standard POSTSCRIPT interpreter errors.
- 'dps_err_nameTooLong' flags user names that are too long. 128 characters is the maximum length for POSTSCRIPT language names.
- 'dps_err_resultTagCheck' flags erroneous result tags, most likely due to erroneous explicit use of the **printobject** operator.
- 'dps_err_resultTypeCheck' flags incompatible result types.
- 'dps_err_invalidContext' flags an invalid 'DPSContext' argument. An attempt to send POSTSCRIPT language code to a context that has terminated is the most likely cause of this error.

For more information, see *DPSErrorProc* in Section 9.2.

9.2 *DPSCLIENT.H* PROCEDURES

This section contains descriptions of the procedures in the Client Library header file *dpsclient.h*, listed alphabetically.

DPSChainContext

```
int DPSChainContext(parent, child);
DPSContext parent, child;
```

links 'child' onto the context chain of 'parent'. This is the chain of contexts that automatically receive a copy of any code or data sent to 'parent'. A context appears on only one such chain.

DPSChainContext returns zero if it successfully chains 'child' to 'parent'. It fails if 'child' is on another context's chain; in that case it returns –1.

See Section 6.1 for more information.

DPSDefaultErrorProc

```
void DPSDefaultErrorProc(ctxt, errorCode, arg1, arg2);
  DPSContext ctxt;
  DPSErrorCode errorCode;
  long unsigned int arg1, arg2;
```

is a sample *DPSErrorProc* for handling errors from the POSTSCRIPT interpreter. See Appendix B for a listing of the code and a description of the procedure.

The meaning of 'arg1' and 'arg2' depend on 'errorCode'. See *DPSErrorProc*.

DPSDestroyContext

```
void DPSDestroyContext(ctxt)
DPSContext ctxt;
```

destroys the context represented by 'ctxt'. The context is first unchained if it is on a chain.

What happens to buffered input and output when a context is destroyed is system specific; in the X Window System it is discarded.

Destroying a context does not destroy its space; see *DPSDestroySpace*.

DPSDestroySpace

```
void DPSDestroySpace(spc)
DPSSpace spc;
```

destroys the space represented by ‘spc’. This is necessary for application termination and clean-up. It also destroys all contexts within ‘spc’.

DPSErrorProc

```
typedef void (*DPSErrorProc)(/*
  DPSContext ctxt;
  DPSErrorCode errorCode;
  long unsigned int arg1, arg2;*/);
```

handles errors caused by the context. These can be POSTSCRIPT language errors reported by the interpreter or errors that occur when the Client Library is called with a context. 'errorCode' is one of the predefined codes that specify the type of error encountered; see 'DPSErrorCode' in Section 9.1 for its type definition. 'errorCode' determines the interpretation of 'arg1' and 'arg2'.

The following list shows how 'arg1' and 'arg2' are handled for each 'errorCode':

'dps_err_ps'
: POSTSCRIPT language error. 'arg1' is the address of the binary object sequence sent by the **handleerror** operator to report the error. The sequence has one object, which is an array of four objects. 'arg2' is the number of bytes in the entire binary object sequence.

'dps_err_nameTooLong'
: Error in wrap argument. The POSTSCRIPT user name and its length are passed as 'arg1' and 'arg2'. A name of more than 128 characters causes an error.

'dps_err_resultTagCheck'
: Error in formulation of wrap. The pointer to the binary object sequence and its length are passed as 'arg1' and 'arg2'. There is one object in the sequence.

'dps_err_resultTypeCheck'
: Incompatible result types. A pointer to the binary object is passed as 'arg1'; 'arg2' is unused.

'dps_err_invalidContext'
: Stale context handle (probably terminated). 'arg1' is a context identifier; 'arg2' is unused.

DPSFlushContext

```
void DPSFlushContext(ctxt)
DPSContext ctxt;
```

forces any buffered code or data to be sent to 'ctxt'. Some Client Library implementations use buffering to optimize performance.

DPSGetCurrentContext

```
DPSContext DPSGetCurrentContext();
```

returns the current context.

DPSGetCurrentErrorBackStop

```
DPSErrorProc DPSGetCurrentErrorBackstop();
```

returns the 'errorProc' passed most recently to *DPSSetErrorBackstop*, or 'NULL' if none was set.

DPSGetCurrentTextBackstop

```
DPSTextProc DPSGetCurrentTextBackstop();
```

returns the 'textProc' passed most recently to *DPSSetTextBackstop*, or 'NULL' if none was set.

DPSInterruptContext

```
void DPSInterruptContext(ctxt)
DPSContext ctxt;
```

notifies the interpreter to interrupt the execution of the context, resulting in the POSTSCRIPT language **interrupt** error. The procedure returns immediately after sending the notification.

DPSPrintf

```
void DPSPrintf(ctxt, fmt, [, arg ...]);
DPSContext ctxt;
char *fmt;
```

sends string 'fmt' to 'ctxt' with the optional arguments converted, formatted, and logically inserted into the string in a manner identical to the Standard C Library routine *printf*. It is useful for sending formatted data or a short POSTSCRIPT language program to a context.

DPSResetContext

```
void DPSResetContext(ctxt)
DPSContext ctxt;
```

resets the context after an error occurs. It ensures that any buffered I/O is discarded and that the context is ready to read and execute more input. *DPSResetContext* works in conjunction with **resynchandleerror.**

DPSSetContext

```
void DPSSetContext(ctxt)
DPSContext ctxt;
```

sets the current context. Call *DPSSetContext* before calling any procedures defined in *psops.h.*

DPSSetErrorBackstop

```
void DPSSetErrorBackstop(errorProc)
DPSErrorProc errorProc;
```

establishes 'errorProc' as a pointer to the backstop error handler. This error handler handles errors that are not handled by any other error handler. 'NULL' will be passed as the 'ctxt' argument to the backstop error handler.

DPSSetErrorProc

```
void DPSSetErrorProc(ctxt, errorProc)
DPSContext ctxt;
DPSErrorProc errorProc;
```

changes the context's error handler.

DPSSetTextBackstop

```
void DPSSetTextBackstop(textProc)
DPSTextProc textProc;
```

establishes the procedure pointed to by 'textProc' as the handler for text output for which there is no other handler. The text handler acts as a backstop for text output.

DPSSetTextProc

```
void DPSSetTextProc(ctxt, textProc)
DPSContext ctxt;
DPSTextProc textProc;
```

changes the context's text handler.

DPSSpaceFromContext

```
DPSSpace DPSSpaceFromContext(ctxt)
DPSContext ctxt;
```

returns the space handle for the specified context. It returns 'NULL' if 'ctxt' does not represent a valid execution context.

DPSTextProc

```
typedef void (*DPSTextProc)(/*
  DPSContext ctxt;
  char *buf;
  long unsigned int count; */);
```

handles text emitted from the interpreter — for example, by the == operator. 'buf' is a pointer to 'count' characters.

DPSUnchainContext

```
void DPSUnchainContext(ctxt)
DPSContext ctxt;
```

removes 'ctxt' from the chain that it is on, if any. The parent and child pointers of the unchained context are set to 'NULL'.

DPSWaitContext

```
void DPSWaitContext(ctxt)
DPSContext ctxt;
```

flushes output buffers belonging to 'ctxt' and then waits until the interpreter is ready for more input to 'ctxt'. It is not necessary to call *DPSWaitContext* after calling a wrapped procedure that returns a value.

Before calling *DPSWaitContext*, you must ensure that the last code sent to the context is syntactically complete, such as a wrap or a correctly terminated POSTSCRIPT operator or composite object.

DPSWriteData

```
void DPSWriteData(ctxt, buf, count)
DPSContext ctxt;
char *buf;
unsigned int count;
```

sends 'count' bytes of data from 'buf' to 'ctxt'. 'ctxt' specifies the destination context. 'buf' points to a buffer that contains 'count' bytes of data. The data will not be changed.

DPSWritePostScript

```
void DPSWritePostScript(ctxt, buf, count);
DPSContext ctxt;
char *buf;
unsigned int count;
```

sends POSTSCRIPT language to a context in any of the three language encodings. 'ctxt' specifies the destination context. 'buf' points to a buffer that contains 'count' bytes of POSTSCRIPT language code. The code in the buffer will be converted according to the context's encoding parameters as needed; refer to the system-specific documentation for a list of supported conversions.

10 SINGLE-OPERATOR PROCEDURES

For each operator defined in the POSTSCRIPT language, the Client Library provides a procedure to invoke the most common usage of the operator. These are called the single-operator procedures. (See the *POSTSCRIPT Language Reference Manual* and *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System* for complete information about how these POSTSCRIPT operators work.) If the predefined usage is not the one you need, it's easy to write wraps for variant forms of the operators.

There are two Client Library header files for single-operator procedures: *dpsops.h* and *psops.h*. The name of the Client Library single-operator procedure is the name of the POSTSCRIPT operator preceded by either DPS or PS[3]:

DPS prefix	Used when the context is explicitly specified; for example, *DPSgsave*. The first argument must be of type 'DPSContext'. These single-operator procedures are defined in *dpsops.h*.
PS prefix	Used when the context is assumed to be the current context; for example, *PSgsave*. These single-operator procedures are defined in *psops.h*. The procedure *DPSSetContext*, defined in *dpsclient.h*, sets the current context.

For example, to execute the POSTSCRIPT operator **translate**, the application can call

```
DPStranslate(ctxt, 1.23, 43.56)
```

where 'ctxt' is a variable of type 'DPSContext', the handle that represents a POSTSCRIPT execution context.

The *DPStranslate* procedure sends the binary encoding of

```
1.23 43.56 translate
```

to execute in 'ctxt'.

[3]Most POSTSCRIPT operator names are lowercase, but some contain uppercase letters; for example **FontDirectory**. In either case, the name of the corresponding single-operator procedure is formed simply by prefixing PS or DPS.

10.1 SETTING THE CURRENT CONTEXT

The single-operator procedures in *psops.h* assume the current context. The *DPSSetContext* procedure, defined in *dpsclient.h*, sets the current context. When the application deals with only one context it is convenient to use the procedures in *psops.h* rather than those in *dpsops.h*. In this case, the application would set the current context during its initialization phase:

```
DPSSetContext(ctxt);
```

In subsequent calls on the procedures in *psops.h*, 'ctxt' is used implicitly. For example:

```
PStranslate(1.23, 43.56);
```

has the same effect as

```
DPStranslate(ctxt, 1.23, 43.56);
```

The explicit method is preferable for situations that require intermingling of calls to multiple contexts.

