AIXwindows Environment/6000

Display PostScript Reference

SC23-2211-00

First Edition (March 1990)

This edition of the Display PostScript Reference applies to the IBM AIXwindows Environment/6000 Licensed Program and to all subsequent releases of these products until otherwise indicated in new releases or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758–3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

IBM is a registered trademark of International Business Machines Corporation.

PostScript and Adobe are registered trademarks of and the PostScript logo is a trademark of Adobe Systems Incorporated.

Preface

The Display PostScript Reference Manual for the AIXwindows Environment/6000 consists of several publications produced by Adobe Systems Incorporated. The publications are in the following order:

- Perspective for Software Developers
- Extensions for the Display PostScript System
- Color Extensions
- Client Library Reference Manual
- X Window System Programmer's Supplement to the Client Reference Manual
- pswrap Reference Manual
- ENCAPSULATED PostScript FILES Specification Version 2.0
- DOCUMENT STRUCTURING CONVENTIONS Specification Version 2.1
- CHARACTER BITMAP DISTRIBUTION FORMAT Specification Version 2.1

Related Publications

Refer to the following Adobe Systems Incorporated publications for additional information:

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

DISPLAY POSTSCRIPT®
SYSTEM

Perspective for Software Developers

ADOBE SYSTEMS
INCORPORATED

Perspective for Software Developers

January 23, 1990

Written by Amy Davidson.

Contents

List of Figures

1 ABOUT THIS DOCUMENT

This document is your introduction to application development using the text and graphics imaging resources of the Display PostScript® system.

This overview for the application programmer describes:

- The Display PostScript system environment you'll be interacting with.
- The Client Library interface you'll be programming for.
- The use of the *pswrap* translator to prepare C-callable procedures containing PostScript® language programs.
- The manuals you'll need.

2 ABOUT THE DISPLAY POSTSCRIPT SYSTEM

The Display PostScript system provides a device-independent imaging model for displaying information on a screen. This imaging model is fully compatible with the imaging model used in PostScript printers. By allowing you to use the PostScript language to display text, graphics, and sampled images, it frees you from display-specific details such as screen resolution and number of available colors.

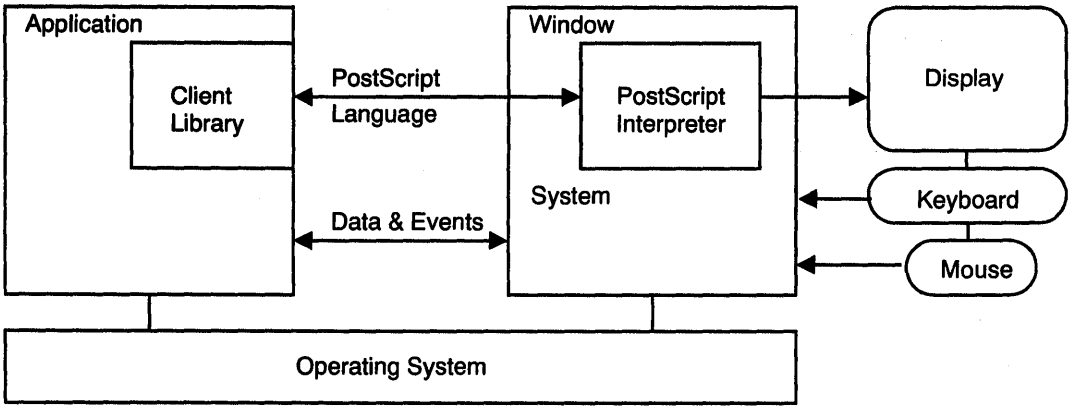
You can look at the Display PostScript system as part of the window system. Your application will use window system features for window placement and sizing, menu creation, and event handling, while using Display PostScript system features to take care of imaging inside the window.

Display PostScript system components include the Client Library, the PostScript interpreter, and the *pswrap* translator. These components are described in the accompanying set of manuals. If you are a new PostScript language programmer, you'll also need copies of the PostScript language manuals. See Section 6.2 for a discussion of documentation required by application developers.

Figure 1 shows the relationship between your application and the

Display PostScript system. The Client Library contains data structures and procedures your application will use to send requests to the PostScript interpreter. No matter what operating system you're dealing with, you can send PostScript language programs to the interpreter in the same way, using the same procedure calls. Therefore portability issues are minimized if you're developing an application for more than one environment.

Figure 1 *The Display PostScript System*



3 APPLICATION BUILDING BLOCKS

Most of your application will be written in C or another high-level languge. You'll call Client Library procedures to start a PostScript execution context, send programs and data to the PostScript interpreter, and get results from the the interpreter. The Client Library is the application's primary interface to the Display PostScript system.

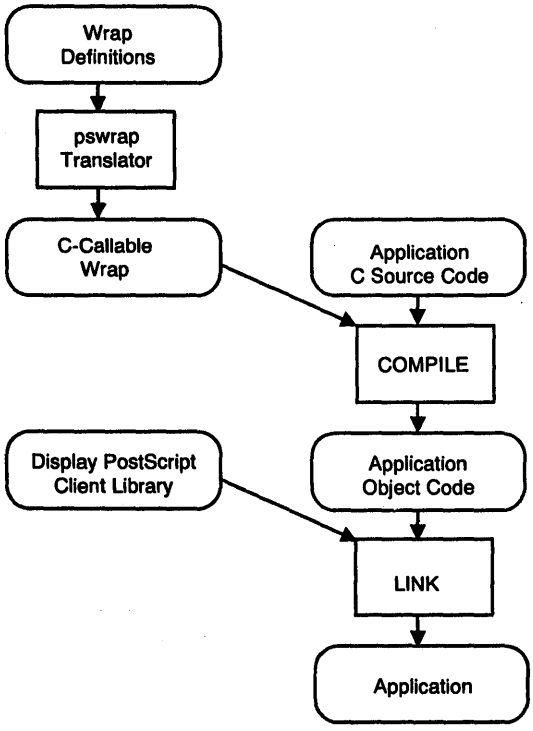
In addition, you'll call *wraps* — PostScript language procedures developed specifically for your application. Wraps, short for *wrapped procedures*, are created by the *pswrap* translator from PostScript language programs written to meet application needs. Figure 2 on page 3 shows how an application program is built.

4 USING *PSWRAP*

Your application will perform calculations, communicate with the window system, read and write files, and do other application processing in C or another high-level language. It will perform imaging tasks by calling wrapped procedures to send PostScript language programs to the interpreter. The *pswrap* translator creates these wraps from PostScript language input.

Figure 2 shows how an application is built of C source code and PostScript language code translated into C-callable procedures by *pswrap*.

Figure 2 *Creating a Display PostScript Application*



Consider a wrap, *PSWDisplayText*, that places text on the screen at a particular *x,y* coordinate. A call to *PSWDisplayText* from the application program might look something like this:

```
PSWDisplayText(72.0, 100.0, "Hello World");
```

The body of the *PSWDisplayText* procedure, is actually written in the PostScript language. It was defined to *pswrap* as follows:

```
defineps PSWDisplayText(float X,Y; char *text)
   X Y moveto
   (text) show
endps
```

In the wrap definition, the 'defineps' and 'endps' keywords tell *pswrap* where a given PostScript language program begins and ends. The 'defineps' statement defines the resulting procedure call. The *pswrap* translator processes this input and produces a C language source-code file. When compiled and linked with the application, the *PSWDisplayText* procedure sends a PostScript language program to the interpreter (binary-encoded for more efficient processing), causing the specified text to be displayed.

To summarize, *pswrap* takes a PostScript language program as input and gives you back a C language program. After you compile the resulting C program and link it into your application, calling the wrap will transmit a stream of PostScript language binary objects to the interpreter. See the *pswrap Reference Manual* for further information.

5 THE CLIENT LIBRARY INTERFACE

The Client Library is a linkable library of compiled C procedures that provides an interface between the application and the Display PostScript system. It creates an environment for handling imaging calls to specific Client Library procedures like *DPSmoveto* and custom wraps written for the application.

The Client Library is tailored by the Display PostScript system vendor for your particular operating environment. The *Client Library Reference Manual* describes a generic interface; your system vendor may add additional features or slightly modify the interface to suit your hardware and software. See your vendor's system-specific documentation for details.

6 SUPPORT FOR APPLICATION DEVELOPERS

Adobe supports application developers by means of the following services:

Adobe Developer Support Line
: A voice mailbox that puts you in touch with Adobe's support services for developers. See Section 6.1.

Technical Literature Catalog
: A free catalog of documentation, available on request.

Documentation
: PostScript language manuals (see Section 6.2), developer reference manuals (included with this release), and additional technical literature defining conventions and standards for PostScript language applications (see Section 6.5).

Adobe Developer Association
: A membership program for developers, providing access to technical resources and monthly mailings. See Section 6.4.

Public Access File Server
: Contains technical documents, code examples, AFM files, and documentation updates. See Section 6.6.

6.1 THE ADOBE DEVELOPER SUPPORT LINE

You can call the Adobe Developer Support Line, (415) 961-4111, for the following kinds of support:

- To receive a free Technical Literature Catalog.
- To order technical literature. (See Section 6.2 for the toll-free number to use when ordering PostScript language manuals.)
- To find out how to become a member of the Adobe Developer Association.
- To request technical assistance (for members of the Adobe Developer Association only).

If you prefer, you can write to us for information. Our mailing address is:

PostScript Developer Support
Adobe Systems Incorporated
1585 Charleston Road, P.O. Box 7900
Mountain View, CA 94039-7900

6.2 MANUALS FOR APPLICATION DEVELOPERS

The accompanying set of developer reference manuals contains information needed by a programmer developing an application for the Display PostScript system. If you're new to the PostScript language, you should first read the following manuals (published by Addison-Wesley and available from Adobe by calling 1-800-344-8335 or through your technical bookstore):

PostScript Language Reference Manual
: The standard reference for the PostScript language. Describes the PostScript imaging model and the concepts and facilities of the PostScript interpreter. Documents the PostScript language. Required reading.

PostScript Language Tutorial and Cookbook
: Introduction to the PostScript language in an informal, interactive style. Contains a collection of example programs that illustrate the PostScript imaging model.

PostScript Language Program Design
: Guidelines for the advanced developer to use in designing and debugging PostScript language programs. Printer-oriented, but most of the information is relevant to writing a Display PostScript application.

Once you're up to speed in the PostScript language, read the following manuals:

PostScript Language Extensions for the Display PostScript System
: Describes the extensions to the PostScript language that were made for the Display PostScript system, such as alternative PostScript language encodings, multiple execution contexts, user paths, window system support, and memory management. Introduces important system concepts and documents additional PostScript operators.

Client Library Reference Manual
: Describes the procedural interface to the Display PostScript system. Tells how to send programs and data to a PostScript execution context, how to handle context output, how to create and terminate a context. Contains procedure definitions, programming tips, and a sample application program.

pswrap Reference Manual
: Describes how to define C-callable procedures that contain PostScript language programs. Tells how to declare input arguments and output to be received from the interpreter. Documents the *pswrap* command line options.

PostScript Language Color Extensions
: Describes color extensions to the PostScript language, including multiple color images, color halftone screen definitions, color correction, and CMYK color specification.

Because the Display PostScript system has been implemented on various system platforms, some of the information required by application programmers is necessarily system-specific. Therefore in addition to the manuals listed above you'll need to consult the documentation provided by your system software vendor. The following system-specific documentation is available from Adobe:

X Window System Programmer's Supplement to the Client Library Reference Manual
Describes information about the Client Library interface that is specific to the X Window System™, such as context creation and additional error codes.

6.3 CLASSES FOR APPLICATION DEVELOPERS

Adobe offers regularly scheduled classes in:

- Programming in the PostScript language.
- The Display PostScript environment.

These classes are held in our East Coast and West Coast facilities and in our European office. Classes in Japan are planned for the future.

To receive a schedule of Adobe classes, please call our Training Support Line at (415) 961-4949.

6.4 THE ADOBE DEVELOPER ASSOCIATION

The Adobe Developer Association is a fee-based membership program for active developers using the PostScript page description language or the Display PostScript system to enhance their application products.

You must be a member of the Adobe Developer Association in order to take advantage of the following offerings:

- Technical support.
- Monthly mailings.
- Discounts on Adobe Systems application products.
- Free technical literature.

You can request a membership application by calling the Adobe Developer Support Line.

6.5 POSTSCRIPT STANDARDS AND CONVENTIONS

The following documents define important conventions and standards that promote compatibility, efficiency, and quality for all PostScript language applications. These documents are available through the Adobe Developer Support Line or the public access file server.

Document Structuring Conventions Specification
: Replaces Appendix C of the *PostScript Language Reference Manual* (Version 1.0 of the PostScript language document structuring conventions). These conventions are important for generating page description files suitable for print spoolers, previewer applications, and post-processors, as well as PostScript printers.

Encapsulated PostScript File Format (EPSF)
: Specifies the document format required for exchange of PostScript language files. This specification suggests a standard for importing PostScript language files in all environments. Applications that support EPSF can exchange graphical output with each other.

PostScript Printer Description Files Specification
: Describes the Adobe Systems PostScript Printer Description (PPD) files and their usage. PPD files are text files in a format that can be read by people and parsed by computers. They are useful for determining and using the special features available on printers with PostScript interpreters.

Adobe Font Metric Files Specification
: Describes the Adobe standard interchange format for communicating font metric information to people and programs.

6.6 THE PUBLIC ACCESS FILE SERVER

If you have access to Internet or UUCP electronic mail, you can use Adobe's public access file server to obtain the following information:

- Code examples.
- AFM files.
- Documentation updates.
- Conventions and standards documents listed in Section 6.5.

The public access file server is a mail-response program. That is, you send it a request by electronic mail and it mails back a response. (The "Subject:" line is treated as part of the message by the file server.)