Note: It is important to pass the correct C types to the single-operator procedures. (See Section 10.3 for the procedure declarations.) In general, if a POSTSCRIPT operator takes operands of arbitrary numeric type, the corresponding single-operator procedure takes parameters of type 'float'. Coordinates are always type 'float'. Passing an integer literal to a procedure that expects a floating-point literal is a common error:

incorrect:	PSlineto(72, 72);
correct:	PSlineto(72.0, 72.0);

Procedures that appear to have no input arguments may actually take their operands from the operand stack — for example, *PSdef* and *DPSdef.*

10.2 TYPES IN SINGLE-OPERATOR PROCEDURES

When using single-operator procedures, be sure to inspect the calling protocol (that is, order and types of formal parameters) for every procedure to be called; these are listed in Section 10.3.

Note: Throughout Section 10.2, references to single-operator procedures with a DPS prefix are equally applicable to the equivalent procedures with a PS prefix.

10.2.1 Rules of Thumb

There is no completely consistent system for associating data types with particular single-operator procedures. In general, it's safest to *look up the definition* in Section 10.3 or in the header file. However, there are a few rules of thumb that can be applied. Note that all of these rules have exceptions.

- Coordinates are specified as type 'float'. For example, all of the standard path construction operators (**moveto**, **lineto**, **curveto**, and so on), take type 'float'.
- Booleans are always type 'int'. The comment '/* int *b */' or '/* int *it */' in the header file means that the procedure returns a boolean.
- If the operator takes either integer or floating-point numbers, the corresponding procedure takes type 'float'. If the operator specifies a number type (such as **rand** and **vmreclaim**), then the procedure takes arguments of that type (typically type 'int').
- Operators that return values must always be specified with a pointer to the appropriate data type. For example, **currentgray** returns the current gray value of the graphics state. You must pass *DPScurrentgray* a pointer to a variable of type 'float'.
- If an operator takes a data type that does not have a directly analogous C type, such as dictionaries, graphic states, and executable arrays, the single-operator procedure takes no arguments. It is assumed that the programmer will arrange for the appropriate data to be on the operand stack before calling the procedure; see *DPSsendchararray* and *DPSsendfloat*, among others.
- If a single-operator procedure takes or returns a matrix, the matrix is specified as 'float m[]', which is an array of six floating-point numbers.
- In general, the integer parameter 'size' is used to specify

the length of a variable-length array; see, for example, *DPSxshow*. For single-operator procedures that take *two* variable-length arrays as parameters, the length of the first array is specified by the integer 'n'; the length of the second array is specified by the integer 'l'; see, for example, *DPSustroke*.

The following operators are worth noting for unusual order and types of arguments, or for other irregularities. After reading these descriptions, inspect the declarations in the listing in this document or in the header file:

- *DPSdefineuserobject* takes no arguments. One would expect it to take at least the index argument, but because of the requirement to have the arbitrary object on the top of the stack, it is probably better to send the index down separately, perhaps via *DPSsendint*.
- *DPSgetchararray*, *DPSgetfloatarray*, and other "get array" operators specify the length of the array first, followed by the array. (Mnemonic: Get the array last.)
- *DPSsendchararray*, *DPSsendfloatarray*, and other "send array" operators specify the array first, followed by the length of the array. (Mnemonic: Send the array first.)
- *DPSinfill*, *DPSinstroke*, and *DPSinufill* support only the *x,y*-coordinate version of the operator. The optional second userpath argument is not supported.
- *DPSinueofill*, *DPSinufill*, *DPSinustroke*, *DPSuappend*, *DPSueofill*, *DPSufill*, *DPSustroke*, and *DPSustrokepath* take a userpath in the form of an encoded number string and operator string. Note that the lengths of the strings follow the strings themselves, as arguments. See *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System* for details.
- *DPSsetdash* takes an array of numbers of type 'float' for the dash pattern.
- *DPSselectfont* takes type 'float' for the font scale parameter.
- *DPSsetgray* takes type 'float'. ('DPSsetgray(1)' is wrong.)
- *DPSxshow*, *DPSxyshow*, *DPSyshow* take an array of numbers of type 'float' for specifying the coordinates of each character.

- *DPSequals* is the procedure equivalent to the = operator.
- *DPSequalsequals* is the procedure equivalent to the == operator.
- *DPSversion* returns the version number in a character array 'buf[]' whose length is specified by 'bufsize'.

10.2.2 Special Cases

A few of the single-operator procedures have been optimized to take user objects for arguments, since they are most commonly used in this way. In the listing in Section 10.3, these user object arguments are specified as type 'int', which is the correct type of a user object.

- *DPScurrentgstate* takes a user object that represents the *gstate* object into which the current graphics state should be stored. The *gstate* object is left on the stack.
- *DPSsetfont* takes a user object that represents the font dictionary.
- *DPSsetgstate* takes a user object that represents the *gstate* object that the current graphics state should be set to.

10.3 *DPSOPS.H* PROCEDURE DECLARATIONS

The procedures in *dpsops.h* and *psops.h* are identical except for the first argument. *dpsops.h* procedures require the 'ctxt' argument; *psops.h* procedures do not. The procedure name is the lowercase POSTSCRIPT operator name preceded by "DPS" or "PS" as appropriate. For the sake of brevity, only the *dpsops.h* procedures are listed here.