To send mail to the file server, use one of the following addresses:

Internet	ps-file-server@adobe.com
UUCP	...!decwrl!adobe!ps-file-server

To receive a quick summary of file server commands, send the following message:

```
help
```

To receive detailed information on how to use the file server, send the following message:

```
send Documents long.help
```

A CHANGES SINCE LAST PUBLICATION OF THIS DOCUMENT

The changes to *Perspective for Software Developers* from the document dated October 25, 1989, are noted in the paragraphs below.

A section has been added to describe support services available from Adobe Systems, including the Adobe Developer Support Line, the Technical Literature Catalog, the Adobe Developer Association, and the public access file server.

The changes to *Perspective for Software Developers* from the document dated October 10, 1988, are noted in the paragraphs below.

The manual has been rewritten and reorganized. A list of suggested reading for software developers has been added. Diagrams have been provided to illustrate the following:

- The relationship of the Client Library and the PostScript interpreter to application and system software.
- The creation of a Display PostScript system application.

Index

POSTSCRIPT®
LANGUAGE

Extensions for the Display PostScript® System

ADOBE SYSTEMS
INCORPORATED

PostScript Language Extensions for the Display PostScript System

January 23, 1990

PostScript and Display PostScript are registered trademarks of Adobe Systems Incorporated.

*Helvetica, *Palatino, and *Times are trademarks of Linotype AG and/or its subsidiaries.
UNIX is a registered trademark of AT&T Information Systems.

Revised by Amy Davidson.

Contents

1 INTRODUCTION

This manual describes a number of extensions to the PostScript® language. These extensions are initially implemented in the Display PostScript® system. However, the utility of most of the extensions is not limited to display applications; we anticipate that those extensions will eventually be incorporated into printer products as well.

As a matter of policy, we do not refer to a 'Display PostScript language'. There is only one PostScript language, which evolves over time to encompass a wider variety of device technologies, environments, and applications. The new facilities described in this manual constitute a set of extensions to the existing PostScript language. Considerable effort has gone into making these extensions upward-compatible from the existing language and integrating them with the PostScript language and imaging models in a harmonious way. We intend that a majority of these extensions will ultimately become a standard part of the language.

Most of the extensions fall into a few major categories:

- The language model is enhanced in several ways. Memory management is more flexible to accommodate applications whose resource requirements are dynamic and unpredictable. Multiple PostScript execution contexts can execute simultaneously on behalf of separate applications sharing a single display system. There are alternative external encodings of the language for greater efficiency of generation and interpretation.
- The set of built-in imaging operations is considerably expanded, though the basic imaging model is unchanged. Operations that are performed frequently by most applications are streamlined to provide convenient generation and highly optimized execution. Although these extensions are motivated by the needs of display-based applications, their utility is not limited to those applications.
- Additional extensions are included to serve the special needs of computer display systems. These extensions adapt the PostScript imaging model to the interactive, dynamic display environment presented by an underlying window

system. Some extensions are generic and apply to all environments; those are described in this manual. Others are specialized to a particular environment or a particular integration of the Display PostScript system with a window system; those are described in documentation provided by the window system developer.

Conceptual and descriptive information regarding the various types of extensions is presented in Sections 2 through 15, roughly in the above order. Individual operator descriptions are listed alphabetically in Section 16. Although we attempt to give a rationale for each extension individually, appreciating the full purpose of the extensions as a whole requires an overall understanding of the Display PostScript system and the environments in which it is designed to operate. This topic is discussed in *Perspective for Software Developers*; see also its list of manuals that document the Display PostScript system.

A majority of the extensions can be implemented in terms of the existing PostScript language, though not necessarily with great efficiency. Through such emulation, we can provide backward compatibility between applications that use the extensions and existing PostScript language implementations that do not support them directly. A few extensions are unique to display applications and are not relevant to printing applications; those, obviously, cannot be emulated.

Other extensions

The PostScript language has already received one major extension, which has appeared in all printers with PostScript interpreter versions 25.0 and greater. This extension was not documented in the original edition of *PostScript Language Reference Manual*, but it is in editions copyright 1986 or later, as well as in Appendix B of this manual.

In order to deal with color output devices, several new operators have been defined. Most of these operators provide control over various aspects of the color rendering process, including color halftoning and undercolor removal. (The halftone dictionary extension, described in this manual, encompasses the functions of some of those operators.) Additionally, there is a **colorimage**

operator for rendering color sampled images. See *PostScript Language Color Extensions* for detailed information.

2 ALTERNATIVE LANGUAGE ENCODINGS

The standard PostScript language is based on the printable subset of the ASCII character set, as described in Section 3.3 of the *PostScript Language Reference Manual.* This representation is highly portable; it is easy to transmit and to store in a wide variety of operating system and communications environments. We refer to this representation as the *ASCII encoding* of the PostScript language.

Although it is portable, the ASCII encoding of the language is not particularly compact, nor is it efficient to generate and to interpret. In environments served by the Display PostScript system, there is a much closer coupling between the producer and the consumer of PostScript language programs than is typical when sending page descriptions to a printer. The application program and the PostScript interpreter communicate in real time; usually they are either in the same machine or are connected by a high-performance communication system. In such environments, compactness or efficiency are more important than maximum portability.

Binary encodings

The Display PostScript system supports two additional encodings of the PostScript language: the *binary token* encoding and the *binary object sequence* encoding. These encodings are extensions to the syntax of the language; that is, they provide different ways to express programs, but they introduce no new semantics. The PostScript language scanner (see *PostScript Language Reference Manual,* Section 3.3) has been extended to recognize the binary encodings in addition to the existing ASCII encoding.

The ASCII and binary encodings can be freely intermixed in any program; the scanner produces the same sequence of objects for a given program, regardless of how the program is encoded. It should be straightforward to translate from one encoding of the language to another.

The binary encodings are intended exclusively for machine generation; it is unreasonable for a human programmer to deal with them. Furthermore, applications using the Display PostScript system are encouraged to make use of the *Client Library*, a procedural interface to capabilities of the PostScript language, and *pswrap*, a translator for PostScript language program fragments. The design of the binary encodings is based primarily on the needs of those facilities. Most applications, therefore, need not be concerned with the details of these encodings.

The *binary token* encoding represents elements of the PostScript language as individual syntactic entities. This encoding emphasizes compactness over efficiency of generation or interpretation. Most elements of the language, such as integers, reals, and operator names, are represented by fewer characters in the binary encoding than in the ASCII encoding. This encoding is most suitable for environments in which communication bandwidth or storage space is the scarce resource.

The *binary object sequence* encoding represents a sequence of one or more PostScript objects as a single syntactic entity. This encoding is not compact, but it can be generated and interpreted very efficiently. Most elements of the language are in a natural machine representation or something very close to one. Additionally, this encoding is oriented toward sending fully or partially precompiled sequences of objects as opposed to ones generated on the fly. This organization matches that of the Client Library, which is the principal interface between applications and the Display PostScript system. It is most suitable for environments in which execution costs dominate communication costs.

Use of the binary encodings requires that the communication channel between the application and the PostScript interpreter be fully transparent. That is, it must be capable of carrying an arbitrary sequence of arbitrary 8-bit character codes, with no characters reserved for communication functions, no 'line' or 'record' length restrictions, etc. If the communication channel is not transparent, the ASCII encoding must be used.

Remember that the various language encodings apply only to

characters consumed by the PostScript language scanner. Applying **exec** to an executable file or string object invokes the scanner, as does the **token** operator. File operators such as **read** and **readstring**, however, simply read the incoming sequence of characters as data, not as encoded PostScript language programs.

The first character of each token determines what encoding is to be used for that token. If it is in the range 128 to 159 inclusive (that is, one of the first 32 codes with the high-order bit set), one of the binary encodings is used;[1] For binary encodings, the character code is treated as a *token type*: it determines which encoding is used and sometimes also specifies the type and representation of the token.

Note that the determination of encoding occurs only on the *first* character of each token (ignoring any white space that precedes the token). Subsequent characters are interpreted according to that encoding until the end of the token is reached, regardless of character codes. For example, a character code in the range 128 to 159 can appear within an ASCII string literal or a comment (however, a binary token type character does terminate a preceding ASCII name or number token). Similarly, a character code outside the range 128 to 159 can appear within a multiple-byte binary encoding.

Token type 159 is reserved for introducing tokens whose syntax and semantics are specific to a particular implementation of the PostScript interpreter (or a particular integration with a window system). The standard language does not specify anything about such tokens, even to say how long they are.

Number representations

Binary tokens and binary object sequences use various representations for numbers. Some numbers are the values of PostScript number objects (integers and reals); others provide structural information, such as lengths and offsets within binary object sequences.

[1]These codes are considered to be 'control characters' in most standard character sets, such as ISO and JIS; they do not have glyphs assigned to them and are therefore unlikely to be used to construct names in PostScript language programs. A means exists to disable interpretation of binary encodings; see the **setobjectformat** operator in Section 3.

Different machine architectures use different representations for numbers. The two most common variations are the byte order within multiple-byte integers and the format of real (floating-point) numbers.

Rather than specify a single convention for representing numbers, the language provides a choice of representations. The application program chooses whichever convention is most appropriate for the machine on which it is running. The PostScript language scanner accepts numbers conforming to any of the conventions, translating to its own internal representation when necessary. This translation is needed only when the application and the PostScript interpreter are running on machines with different architectures.

The number representation to be used is specified as part of the token type (the initial character of the binary token or binary object sequence). There are two independent choices, one for byte order and one for real format. The byte order choices are:

- *High-order byte first* ('big-endian') – in a multiple-byte integer or fixed-point number, the high-order byte comes first, followed by successively lower-order bytes.
- *Low-order byte first* ('little-endian') – in a multiple-byte integer or fixed-point number, the low-order byte comes first, followed by successively higher-order bytes.

The real format choices are:

- *IEEE standard* – a real number is represented in IEEE 32-bit floating-point format.[2] The order of the bytes is the same as the integer byte order, as specified above. For example, if the high-order byte of an integer comes first, then the sign and first 7 exponent bits of an IEEE standard real come first.
- *Native* – a real is represented in the native format for the machine on which the PostScript interpreter is running. This may be a standard format or something completely different; the choice of byte order is not relevant. The application program is responsible for finding out what the correct format is. In general, this is useful only in environ-

[2]IEEE 754: *Standard for Binary Floating-Point Arithmetic*, 1985.

ments where it is known that the application and the PostScript interpreter are running on the same machine or on machines with compatible architectures. Obviously, PostScript language programs that use this real number representation are not portable.

Since each binary token and binary object sequence defines its own number representation, binary encoded programs with different number representations can be freely intermixed. This is a convenience for applications that obtain portions of PostScript language programs from different sources.

Binary tokens

Binary tokens are variable-length binary encodings of certain types of PostScript objects. A binary token represents an object that can also be represented in the ASCII encoding, but usually with fewer characters. Thus, the binary encoding is usually the most compact representation of a program, though not necessarily the most efficient to execute.

Semantically, a binary token is equivalent to some corresponding ASCII token. When the scanner encounters the binary encoding for the integer 123, it produces the same result as when it encounters an ASCII token consisting of the characters '123'. That is, it produces an integer object whose value is 123; the object is the same (and occupies the same amount of space if stored in VM) whether it came from a binary or an ASCII token.

Unlike the ASCII and binary object sequence encodings, the binary token encoding is incomplete: not everything in the language can be expressed as binary tokens. For example, it makes no sense to have binary token encodings of '{' and '}', since their ASCII encodings are already compact. Similarly, it makes no sense to have binary encodings for the names of operators that are rarely used, since their contribution to the overall length of a PostScript language program is negligible. The incompleteness of the binary token encoding is not a problem, since ASCII and binary tokens can be freely intermixed.

The binary token encoding is summarized in the following table. A binary token begins with a token type character, as discussed

earlier. A majority of the token types (128 to 159) are used for binary tokens; the remainder are used for binary object sequences or are unassigned. The token type determines how many additional characters comprise the token and how the token is interpreted.

Token type(s)	*Additional characters*	*Interpretation*
128 – 131	–	binary object sequence; this encoding is described in the next section.
132	4	32-bit integer, high-order byte first.
133	4	32-bit integer, low-order byte first.
134	2	16-bit integer, high-order byte first.
135	2	16-bit integer, low-order byte first.
136	1	8-bit integer, treating the character after the token type as a signed number n; $-128 \le n \le 127$.
137	3 or 5	16- or 32-bit fixed-point number. The number representation (size, byte order, and scale) is encoded in the character immediately following the token type; the remaining two or four characters are the number itself. The representation parameter is treated as an unsigned integer r in the range 0 to 255: $0 \le r \le 31$ 32-bit fixed-point number, high-order byte first; the *scale* parameter (number of bits of fraction) is equal to r. $32 \le r \le 47$ 16-bit fixed-point number, high-order byte first; *scale* = $r - 32$. $r \ge 128$ same as $r - 128$ except that all numbers are given low-order byte first.
138	4	32-bit IEEE standard real, high-order byte first.
139	4	32-bit IEEE standard real, low-order byte first.
140	4	32-bit native real.
141	1	boolean. The character following the token type gives the value: 0 for *false*, 1 for *true*.
142	1 + n	string of length n. The parameter n is contained in the character after the token type; $0 \le n \le 255$. The n characters of the string follow the parameter.
143	2 + n	long string of length n. The 16-bit parameter n is contained in the two characters after the token type, represented high-order byte first; $0 \le n \le 65535$. The n characters of the string follow the parameter.
144	2 + n	long string of length n. The 16-bit parameter n is contained in the two characters after the token type, represented low-order byte first; $0 \le n \le 65535$. The n characters of the string follow the parameter.