Note: *DPSSetContext* must have been called before calling any procedure in *psops.h.*

```
extern void DPSFontDirectory( /* DPSContext ctxt; */ );
extern void DPSISOLatin1Encoding( /* DPSContext ctxt; */ );
extern void DPSSharedFontDirectory( /* DPSContext ctxt; */ );
extern void DPSStandardEncoding( /* DPSContext ctxt; */ );
extern void DPSUserObjects( /* DPSContext ctxt; */ );
extern void DPSabs( /* DPSContext ctxt; */ );
extern void DPSadd( /* DPSContext ctxt; */ );
extern void DPSaload( /* DPSContext ctxt; */ );
extern void DPSanchorsearch( /* DPSContext ctxt; int *truth; */ );
extern void DPSand( /* DPSContext ctxt; */ );
extern void DPSarc( /* DPSContext ctxt; float x, y, r, angle1, angle2; */ );
extern void DPSarcn( /* DPSContext ctxt; float x, y, r, angle1, angle2; */ );
extern void DPSarct( /* DPSContext ctxt; float x1, y1, x2, y2, r; */ );
extern void DPSarcto( /* DPSContext ctxt; float x1, y1, x2, y2, r; float *xt1, *yt1, *xt2, *yt2; */ );
extern void DPSarray( /* DPSContext ctxt; int len; */ );
extern void DPSashow( /* DPSContext ctxt; float x, y; char *s; */ );
extern void DPSastore( /* DPSContext ctxt; */ );
extern void DPSatan( /* DPSContext ctxt; */ );
extern void DPSawidthshow( /* DPSContext ctxt; float cx, cy; int c; float ax, ay; char *s; */ );
extern void DPSbanddevice( /* DPSContext ctxt; */ );
extern void DPSbegin( /* DPSContext ctxt; */ );
extern void DPSbind( /* DPSContext ctxt; */ );
extern void DPSbitshift( /* DPSContext ctxt; int shift; */ );
extern void DPSbytesavailable( /* DPSContext ctxt; int *n; */ );
extern void DPScachestatus( /* DPSContext ctxt; */ );
extern void DPSceiling( /* DPSContext ctxt; */ );
extern void DPScharpath( /* DPSContext ctxt; char *s; int b; */ );
extern void DPSclear( /* DPSContext ctxt; */ );
extern void DPScleardictstack( /* DPSContext ctxt; */ );
extern void DPScleartomark( /* DPSContext ctxt; */ );
extern void DPSclip( /* DPSContext ctxt; */ );
extern void DPSclippath( /* DPSContext ctxt; */ );
extern void DPSclosefile( /* DPSContext ctxt; */ );
extern void DPSclosepath( /* DPSContext ctxt; */ );
extern void DPScolorimage( /* DPSContext ctxt; */ );
extern void DPSconcat( /* DPSContext ctxt; float m[]; */ );
extern void DPSconcatmatrix( /* DPSContext ctxt; */ );
extern void DPScondition( /* DPSContext ctxt; */ );
extern void DPScopy( /* DPSContext ctxt; int n; */ );
extern void DPScopypage( /* DPSContext ctxt; */ );
extern void DPScos( /* DPSContext ctxt; */ );
extern void DPScount( /* DPSContext ctxt; int *n; */ );
extern void DPScountdictstack( /* DPSContext ctxt; int *n; */ );
extern void DPScountexecstack( /* DPSContext ctxt; int *n; */ );
extern void DPScounttomark( /* DPSContext ctxt; int *n; */ );
extern void DPScurrentblackgeneration( /* DPSContext ctxt; */ );
extern void DPScurrentcacheparams( /* DPSContext ctxt; */ );
extern void DPScurrentcmykcolor( /* DPSContext ctxt; float *c, *m, *y, *k; */ );
extern void DPScurrentcolorscreen( /* DPSContext ctxt; */ );
extern void DPScurrentcolortransfer( /* DPSContext ctxt; */ );
extern void DPScurrentcontext( /* DPSContext ctxt; int *cid; */ );
extern void DPScurrentdash( /* DPSContext ctxt; */ );
extern void DPScurrentdict( /* DPSContext ctxt; */ );
extern void DPScurrentfile( /* DPSContext ctxt; */ );
extern void DPScurrentflat( /* DPSContext ctxt; float *flatness; */ );
extern void DPScurrentfont( /* DPSContext ctxt; */ );
extern void DPScurrentgray( /* DPSContext ctxt; float *gray; */ );
extern void DPScurrentgstate( /* DPSContext ctxt; int gst; */ );
extern void DPScurrenthalftone( /* DPSContext ctxt; */ );
extern void DPScurrenthalftonephase( /* DPSContext ctxt; float *x, *y; */ );
extern void DPScurrenthsbcolor( /* DPSContext ctxt; float *h, *s, *b; */ );
extern void DPScurrentlinecap( /* DPSContext ctxt; int *linecap; */ );
extern void DPScurrentlinejoin( /* DPSContext ctxt; int *linejoin; */ );
extern void DPScurrentlinewidth( /* DPSContext ctxt; float *width; */ );
extern void DPScurrentmatrix( /* DPSContext ctxt; */ );
extern void DPScurrentmiterlimit( /* DPSContext ctxt; float *limit; */ );
extern void DPScurrentobjectformat( /* DPSContext ctxt; int *code; */ );
extern void DPScurrentpacking( /* DPSContext ctxt; int *b; */ );
extern void DPScurrentpoint( /* DPSContext ctxt; float *x, *y; */ );
extern void DPScurrentrgbcolor( /* DPSContext ctxt; float *r, *g, *b; */ );
extern void DPScurrentscreen( /* DPSContext ctxt; */ );
extern void DPScurrentshared( /* DPSContext ctxt; int *b; */ );
extern void DPScurrentstrokeadjust( /* DPSContext ctxt; int *b; */ );
extern void DPScurrenttransfer( /* DPSContext ctxt; */ );
extern void DPScurrentundercolorremoval( /* DPSContext ctxt; */ );
extern void DPScurveto( /* DPSContext ctxt; float x1, y1, x2, y2, x3, y3; */ );
extern void DPScvi( /* DPSContext ctxt; */ );
extern void DPScvlit( /* DPSContext ctxt; */ );
extern void DPScvn( /* DPSContext ctxt; */ );
extern void DPScvr( /* DPSContext ctxt; */ );
extern void DPScvrs( /* DPSContext ctxt; */ );
extern void DPScvs( /* DPSContext ctxt; */ );
extern void DPScvx( /* DPSContext ctxt; */ );
extern void DPSdef( /* DPSContext ctxt; */ );
extern void DPSdefaultmatrix( /* DPSContext ctxt; */ );
extern void DPSdefinefont( /* DPSContext ctxt; */ );
extern void DPSdefineusername( /* DPSContext ctxt; int i; char *username; */ );
extern void DPSdefineuserobject( /* DPSContext ctxt; */ );
extern void DPSdeletefile( /* DPSContext ctxt; char *filename; */ );
extern void DPSdetach( /* DPSContext ctxt; */ );
extern void DPSdeviceinfo( /* DPSContext ctxt; */ );
extern void DPSdict( /* DPSContext ctxt; int len; */ );
extern void DPSdictstack( /* DPSContext ctxt; */ );
extern void DPSdiv( /* DPSContext ctxt; */ );
extern void DPSdtransform( /* DPSContext ctxt; float x1, y1; float *x2, *y2; */ );
extern void DPSdup( /* DPSContext ctxt; */ );
extern void DPSecho( /* DPSContext ctxt; int b; */ );
extern void DPSend( /* DPSContext ctxt; */ );
extern void DPSeoclip( /* DPSContext ctxt; */ );
extern void DPSeofill( /* DPSContext ctxt; */ );
extern void DPSeoviewclip( /* DPSContext ctxt; */ );
extern void DPSeq( /* DPSContext ctxt; */ );
extern void DPSequals( /* DPSContext ctxt; */ );
extern void DPSequalsequals( /* DPSContext ctxt; */ );
extern void DPSerasepage( /* DPSContext ctxt; */ );
extern void DPSerrordict( /* DPSContext ctxt; */ );
extern void DPSexch( /* DPSContext ctxt; */ );
extern void DPSexec( /* DPSContext ctxt; */ );
extern void DPSexecstack( /* DPSContext ctxt; */ );
extern void DPSexecuserobject( /* DPSContext ctxt; int userObjIndex; */ );
extern void DPSexecuteonly( /* DPSContext ctxt; */ );
extern void DPSexit( /* DPSContext ctxt; */ );
extern void DPSexp( /* DPSContext ctxt; */ );
extern void DPSfalse( /* DPSContext ctxt; */ );
extern void DPSfile( /* DPSContext ctxt; char *name, *access; */ );
extern void DPSfilenameforall( /* DPSContext ctxt; */ );
extern void DPSfileposition( /* DPSContext ctxt; int *pos; */ );
extern void DPSfill( /* DPSContext ctxt; */ );
extern void DPSfindfont( /* DPSContext ctxt; char *name; */ );
extern void DPSflattenpath( /* DPSContext ctxt; */ );
extern void DPSfloor( /* DPSContext ctxt; */ );
extern void DPSflush( /* DPSContext ctxt; */ );
extern void DPSflushfile( /* DPSContext ctxt; */ );
extern void DPSfor( /* DPSContext ctxt; */ );
extern void DPSforall( /* DPSContext ctxt; */ );
extern void DPSfork( /* DPSContext ctxt; */ );
extern void DPSframedevice( /* DPSContext ctxt; */ );
extern void DPSge( /* DPSContext ctxt; */ );
extern void DPSget( /* DPSContext ctxt; */ );
extern void DPSgetboolean( /* DPSContext ctxt; int *it; */ );
extern void DPSgetchararray( /* DPSContext ctxt; int size; char s[]; */ );
extern void DPSgetfloat( /* DPSContext ctxt; float *it; */ );
extern void DPSgetfloatarray( /* DPSContext ctxt; int size; float a[]; */ );
extern void DPSgetint( /* DPSContext ctxt; int *it; */ );
extern void DPSgetintarray( /* DPSContext ctxt; int size; int a[]; */ );
extern void DPSgetinterval( /* DPSContext ctxt; */ );
extern void DPSgetstring( /* DPSContext ctxt; char *s; */ );
extern void DPSgrestore( /* DPSContext ctxt; */ );
extern void DPSgrestoreall( /* DPSContext ctxt; */ );
extern void DPSgsave( /* DPSContext ctxt; */ );
extern void DPSgstate( /* DPSContext ctxt; */ );
extern void DPSgt( /* DPSContext ctxt; */ );
extern void DPSidentmatrix( /* DPSContext ctxt; */ );
extern void DPSidiv( /* DPSContext ctxt; */ );
extern void DPSidtransform( /* DPSContext ctxt; float x1, y1; float *x2, *y2; */ );
extern void DPSif( /* DPSContext ctxt; */ );
extern void DPSifelse( /* DPSContext ctxt; */ );
extern void DPSimage( /* DPSContext ctxt; */ );
extern void DPSimagemask( /* DPSContext ctxt; */ );
extern void DPSindex( /* DPSContext ctxt; int i; */ );
extern void DPSineofill( /* DPSContext ctxt; float x, y; int *b; */ );
extern void DPSinfill( /* DPSContext ctxt; float x, y; int *b; */ );
extern void DPSinitclip( /* DPSContext ctxt; */ );
extern void DPSinitgraphics( /* DPSContext ctxt; */ );
extern void DPSinitmatrix( /* DPSContext ctxt; */ );
extern void DPSinitviewclip( /* DPSContext ctxt; */ );
extern void DPSinstroke( /* DPSContext ctxt; float x, y; int *b; */ );
extern void DPSinueofill( /* DPSContext ctxt; float x, y; char nums[]; int n; char ops[]; int l; int *b; */ );
extern void DPSinufill( /* DPSContext ctxt; float x, y; char nums[]; int n; char ops[]; int l; int *b; */ );
extern void DPSinustroke( /* DPSContext ctxt; float x, y; char nums[]; int n; char ops[]; int l; int *b; */ );
extern void DPSinvertmatrix( /* DPSContext ctxt; */ );
extern void DPSitransform( /* DPSContext ctxt; float x1, y1; float *x2, *y2; */ );
extern void DPSjoin( /* DPSContext ctxt; */ );
extern void DPSknown( /* DPSContext ctxt; int *b; */ );
extern void DPSkshow( /* DPSContext ctxt; char *s; */ );
extern void DPSle( /* DPSContext ctxt; */ );