145	1	literal name from system name table indexed by *index*. The *index* parameter is contained in the character after the token type; 0 ≤ *index* ≤ 255.
146	1	executable name from system name table indexed by *index*. The *index* parameter is contained in the character after the token type; 0 ≤ *index* ≤ 255.
147	1	literal name from user name table indexed by *index*. The *index* parameter is contained in the character after the token type; 0 ≤ *index* ≤ 255.
148	1	executable name from user name table indexed by *index*. The *index* parameter is contained in the character after the token type; 0 ≤ *index* ≤ 255.
149	3 + data	homogeneous number array. This consists of a four-character header (including the token type) followed by a variable length array of numbers whose size and representation are specified in the header. This is described in detail below.
150 – 158	–	unassigned; occurrence of a token with these types will cause a **syntaxerror**.
159	unspecified	reserved for token types that are implementation or window system specific.

The encodings for integers, reals, and booleans are straightforward and require no further explanation. The other token types require additional discussion.

A *fixed-point* number is a binary number having integer and fractional parts; the position of the binary point is specified by a separate *scale* value. In a fixed-point number of *n* bits, the high-order bit is the sign, the next *n* – *scale* – 1 bits are the integer part, and the low-order *scale* bits are the fractional part. For example, if the number is 16 bits wide and *scale* is 5, it is interpreted as a sign, a 10-bit integer part, and a 5-bit fractional part. A negative number is represented in two's complement form.

There are both 16- and 32-bit fixed-point numbers, allowing an application to make a tradeoff between compactness and precision. Regardless of the token's length, the object produced by the scanner for a fixed-point number is an integer if *scale* is zero; otherwise it is a real. Note that a 32-bit fixed-point number actually takes *more* characters to represent than a 32-bit real; it is useful only if the application already represents numbers that way. (Using this representation makes somewhat more sense in homogeneous number arrays, described below.)

A string token specifies the string's length as a one- or two-character unsigned integer. The specified number of characters of the string follow immediately. Note that all the characters are

treated literally; there is no special treatment of '\' or other characters. The main purpose of the binary token encoding of strings is to allow arbitrary binary data to be represented straightforwardly, not to save space.

The name encodings specify a *system name index* or a *user name index* that selects a name object from the system or user name table and uses it as either a literal or an executable name. This mechanism is described below. Note that only the first 256 elements of each array can be accessed by this means.

A *homogeneous number array* is a single binary token that represents a PostScript literal array object whose elements are all numbers. The token consists of a four-character header (including the token type) followed by a variable-length sequence of numbers. All of the numbers are represented in one way, which is specified in the header.

The header consists of the token type character (149, denoting a homogeneous number array), a character that describes the number representation, and two characters that specify the array length (number of elements). The number representation is treated as an unsigned integer r in the range 0 to 255 and is interpreted as follows:

$0 \le r \le 31$	32-bit fixed-point number, high-order byte first; the *scale* parameter (number of bits of fraction) is equal to r.
$32 \le r \le 47$	16-bit fixed-point number, high-order byte first; *scale* = $r - 32$.
48	32-bit IEEE standard real, high-order byte first.
49	32-bit native real.
$128 \le r \le 177$	same as $r - 128$ except that all numbers are given low-order byte first.

Note that this interpretation is similar to that of the representation parameter r in individual fixed-point number tokens.

The array's *length* is given by the last two characters of the header, treated as an unsigned 16-bit number. The byte order in this field is as specified by the number representation: $r < 128$ indicates high-order byte first; $r \ge 128$ indicates low-order byte first.

Following the header are 2 × *length* or 4 × *length* characters, depending on representation, that encode successive numbers of the array.

When this class of token is consumed by the PostScript language scanner, it produces a literal array object. The elements of this array are all integers if the representation parameter *r* is 0, 32, 128, or 160 (specifying fixed-point numbers with a *scale* of zero); otherwise they are all reals. Once scanned, such an array is indistinguishable from an array produced by other means (and occupies the same amount of space).

Although the homogeneous number array representation is useful in its own right, it is particularly useful in conjunction with operators that take an *encoded number string* as an operand. This is described later in this section.

Binary object sequences

A binary object sequence is a single token that describes an executable array of objects, each of which may be a simple object, a string, or another array nested to arbitrary depth. The entire sequence can be constructed, transmitted, and scanned as a single self-contained syntactic entity.

Semantically, a binary object sequence is an ordinary executable array, as if the objects in the sequence had been surrounded by '{' and '}', but with one important difference: its execution is immediate instead of deferred. That is, when a binary object sequence is encountered in a file being executed directly by the PostScript interpreter, the interpreter performs an implicit **exec** instead of pushing the array on the operand stack as it would ordinarily. (This special treatment does not apply when a binary object sequence appears in a context where execution is already deferred, e.g., nested in ASCII-encoded '{' and '}' or consumed by the **token** operator.)

Since a binary object sequence is syntactically a single token, it is completely processed by the scanner before any of it is executed by the interpreter. The entire array and all its subsidiary composite objects are allocated in private or shared VM according to the VM allocation mode in effect at the time the binary

object sequence is scanned (see Section 4). Similarly, encoded name bindings are those in effect at scan time (see below).

The encoding emphasizes ease of construction and interpretation over compactness. Each object is represented by eight successive characters. In the case of simple objects, these eight characters describe the entire object (type, attributes, and value). In the case of composite objects, the eight characters include a reference to some other part of the binary object sequence where the value of the object resides. The entire structure is easy to describe using the data type definition facilities of implementation languages such as C and Pascal.

A binary object sequence consists of four parts in the following order:

- *header* — four or eight characters of information about the binary object sequence as a whole;
- *top-level array* — a sequence of objects, eight characters each, which constitute the value of the main array object;
- *subsidiary arrays* — more eight-character objects, which constitute the values of nested array objects;
- *string values* — an unstructured sequence of characters, which constitute the values of string objects and the text of name objects.

The first character of the header is the token type, mentioned earlier. Four token types denote a binary object sequence and select a number representation for all integers and reals embedded within it:

128	high-order byte first; IEEE standard real format
129	low-order byte first; IEEE standard real format
130	high-order byte first; native real format
131	low-order byte first; native real format

At this point, the header can take one of two forms, depending upon the number of elements in the top level array and the overall length of the object sequence. If there are 255 top-level elements or fewer and the overall length of the object sequence is 65,535 characters or fewer, the second character specifies the number of elements in the top-level array and the third and

fourth characters, taken together as a 16-bit unsigned integer, specify the size in characters of the entire binary object sequence, including header, top-level, and subsidiary arrays, and string values. (The order of characters that constitute this size field is according to the number representation specified by the token type. This is true of *all* multi-character numbers in the binary object sequence.) If there are more than 255 top-level objects or the overall length of the object sequence is greater than 65,535 characters, the second character is set to zero. The next two bytes are the number of top-level elements and the next four bytes are the overall length of the object sequence (again, the order of characters that constitute these size fields is according to the number representation specified by the token type).

Following the header is an uninterrupted sequence of eight-character objects that constitute both the top-level array and subsidiary arrays. The length of this sequence is not given explicitly; it continues until the earliest string value referenced from an object in the sequence, or until the end of the entire token.

The first character of each object in the sequence gives the object's literal/executable attribute in the high-order bit and its type in the low-order 7 bits.[3] The attribute values are:

0 literal
1 executable

The meaning of the object type field is given below.

The second character of an object is unused; its value must be zero. The third and fourth characters constitute a 16-bit integer, referred to as the *length*. The fifth through eighth characters constitute the *value*. The interpretation of the length and value fields depends on the object's type. (Once again, the character order within these fields is according to the number representation for the binary object sequence overall.)

The object types and the interpretation of the length and value fields are:

[3]Note that the positions of these fields within the character are *not* influenced by the prevailing number representation. To describe these as distinct fields in a C 'struct' requires different type definitions for big-endian and little-endian machines.

0 null: length and value are unused
1 integer: length is unused; value is a signed 32-bit integer
2 real: length is unused; value is a real
3 name: see below
4 boolean: length is unused; value is 0 for false, 1 for true
5 string: see below
6 immediately evaluated name: see below
9 array: see below
10 mark: length and value are unused

For types *string* and *array*, the length field specifies the number of elements (characters in a string or objects in an array); it is treated as an unsigned 16-bit integer. The value field specifies the offset, in characters, of the start of the object's value relative to the first character of the first object in the top-level array. An array offset must refer somewhere within the top-level or subsidiary arrays; it must be a multiple of 8. A string offset must refer somewhere within the string values; the strings have no alignment requirement and need not be null-terminated or otherwise delimited. (If the length of a string or array object is zero, its value is disregarded.)

For the *name* type, the length field is treated as a *signed* 16-bit integer that selects one of three interpretations of the value field:

$n > 0$ value is an offset to the text of the name, just the same as for a string; n is the name's length (which must be within the implementation limit for names)
0 value is a *user name index* (see below)
−1 value is a *system name index* (see below)

An *immediately evaluated name* object is analogous to the '*//name*' syntax of the ASCII encoding. (See Appendix B.) This object is treated just the same as a name, as described above. However, the scanner then immediately looks up the name in the context of the current dictionary stack and substitutes the corresponding value for that name. If the name is not found, an **undefined** error occurs.

For the composite objects, there are no enforced restrictions against multiple references to the same value or recursive or self-referential arrays. However, such structures cannot be expressed

directly in the ASCII or binary token encodings of the language; their use violates the interchangeability of the encodings. Therefore, the recommended structure of a binary object sequence is for each composite object to refer to a distinct value. There is one exception: references from multiple name objects to the same string value are specifically encouraged, since name objects are unique by definition.

The scanner will generate a **syntaxerror** upon encountering a binary object sequence that is malformed in any way. Possible causes include:

- an object type that is undefined;
- an 'unused' field that is not zero;
- lengths and offsets that, in combination, would refer outside the bounds of the binary object sequence;
- an array offset that is not a multiple of 8 or that refers beyond the earliest string offset.

As is true for all errors, when a **syntaxerror** occurs, the PostScript interpreter pushes onto the operand stack the object that caused the error. For an error detected by the scanner, however, there is not actually such an object, since the error occurred before the scanner had finished creating one. Instead, the scanner fabricates a string object consisting of the characters encountered so far in the current token. If a binary token or binary object sequence was being scanned, the string object produced is a description of the token rather than the literal characters (which would be gibberish if printed as part of an error message). For example:

```
(bin obj seq, type=128, elements=23, size=234,
 array out of bounds)
```

System and user name encodings

Both the binary token and binary object sequence encodings provide optional means for representing names as small integers instead of as full text strings. Such an integer is either a *system name index* or a *user name index*. Careful use of encoded names can result in substantial space savings and execution performance improvement.

A name index is a reference to an element of a name table already known to the PostScript interpreter. When the scanner encounters a name token that specifies a name index (rather than a text name), it immediately substitutes the corresponding element of the appropriate table. This substitution occurs at scan time, not at execution time; the result of the substitution is an ordinary PostScript name object.

A system name index is an index into the system name table, which is built-in and has a standard value. The elements of this table are standard operator names, font names, character names, and other names that are a standard part of the PostScript VM. The contents of this table are documented in appendix C; they are also available as a machine-readable file for use by *pswrap*, translators, and other programs that deal with binary encodings.

A user name index is an index into the user name table, whose contents may be defined by a PostScript language program by means of the **defineusername** operator. This provides efficient encodings of non-system names that are used frequently. However, there are various restrictions on user name encodings; additions to the user name table must be made in a stylized way to ensure correct behavior.

If there is no name associated with a system or user name index, the scanner generates an **undefined** error; the offending command is 'system*n*' or 'user*n*', where *n* is the decimal representation of the index.

An encoded binary name specifies (as part of the encoding) whether the name is to be literal or executable; this overrides the corresponding attribute of the replacement name object. Thus, a given element of the system or user name table can be treated as either literal or executable when referenced from a binary token or object sequence. In the binary object sequence encoding, one can also specify an immediately evaluated name object, analogous to '*//name*'. When such an object specifies a name index, note that there are *two* substitutions: the first obtains a name object from the appropriate table; the second looks up that name object in the current dictionary context.

One should be aware that the binary token encoding provides means to reference only the first 256 elements of either of the

name tables. (The binary object sequence encoding does not have this limitation.) Maximum program compactness can be achieved by organizing the user name table in such a way that the most commonly used names are in the first 256 elements.

Like everything else having to do with binary encodings, encoded names are intended for machine generation only. The *pswrap* and Client Library facilities are the preferred means for application programs to generate binary encoded programs. In particular, those facilities maintain the user name table automatically and encode names using both the system and user name tables. An application should not attempt to alter the user name table itself, since that would interfere with the activity of the Client Library.

A program can depend on a given system name index representing a particular name object. Applications that generate binary encoded PostScript language programs are encouraged to take advantage of system name index encodings, since they save both space and time.

The meaning of a given user name index is local to a specific PostScript execution context – more precisely, to a context's private VM or *space* (see Sections 4 and 5). If several contexts are associated with the same space, a user name index defined in one context may be used in another context. (It is the client's responsibility to synchronize execution of the contexts so that definition and use occur in the correct order.)

The user name index facility is intended for use only during interactive sessions with a Display PostScript system. It should not be used in a PostScript language program that must stand by itself, such as one sent to a printer or written to a file for later use. If a program contains user name index encodings, it cannot be composed with or embedded in other PostScript language programs and it cannot easily be translated to the ASCII encoding. PostScript printers may not support user definition of name encodings. The Client Library has an option to disable use of user name encodings and produce text encoded names always; this option may be invoked dynamically by an application program to produce a PostScript language program that is to be captured in a file or diverted to a printer.

Encoded number strings

Several of the new operators require as operands an indefinitely long sequence of numbers to be used as coordinate values (either absolute or relative). The operators include those dealing with user paths, rectangles, and explicitly positioned text, all of which are described in other parts of this manual. In the most common use of these operators, all the numbers are provided as literal values by the application as opposed to being computed by the PostScript language program.

In order to facilitate this common use and to streamline both the generation and the interpretation of numeric operand sequences, we have defined a standard facility for presenting such operands to an operator. A number sequence may be represented either as an ordinary PostScript array object (whose elements are to be used successively) or as an *encoded number string*.

An encoded number string is a PostScript string object that consists of a single *homogeneous number array* according to the binary token encoding described above. That is, the first four characters are treated as a header; the remaining characters are treated as a sequence of numbers encoded as described in the header.

The attractive feature of an encoded number string is that it is a compact representation of a number sequence both in its external form *and in VM*. Syntactically, it is simply a string object; it remains in that form after being scanned and placed in VM. It is interpreted as a sequence of numbers only when it is actually used as an operand of an operator that is expecting a number array. Furthermore, even then it is neither processed by the scanner nor expanded into a PostScript array object; instead, the numbers are consumed directly by the operator. This arrangement is both compact and efficient.

The following are equivalent ways of invoking **rectfill**, which is one of the new operators that expect number sequences as operands:

```
[ ASCII-encoded numbers ] rectfill
homogeneous number array  rectfill
string  rectfill
```

The first line constructs an ordinary PostScript array object containing the numbers and passes it to **rectfill**. (This is actually the most general form, since the '[' and ']' could enclose an arbitrary computation that produces the numbers and pushes them on the stack.)

On the second line, a binary token representing a homogeneous number array appears directly in the program. In this instance, the scanner produces an array object, which is then consumed by **rectfill**. The **rectfill** operator treats this case as indistinguishable from the first one.

On the third line, a *string* object appears in the program. This string object is most likely encoded as a binary token or an element of a binary object sequence, but conceivably it could be an ASCII-encoded hexadecimal string enclosed in '<' and '>' or a string value read by **readstring**. (An ordinary ASCII string enclosed in '(' and ')' is less suitable because of the need to quote special characters.) When **rectfill** is given a string object, it interprets the *value* of the string as the binary token encoding of a homogeneous number array. The result produced is equivalent to:

```
string cvx exec rectfill
```

Here, **exec** interprets *string* as a PostScript language program. The scanner, finding that the first (and only) token in *string* is a binary token encoding of a homogeneous number array, produces that array and pushes it on the operand stack. The **rectfill** now sees an array operand, as in one of the first two lines in the earlier example. However, although the end result is the same, passing *string* directly to **rectfill** is much more efficient (in both time and space), since it bypasses creating the array object in VM.

The operators that use encoded number strings include **rectfill**, **rectstroke**, **rectclip**, **rectviewclip**, **xshow**, **yshow**, and **xyshow**. Additionally, an encoded user path represents its numeric operands as an encoded number string; the relevant operators are **ufill**, **ueofill**, **uappend**, **inufill**, **inueofill**, and **inustroke**.

3 STRUCTURED OUTPUT

The Display PostScript system provides a means for a program to send various kinds of information back to the application (via the Client Library). This information includes the values of objects produced by queries, error messages, unstructured text generated by **print**, and perhaps window system specific events. A PostScript context writes all of this data to its standard output file. The Client Library requires a way to distinguish among these different kinds of information received from a context.

To serve this need, we have defined a *structured output format* and provided means for a PostScript language program to generate output conforming to it. The format is basically the same as the *binary object sequence* representation for input, described in Section 2.

A program that writes structured output should be judicious in its use of unstructured output primitives such as **print** and '**=**'. In particular, since the start of a binary object sequence is indicated by a character whose code is in the range 128 to 159 inclusive, unstructured output should consist only of character codes outside that range; otherwise, confusion will ensue in the Client Library or the application. (Of course, this is only a convention; by prior arrangement, a program may send unstructured data to the application.)

The new operator **printobject** writes an object to the standard output file as a binary object sequence. A similar operator, **writeobject**, writes to a file. The binary object sequence contains a top-level array consisting of one element which is the object being written; see the description of binary object sequences in Section 2. That object, however, can be composite, so the binary object sequence may include subsidiary arrays and strings.

In the binary object sequences produced by **printobject** and **writeobject**, the number representation is controlled by the **setobjectformat** operator. The binary object sequence has a token type that identifies the representation used.

Accompanying the top-level object in the object sequence is a one-character *tag*, which is specified as an operand of

printobject. This tag is carried in the second character of the object, which is otherwise unused (see Section 2). Only the top-level object receives a tag; the second byte of subsidiary objects is zero. In spite of its physical position, the tag is logically associated with the object sequence as a whole.

The purpose of the tag is to enable the PostScript language program to specify the intended disposition of the object sequence. A few tag values are reserved for reporting errors (see below); the remaining tag values may be used arbitrarily. The Client Library uses tags when it issues a query to the PostScript context. The query consists of a PostScript language program that includes one or more instances of **printobject** to send responses back to the Client Library. A different tag is specified for each **printobject** so that the Client Library can distinguish among the responses as they arrive.

Tag values 0 through 249 are available for general use. Tag values 250 through 255 are reserved to identify object sequences that have special significance. Of these, only tag value 250 is presently defined: it is used to report errors.

Errors are initiated as described in Sections 3.6 and 3.8 of the *PostScript Language Reference Manual*. Normally when an error occurs, control automatically passes from the PostScript language program to an error-handling procedure in the root control program of the context. If binary encoding is disabled (see **setobjectformat**), the error handler generates a text message similar to an error message on a PostScript printer. Otherwise it writes a binary object sequence with a tag value of 250.

The binary object sequence that reports an error contains a four-element array as its top-level object. The array elements, ordered as they appear, are:

- The name 'Error' (indicates an ordinary error detected by the PostScript interpreter; a different name could indicate another class of errors, in which case the meanings of the other array elements might be different).
- The name that identifies the specific error (e.g., **typecheck**).
- The object that was being executed when the error oc-

curred; if the object that raised the error is not printable, some suitable substitute is provided — for example, an operator name in place of an operator object.

- An error-handler flag (a boolean object whose value is *true* if the program expects to resynchronize with the client and *false* otherwise).

The normal error handler, **handleerror**, sets the flag to *false*. An alternate error handler, **resynchhandleerror**, sets the flag to *true*; it should be used when the program expects to resynchronize with the client. See the section on handling errors in the *Client Library Reference Manual* for more information on **handleerror** and **resynchandleerror**.

In addition to binary object sequences and unstructured text, a program may need to send special tokens whose syntax and semantics are implementation or environment dependent. For example, if a PostScript language program is able to intercept window system events, it may need to send some of those events to the application. Binary token type 159 is reserved for this purpose (see Section 2).

4 MEMORY MANAGEMENT

The PostScript interpreter used in printers has a very simple approach to management of virtual memory (VM) resources. Memory consumed by creating new composite objects is simply not reclaimed until a **restore** is executed; the VM then reverts to the state it was in at the time of the matching **save**.

This approach works well for PostScript printers. Execution of a PostScript page description should ordinarily have no lasting side effects. A page description is divided into pages; individual pages should have no lasting side effects that would influence the execution of subsequent pages. The strict nesting of VM states imposed by the **save/restore** facility matches this structure well. To ensure portability, PostScript language programs that are page descriptions should assume that VM is managed in this way.

Interactive display applications, on the other hand, perform

operations in a much less structured fashion. The stream of PostScript language text generated by an application is typically not divided into 'pages' and may have no obvious overall structure. Furthermore, an interactive session may never terminate; there is no opportunity to reclaim VM resources consumed during the session. Thus, **save** and **restore** are much less suitable for overall memory management, though they can still be useful for encapsulating isolated computations.

Garbage collection

A more sophisticated approach to memory management is clearly required. The Display PostScript system includes an automatic VM reclamation facility, popularly known as a 'garbage collector'. This facility automatically reclaims the memory occupied by composite objects that are no longer accessible to the PostScript language program (i.e., do not appear on any of the stacks or as elements of other composite objects).

Garbage collection is not a language feature *per se*, since it normally takes place without explicit action on the part of the PostScript language program being executed. However, the presence of a garbage collector strongly influences the style of programming that is permissible. A program that endlessly consumes VM and never executes **save** and **restore** will eventually encounter a **VMerror** if executed by a PostScript interpreter that does not have garbage collection.

Of course, garbage collection is not entirely free. There is a certain cost associated with creating and destroying composite objects in VM. The most common case is that of literal objects (strings, user path procedures, etc.) that are immediately consumed by operators such as **show** and **ufill** and then never used again. The garbage collector is engineered to deal with this case inexpensively, so application programs should not hesitate to take advantage of it. However, the cost of garbage collection is greater for objects that have longer lifetimes or that are allocated explicitly. Programs that frequently require temporary objects are encouraged to create them once and reuse them instead of creating new ones on every use.

Even with garbage collection, the **save** and **restore** operators still

have their standard behavior. That is, **restore** still resets all objects visible to the PostScript language program to their state at the time of the matching **save**. It still reclaims all composite objects created since the matching **save** (and does so very cheaply). Thus, a Display PostScript application may continue to use the **save/restore** facility in cases where its semantics are useful.

In an environment with garbage collection, the semantics of **vmstatus** are not as well defined as they are in an environment with explicit memory management. The garbage collection process operates intermittently, not continously; some inaccessible objects cannot immediately be recognized as such. Thus, the *used* value returned by this operator is only meaningful immediately after a garbage collection has taken place. This can be invoked explicitly by the **vmreclaim** operator. The **setvmthreshold** operator provides additional control over the behavior of the garbage collector.

Deliberate discard and **undef**

With garbage collection comes the opportunity to deliberately discard composite objects that are no longer needed and to do so in an order unrelated to the time of creation of those objects. This is particularly valuable for very large objects such as font definitions. In order for this to be done effectively, certain programming considerations must be observed; these considerations arise mainly from interactions with **save** and **restore**.

As explained above, the VM occupied by a composite object can be reclaimed by the garbage collector as soon as it becomes inaccessible to the PostScript language program. For example, if the only reference to a particular composite object consists of an element of some array or dictionary, replacing that element with a null object (say, using **put**) renders the former object's value inaccessible and reclaimable.

In the case of a dictionary, it is useful to be able to remove an entry entirely, that is, to remove both the key and the value of a key-value pair, as opposed to replacing the value with some other value. This action is performed by the new operator **undef**, which is described below. Removing an entry from the **FontDirectory** dictionary requires another new operator,

undefinefont, since **FontDirectory** is read-only except by font specific operators.

Regardless of the means used to remove references to a composite object, the action will be undone by a subsequent **restore** if the reference existed at the time of the matching **save**. This is true even for **undef**: **restore** reinstates the deleted dictionary entry. In this situation, the referenced object has never become truly inaccessible, since access to it can be reinstated by executing **restore**. Consequently, the VM occupied by that object is not reclaimed.

As a practical matter, this means that a PostScript language program can successfully discard a composite object only while executing at the same depth of **save/restore** nesting as was in effect when the object was created. Fonts are typically defined at the outermost level of **save/restore** nesting (or in shared VM, as described below). To discard a font definition and reclaim the VM that it occupies, one must execute **undefinefont** at the same level of **save/restore** nesting.

Shared VM

The existing model of VM is that of a uniform, unstructured store of composite objects. This model has been extended to support multiple VMs whose contents have different lifetime and visibility and whose behavior with respect to **save** and **restore** is decoupled.

The motivation for introducing multiple VMs is the need to support multiple, concurrent execution contexts in the Display PostScript system. The facilities that deal with multiple contexts are described in Section 5. However, most of the semantics of the multiple VM facility can be described independently of contexts and are therefore presented here.

Each PostScript execution context has a *private* VM that is visible only within that context. Additionally, there is a single *shared* VM that is visible to all contexts and that can be updated

by any context under suitable conditions.[4]

Of all the objects visible to a PostScript language program, some are in private VM and some are in shared VM. New composite objects, whether created implicitly by the PostScript language scanner or explicitly by operators, are normally allocated in private VM. A program can read and alter the values of objects in private VM in the usual way, subject only to the access attributes of the objects involved. A program can also read the values of objects in shared VM without any unusual restrictions. Thus, for most purposes, the behavior of the two-part VM is virtually indistinguishable from the behavior of the conventional one-part VM.

The ability to alter the values of objects in shared VM is restricted in one important way.[5] It is illegal to store a private object as an element of a shared object. More precisely, a composite object whose value was created by ordinary means (and is therefore in private VM) cannot be stored as an element of an existing composite object whose value is in shared VM. An attempt to do so will result in an **invalidaccess** error. On the other hand, there are no restrictions on storing simple objects, such as integers and names, as elements of shared objects; nor are there restrictions on storing shared objects as elements of private objects. In this connection, name objects are always treated as if they were shared. The **scheck** operator inquires whether an object is private or shared.

In order to create a new composite object in shared VM, a program must explicitly enter *shared VM allocation mode*. This is done by executing the **setshared** operator, which switches between private and shared VM allocation modes. This mode controls the VM region in which the values of new composite ob-

[4]Even if a PostScript interpreter supports only one context, as in a printer, having a 'shared' VM is still useful. The shared VM holds objects whose lifetime is independent of the lifetime of objects in the (single) private VM. Such objects may include font definitions that are to persist through execution of multiple print jobs. In this respect, shared VM is a replacement for the cumbersome and less general **exitserver** mechanism.

[5]The ability to alter the shared VM may be further restricted in some environments. For example, a PostScript printer may require a program to present a password to some **statusdict** operator before attempting to alter the shared VM. Such restrictions do not ordinarily make sense in environments served by the Display PostScript system.

jects are subsequently allocated; it affects both objects created implicitly by the scanner and ones created explicitly by operators. Such objects can be stored as elements of other objects (both shared and private) without restriction. The allocation mode also has certain other effects that are explained below.