extern void DPSlength( /* DPSContext ctxt; int *len; */ );
extern void DPSlineto( /* DPSContext ctxt; float x, y; */ );
extern void DPSln( /* DPSContext ctxt; */ );
extern void DPSload( /* DPSContext ctxt; */ );
extern void DPSlock( /* DPSContext ctxt; */ );
extern void DPSlog( /* DPSContext ctxt; */ );
extern void DPSloop( /* DPSContext ctxt; */ );
extern void DPSlt( /* DPSContext ctxt; */ );
extern void DPSmakefont( /* DPSContext ctxt; */ );
extern void DPSmark( /* DPSContext ctxt; */ );
extern void DPSmatrix( /* DPSContext ctxt; */ );
extern void DPSmaxlength( /* DPSContext ctxt; int *len; */ );
extern void DPSmod( /* DPSContext ctxt; */ );
extern void DPSmonitor( /* DPSContext ctxt; */ );
extern void DPSmoveto( /* DPSContext ctxt; float x, y; */ );
extern void DPSmul( /* DPSContext ctxt; */ );
extern void DPSne( /* DPSContext ctxt; */ );
extern void DPSneg( /* DPSContext ctxt; */ );
extern void DPSnewpath( /* DPSContext ctxt; */ );
extern void DPSnoaccess( /* DPSContext ctxt; */ );
extern void DPSnot( /* DPSContext ctxt; */ );
extern void DPSnotify( /* DPSContext ctxt; */ );
extern void DPSnull( /* DPSContext ctxt; */ );
extern void DPSnulldevice( /* DPSContext ctxt; */ );
extern void DPSor( /* DPSContext ctxt; */ );
extern void DPSpackedarray( /* DPSContext ctxt; */ );
extern void DPSpathbbox( /* DPSContext ctxt; float *llx, *lly, *urx, *ury; */ );
extern void DPSpathforall( /* DPSContext ctxt; */ );
extern void DPSpop( /* DPSContext ctxt; */ );
extern void DPSprint( /* DPSContext ctxt; */ );
extern void DPSprintobject( /* DPSContext ctxt; int tag; */ );
extern void DPSprompt( /* DPSContext ctxt; */ );
extern void DPSpstack( /* DPSContext ctxt; */ );
extern void DPSput( /* DPSContext ctxt; */ );
extern void DPSputinterval( /* DPSContext ctxt; */ );
extern void DPSquit( /* DPSContext ctxt; */ );
extern void DPSrand( /* DPSContext ctxt; */ );
extern void DPSrcheck( /* DPSContext ctxt; int *b; */ );
extern void DPSrcurveto( /* DPSContext ctxt; float x1, y1, x2, y2, x3, y3; */ );
extern void DPSread( /* DPSContext ctxt; int *b; */ );
extern void DPSreadhexstring( /* DPSContext ctxt; int *b; */ );
extern void DPSreadline( /* DPSContext ctxt; int *b; */ );
extern void DPSreadonly( /* DPSContext ctxt; */ );
extern void DPSreadstring( /* DPSContext ctxt; int *b; */ );
extern void DPSrealtime( /* DPSContext ctxt; int *i; */ );
extern void DPSrectclip( /* DPSContext ctxt; float x, y, w, h; */ );
extern void DPSrectfill( /* DPSContext ctxt; float x, y, w, h; */ );
extern void DPSrectstroke( /* DPSContext ctxt; float x, y, w, h; */ );
extern void DPSrectviewclip( /* DPSContext ctxt; float x, y, w, h; */ );
extern void DPSrenamefile( /* DPSContext ctxt; char *old, *new; */ );
extern void DPSrenderbands( /* DPSContext ctxt; */ );
extern void DPSrepeat( /* DPSContext ctxt; */ );
extern void DPSresetfile( /* DPSContext ctxt; */ );
extern void DPSrestore( /* DPSContext ctxt; */ );
extern void DPSreversepath( /* DPSContext ctxt; */ );
extern void DPSrlineto( /* DPSContext ctxt; float x, y; */ );
extern void DPSrmoveto( /* DPSContext ctxt; float x, y; */ );
extern void DPSroll( /* DPSContext ctxt; int n, j; */ );
extern void DPSrotate( /* DPSContext ctxt; float angle; */ );
extern void DPSround( /* DPSContext ctxt; */ );
extern void DPSrrand( /* DPSContext ctxt; */ );
extern void DPSrun( /* DPSContext ctxt; char *filename; */ );
extern void DPSsave( /* DPSContext ctxt; */ );
extern void DPSscale( /* DPSContext ctxt; float x, y; */ );
extern void DPSscalefont( /* DPSContext ctxt; float size; */ );
extern void DPSscheck( /* DPSContext ctxt; int *b; */ );
extern void DPSsearch( /* DPSContext ctxt; int *b; */ );
extern void DPSselectfont( /* DPSContext ctxt; char *name; float scale; */ );
extern void DPSsendboolean( /* DPSContext ctxt; int it; */ );
extern void DPSsendchararray( /* DPSContext ctxt; char s[]; int size; */ );
extern void DPSsendfloat( /* DPSContext ctxt; float it; */ );
extern void DPSsendfloatarray( /* DPSContext ctxt; float a[]; int size; */ );
extern void DPSsendint( /* DPSContext ctxt; int it; */ );
extern void DPSsendintarray( /* DPSContext ctxt; int a[]; int size; */ );
extern void DPSsendstring( /* DPSContext ctxt; char *s; */ );
extern void DPSsetbbox( /* DPSContext ctxt; float llx, lly, urx, ury; */ );
extern void DPSsetblackgeneration( /* DPSContext ctxt; */ );
extern void DPSsetcachedevice( /* DPSContext ctxt; float wx, wy, llx, lly, urx, ury; */ );
extern void DPSsetcachelimit( /* DPSContext ctxt; float n; */ );
extern void DPSsetcacheparams( /* DPSContext ctxt; */ );
extern void DPSsetcharwidth( /* DPSContext ctxt; float wx, wy; */ );
extern void DPSsetcmykcolor( /* DPSContext ctxt; float c, m, y, k; */ );
extern void DPSsetcolorscreen( /* DPSContext ctxt; */ );
extern void DPSsetcolortransfer( /* DPSContext ctxt; */ );
extern void DPSsetdash( /* DPSContext ctxt; float pat[]; int size; float offset; */ );
extern void DPSsetfileposition( /* DPSContext ctxt; int pos; */ );
extern void DPSsetflat( /* DPSContext ctxt; float flatness; */ );
extern void DPSsetfont( /* DPSContext ctxt; int f; */ );
extern void DPSsetgray( /* DPSContext ctxt; float gray; */ );
extern void DPSsetgstate( /* DPSContext ctxt; int gst; */ );
extern void DPSsethalftone( /* DPSContext ctxt; */ );
extern void DPSsethalftonephase( /* DPSContext ctxt; float x, y; */ );
extern void DPSsethsbcolor( /* DPSContext ctxt; float h, s, b; */ );
extern void DPSsetlinecap( /* DPSContext ctxt; int linecap; */ );
extern void DPSsetlinejoin( /* DPSContext ctxt; int linejoin; */ );
extern void DPSsetlinewidth( /* DPSContext ctxt; float width; */ );
extern void DPSsetmatrix( /* DPSContext ctxt; */ );
extern void DPSsetmiterlimit( /* DPSContext ctxt; float limit; */ );
extern void DPSsetobjectformat( /* DPSContext ctxt; int code; */ );
extern void DPSsetpacking( /* DPSContext ctxt; int b; */ );
extern void DPSsetrgbcolor( /* DPSContext ctxt; float r, g, b; */ );
extern void DPSsetscreen( /* DPSContext ctxt; */ );
extern void DPSsetshared( /* DPSContext ctxt; int b; */ );
extern void DPSsetstrokeadjust( /* DPSContext ctxt; int b; */ );
extern void DPSsettransfer( /* DPSContext ctxt; */ );
extern void DPSsetucacheparams( /* DPSContext ctxt; */ );
extern void DPSsetundercolorremoval( /* DPSContext ctxt; */ );
extern void DPSsetvmthreshold( /* DPSContext ctxt; int i; */ );
extern void DPSshareddict( /* DPSContext ctxt; */ );
extern void DPSshow( /* DPSContext ctxt; char *s; */ );
extern void DPSshowpage( /* DPSContext ctxt; */ );
extern void DPSsin( /* DPSContext ctxt; */ );
extern void DPSsqrt( /* DPSContext ctxt; */ );
extern void DPSsrand( /* DPSContext ctxt; */ );
extern void DPSstack( /* DPSContext ctxt; */ );
extern void DPSstart( /* DPSContext ctxt; */ );
extern void DPSstatus( /* DPSContext ctxt; int *b; */ );
extern void DPSstatusdict( /* DPSContext ctxt; */ );
extern void DPSstop( /* DPSContext ctxt; */ );
extern void DPSstopped( /* DPSContext ctxt; */ );
extern void DPSstore( /* DPSContext ctxt; */ );
extern void DPSstring( /* DPSContext ctxt; int len; */ );
extern void DPSstringwidth( /* DPSContext ctxt; char *s; float *xp, *yp; */ );
extern void DPSstroke( /* DPSContext ctxt; */ );
extern void DPSstrokepath( /* DPSContext ctxt; */ );
extern void DPSsub( /* DPSContext ctxt; */ );
extern void DPSsystemdict( /* DPSContext ctxt; */ );
extern void DPStoken( /* DPSContext ctxt; int *b; */ );
extern void DPStransform( /* DPSContext ctxt; float x1, y1; float *x2, *y2; */ );
extern void DPStranslate( /* DPSContext ctxt; float x, y; */ );
extern void DPStrue( /* DPSContext ctxt; */ );
extern void DPStruncate( /* DPSContext ctxt; */ );
extern void DPStype( /* DPSContext ctxt; */ );
extern void DPSuappend( /* DPSContext ctxt; char nums[]; int n; char ops[]; int l; */ );
extern void DPSucache( /* DPSContext ctxt; */ );
extern void DPSucachestatus( /* DPSContext ctxt; */ );
extern void DPSueofill( /* DPSContext ctxt; char nums[]; int n; char ops[]; int l; */ );
extern void DPSufill( /* DPSContext ctxt; char nums[]; int n; char ops[]; int l; */ );
extern void DPSundef( /* DPSContext ctxt; char *name; */ );
extern void DPSundefinefont( /* DPSContext ctxt; char *name; */ );
extern void DPSundefineuserobject( /* DPSContext ctxt; int userObjIndex; */ );
extern void DPSupath( /* DPSContext ctxt; int b; */ );
extern void DPSuserdict( /* DPSContext ctxt; */ );
extern void DPSusertime( /* DPSContext ctxt; int *milliseconds; */ );
extern void DPSustroke( /* DPSContext ctxt; char nums[]; int n; char ops[]; int l; */ );
extern void DPSustrokepath( /* DPSContext ctxt; char nums[]; int n; char ops[]; int l; */ );
extern void DPSversion( /* DPSContext ctxt; int bufsize; char buf[]; */ );
extern void DPSviewclip( /* DPSContext ctxt; */ );
extern void DPSviewclippath( /* DPSContext ctxt; */ );
extern void DPSvmreclaim( /* DPSContext ctxt; int code; */ );
extern void DPSvmstatus( /* DPSContext ctxt; int *level, *used, *maximum; */ );
extern void DPSwait( /* DPSContext ctxt; */ );
extern void DPSwcheck( /* DPSContext ctxt; int *b; */ );
extern void DPSwhere( /* DPSContext ctxt; int *b; */ );
extern void DPSwidthshow( /* DPSContext ctxt; float x, y; int c; char *s; */ );
extern void DPSwrite( /* DPSContext ctxt; */ );
extern void DPSwritehexstring( /* DPSContext ctxt; */ );
extern void DPSwriteobject( /* DPSContext ctxt; int tag; */ );
extern void DPSwritestring( /* DPSContext ctxt; */ );
extern void DPSwtranslation( /* DPSContext ctxt; float *x, *y; */ );
extern void DPSxcheck( /* DPSContext ctxt; int *b; */ );
extern void DPSxor( /* DPSContext ctxt; */ );
extern void DPSxshow( /* DPSContext ctxt; char *s; float numarray[]; int size; */ );
extern void DPSxyshow( /* DPSContext ctxt; char *s; float numarray[]; int size; */ );
extern void DPSyield( /* DPSContext ctxt; */ );
extern void DPSyshow( /* DPSContext ctxt; char *s; float numarray[]; int size; */ );
```

11 RUNTIME SUPPORT FOR WRAPPED PROCEDURES

This section describes the procedures in the *dpsfriends.h* header file that are called by wrapped procedures — the C-callable procedures that are output by the *pswrap* translator. This information is not normally required by the application programmer.

A description of the *dpsfriends.h* header file is provided for application or toolkit programmers who need finer control over these areas:

- Transmission of code for execution.
- Handling of result values.
- Mapping of user names to user name indices.

This section also contains a discussion of the structure of binary object sequences.

11.1 MORE ABOUT SENDING CODE FOR EXECUTION

One of the primary purposes of the Client Library is to provide runtime support for the code generated by *pswrap*. Each wrapped procedure builds a binary object sequence that represents the POSTSCRIPT language code to be executed. Since a binary object sequence is structured, the procedures for sending a binary object sequence are designed to take advantage of this structure.

The following procedures efficiently process binary object sequences generated by wrapped procedures:

- *DPSBinObjSeqWrite* sends the beginning of a new binary object sequence generated by a wrapped procedure. This initial part includes, at minimum, the header and the entire top-level sequence of objects. It can also include subsidiary array elements and/or string characters if those arrays and strings are static — that is, if their lengths are known at compile time and there are no intervening arrays or strings of varying length. *DPSBinObjSeqWrite* may convert the binary object sequence to another encoding, depending upon the 'DPSContextRec' encoding variables. For a particular wrapped procedure, *DPSBinObjSeqWrite* is called exactly once.

- *DPSWriteTypedObjectArray* sends arrays (excluding strings) that were specified as input arguments to a wrapped procedure. It writes POSTSCRIPT language code specified by the context's format and encoding variables, doing appropriate conversions as needed. For a particular wrapped procedure, *DPSWriteTypedObjectArray* is called zero or more times — once for each input array specified.
- *DPSWriteStringChars* sends the text of strings or names. It appends characters to the current binary object sequence. For a particular wrapped procedure, *DPSWriteStringChars* is called zero or more times to send the text of names and strings.

The overall length of arrays and strings sent by *DPSWriteTypedObjectArray* and *DPSWriteStringChars* must be consistent with the length information specified in the binary object sequence header sent by *DPSBinObjSeqWrite*. In particular, don't rely on 'sizeof()' to return the correct size value of the binary object sequence.

11.2 RECEIVING RESULTS

Each wrapped procedure with output arguments constructs an array containing elements of type 'DPSResultsRec'. This array is called the *result table*. The index position of each element corresponds to the ordinal position of each output argument as defined in the wrapped procedure: the first table entry (index 0) corresponds to the first output argument, the second table entry (index 1) corresponds to the second argument, and so on. Each entry defines one of the output arguments of a wrapped procedure by specifying a data type, a count, and a pointer to the storage for the value. *DPSSetResultTable* registers the result table with the context.

The interpreter sends return values to the application as binary object sequences. Wrapped procedures that have output arguments use the **printobject** operator to tag and send each return value. (See the discussion of the **printobject** operator in *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System.*) The tag corresponds to the index of the output argument in the result table. After the wrapped procedure finishes sending the POSTSCRIPT language program, it calls

DPSAwaitReturnValues to wait for all of the results to come back.

As the Client Library receives results from the interpreter, it places each result into the output argument specified by the result table. The tag of each result object in the sequence is used as an index into the result table. When the Client Library receives a tag that is greater than the last defined tag number, *DPSAwaitReturnValues* returns. This final tag is called the termination tag.

Certain conventions must be followed to handle return values for wrapped procedures properly:

- The tag associated with the return value is the ordinal of the output parameter as listed in the definition of the wrapped procedure, starting from 0 and counting from left to right (see example below).
- If the 'count' field of the 'DPSResultsRec' is –1, the expected result is a single element, or ''scalar,'' and return values with the same tag overwrite previous values. Otherwise, the 'count' indicates the number of array elements that remain to be received. In this case, a series of return values with the same tag are stored in successive elements of the array. If the value of 'count' is zero, further array elements of the same tag value are ignored.
- *DPSAwaitReturnValues* returns when it notices that the 'resultTable' pointer in the 'DPSContextRec' data object is 'NULL'. The code that handles return values should note the reception of the termination tag by setting the 'resultTable' to 'NULL' to indicate that there are no more return values to receive for this wrapped procedure.

Here is an example of a wrap with return values:

```
defineps Example(| int *x, *y, *z)
   10 20 30 x y z
endps
```

The code generated for this wrapped procedure is actually:

```
10 20 30
0 printobject
   % pop integer 30 off the operand stack,
   % use tag = 0 (result table index = 0, first parameter 'x')
   % write binary object sequence
1 printobject
   % pop integer 20 off the operand stack,
   % use tag = 1 (result table index = 1, second parameter 'y')
   % write binary object sequence
2 printobject
   % pop integer 10 off the operand stack,
   % use tag = 2 (result table index = 2, third parameter 'z')
   % write binary object sequence
0 3 printobject
   % push dummy value 0 on operand stack
   % pop integer 0 off operand stack,
   % use tag = 3 (termination tag)
   % write binary object sequence
flush
   % make sure all data is sent back to the application
```

11.3 MANAGING USER NAMES

Name indices are the most efficient way to specify names in a binary object sequence; refer to *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System* for a full description. The Client Library manages the mapping of user names to indices. Wrapped procedures map user names automatically. The first time a wrapped procedure is called, it calls *DPSMapNames* to map all user names specified in the wrapped procedure into indices. The application may also call *DPSMapNames* directly to obtain name mappings.

A name map is stored in a space. All contexts associated with that space have the same name map. The name mapping for the context is automatically kept up to date by the Client Library in the following way:

- Every wrapped procedure calls *DPSBinObjSeqWrite*, which, in addition to sending the binary object sequence, checks to see if the user name map is up to date.
- *DPSBinObjSeqWrite* calls *DPSUpdateNameMap* if the name map of the space does not agree with the Client Library's name map. *DPSUpdateNameMap* may send a

series of **defineusername** operators to the POSTSCRIPT interpreter.

DPSNameFromIndex returns the text for the user name with the given index. The string returned is owned by the Client Library; treat it as read-only.

11.4 BINARY OBJECT SEQUENCES

Syntactically, a binary object sequence is a single token. The structure is described in detail in *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System.* The definitions in this section correspond to the components of a binary object sequence.

```
#define DPS_HEADER_SIZE       4

#define DPS_HI_IEEE           128
#define DPS_LO_IEEE           129
#define DPS_HI_NATIVE         130
#define DPS_LO_NATIVE         131

#ifndef DPS_DEF_TOKENTYPE
#define DPS_DEF_TOKENTYPE  DPS_HI_IEEE
#endif  DPS_DEF_TOKENTYPE

typedef struct {
   unsigned char tokenType;
   unsigned char nTopElements;
   unsigned short length;
   DPSBinObjRec objects[1];
} DPSBinObjSeqRec, *DPSBinObjSeq;
```

A binary object sequence begins with a four-byte header. The first byte indicates the token type. A binary object is defined by one of the four token type codes listed above. 'DPS_DEF_TOKENTYPE' defines the default token type for binary object sequences generated by a particular implementation of the Client Library. 'DPS_DEF_TOKENTYPE' must be consistent with the machine architecture upon which the Client Library is implemented.

The 'nTopElements' byte indicates the number of top-level objects in the sequence. A binary object sequence can have from 1 to 255 top-level objects. If more top-level objects are required, use an *extended* binary object sequence (described in Section 11.5).

The next two bytes form a nonzero 16-bit integer that is the total byte length of the binary object sequence.

The header is followed by a sequence of objects.

```
#define DPS_NULL              0
#define DPS_INT               1
#define DPS_REAL              2
#define DPS_NAME              3
#define DPS_BOOL              4
#define DPS_STRING            5
#define DPS_IMMEDIATE         6
#define DPS_ARRAY             9
#define DPS_MARK              10
```

The first byte of an object describes its attributes and type. The types are listed above and correspond to the POSTSCRIPT language objects that *pswrap* generates.

```
#define DPS_LITERAL 0
#define DPS_EXEC    0x080
```

The high-order bit indicates whether the object has the literal (0) or executable (1) attribute.

The next byte is the tag byte, which must be zero for objects sent to the interpreter. Result values sent back from the interpreter will use the tag field, as described in Section 11.2.

The next two bytes form a 16-bit integer that is the length of the object. The unit value of the length field depends upon the type of the object. For arrays, the length indicates the number of elements in the array. For strings, the length indicates the number of characters.

The last four bytes of the object form the value field. The interpretation of this field depends upon the type of the object.

```
typedef struct {
    unsigned char attributedType;
    unsigned char tag;
    short length;
    long int val;
} DPSBinObjGeneric;     /* boolean, int, string, name and array */

typedef struct {
    unsigned char attributedType;
    unsigned char tag;
    short length;
    float realVal;
} DPSBinObjReal;        /* float */
```

'DPSBinObjGeneric' and 'DPSBinObjReal' are defined for the use of wraps. They make it easier to initialize the static portions of the binary object sequence.

```
typedef struct {
    unsigned char attributedType;
    unsigned char tag;
    short length;
    union {
        long int integerVal;
        float realVal;
        long int nameVal;    /* offset or index */
        long int booleanVal;
        long int stringVal;  /* offset */
        long int arrayVal;  /* offset */
    } val;
} DPSBinObjRec;
```

'DPSBinObjRec' is a general-purpose variant record for interpreting an object in a binary object sequence.

11.5 EXTENDED BINARY OBJECT SEQUENCES

If there are more than 255 top-level objects in the sequence, an *extended binary object sequence* is required; it is represented by 'DPSExtendedBinObjSeqRec', as follows:

Byte 0	Same as for a normal binary object sequence; it represents the token type.
Byte 1	Set to zero; indicates that this is an extended binary object sequence. (In a normal binary object sequence, this byte represents the number of top-level objects.)

Bytes 2-3	A 16-bit value representing the number of top-level elements.
Bytes 4-7	A 32-bit value representing the overall length of the extended binary object sequence.

The byte order in numeric fields is according to the number representation specified by the token type.

The layout of the remainder of the extended binary object sequence is identical to that of a normal binary object sequence.

11.6 *DPSFRIENDS.H* DATA STRUCTURES

This section describes the data structures used by the *pswrap* program as part of its support for wrapped procedures.

Note: The 'DPSContextRec' data structure and its handle, 'DPSContext', are part of the *dpsfriends.h* header file. They are documented in Section 9.1 because they are also used by *dpsclient.h* procedures.