The modifications made to the shared VM, including creation of new shared objects while in shared VM allocation mode, are not affected by subsequent execution of **restore**. That is, a **restore** does *not* undo the modifications to the shared VM, even if the matching **save** preceded the modificatons. It does, however, undo changes made to the private VM. Objects in shared VM are reclaimed only by the garbage collector; this occurs when those objects are no longer accessible from any context.

Certain standard dictionaries are located in shared VM and others in private VM. Storing a shared object into a shared dictionary is the normal way of making that object visible to other contexts. The standard shared dictionaries are:

---

systemdict the standard system dictionary, which is always read-only.

shareddict a new standard shared dictionary, which is writable by any context. This dictionary is stored as **shareddict** in **systemdict**. It is permanently on the dictionary stack, below **userdict** and above **systemdict**.

SharedFontDirectory a dictionary consisting of fonts installed by executing **definefont** while in shared VM allocation mode. This dictionary is stored as **SharedFontDirectory** in **systemdict**. The **findfont** procedure looks first in the private **FontDirectory**, then in **SharedFontDirectory**. This is also the case for the new **selectfont** operator (see Section 10).

The standard private dictionaries[6] are:

---

userdict the standard user dictionary. This dictionary is stored as **userdict** in **systemdict**; however, as viewed by each context, the value of **userdict** is the one located in that context's private VM.

errordict the standard error dictionary (stored as **errordict** in **systemdict** the same way as **userdict**).

[6]Although logically there is a separate instance of each of these dictionaries in each context's private VM, they are implemented in such a way that a separate instance is created only if the dictionary is modified. This optimization is invisible to a PostScript language program.

statusdict the standard dictionary for product-specific operators, procedures, and parameters (stored as **statusdict** in **systemdict** the same way as **userdict**).[7]

FontDirectory a dictionary consisting of fonts installed by executing **definefont** while in private VM allocation mode. Fonts so defined are private to the context that defined them. The **findfont** procedure looks first in **FontDirectory**, then in **SharedFontDirectory**. This dictionary is stored as **FontDirectory** in **systemdict** the same way as **userdict**. However, when shared VM allocation mode is in effect, the name **FontDirectory** is temporarily rebound to the value of **SharedFontDirectory** so that only shared fonts are visible; this ensures correct behavior of fonts that are defined in terms of other fonts.

$error a dictionary accessed by the built-in error handler procedures (stored as **$error** in **userdict**).

This organization is designed to permit font definitions to be executed in either private or shared VM allocation mode. In the latter case, the font dictionary is created in shared VM and the **definefont** enters it into **SharedFontDirectory**, where it is available to all contexts.

Although the principal intended use of shared VM is to hold font definitions, it is not limited to such use. Any definitions that are needed by several contexts may be placed in shared VM, saving both space and time. Additionally, shared VM can be used for active communication among contexts. However, several guidelines on use of shared VM must be observed in order to avoid unexpected behavior:

- If a shared program defines a dictionary (or other data structure) to hold temporary data during execution of the program, it should create the dictionary in private VM upon first use of the program in a given execution context. Using a shared dictionary for this purpose could result in interference between multiple contexts executing the same program.
- For the reason just given, the prologues for most existing PostScript language applications may not work correctly if loaded into shared VM. Such prologues need to be restructured to segregate the constant information, such as procedure definitions, from the variable information.

[7] **statusdict** is private instead of shared for compatibility with PostScript printers, in which certain device specific parameters are set by storing into **statusdict**.

- Programs that deliberately modify shared VM in order to accomplish intercontext communication may wish to take advantage of the mutual exclusion and synchronization primitives described in Section 5.

5 MULTIPLE EXECUTION CONTEXTS

The Display PostScript system is able to support the execution of multiple PostScript language programs concurrently. This capability is required when multiple application programs share a single display system and window system. Additionally, it is sometimes advantageous for a single application to be structured as multiple concurrent processes. In this section, we describe the language extensions for managing the interactions between multiple execution contexts.

Applications normally access the Display PostScript system through the Client Library, which provides access to the PostScript imaging capabilities via procedures that can be called from an implementation language such as C or Pascal. The Client Library includes procedures for creating, communicating with, and destroying PostScript execution contexts. Strictly speaking, the Client Library facilities are not part of the PostScript language definition; they are described in the *Client Library Reference Manual.*

Terminology and execution model

A *PostScript execution context* (hereafter called simply a 'context') consists of all the state that is visible to a running PostScript language program. This state includes:

- an independent thread of control. Multiple threads can be in progress concurrently.
- a set of stacks: operand stack, dictionary stack, execution stack, and graphics state stack. Starting from these stacks, one can access all state visible to a PostScript language program, such as dictionaries, paths, devices, etc.
- a private VM or *space*, discussed below.
- a shared VM, which is uniformly visible to all contexts (see Section 4).

- standard input and output files. In the Display PostScript system, these provide a means for communicating with an application program.
- miscellaneous state variables, such as the current view clip (see Section 13), garbage collector control parameter (Section 4), object output format parameter (Section 3), and array packing mode (Appendix B). Unless otherwise documented, any parameter that is not part of VM is private to each context. When a new context is created, all such parameters are initialized to their default state.

A *space* is what we have called a *private VM* in Section 4. It includes **userdict** and all new composite objects created during normal execution of a context (except when the context invokes **setshared** and alters shared VM).

In the usual case of multiple independent contexts serving multiple independent applications, each context has its own space. Thus, the behavior of the contexts is decoupled to the maximum extent possible. Contexts can interact only by deliberately altering shared VM; this is normally done only for the purpose of installing shared definitions such as fonts. At all other times, one can think of each context as a self-contained 'virtual printer'.

However, it is also possible for two or more contexts to use the same space. This implies a much closer degree of coupling among the contexts, since they must cooperate closely to maintain their common space in a consistent state. This arrangement makes sense when multiple contexts are serving a single application program. For example, an application may manage multiple instances of itself, as in a text editor with multiple windows. Or an application may itself be organized as several concurrent activities, such as tracking user interactions in the foreground while updating the displayed image in the background.

An application program can call Client Library procedures, not described here, to create multiple contexts that use the same space. Additionally, an executing PostScript language program can create a new context sharing the current context's space by executing the **fork** operator. It can also await completion of a previously forked context by executing the **join** operator.

When multiple contexts share a single space, they require a

means to synchronize their activities. To facilitate this, the language has been extended to include two new types of objects and several new operators for manipulating them.

A *lock* is a mutual exclusion semaphore that can be used by cooperating contexts to guard against concurrent access to data that they are sharing. A context acquires a lock before accessing the data and releases it afterward. During that time, other contexts are prevented from acquiring the lock, thus preventing them from accessing the data when it is in a possibly inconsistent state. The association between a lock object and the data protected by the lock is entirely a matter of programming convention.

A *condition* is a binary semaphore that can be used by cooperating contexts to synchronize their activity. One or more contexts can wait on a condition, i.e., suspend execution for an arbitrary length of time until notified by another context that the condition has been satisfied. Once again, the association between the condition object and the actual event or state that it represents is a matter of programming convention.

Although the synchronization primitives are primarily intended for use by multiple contexts that share a single space, they can also be used by any contexts to synchronize access to data in shared VM. Of course, this requires prearrangement among all contexts involved; the lock and condition objects used for this purpose must themselves be in shared VM.

Programming considerations

In any environment that supports concurrent execution of independent threads of control, there is always the possibility of deadlock. The most familiar form of deadlock arises among two or more contexts when each waits for a notification from the other or each attempts to acquire a lock already held by the other. Another deadlock situation arises when all available communication buffers become filled with data for a context that is waiting for notification from some other context, but the other context cannot proceed because it has no way to communicate. Such deadlocks can be avoided only through careful system and application design.

Scheduling of contexts in unpredictable. In some environments, the PostScript interpreter may switch control among contexts at arbitrary times (i.e., preemptively); therefore, program execution in different contexts may be interleaved arbitrarily. Preemption may occur even within a single operator, such as one that causes a PostScript language procedure to be executed or that reads or writes a file. Therefore, to ensure predictable behavior, contexts should use the synchronization primitives to control access to shared data.

Locks and conditions are ordinarily used together in a fairly stylized way; the language primitives are organized with this way of using them in mind. The **monitor** operator acquires a lock (waiting if necessary), executes an arbitrary PostScript language procedure, then releases the lock. The **wait** operator is executed within a procedure invoked by **monitor**; it releases the lock, waits for the condition to be satisfied, and reacquires the lock. The **notify** operator indicates that a condition has been satisfied and resumes any contexts waiting on that condition.

The recommended style of use of **wait** and **notify** is based on the notion that a context first waits for a shared data structure to reach some desired state, then performs some computation based on that state, and finally alerts other contexts of any changes it has made to the data. A lock and a condition are used to implement this protocol. The lock protects against concurrent access to the data; the condition is used to notify other contexts that some potentially interesting change has taken place.[8]

This protocol is illustrated by the following two program fragments; note that they are likely to be executed by different contexts.

[8]Locks and conditions are treated separately because one may want to have several conditions that represent distinct states of the same shared data.

```
lock1
{
   {
      ... boolean expression testing monitored data ...
      {exit} {lock1 cond1 wait} ifelse
   } loop
   ... computation involving monitored data ...
} monitor

lock1
{
   ... computation that changes monitored data ...
   cond1 notify
} monitor
```

The first program executes **monitor** to acquire the lock *lock1*; it must do so to safely access the shared data associated with it. The program then checks whether the *boolean expression* has become true; it waits on the condition *cond1* (repeatedly if necessary) until the expression evaluates to true. Now, while still holding the lock, it performs some *computation* based on this state of the shared data; note that it might alter the data in such a way that the *boolean expression* would evaluate false. Finally, it releases *lock1* by leaving the procedure invoked by **monitor.**

The second program acquires *lock1* and then performs some computation that alters the data in a way that might favorably affect the outcome of the *boolean expression.* It then notifies *cond1* and releases *lock1*. Any other context that is suspended at the **wait** in the first program now resumes and gets a chance to re-evaluate the *boolean expression.*

Note that it is unsafe to assume that the state tested by the *boolean expression* is true immediately after resumption from a **wait**. Even if it was true at the moment of the **notify**, it might have become false due to intervening execution by some other context. Notifying *cond1* does not necessarily certify that the value of the *boolean expression* is true, only that it might be true. Programs that conform to this protocol are immune from deadlocks due to 'lost notifies' or malfunctions due to 'extra notifies'.

Restrictions

Each context has its own private pair of standard input and output files. That is, different contexts obtain different file objects as a result of executing **currentfile** or applying the **file** operator to the names '%stdin' and '%stdout'. A context should not attempt to make its standard input and output files available for use by other contexts; doing so will cause unpredictable behavior.

The standard input file carries data addressed to this context by the application; the standard output file carries data identified as coming from the current context to the application. Obviously, a program that executes **fork** must transmit the identity of the new context to the application in order for the application to address data to that context. (However, doing so is not always required, since some forked contexts have no need to communicate over their standard input and output files.)

If multiple contexts share the same space, the semantics of **save** and **restore** become somewhat problematical. The operation performed by **restore** is logically to restore the entire space (i.e., the private VM) to its state as of the matching **save**. If one context does this, another context sharing the same space might observe the effect of the **restore** at some totally unpredictable time during its own execution; that is, its recent computations would be undone unexpectedly. This behavior is clearly not useful.

Therefore, if any context executes a **save**, all other contexts sharing the same space are suspended until the original context executes the matching **restore**. This ensures that the **restore** does not disrupt the activities of those other contexts. This restriction applies only to contexts sharing the same space; contexts associated with other spaces proceed unhindered.[9]

Additionally, there are some restrictions on the synchronization operators that a context may execute while it has an unmatched **save** pending. For example, attempting to acquire a lock that is already held by another context sharing the same space is not allowed since it would surely lead to deadlock.

[9]Note that **save** and **restore** do not affect shared VM; therefore, contexts with separate spaces cannot interfere with each other by executing **save** and **restore**.

If a context terminates when it has an unmatched **save** pending, an automatic **restore** is executed, thereby allowing other contexts to proceed.

As a practical matter, **save** and **restore** are not of much use when a space is shared among multiple contexts. Programs that are organized in this way should avoid using **save** and **restore**. On the other hand, programs that are organized as one space per context can use **save** and **restore** without restriction. This is especially important to maintain compatibility with existing printing applications, font products, etc.

Operators

For the context operators, a *context* is an integer that identifies a PostScript execution context. Each context has a unique identifier, whether it is created by calling a Client Library procedure or by executing the PostScript **fork** operator. This integer identifies the context during communication between the application and the Display PostScript system as well as during execution of the **join** and **detach** operators. Identifiers for contexts that have terminated become invalid and are not reused during the lifetime of any currently active session. The **currentcontext** operator returns the identifier for the context that is executing.

A context can *suspend* its own execution by any of a variety of means: execute the **wait**, **monitor**, or **yield** operators or return from its top-level procedure to await a **join**. The context retains all the state it had at the moment of suspension and can ordinarily be resumed from the point of suspension.

A context can *terminate* by executing the **quit** operator or as a result of an explicit termination request from the Client Library. Termination also occurs if an error occurs that is not caught by an explicit use of **stopped**. When a context terminates, its stacks are destroyed, its standard input and output files are closed, and its context identifier becomes invalid.

There is no hierarchical relationship among contexts. Termination of a context has no effect on other contexts that it may have created. An integer that identifies a context has the same meaning in every context; it may be referenced in a context different from the one that created it.

The objects *lock* and *condition* are distinct types of PostScript object. They are composite objects in the sense that their values occupy space in VM separate from the objects themselves; when a lock or condition object is stored in multiple places, all the instances share the same value. However, the values of locks and conditions are not directly accessible; they are accessed implicitly by the synchronization operators described above.

An **invalidcontext** error occurs if an invalid context identifier is presented to any of the context operators or if any of the programming restrictions are violated.