DPSBinObjGeneric

```
typedef struct {
   unsigned char attributedType;
   unsigned char tag;
   unsigned short length;
   long int val;
} DPSBinObjGeneric;      /* boolean, int, string, name and array */
```

is defined for the use of wraps. It is used to initialize the static portions of the binary object sequence. See 'DPSBinObjReal' for type 'real'.

DPSBinObjReal

```
typedef struct {
   unsigned char attributedType;
   unsigned char tag;
   unsigned short length;
   float realVal;
} DPSBinObjReal;         /* float */
```

is similar to 'DPSBinObjGeneric', but represents a real number.

DPSBinObjRec

```
typedef struct {
    unsigned char attributedType;
    unsigned char tag;
    unsigned short length;
    union {
        long int integerVal;
        float realVal;
        long int nameVal;    /* offset or index */
        long int booleanVal;
        long int stringVal;  /* offset */
        long int arrayVal;  /* offset */
    } val;
} DPSBinObjRec;
```

is a general-purpose variant record for interpreting an object in a binary object sequence.

DPSBinObjSeqRec

```
typedef struct {
    unsigned char tokenType;
    unsigned char nTopElements;
    unsigned short length;
    DPSBinObjRec objects[1];
} DPSBinObjSeqRec, *DPSBinObjSeq;
```

This data type is provided as a convenience for accessing a binary object sequence copied from an I/O buffer.

DPSDefinedType

```
typedef enum {
  dps_tBoolean,
  dps_tChar,    dps_tUChar,
  dps_tFloat,   dps_tDouble,
  dps_tShort,   dps_tUShort,
  dps_tInt,     dps_tUInt,
  dps_tLong,    dps_tULong } DPSDefinedType;
```

enumerates the C data types used to describe wrap arguments.

DPSExtendedBinObjSeqRec

```
typedef struct {
    unsigned char tokenType;
    unsigned char escape;  /* zero if this is an extended sequence */
    unsigned short nTopElements;
    unsigned long length;
    DPSBinObjRec objects[1];
} DPSExtendedBinObjSeqRec, *DPSExtendedBinObjSeq;
```

This data type has a purpose similar to 'DPSBinObjSeqRec', but is used for extended binary object sequences.

DPSNameEncoding

```
typedef enum {
  dps_indexed, dps_strings
  } DPSNameEncoding;
```

defines the two possible encodings for user names in the 'dps_binObjSeq' and 'dps_encodedTokens' forms of POSTSCRIPT language programs.

DPSProcs

```
/* pointer to procedures record */
```

See 'DPSProcsRec'.

DPSProcsRec

```
typedef struct {
  void (*BinObjSeqWrite)( /* DPSContext ctxt, char *buf, unsigned int count */ );
  void (*WriteTypedObjectArray)( /*
          DPSContext ctxt,
          DPSDefinedType type;
          char *array,
          unsigned int length */ );
  void (*WriteStringChars)( /* DPSContext ctxt; char *buf; unsigned int count; */ );
  void (*WriteData)( /* DPSContext ctxt, char *buf, unsigned int count */ );
  void (*WritePostScript)( /* DPSContext ctxt, char *buf, unsigned int count */ );
  void (*FlushContext)( /* DPSContext ctxt */ );
  void (*ResetContext)( /* DPSContext ctxt */ );
  void (*UpdateNameMap)( /* DPSContext ctxt */ );
  void (*AwaitReturnValues)( /* DPSContext ctxt */ );
  void (*Interrupt)( /* DPSContext ctxt */ );
  void (*DestroyContext)( /* DPSContext ctxt */ );
  void (*WaitContext)( /* DPSContext ctxt */ );
  } DPSProcsRec, *DPSProcs;
```

defines the data structure pointed to by 'DPSProcs'.

This record contains pointers to procedures that implement all of the operations that can be performed on a context. These procedures are analogous to the instance methods of an object in an object-oriented language.

Note: Application developers need not be concerned with the contents of this data structure. Do not change the 'DPSProcs' pointer. Do not change the contents of 'DPSProcsRec'.

DPSProgramEncoding

```
typedef enum {
  dps_ascii, dps_binObjSeq, dps_encodedTokens
  } DPSProgramEncoding;
```

defines the three possible encodings of POSTSCRIPT language programs: ASCII encoding, binary object sequence encoding, and binary token encoding.

DPSResultsRec

```
typedef struct {
  DPSDefinedType type;
  int count;
  char *value;
  } DPSResultsRec, *DPSResults;
```

Each wrapped procedure constructs an array called the *result table*, which consists of elements of type 'DPSResultsRec'. The index position of each element corresponds to the ordinal position of each output parameter as defined in the wrapped procedure; for example, index 0 (the first table entry) corresponds to the first output parameter, index 1 corresponds to the second output parameter, and so on.

'type' specifies the formal type of the return value. 'count' specifies the number of values expected; this supports array formals. 'value' points to the location of the first value; the storage beginning there must have room for 'count' values of type 'type'. If 'count' is –1, 'value' points to a scalar (single) result argument. If 'count' is zero, any subsequent return values are ignored.

DPSSpace

```
/* handle for space record */
```

See 'DPSSpaceRec'.

DPSSpaceRec

```
typedef struct {
  DPSSpaceProcs procs;
  } DPSSpaceRec, *DPSSpace;

typedef struct {
  void (*DestroySpace)(/* DPSSpace space */);
  } DPSSpaceProcsRec, *DPSSpaceProcs;
```

provides a representation of a space. See also *DPSDestroySpace* in Section 9.2.

11.7 *DPSFRIENDS.H* PROCEDURES

The following is an alphabetical listing of the procedures in the Client Library header file *dpsfriends.h.* These procedures are for experts only; most application programmers don't need them. The *pswrap* translator inserts calls to these procedures when it creates C-callable wrapped procedures specified by the application programmer.

DPSAwaitReturnValues

```
void DPSAwaitReturnValues(ctxt)
DPSContext ctxt;
```

waits for all results described by the result table; see 'DPSResultRec'. It uses the tag of each object in the sequence to find the corresponding entry in the result table. When *DPSAwaitReturnValues* receives a tag that is greater than the last defined tag number, there are no more return values to be received and the procedure returns. This final tag is called the termination tag. *DPSSetResultTable* must be called to set the result table before any calls to *DPSBinObjSeqWrite*.

DPSAwaitReturnValues can call the context's error procedure with 'dps_err_resultTagCheck' or 'dps_err_resultTypeCheck'. It will return prematurely if it encounters a 'dps_err_ps' error.

DPSBinObjSeqWrite

```
void DPSBinObjSeqWrite(ctxt, buf, count)
DPSContext ctxt;
char *buf;
unsigned int count;
```

sends the beginning of a binary object sequence generated by a wrap. 'buf' points to a buffer containing 'count' bytes of a binary object sequence. 'buf' must point to the beginning of a sequence, which includes at least the header and the entire top-level sequence of objects.

DPSBinObjSeqWrite may also include subsidiary array elements and/or strings. It writes POSTSCRIPT language as specified by the format and encoding variables of 'ctxt', doing appropriate conversions as needed. If the buffer does not contain the entire binary object sequence, one or more calls to *DPSWriteTypedObjectArray* and/or *DPSWriteStringChars* must follow immediately; 'buf' and its contents must remain valid until the entire binary object sequence has been written. *DPSBinObjSeqWrite* ensures that the user name map is up to date.

DPSMapNames

```
void DPSMapNames(ctxt, nNames, names, indices)
DPSContext ctxt;
unsigned int nNames;
char **names;
long int **indices;
```

maps all specified names into user name indices, sending new **defineusername** definitions as needed. 'names' is an array of strings whose elements are the user names. 'nNames' is the number of elements in the array. 'indices' is an array of pointers to '(long int *)' integers, which are the locations in which to store the indices. *DPSMapNames* is normally called automatically from within wraps. The application can also call this procedure directly to obtain name mappings.

DPSMapNames calls the context's error procedure with 'dps_err_nameTooLong'.

Note: The caller must ensure that the string pointers remain valid after the procedure returns. The Client Library becomes the owner of all strings passed to it with *DPSMapNames.*

The same name may be used several times in a wrap. To reduce string storage, these duplicates can be eliminated by using an optimization recognized by *DPSMapNames*. If the pointer to the string in the array 'names' is null — that is, '(char *)0' — *DPSMapNames* uses the nearest non-null name that precedes the '(char *)0' entry in the array. The first element of 'names' must be non-null. This optimization works best if you sort the names so that duplicate occurrences are adjacent.

Example: *DPSMapNames* treats the following arrays as equivalent, but the one on the right saves storage.

```
{                           {
"boxes",                    "boxes",
"drawMe",                   "drawMe",
"drawMe",                   (char *)0,
"init",                     "init",
"makeAPath",                "makeAPath",
"returnAClip",              "returnAClip",
"returnAClip",              (char *)0,
"returnAClip"               (char *)0
}                           }
```

DPSNameFromIndex

```
char *DPSNameFromIndex(index)
long int index;
```

returns the text for the user name with the given index. The string returned must be treated as read-only. 'NULL' will be returned if 'index' is invalid.

DPSSetResultTable

```
void DPSSetResultTable(ctxt, tbl, len)
DPSContext ctxt;
DPSResults tbl;
unsigned int len;
```

sets the result table and its length in 'ctxt'. This operation must be performed before a wrap body that can return a value is sent to the interpreter.

DPSUpdateNameMap

```
void DPSUpdateNameMap(ctxt)
DPSContext ctxt;
```

sends a series of **defineusername** commands to the interpreter. This procedure is called if the name map of the context's space is not synchronized with the Client Library name map.

DPSWriteStringChars

```
void DPSWriteStringChars(ctxt, buf, count);
DPSContext ctxt;
char *buf;
unsigned int count;
```

appends strings to the current binary object sequence. 'buf' contains 'count' characters that form the body of one or more strings in a binary object sequence. 'buf' and its contents must remain valid until the entire binary object sequence has been sent.

DPSWriteTypedObjectArray

```
void DPSWriteTypedObjectArray(ctxt, type, array, length)
DPSContext ctxt;
DPSDefinedType type;
char *array;
unsigned int length;
```

writes POSTSCRIPT language code as specified by the format and encoding variables of 'ctxt', doing appropriate conversions as needed. 'array' points to an array of 'length' elements of type 'type'. 'array' contains the element values for the body of a subsidiary array that was passed as an input argument to *pswrap*. 'array' and its contents must remain valid until the entire binary object sequence has been sent.