6 USER OBJECTS

Some applications require a convenient and efficient way to refer to PostScript language objects previously constructed in VM. Some types of objects, such as dictionaries and gstates, are not visible as data outside the PostScript interpreter; that is, they cannot be represented or referenced directly in any encoding of the language, even binary object sequences. Instead, the application must refer to such objects by means of surrogate objects, such as names or integers, whose values can be encoded and communicated easily.

The traditional way to accomplish this is to store such objects as elements of dictionaries or arrays and later to refer to them with their dictionary keys or array indices. In a PostScript language program written by a programmer, this approach is natural and straightforward. When the program is generated mechanically by another program, however, managing the space of surrogate objects (names or integers) requires additional bookkeeping. This is true particularly when the set of objects being managed is dynamically varying and when the responsibility for creating and referencing them is distributed among multiple libraries or packages.

pswrap provides a way for an application program to refer to user objects conveniently. This facility is described in the *pswrap Reference Manual.*

To support user objects, the Display PostScript system provides

three new operations: **defineuserobject**, **undefineuserobject**, and **execuserobject**, which manipulate an array named **UserObjects**. These operations introduce no fundamentally new capabilities; their behavior can be described entirely in the PostScript language and they can be implemented as procedures rather than as operators.[10] They have been made a standard part of the language so that *pswrap* can depend on their being available.

The following example illustrates the intended use of user objects.

```
/Times-Roman findfont 12 scalefont
17 exch defineuserobject
```

The first line of the example obtains a user object (in this case, a font dictionary). The second line associates the user object index 17 with this dictionary. Subsequently,

```
17 execuserobject setfont
```

pushes the font dictionary on the operand stack, from which it is taken by **setfont**. **execuserobject** executes the user object associated with index 17; however, since the object in this example is not executable, the result of the execution is to push the object onto the operand stack.

7 GRAPHICS STATE OBJECTS

The PostScript graphics state consists of a large collection of parameters that are accessed implicitly by the imaging operators. These parameters can be read and altered individually; the entire graphics state can be saved by pushing it on a stack (**gsave**) and restored by popping it from the stack (**grestore**).

This organization serves the needs of printing applications very well, assuming that the documents to be printed are reasonably structured. However, in interactive applications to be served by the Display PostScript system, a program needs to switch its at-

[10]User objects are entirely different from user names, described in Section 2. User names are part of the binary encoding extensions of the PostScript language syntax.

tention among multiple, more-or-less independent imaging contexts in an unpredictable order. Switching entire graphics states by altering their components individually is cumbersome and inefficient.

To address this need, we have introduced a new type of object, the *gstate*, that is capable of representing an entire graphics state. A gstate is a composite object that occupies VM and that conforms to the normal **save/restore** discipline; it is created by the **gstate** operator. The **type** operator returns the name *gstatetype* when a gstate is its operand.

The operators **setgstate**, **currentgstate**, and **copy** read and alter a gstate's value as a whole by copying it to or from the current graphics state or another gstate object. There is no way to select individual elements of a gstate's value directly; however, this can be accomplished by copying the gstate to the current graphics state temporarily and then accessing it using the regular graphics state operators.

Note that a gstate object captures *every* element of a graphics state, including such things as the current path and current clip path. For example, if a non-empty current path exists at the time **gstate** or **currentgstate** is executed, that path will be reinstated by the corresponding **setgstate**. Unless this effect is specifically desired, it is best to snapshot a graphics state only when the current path is empty and the current clip path is in its default state.

8 USER PATHS

A *user path* is a PostScript language procedure consisting entirely of path construction operators and their coordinate operands expressed as literal numbers. In other words, it is a completely self-contained description of a path in user space. There exist several new operators that combine execution of a user path description with rendering the resulting path (i.e., using it for filling or stroking).

The construction and use of a user path are best illustrated by an example:

```
{
ucache                          % this is optional
100 200 400 500 setbbox  % this is required
150 200 moveto
250 200 400 390 400 460 curveto
400 480 350 500 250 500 curveto
100 400 lineto
closepath
}
ufill
```

The tokens enclosed in '{' and '}' constitute a user path definition. The **setbbox** operator, with its four numeric operands (integers or reals), must appear first, or immediately after the optional **ucache**; the **setbbox** and **ucache** operators are described below. The remainder of the user path consists of path construction operators and their operands, in any sensible order. The path is assumed to start out empty, so the first operator after the **setbbox** must be an absolute positioning operator (**moveto, arc,** or **arcn**).

ufill is one of the new combined path construction and rendering operators. Its effect is to interpret the user path as if it were an ordinary PostScript language procedure (in the context of **systemdict**), then to perform a **fill**. Moreover, it performs a **newpath** prior to interpreting the user path and it encloses the entire operation with a **gsave** and a **grestore**. Thus, the overall effect of the above example is to define a path and to paint its interior with the current color; it leaves no side effects in the graphics state (or anywhere else except in raster memory).

The user path rendering operators can be fully described in terms of the existing PostScript language facilities; they introduce no fundamentally new capability. There are several motivations for having an integrated user path facility as a standard part of the language:

- It closely matches the needs of many application programs. In particular, it fits very well with the Display PostScript Client Library organization. If the language did not provide a user path facility, most applications would have to invent one.
- A user path consists of path construction operators and

numeric operands, not arbitrary computations. Thus, the user path is self-contained; its semantics are guaranteed not to depend on an unpredictable execution environment. Additionally, the information provided by **setbbox** assures that the coordinates of the path will be within predictable bounds. As a result, interpretation of a user path may be much more efficient than execution of an arbitrary PostScript language procedure.[11]

- Because a user path is represented as a procedure object and is self-contained, the PostScript interpreter can save the results of executing it in a cache. This may eliminate redundant interpretation of the same path definition, which is important in some Display PostScript applications that update the display frequently.

User path construction

A user path is an array or packed array object consisting of the following sequences of elements:

ucache
$ll_x\ ll_y\ ur_x\ ur_y$ **setbbox**
x y **moveto**
dx dy **rmoveto**
x y **lineto**
dx dy **rlineto**
$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$ **curveto**
$dx_1\ dy_1\ dx_2\ dy_2\ dx_3\ dy_3$ **rcurveto**
$x\ y\ r\ ang_1\ ang_2$ **arc**
$x\ y\ r\ ang_1\ ang_2$ **arcn**
$x_1\ y_1\ x_2\ y_2\ r$ **arct**
closepath

The permitted operators are all the standard PostScript operators that append to the current path, with the exception of **arcto** and **charpath**, which are not allowed. Additionally, there are three new user path construction operators: **ucache**, **setbbox**, and **arct**, which are described below. The permitted operands are number

[11]The user path rendering operators that are defined not to alter the current path may not create an explicit path at all. Indeed, if the bounding box lies completely outside the current clipping path, execution of the path definition and the rendering operation may be bypassed altogether. This behavior is, however, completely invisible to the PostScript language program.

literals, i.e., integers and reals. The correct number of operands must be supplied to each operator. Any deviation from these rules will result in a **typecheck** error when the user path is interpreted.

The user path begins with an optional **ucache**, whose purpose is described below. Immediately following this must be a **setbbox** sequence, which establishes a bounding box (in user space) enclosing the entire path. All coordinates specified as operands to the subsequent path construction operators must fall within these bounds; if they don't, a **rangecheck** error will occur when the user path is interpreted.

The path construction operators in a user path may appear either as executable name objects, such as 'moveto', or as actual PostScript operator objects, such as the value of 'moveto' in **systemdict**. An application program constructing a user path specifies name objects; however, applying **bind** to the user path (or to a procedure containing it) ordinarily causes the names to be replaced by the operator objects themselves.

The user path rendering operators interpret a user path as if **systemdict** were the current dictionary (see the definition of **uappend**); thus, the path construction operators contained in the user path are guaranteed to have their standard meanings. It is illegal for a user path to contain names other than the standard path construction operator names. Aliases are prohibited so as to ensure that the user path definition is self-contained and its meaning is entirely independent of its execution environment.

Encoded user paths

An *encoded user path* is a very compact representation of a user path. It is an array consisting of two PostScript string objects (or an array and a string). The strings effectively encode the operands and operators of an equivalent user path procedure, using a compact binary encoding.

The encoded user path representation is accepted and understood by the user path rendering operators such as **ufill**. Those operators interpret the data structure and perform the encoded operations; it does not make sense to think of 'executing' the

encoded user path directly.[12] When we say that an encoded value represents an operation such as **moveto**, we mean the standard **moveto** operation; as with unencoded user paths, there is no opportunity to redefine the meanings of operators represented in an encoded user path.

The first element of an encoded user path is a *data string* or *data array* containing numeric operands; the second is an *operator string* containing encoded operators. This two-part organization is for the convenience of application programs that generate encoded user paths; in particular, operands always fall on natural addressing boundaries. All the characters in both strings are interpreted as binary numbers, not as ASCII character codes.

If the first element is a string, it is interpreted as an encoded number string, whose representation is described in Section 2. If it is an array, its elements are simply used in sequence; they must all be numbers.

The operator string is interpreted as a sequence of encoded path construction operators, one operation code (opcode) per character. (Unlike the data string, the operator string is not interpreted as an encoded number string.) The allowed opcode values are as follows:

0	**setbbox**
1	**moveto**
2	**rmoveto**
3	**lineto**
4	**rlineto**
5	**curveto**
6	**rcurveto**
7	**arc**
8	**arcn**
9	**arct**
10	**closepath**
11	**ucache**
$n > 32$	repetition count: repeat next opcode $n - 32$ times

[12]In principle, one could write a PostScript language program to perform this interpretation; this is analogous to writing an emulator for another language. Note that the operator encoding is specialized to user path definitions; it has nothing to do with the alternative external encodings of the PostScript language, which are described in Section 2.

Associated with each opcode in the operator string are zero or more operands in the data string or data array. The order of the operands is the same as in an ordinary user path. For example, execution of a **lineto** (opcode 3) consumes an *x* operand and a *y* operand from the data sequence.

If the encoded user path does not conform to the rules described above, a **typecheck** error will occur when the path is interpreted. Possible errors include invalid opcodes in the operator string or premature end of the data sequence.

User path cache

Interactive applications using the Display PostScript system typically define certain paths that must be redisplayed frequently or that are repeated many times. To optimize interpretation of such paths, the Display PostScript system provides a facility called the *user path cache*. This cache, analogous to the font cache, retains the results of interpreting user path definitions. When the PostScript interpreter encounters a user path that is already in the cache, it substitutes the cached results instead of reinterpreting the path definition.

There is a non-trivial cost associated with placing a user path in the cache: extra computation is required and existing paths may be displaced from the cache. Since most user paths are used once and immediately thrown away, it does not make sense to place every user path in the cache. Instead, the application program must explicitly identify the user paths that are to be cached. It does so by including the **ucache** operator as the first element of the user path definition (before the **setbbox** sequence), as shown in the following example:

```
/Circle1 {ucache -1 -1 1 1 setbbox 0 0 1 0 360 arc}
cvlit def

Circle1 ufill
```

The **ucache** operator notifies the PostScript interpreter that the enclosing user path should be placed in the cache if it is not already there or obtained from the cache if it is. This cache management is not performed directly by **ucache**; instead, it is

performed by the user path rendering operator that interprets the user path (**ufill** in this example). This is because the results retained in the cache differ according to what rendering operation is performed.[13] The **ufill** produces the same effects on the current page whether or not the cache is accessed.

Caching is based on the *value* of a user path object. That is, two user paths are considered the same for caching purposes if all elements of one are equal to the corresponding elements of the other, even if the objects themselves are not equal. Thus, a user path placed in the cache need not be explicitly retained in VM; an equivalent user path appearing literally later in the program can take advantage of the cached information. (Of course, if it is known that a given user path will be used many times, defining it explicitly in VM avoids creating it multiple times.)

User path caching, like font caching, is effective across translations of the user coordinate system, but not across other transformations such as scaling or rotation. In other words, multiple instances of a given user path rendered at different places on the page take advantage of the user path cache when the CTM is altered only by **translate**. If the CTM is altered by **scale** or **rotate**, the instances will be treated as if they were described by different user paths.

Two other features of the above example should be noted. First, the user path object is explicitly saved for later use (as the value of 'Circle1' in this example). This is done in anticipation of rendering the same path multiple times (in this case, a one-unit circle). Second, the **cvlit** operator is applied to the user path object in order to remove its executable attribute. This is to ensure that the subsequent reference to 'Circle1' simply pushes the object on the operand stack rather than inappropriately executing it as a procedure. (It is unnecessary to do this if the user path isn't saved for later use but is simply consumed immediately by a user path rendering operator.)

[13]For this reason, it does not make sense to invoke **ucache** outside a user path; doing so has no effect.

Operators

There are four categories of user path operators:

- New path construction operators, intended for inclusion in user path definitions (but not limited to such use), i.e., **setbbox, arct.**
- User path rendering operators, combining interpretation of a user path with a rendering operation (fill or stroke), i.e., **ufill, ueofill, ustroke.**
- User path cache operators, providing the ability to control and query the operation of the user path cache, i.e., **ucache, ucachestatus, setucacheparams.**
- miscellaneous operators that involve user paths, i.e., **uappend, upath, ustrokepath, inufill, inueofill, inustroke**

A *userpath* is one of the following:

- an ordinary user path: an array (which need not be executable) whose length is at least 5;
- an encoded user path: an array of two elements. The first element must be either an array whose elements are all numbers or a string that can be interpreted as an encoded number string (see Section 2). The second must be a string that encodes a sequence of operators, as described above.

In either case, the value of the object must conform to the rules for constructing user paths, as detailed in preceding sections; that is, the operands and operators must appear in the correct sequence. If the user path is malformed, a **typecheck** error will occur.

Several of the operators take an optional *matrix* as their topmost operand. This is a six-element array of numbers that describe a transformation matrix, as described in Section 4.4 of the *PostScript Language Reference Manual.* A matrix is distinguished from a user path (which is also an array) by the number and types of its elements.

In several of the descriptions of user path operators, the semantics of an operator are described as being 'equivalent' to a

PostScript language program making use of lower-level operators. This does not necessarily mean that the implementation executes those lower-level operators explicitly; in particular, redefining those operator names will not affect the behavior of the high-level operator. The effect is as if the 'equivalent' PostScript language program has had **bind** applied to it with **systemdict** as the current dictionary. Furthermore, the 'equivalent' program cannot take advantage of the user path cache.

Most of the user path rendering operators have no effect on the graphics state. The absence of side effects is a significant reason for the efficiency of the operations; in particular, there is no need to build up an explicit current path only to discard it after one use. Although the behavior of the operators can be described as if the path were built up, rendered, and discarded in the usual way, the actual implementation of the operators is optimized to avoid unnecessary work. Note that there is no user path clip operation. Since the whole purpose of the clip operation is to alter the current clipping path, there is no way to avoid actually building the path. The best way to clip with a user path is:

```
newpath userpath uappend clip newpath
```

This operation can still take advantage of information in the user path cache under favorable conditions.

The **uappend** operator and the rendering operators defined in terms of **uappend**, such as **ufill**, perform a temporary adjustment to the current transformation matrix as part of their execution. This adjustment consists of rounding the t_x and t_y components of the CTM to the nearest integer values. The purpose of this is to ensure that scan conversion of the user path produces uniform results when it is placed at different positions on the page through translation; it is especially important if the user path is cached. This adjustment is not ordinarily visible to a PostScript language program; it is not mentioned in the descriptions of the individual operators.

9 RECTANGLES

Rectangles are used very frequently, especially in display applications. Thus, it is useful to have a few primitives to render rectangles directly. This is a convenience to application programs; additionally, the foreknowledge that the figure will be a rectangle results in significantly optimized execution.

A rectangle is defined in the user coordinate system. The result produced is identical to that of a rectangle defined as an ordinary path. The rectangle operators are **rectfill**, **rectstroke**, **rectclip**, and **rectviewclip**.

The rectangle operators accept three different forms of operands. The first form is simply four numbers: *x*, *y*, *width*, and *height*, which describe a single rectangle. The rectangle's sides are parallel to the user space axes; it has corners located at (*x*, *y*), (*x* + *width*, *y*), (*x* + *width*, *y* + *height*), and (*x*, *y* + *height*). Note that *width* and *height* can be negative.

The other two forms are an indefinitely long sequence of numbers, represented either as an array or as an encoded number string; this representation is described in Section 2. The sequence must contain a multiple of four numbers; each group of four consecutive numbers is interpreted as the *x*, *y*, *width*, and *height* values defining a single rectangle. The effect produced is equivalent to specifying all the rectangles as separate subpaths of a single combined path, which is then rendered by a single **fill**, **stroke**, or **clip** operator.

All rectangles are drawn in a counterclockwise direction in user space, regardless of the signs of the *width* and *height* operands. This ensures that when multiple rectangles overlap, all of their interiors are treated as 'inside' the path according to the non-zero winding number rule. In the operator descriptions in Section 16, the programs stated to be 'equivalent' to the operators are valid only for positive *width* and *height* values; more complex programs are required to deal with negative values.

10 FONT-RELATED EXTENSIONS

Explicit character positioning

The standard operators for setting text (**show** and its variants) are designed according to the assumption that characters are ordinarily shown with their standard metrics. Means are provided to vary the metrics in certain limited ways: the **ashow** operator systematically adjusts the widths of all characters of a string during one show operation; the optional **Metrics** entry of a font dictionary adjusts the widths of all instances of particular characters of a font.

Certain applications that set text require very precise control over the positioning of each character. Although it is possible to position characters individually by executing a **moveto** and a single character **show** for each one, this approach is too cumbersome and expensive for setting more than small amounts of text. When an application has gone to the trouble of computing the positions of individual characters, it should have a reasonable way to express those positions directly.

Three new variants of the **show** operator have been defined to streamline the setting of individually positioned characters: **xyshow**, **xshow**, and **yshow**. Each operator is given a string of text to be shown, just the same as **show**. Additionally, it expects a second operand, which is either an array composed of numbers or a string that can be interpreted as an encoded number string as described in Section 2. The numbers are used in sequence to control the widths of the characters being shown, i.e., the spacing between each character and the next. They completely override the standard widths of the characters.

Each number (or, for **xyshow**, each pair of consecutive numbers) is associated with the corresponding character of the text string being shown. For a basic PostScript font, this is the entire story. For a composite font, which may have a complex mapping from characters in the show string to glyphs rendered on the page, successive elements of the number array or the encoded number string are associated with successively rendered glyphs.

Font selection

Applications that frequently switch fonts require a streamlined means for doing so. The canonical sequence **findfont, scalefont** (or **makefont**), and **setfont** appears so frequently that most applications define a procedure to perform it. The cost of this procedure, as well as **findfont** (which is itself a procedure) and **scalefont** (which performs rather extensive computations), can have a serious impact on efficiency.

To better support the needs of applications, we have introduced a new operator, **selectfont**, that combines the actions of the above three operators. This operator takes advantage of information in the font cache in order to avoid calling **findfont** or performing the **scalefont** or **makefont** computations unnecessarily. Thus, in the common case of selecting a font and size combination that has been used recently, **selectfont** works with great efficiency.

Outline and bitmap font coordination

In display systems, the resolution of the device is typically quite low; resolutions in the range of 60 to 100 pixels per inch are common. When characters are produced algorithmically from outlines in typical sizes (10 to 12 points), the results are often not as legible as they need to be for most comfortable reading. The usual way to deal with this problem is to use *screen fonts* consisting of bitmap characters that have been tuned manually. The hand tuning increases legibility, possibly at the expense of fidelity to the original character shapes.

The Display PostScript system includes the ability to take advantage of hand-tuned bitmap fonts when they are available. This facility is fully integrated with the standard PostScript font machinery; its operation is almost totally invisible to a PostScript language program.

When a program sets text by executing an operator such as **show**, the PostScript interpreter first consults the font cache in the usual way. If the character is not there, it next consults the current device, requesting it to provide a bitmap form of the character at the required size. If the device can provide such a bitmap, it does so; the PostScript interpreter places the bitmap in

the font cache for subsequent use. If there is no such character, the interpreter executes the character description in the usual way, placing the scan converted result in the font cache.

The mechanism by which bitmap characters are provided by a device is not part of the language and is entirely hidden from a PostScript language program. In an integration of the Display PostScript system with a window system, the implementation of the device is the responsibility of the window system. Thus, the conventions for locating and representing bitmap characters are environment dependent. (Re-encoding a font preserves the association with bitmap characters; most other modifications to a font dictionary destroy the association.)

Bitmap fonts are typically provided in one orientation and a range of sizes from 10 to 24 points. (Beyond 24 points, characters scan converted from outlines are perfectly acceptable.) The PostScript interpreter can usually choose a bitmap character whose size is sufficiently close to the one requested and render it directly.

Associated with each hand-tuned bitmap is a *width*, i.e., displacement from the origin of the character to the origin of the next character. This width is also hand tuned for maximum legibility; it is an *integer* interpreted in device space (i.e., in character space, since pixels are pixels). It is usually different from the width produced when the same character is scan converted from the font definition, since that width (the *scaleable width*) is defined by real numbers that are scaled according to the requested font size.[14]

To achieve true fidelity between displays and printers when rendering characters, an application must use the scaleable widths to position characters on the display. Unfortunately, this leads to uneven letter spacing due to the need to round character positions to device pixel boundaries; at display resolution, this unevenness is objectionable. On the other hand, using the integer bitmap widths to produce evenly spaced text on the display leads to incorrect results on the printer. The only reasonable solution is

[14]Hand tuned bitmaps are carefully designed so that the bitmap widths and scaleable widths are as similar as possible when averaged over large amounts of text.

to use bitmap widths on the display and scaleable widths on the printer and to compensate for the positioning discrepancies in some other way.

Many word processing and page layout programs already use the following technique when rendering text on the display:

- Set the characters according to their integer bitmap widths, but keep track of the accumulated difference between the bitmap widths and the true scaleable widths.
- Adjust the spaces between words to compensate for the accumulated error. The most accurate way to do this is first to compute the error for an entire line and then to distribute the accumulated error among all the spaces in that line.

This technique maintains fidelity between display and printer on a line-by-line basis.

An application can control whether bitmap widths or scaleable widths are to be used on a per-font basis by adding a new entry, **BitmapWidths**, to the top-level font dictionary. If this entry is present, it must have a boolean value: *true* indicates that bitmap widths are to be used when the device provides bitmaps for this font; *false* indicates that scaleable widths are to be used. If the entry is not present or if the device does not provide bitmaps for this font, the normal scaleable widths are used always.

A device implementation ordinarily uses hand-tuned bitmaps only when the following conditions are met:

- The coordinate system axes are perpendicular (that is, the transformations are not skewed).
- The scale is uniform (reflections about axes are allowed).
- The angle of rotation is an even multiple of 90 degrees (0, 90, 180, or 270).

The appearance of the hand-tuned bitmaps is usually preferable to that of scan-converted outlines for a given character at a given point size.

Hand-tuned bitmaps are provided in a range of discrete sizes. When a requested size falls between two discrete sizes, the closest discrete size can be used and the widths are scaled ac-

cordingly. In all other cases, the default is to use scan-converted outlines. In certain cases a developer may deem it preferable to produce transformations of the bitmaps rather than scan-converting the transformed outlines.

Three keys can be added to the top-level font dictionary to control these transformations:

ExactSize Refers to cases where there is an exact match between the requested size and a hand-tuned bitmap when the coordinate system axes are perpendicular, the scale is uniform, and the angle of rotation is an even multiple of 90 degrees.

InBetweenSize Refers to cases where the requested size falls between discrete hand-tuned bitmap sizes under the same conditions as **ExactSize**.

TransformedChar
Refers to cases where the transformation is other than those mentioned under **ExactSize** conditions.

Each of these keys (**ExactSize**, **InBetweenSize**, and **TransformedChar**) can take one of the following values to control the use of hand-tuned bitmaps:

0	Use outline
1	Use closest hand-tuned bitmap size
2	Use transformed hand-tuned bitmap

Not all implementations are able to transform hand-tuned bitmaps. The default values for the additional keys are specified below:

Key	Default Value
BitmapWidths	false
ExactSize	1 (closest hand-tuned bitmap)
InBetweenSize	1 (closest hand-tuned bitmap)
TransformedChar	0 (outline)

11 HALFTONE DEFINITION

Halftoning is the process by which continuous gray tones are approximated by a pattern of pixels that can achieve only a limited number of discrete gray tones. The most familiar case of this is rendering of gray tones with black and white pixels. In the original PostScript language, program control of the halftoning process is provided by means of the **setscreen** operator, described in Section 4.8 of the *PostScript Language Reference Manual.*

As PostScript interpreters are integrated with a wider assortment of printing and display technologies, the language must be extended to provide more control over details of the halftoning process. For example, in color printing, one must specify independent halftone screens for each of three or four color separations. In imaging on low-resolution displays, one must have finer control over the halftoning process in order to achieve the best approximations of gray levels or colors and to minimize artifacts.

In recognition of the need to provide new halftoning processes with new printing and display technologies, we have introduced an extensible mechanism for defining halftones. This mechanism, called the *halftone dictionary*, provides means to define any of several types of halftones. The **setscreen** style of halftone is one of these types; new types (or new variations on those types) do not require fundamental language changes.

Remember that everything relating to halftones is, by definition, device dependent. In general, when an application defines its own halftones, it sacrifices portability. Associated with every device is a default halftone definition that is appropriate for most

applications. Only relatively sophisticated applications need to define their own halftones to achieve special effects.

Halftone dictionaries

A *halftone dictionary* is an ordinary PostScript dictionary object, certain of whose key-value pairs have special meanings. Some of the contents of a halftone dictionary are optional and user-definable, while other key-value pairs *must* be present and have the correct semantics in order for the halftone machinery to operate properly. In this respect (as in several others), a halftone dictionary is analogous to a font dictionary.

The graphics state includes a *current halftone dictionary*, which specifies the halftoning process to be used by the painting operators. The operator **currenthalftone** returns the current halftone dictionary; **sethalftone** establishes a different halftone dictionary as the current one.

A halftone dictionary is a self-contained description of a halftoning process. Painting operations, such as **fill**, **stroke**, and **show**, consult the current halftone dictionary when they require information about the halftoning process. Some of the entries in the dictionary are procedures that are called to compute the required information.

The PostScript interpreter consults the halftone dictionary at unpredictable times. Furthermore, it can cache the results internally for later use; such caching may persist through switches of halftone dictionaries caused by **sethalftone**, **gsave**, and **grestore**. For these reasons, once a halftone dictionary has been passed to **sethalftone**, its contents should be considered read-only. Procedures in the halftone dictionary must compute results that depend only on information in the halftone dictionary, not on outside information, and they must not have side-effects.[15]

Every halftone dictionary must have a **HalftoneType** entry whose value is an integer. This specifies the major type of

[15]This rules out certain 'tricks', such as the pattern fill example in the *PostScript Language Tutorial and Cookbook*, that depend on the spot function being executed at predictable times. Such tricks continue to work for halftones defined by **setscreen**, but not for halftones defined by halftone dictionaries.

halftoning process; the remaining entries in the dictionary are interpreted according to the type. The halftone types currently defined are:

1 The halftone is defined by frequency, angle, and spot function (corresponding to the existing **setscreen** facility).

2 The halftone is defined by four separate frequency, angle, and spot functions: one for each of the three primary colors (red, green, and blue) plus gray.