A CHANGES SINCE LAST RELEASE

This manual has been completely reorganized and rewritten. Changes to the *Client Library Reference Manual* from the document dated October 7, 1988, are noted in the paragraphs below.

The example text handler program in Section 5.3 has been changed from a Macintosh DISPLAY POSTSCRIPT program to an X11/DPS program. The example application program in Section 8 has been changed from a Macintosh DISPLAY POSTSCRIPT program to an X11/DPS program.

An example error handler program, *DPSDefaultErrorProc*, has been provided in Appendix B. This is the default error handler in the DISPLAY POSTSCRIPT extension for the X Window System.

The synchronization example in Section 6.4 has been replaced by an X-specific example.

The specifications for *dpsclient.h* and *dpsfriends.h* procedures are now in separate chapters.

Listings of the header files have been removed, except for *dpsops.h* (representing itself and *psops.h*), whose procedure declarations are not listed elsewhere in this manual.

Numerous inconsistencies in the arguments to some of the single-operator procedures have been cleaned up.

The document has been updated to be consistent with the latest versions of *dpsfriends.h*, *dpsclient.h*, *dpsops.h*, and *psops.h*. The following are no longer defined by Adobe:

- *DPSGetLastNameIndex*
- *DPSLastNameIndex*
- *DPSLastObjectIndex*
- *DPSNewUserObject*

References to system-specific issues have been added throughout the manual, including the following:

- Context creation routines.

- Behavior of default and backstop error and text handlers.
- Automatic encoding translation (for example, binary object sequence to tokens).
- Additional error codes.
- Exception handling and error recovery.
- Programming examples and code fragments.

A section on programming tips has been added.

The index has been enhanced.

B EXAMPLE ERROR HANDLER

An error handler must deal with all errors defined in *dpsclient.h* as well as any additional errors defined in system-specific header files.

This appendix contains an example of an error handler for the X Window System extension of the DISPLAY POSTSCRIPT system.

B.1 ERROR HANDLER IMPLEMENTATION

An example implementation of an error handler, *DPSDefaultErrorProc*, follows. The code is followed by explanatory text.

```
#include "dpsclient.h"

void
DPSDefaultErrorProc(ctxt, errorCode, arg1, arg2)
  DPSContext ctxt;
  DPSErrorCode errorCode;
  long unsigned int arg1, arg2; {

  DPSTextProc textProc = DPSGetCurrentTextBackstop();

  char *prefix = "%%[ Error: ";
  char *suffix = " ]%%\n";

  char *infix = "; OffendingCommand: ";
  char *nameinfix = "User name too long; Name: ";
  char *contextinfix = "Invalid context: ";
  char *taginfix = "Unexpected wrap result tag: ";
  char *typeinfix = "Unexpected wrap result type; tag: ";

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

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

      elements = (DPSBinObj)(((char *) ary) + ary->val.arrayVal);
  errorName = (char *)(((char *) ary) + elements[1].val.nameVal);
  errorNameCount = elements[1].length;

  error = (char *)(((char *) ary) + elements[2].val.nameVal);
  errorCount = elements[2].length;

  resyncFlg = elements[3].val.booleanVal;

  if (textProc != NIL) {
    (*textProc)(ctxt, prefix, strlen(prefix));
    (*textProc)(ctxt, errorName, errorNameCount);
    (*textProc)(ctxt, infix, strlen(infix));
    (*textProc)(ctxt, error, errorCount);
    (*textProc)(ctxt, suffix, strlen(suffix));
    }
  if (resyncFlg && (ctxt != dummyCtx)) {
    RAISE(dps_err_ps, ctxt);
    CantHappen();
    }
  break;
  }
case dps_err_nameTooLong:
  if (textProc != NIL) {
    char *buf = (char *)arg1;
    (*textProc)(ctxt, prefix, strlen(prefix));
    (*textProc)(ctxt, nameinfix, strlen(nameinfix));
    (*textProc)(ctxt, buf, arg2);
    (*textProc)(ctxt, suffix, strlen(suffix));
    }
  break;
case dps_err_invalidContext:
  if (textProc != NIL) {
    char m[100];
    (void) sprintf(m, "%s%s%d%s", prefix, contextinfix, arg1, suffix);
    (*textProc)(ctxt, m, strlen(m));
    }
  break;
case dps_err_resultTagCheck:
case dps_err_resultTypeCheck:
  if (textProc != NIL) {
    char m[100];
    unsigned char tag = *((unsigned char *) arg1+1);
    (void) sprintf(m, "%s%s%d%s", prefix, typeinfix, tag, suffix);
    (*textProc)(ctxt, m, strlen(m));
    }
  break;
case dps_err_invalidAccess:
  if (textProc != NIL)
    {
    char m[100];
```

```
      (void) sprintf (m, "%sInvalid context access.%s", prefix, suffix);
      (*textProc) (ctxt, m, strlen (m));
      }
      break;
  case dps_err_encodingCheck:
    if (textProc != NIL)
      {
      char m[100];
      (void) sprintf (m, "%sInvalid name/program encoding: %d/%d.%s",
                    prefix, (int) arg1, (int) arg2, suffix);
      (*textProc) (ctxt, m, strlen (m));
      }
      break;
  case dps_err_closedDisplay:
    if (textProc != NIL)
      {
      char m[100];
      (void) sprintf (m, "%sBroken display connection %d.%s",
                    prefix, (int) arg1, suffix);
      (*textProc) (ctxt, m, strlen (m));
      }
      break;
  case dps_err_deadContext:
    if (textProc != NIL)
      {
      char m[100];
      (void) sprintf (m, "%sDead context 0x0%x.%s", prefix,
                    (int) arg1, suffix);
      (*textProc) (ctxt, m, strlen (m));
      }
      break;
  default:;
  }
} /* DPSDefaultErrorProc */
```

B.2 DESCRIPTION OF THE ERROR HANDLER

DPSDefaultErrorProc handles errors that arise when a wrap or Client Library procedure is called for the context. The error code indicates what error occurred. Interpretation of the 'arg1' and 'arg2' values is based on the error code.

The error handler initializes itself by getting the current backstop text handler and assigning string constants that will be used to formulate and report a text message. The section of the program that deals with the various error codes begins with the 'switch' statement. Each error code can be handled differently.

If a 'textProc' was specified, the error handler calls the text handler to formulate an error message, passing it the name of the error, the object that caused the error, and the string constants used to format a standard error message. For example, a **typecheck** error reported by the **cvn** operator would be reported as a 'dps_err_ps' error code and printed as follows:

```
%%[ Error: typecheck; OffendingCommand: cvn ]%%
```

The following error codes are common to all Client Library implementations:

- 'dps_err_ps' represents all POSTSCRIPT language errors reported by the interpreter; that is, the errors listed under each operator in the *POSTSCRIPT Language Reference Manual* and *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System.* See Section B.3 for more information about this error code.
- 'dps_err_nameTooLong' arises if a binary object sequence or encoded token has a name whose length exceeds 128 characters. 'arg1' is the POSTSCRIPT user name; 'arg2' is its length.
- 'dps_err_invalidContext' arises if a Client Library routine was called with an invalid context. This can happen if the client is unaware that the execution context in the interpreter has terminated. 'arg1' is a context identifier; 'arg2' is unused.
- 'dps_err_resultTagCheck' occurs when an invalid tag is received for a result value. There is one object in the sequence. 'arg1' is a pointer to the binary object sequence; 'arg2' is the length of the binary object sequence.
- 'dps_err_resultTypeCheck' occurs when the value returned is of a type incompatible with the output parameter (for example, a string returned to an integer output parameter). 'arg1' is a pointer to the binary object (the result with the wrong type); 'arg2' is unused.

The remainder of the error codes are specific to the X Window System:

- 'dps_err_invalidAccess' indicates that a shared context is being used improperly. For example, result values were erroneously sent to a sharing client other than the creator of the context. 'arg1' and 'arg2' are unused.
- 'dps_err_encodingCheck' indicates that an undefined encoding value has been passed to *DPSChangeEncoding* or that the application is trying to change the name encoding of a shared context. 'arg1' is the new name encoding; 'arg2' is the new program encoding.
- 'dps_err_closedDisplay' indicates that the connection to the server has been lost. 'arg1' is the index number of the display; 'arg2' is unused.
- 'dps_err_deadContext' indicates that a context has terminated in the interpreter, but the resources assigned to the context have not been freed. 'arg1' is the 'DPSContext' handle; 'arg2' is unused.

B.3 HANDLING POSTSCRIPT LANGUAGE ERRORS

The following discussion applies only to the 'dps_err_ps' error code. This error code represents all possible POSTSCRIPT operator errors. Because the interpreter provides a binary object sequence containing detailed information about the error, more options are available to the error handler than for other client errors.

'arg1' points to a binary object sequence that describes the error. The binary object sequence is a four-element array consisting of the name 'Error', the name that identifies the specific error, the object that was executed when the error occurred, and a boolean indicating whether the context expects to be resynchronized. For further details of the format of the binary object sequence, see *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System.*

The type and length of the array are checked with assertions. The body of the array is pointed to by the 'elements' variable. Each element of the array is derived and placed in a variable.

DPSDefaultErrorProc raises an exception only if the context executed **resyncstart** to install **resynchandleerror**. The 'resyncFlag' variable contains the value of the fourth element of the binary object sequence array, the boolean that indicates whether resynchronization is needed. 'resyncFlag' will be *false* if the **handleerror** operator handled the error; it will be *true* if **resynchandleerror** handled the error.

If 'resyncFlag' is *true* and the context handling the error is a context created by the application, the error handler raises the exception by calling *RAISE.* This call never returns. See Appendix C for a discussion of how *RAISE* works.

C EXCEPTION HANDLING

This appendix describes a general-purpose exception-handling facility. It provides help for a narrowly defined problem area — handling POSTSCRIPT language errors that arise from the conditions listed on page 26. *Most application programmers need not be concerned with exception handling.* These facilities can be used in conjunction with POSTSCRIPT language code and a sophisticated error handler such as *DPSDefaultErrorProc* to provide a certain amount of error recovery capability. Consult the system-specific documentation for alternative means of error recovery.

An *exception* is an unexpected condition such as a POSTSCRIPT language error that prevents a procedure from running to normal completion. The procedure could simply return, but data structures might be left in an inconsistent state and returned values might be incorrect. Instead of returning, the procedure can raise the exception, passing a code that indicates what has happened. The exception is intercepted by some caller of the procedure that raised the exception (any number of procedure calls deep); execution then resumes at the point of interception. As a result, the procedure that raised the exception is terminated, as are any intervening procedures between it and the procedure that intercepted the exception, an action which is called "unwinding the call stack."

The Client Library provides a general-purpose exception-handling mechanism in *dpsexcept.h.* This header file provides facilities for placing exception handlers in application subroutines to respond cleanly to exceptional conditions.

Note: Application programs may need to contain the following statement:

```
#include "dpsexcept.h"
```

As an exception propagates up the call stack, each procedure encountered can deal with the exception in one of three ways:

- It ignores the exception, in which case the exception continues on to the caller of the procedure.
- It intercepts the exception and handles it, in which case all procedure calls below the handler are unwound and discarded.
- It intercepts, handles, and then reraises the exception, allowing handlers higher in the stack to notice and react to the exception.

The body of a procedure that intercepts exceptions is written as follows:

```
DURING
  statement1;
  statement2;
  ...
HANDLER
  statement3
  statement4;
  ...
END_HANDLER
```

The statements between 'HANDLER' and 'END_HANDLER' comprise the exception handler for exceptions occurring between 'DURING' and 'HANDLER'. The procedure body works as follows:

- Normally, the statements between 'DURING' and 'HANDLER' are executed.
- If no exception occurs, the statements between 'HANDLER' and 'END_HANDLER' are bypassed; execution resumes at the statement after 'END_HANDLER'.
- If an exception is raised while executing the statements between 'DURING' and 'HANDLER' (including any procedure called from those statements), execution of those statements is aborted and control passes to the statements between 'HANDLER' and 'END_HANDLER'.

In terms of C syntax, you must treat these macros as if they were C code brackets, as follows:

Macro	*C Equivalent*
'DURING'	{{
'HANDLER'	}{
'END_HANDLER'	}}

In general, exception-handling macros should either entirely enclose a code block (the preferred method — see Example 1 below) or should be entirely within the block (see Example 2).

```
DURING
   while (/* Example 1 */) {
      ...
   }
HANDLER
   ...
END_HANDLER
while (/* Example 2 */) {
   DURING
   ...
   HANDLER
   ...
   END_HANDLER
}
```

When a procedure detects an exceptional condition, it can raise an exception by calling *RAISE*. *RAISE* takes two arguments. The first is an error code (for example, one of the values of 'DPSErrorCode'). The second is a pointer, 'char *', which may point to any kind of data structure, such as a string of ASCII text or a binary object sequence.

The exception handler has two local variables, 'Exception.Code' and 'Exception.Message', which are the values passed to the call to *RAISE*. These variables have valid contents only between 'HANDLER' and 'END_HANDLER'.

If the exception handler executes 'END_HANDLER' or returns, propagation of the exception ceases. However, if the exception handler calls *RERAISE*, the exception — along with

‘Exception.Code’ and ‘Exception.Message’ — is propagated to the next outer dynamically enclosing occurrence of ‘DURING ... HANDLER’.

A procedure may choose not to handle an exception, in which case one of its callers must handle it. There are two common reasons for wanting to handle exceptions:

- To deallocate dynamically allocated storage and clean up any other local state, then allow the exception to propagate further. In this case, the handler should perform its cleanup, then call *RERAISE*.
- To recover from certain exceptions that might occur, then continue normal execution. In this case, the handler should compare ‘Exception.Code’ against the set of exceptions it can handle. If it can handle the exception, it should perform the recovery and execute the statement that follows ‘END_HANDLER’; if not, it should call *RERAISE* to propagate the exception to a higher-level handler.

Warning: It is illegal to execute a statement between ‘DURING’ and ‘HANDLER’ that would transfer control outside of those statements. In particular, ‘return’ is illegal: an unspecified error will occur. This restriction does not apply to the statements between ‘HANDLER’ and ‘END_HANDLER’. To return from the exception handler, call ‘E_RETURN_VOID()’; to perform ‘return(x)’, call ‘E_RETURN(x)’.

C.1 RECOVERING FROM POSTSCRIPT LANGUAGE ERRORS

The example *DPSDefaultErrorProc* procedure can be used with the POSTSCRIPT operator **resyncstart** to recover from POSTSCRIPT language errors. If you use this strategy, an exception can be raised by any of the Client Library procedures that write code or data to the context: any wrap, any single-operator procedure, *DPSWritePostScript*, and so on. The strategy is as follows:

- Send the operator **resyncstart** to the context immediately after it is created. **resyncstart** is a simple read-evaluateprint loop enclosed in a **stopped** clause which, on error, executes **resynchandleerror**. **resynchandleerror** reports POSTSCRIPT errors back to the client in the form of a binary object sequence of a single object: an array of four elements as described in *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System*. The fourth element of the binary object sequence, a boolean, is set to *true* to indicate that **resynchandleerror** is executing. The **stopped** clause itself executes within an outer loop.

- When a POSTSCRIPT language error is detected, **resynchandleerror** writes the binary object sequence describing the error, flushes the output stream **%stdout**, then reads and discards any data on the input stream **%stdin** until EOF (an end-of-file marker) is received. This effectively clears out any pending code and data, and makes the context do nothing until the client handles the error.
- The binary object sequence sent by **resynchandleerror** is eventually received by the client and passed to the context's error handler. The error handler formulates a text message from the binary object sequence and displays it, perhaps by calling the backstop text handler. It then inspects the binary object sequence and notices that the fourth element of the array, a boolean, is *true*. This means that **resynchandleerror** is executing and is waiting for the client to recover from the error. At this point, the error handler may raise an exception by calling *RAISE* with 'dps_err_ps' and the 'DPSContext' pointer, in order to allow some exception handler to do specific error recovery.
- The 'dps_err_ps' exception is caught by one of the handlers in the application program. This causes the C stack to be unwound, and the handler body to be executed. To handle the exception, the application can reset the context that reported the error, discarding any waiting code.
- The handler body calls *DPSResetContext*, which resets the context after an error occurs. This procedure guarantees that any buffered I/O is discarded and that the context is ready to read and execute more input. Specifically, *DPSResetContext* causes EOF to be put on the context's input stream.
- We have come full circle now. EOF is received by **resynchandleerror**, which causes it to terminate. The

outer loop of **resyncstart** then reopens the context's input stream **%stdin**, which clears the end-of-file indication and resumes execution at the top of the loop. The context is now ready to read new code.

Although the above strategy works well enough for some applications, it leaves the context and the contents of its private VM in an unknown state. For example, the dictionary and operand stacks may be cluttered, or free-running forked contexts may have been created, or the contents of **userdict** may have been changed. Clearing the state of such a context may be very complicated.

Note: You may not get POSTSCRIPT language error exceptions when you expect them. Because of various delays related to buffering and scheduling, a POSTSCRIPT language error may be reported long after the C procedure responsible for the error has returned. This makes it difficult to write an exception handler for a given section of code. If this code can cause a POSTSCRIPT language error and will therefore cause *DPSDefaultErrorProc* to raise an exception, you can ensure that you get the exception in a timely manner by using synchronization, which is discussed in Section 6.4.

Warning: In multi-context applications that require error recovery, the code to recover from POSTSCRIPT errors can get quite complicated. An exception reporting a POSTSCRIPT error caused by one context can be raised by any call on the Client Library, even one on behalf of some other context, including calls made from wraps. Although *DPSDefaultErrorProc* does pass the context that caused the error as an argument to RAISE, it is difficult in general to deal properly with an exception from one context that arises while the application is working with another.

When the standard **handleerror** procedure is called to report an error, no recovery is possible except to display an error message and destroy the context.

C.2 EXAMPLE EXCEPTION HANDLER

A typical application might have the following main loop. Assume that a context has already been created with *DPSDefaultErrorProc* as its error procedure, and that **resyncstart** has been executed by the context.

```
#include <dpsexcept.h>

    while (/* the user hasn't quit */) {
        /* get an input event */
        event = GetEventFromQueue();
        /* react to event */
        DURING
            switch (event) {
                case EVENT_A:
                    UserWrapA(context, ...);
                    break;
                case EVENT_B:
                    UserWrapB(context, ...);
                    break;
                case EVENT_C:
                    ProcThatCallsSeveralWraps(context);
                    break;
                /* ... */
                default:;
            }
        HANDLER
            /* the context's error proc has already posted an
               error for this exception, so just reset.
               Make sure the context we're using is the
               one that caused the error! */
            if (Exception.Code == dps_err_ps)
                DPSResetContext((DPSContext)Exception.Message);
        END_HANDLER
    }
```

Most of the calls in the 'switch' statement are either direct calls to wrapped procedures or indirect calls (that is, calls to procedures that make direct calls to wrapped procedures or to the Client Library). All of the procedure calls between 'DURING' and 'HANDLER' can potentially raise an exception. The code between 'HANDLER' and 'END_HANDLER' is executed *only* if an exception is raised by the code between 'DURING' and 'HANDLER'. Otherwise, the handler code is skipped.

Suppose *ProcThatCallsSeveralWraps* is defined as follows:

```
void ProcThatCallsSeveralWraps(context)
DPSContext context;
{
   char *s = ProcThatAllocsAString(...);
   int n;

   DURING
      UserWrapC1(context, ...);
      UserWrapC2(context, &n);  /* user wrap returns a value */
      DPSPrintf(context, "/%s %d def\n", s, n); /* client lib proc */
   HANDLER
      if ((DPSContext)Exception.Message == context)
         {
         /* clean up the allocated string */
         free(s);
         s = NULL;
         }
      /* let the caller handle resetting the context */
      RERAISE;
   END_HANDLER

   /* clean up, if we haven't already */
   if (s != NULL) free(s);
}
```

This procedure unconditionally allocates storage, then calls procedures that may raise an exception. If there were no handler here and the exception simply propagated to the main loop, the storage allocated for the string would never be reclaimed. The solution is to define a handler that frees the storage and then calls *RERAISE* to allow another handler to do the final processing of the exception.

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. 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[1]	————	SDC Order Processing - WMO/E15 *or* Software Distribution Center Digital Equipment Corporation Westminster, Massachusetts 01473

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