3 The halftone is defined directly by a threshold array at device resolution.

4 The halftone is defined by four threshold arrays: one for each of the three primary colors plus gray.

If the current halftone has been defined by **sethalftone** instead of by **setscreen**, a subsequent **currentscreen** will return a frequency of 60, an angle of 0, and the halftone dictionary as the spot function. If **setscreen** receives a dictionary as a spot function, it will ignore the frequency and angle parameters and perform the equivalent of **sethalftone** on the dictionary. This behavior is for compatibility with existing applications that attempt to alter the screen frequency or angle without providing a new spot function. Such applications cannot produce the intended effect but still run to completion.

Spot functions

A type 1 halftone dictionary defines a halftone in terms of its frequency, angle, and spot function. These parameters have the same meanings as the operands given to **setscreen**, but they are provided as entries in a halftone dictionary. The entries are as follows:

Key	*Type*	*Semantics*
HalftoneType	integer	must be 1.
Frequency	number	screen frequency, measured in halftone cells per inch in device space.
Angle	number	screen angle: number of degrees by which the screen is to be rotated with respect to the device coordinate system.

SpotFunction	procedure	procedure that defines the order in which device pixels within a screen cell are adjusted for different gray levels.

A halftone defined in this way produces results identical to a halftone defined by **setscreen**. However, the dictionary form of this halftone definition can work more efficiently since the PostScript interpreter can retain information about it in a cache, which it is not permitted to do for a halftone specified by **setscreen**. See the previous section for a discussion of this matter.

A type 2 halftone dictionary defines a halftone as four screens in the same manner as **setcolorscreen**. Instead of a single **Frequency** entry, there are entries for **RedFrequency**, **GreenFrequency**, **BlueFrequency**, and **GrayFrequency**; likewise for **Angle** and **SpotFunction**. Color screens are not further discussed here; see *PostScript Language Color Extensions* for more information.

Threshold arrays

A type 3 halftone dictionary defines a halftone as an array of threshold values that directly control individual device pixels in a halftone cell. This provides a finer degree of control over halftone rendering; also, it permits halftone cells to be rectangular, whereas halftone cells defined by a spot function are always square. Both of these capabilities are important for low-resolution display applications.

A *threshold array* is much like a sampled image: it is a rectangular array of pixel values. However, it is defined entirely in device space and the sample values always occupy 8 bits each. The pixel values nominally represent gray levels in the usual way, where 0 is black and 255 is white. The threshold array is replicated to tile the entire device space; thus, each pixel of device space is mapped to a particular sample of the threshold array.

On a bilevel device (each pixel is either black or white), the halftoning algorithm is as follows. For each device pixel that is to be painted with some gray level, the corresponding pixel of

the threshold array is consulted. If the desired gray level is less than the pixel value in the threshold array, the device pixel is painted black; otherwise it is painted white. For the purpose of this comparison, gray values in the range 0 to 1 (inclusive) correspond to pixel values 0 to 255 in the threshold array.

This scheme easily generalizes to monochrome devices with multiple bits per pixel. For example, if there are 2 bits per pixel, then each pixel can directly represent one of four different gray levels: black, dark gray, light gray, and white, encoded as 0, 1, 2, and 3 respectively. For each device pixel that is to be painted with some in-between gray level, the corresponding pixel of the threshold array is consulted to determine whether to use the next lower or next higher representable gray level. In this situation, the samples in the threshold array do not represent absolute gray values but gradations between two adjacent representable gray levels.

With this approach, it is reasonable to use the same threshold array for monochrome displays having different numbers of gray levels. This works because the threshold values are effectively scaled to span the distance between adjacent representable gray values, regardless of how many distinct gray values there are. (Indeed, the halftone rendering algorithm for a single bit per pixel device is simply a special case of the one for multiple bits per pixel.)

A type 3 halftone dictionary must contain the following entries:

Key	*Type*	*Semantics*
HalftoneType	integer	must be 3.
Width	integer	width of threshold array, in pixels.
Height	integer	height of threshold array, in pixels.
Thresholds	string	threshold values. This string must be *width* × *height* characters long. The individual characters represent threshold values as described above. The order of pixels is the same as for a sampled image mapped directly onto device space, with the first sample at the lower left corner[16] and *x* coordinates changing faster than *y* coordinates.

A halftone defined in this way can also be used with color (RGB) displays. The red, green, and blue values are simply treated independently as gray levels; the same threshold array applies to each color.

However, some devices, particularly color printers, require separate halftones for each primary color (and sometimes also for gray). A type 4 halftone dictionary defines four separate threshold arrays. Instead of a single **Width** entry, there are entries for **RedWidth**, **GreenWidth**, **BlueWidth**, and **GrayWidth**; likewise for **Height** and **Thresholds**.

Halftone phase

In a printer, the gray pattern tiles device space starting at the device space origin. That is, the halftone grid is aligned such that the lower left corner of the lower left halftone cell is positioned at (0, 0) in device space, independent of the value of the current transformation matrix. This ensures that adjacent gray areas will be painted with halftones having the same phase, thereby avoiding 'seams' or other artifacts.

On a display, the phase relationship between the halftone grid and device space needs to be more flexible. This need arises be-

[16]that is, the corner corresponding to the minimum *x* and *y* coordinates in device space; mathematically, this is the 'lower left' corner in a normal, right-handed Cartesian coordinate system. Display devices typically have a left-handed coordinate system in which *y* coordinates increase downward on the screen. For such devices, the mathematical 'lower left' corner is the upper left corner on the physical screen.

cause most window systems provide a *scrolling* operation in which the existing contents of raster memory are copied from one place to another in device space. This operation can alter the phase of halftones that have already been scan converted. It is necessary to alter the phase of the halftone generation algorithm correspondingly so that newly painted halftones will align with the existing ones.

The graphics state includes a pair of *halftone phase* parameters, one for x and one for y. These integers define an offset from the device space origin to the halftone grid origin. Of course, the halftone grid does not actually have an origin, so the offset values are actually interpreted modulo the width and height of the halftone cell. Effectively, they ensure that *some* halftone cell will have its lower left corner at (x, y) in device space.

The intended use of the halftone phase operators (**sethalftonephase** and **currenthalftonephase**) is in conjunction with window system operations that perform scrolling. If the application scrolls the displayed image by (dx, dy) pixels in device space, it should simply add dx and dy to the halftone phase parameters; it should not worry about computing them modulo the size of the halftone cell. This has the correct effect even if the displayed image is composed of several different halftone screens.

Note that the halftone phase is defined to be part of the graphics state, not part of the device. This is because an application may subdivide device space into multiple regions that it scrolls independently. A recommended technique is to associate a separate gstate (graphics state) object with each such region; this object carries all the parameters required to image within that region, including the halftone phase.

12 SCAN CONVERSION DETAILS

As discussed in Section 2.3 of the *PostScript Language Reference Manual*, the PostScript interpreter executes a *scan conversion* algorithm to render abstract graphical shapes in the raster memory of the output device. The details of this algorithm have not been specified until now, since they are of little concern

to a PostScript language program that purports to be device independent. However, at the low resolutions typical of computer displays, one must pay some attention to scan conversion details, since variations of even one pixel's width can have a noticeable effect on appearance.

To ensure consistent and predictable results, the scan conversion algorithm is now specified more rigorously. This is not a language change *per se*; it is a more precise description of the scan conversion process, whose former definition was rather vague. Additionally, the PostScript imaging model has been extended to include a device-independent means for obtaining consistent line widths during stroke operations.

Scan conversion rules

The rules below enable one to determine precisely which device pixels will be affected by a painting operation. These rules apply to the Display PostScript system and to future PostScript products from Adobe Systems based on the same software technology; they do not necessarily apply to older products.

In the following descriptions, all references to coordinates and pixels are in device space. A 'shape' is a path to be painted with the current color or with an image; its coordinates are mapped into device space but not rounded to device pixel boundaries. At this level, curves have been flattened to sequences of straight lines and all 'insideness' computations have been performed.

Pixel boundaries fall on integer coordinates in device space. A pixel is a square region identified by the coordinates of its minimum x, minimum y corner. A pixel is a *half-open* region, meaning that it includes half of its boundary points. More precisely, for any point whose real number coordinate is (x, y), let $i = \text{floor}(x)$ and $j = \text{floor}(y)$. The pixel that contains this point is the one identified as (i, j). The region belonging to that pixel is defined to be the set of points (x', y') such that $i \le x' < i+1$ and $j \le y' < j+1$.

Like pixels, shapes to be filled are also treated as half-open regions that include the boundaries along their 'floor' sides but not along their 'ceiling' sides.

A shape is scan converted by painting any pixel whose square region intersects the shape, no matter how small the intersection is. This ensures that no shape ever disappears as a result of unfavorable placement relative to the device pixel grid (as might happen with other possible scan conversion rules). The area covered by painted pixels is always at least as large as the area of the original shape.

This scan conversion rule applies to both fill operations and to strokes with non-zero width. Zero-width strokes are done in a device dependent manner that may include fewer pixels than this rule specifies.

The region of device space to be painted by a sampled image is determined similarly, though not identically. The image source rectangle is transformed into device space and defines a half-open region, just as for fill operations. However, only those pixels whose *centers* lie within the region are painted. Furthermore, the position of the *center* of such a pixel (i.e., coordinate values whose fractional part is one-half) is mapped back into source space to determine how to color the pixel. There is no averaging over the pixel area; if the resolution of the source image is higher than that of device space, some source samples are not be used.

For clipping, the clip region consists of the set of pixels that would be included by a fill. A subsequent painting operation affects a region that is the intersection of the set of pixels defined by the clip region with the set of pixels for the region to be painted.

Automatic stroke adjustment

When a stroke is drawn along a path, the scan conversion process may produce lines of non-uniform thickness due to rasterization effects. This is because in general the line width and the coordinates of the end points, translated into device space, are arbitrary real numbers, not quantized to device pixels. Thus, a line of a given width can intersect with a different number of device pixels depending on where it is positioned.

For best results, it is important to compensate for the rasteriza-

tion effects so as to produce strokes of uniform thickness; this is especially important in low-resolution display applications. While this can be done explicitly by a PostScript language program (as discussed in the documentation for **itransform** in the *PostScript Language Reference Manual*), doing so is cumbersome and inefficient. The newly introduced user path rendering operators, such as **ustroke**, provide no opportunity for a program to intervene in order to adjust the coordinates and line width. Furthermore, a more sophisticated adjustment algorithm is required to produce the most accurate results.

To meet this need, a *stroke adjustment* mechanism has been introduced as a standard part of the PostScript imaging model. When it is in effect, the line width and the coordinates of a stroke are automatically adjusted as necessary to produce lines of uniform thickness; furthermore, the thickness is as near as possible to the requested line width (i.e., no more than half a pixel different).[17]

Because automatic stroke adjustment can have a substantial effect on the appearance of lines, an application must be able to control whether or not it is performed. The operator **setstrokeadjust** alters a boolean value in the graphics state that determines whether or not stroke adjustment will be performed during subsequent **stroke** and related operators. This allows compatibility with existing PostScript language programs.

When a character description is executed (e.g., the **BuildChar** procedure of a user-defined font), stroke adjustment is initially disabled instead of being inherited from the context of the **show** operation. This is necessary because character descriptions are executed at unpredictable times due to font caching. A **BuildChar** procedure can enable stroke adjustment if it wants to.

[17]If the requested line width, transformed into device space, is less than half a pixel, the stroke is rendered as a single-pixel line. This is the thinnest line that can be rendered at device resolution; it is equivalent to the effect produced by setting the line width to zero.

13 VIEW CLIPS

Interactive applications frequently make incremental updates to the displayed image. Such updates arise both from changes to the displayed graphical objects themselves and from window system manipulations that cause formerly obscured objects to become visible. For efficiency's sake, it is desirable for the application to redraw only those graphical objects that are affected by the change.

One approach to accomplishing this is to define a path that encloses the changed areas of the display, then redraw only those graphical objects that are enclosed (or partially enclosed) within the path. To produce correct results, it is necessary to impose this path as a clipping path while redrawing. If this were not done, portions of objects that are redrawn might incorrectly obscure objects that are not redrawn.

This clipping could be accomplished by adjusting the clipping path in the graphics state in the normal way. However, this is not particularly convenient, since the program that imposes the clipping and the program that is executed to redraw objects on the display may have different ideas about what the clipping path should be. This problem becomes particularly acute given the ability to switch entire graphics states arbitrarily.

To alleviate this, we have extended the PostScript imaging model to introduce another level of clipping, the *view clip*, that is entirely independent of the graphics state. Objects are rendered on the device only in areas that are enclosed by both the current clipping path and the current view clipping path.

The view clipping path is actually part of the PostScript execution context, not the graphics state. Its initial value is a path that encloses the entire imageable area of the output device (see **initviewclip**). The operators that alter the view clipping path do not affect the clipping path in the graphics state or vice versa. The view clipping path is not affected by **gsave** and **grestore**; however, a **restore** will reinstate the view clipping path that was in effect at the time of the matching **save**.[18] Note that view clips

[18]View clipping is temporarily disabled when the current output device is a *mask device*, such as the one installed by **setcachedevice**.

do not nest; rather, a new view clip replaces the existing one. The following operators manipulate view clips: **viewclip, eoviewclip, rectviewclip, viewclippath,** and **initviewclip.**

14 WINDOW SYSTEM SUPPORT

For each integration of the Display PostScript system with a window system, there is a collection of operators for doing such things as specifying the window that is to be affected by subsequent painting operators. These operators are *window system specific* because their syntax and semantics vary according to the properties and capabilities of the underlying window system. They are not documented in this manual.

In addition to the window system specific operators, there are several operators that are window related but have a consistent meaning across all window systems. They are needed to enable an application to associate input events (e.g., mouse clicks) with graphical objects in user space. These operators (i.e., **infill, ineofill, inufill, inueofill, instroke,** and **inustroke**) can be used freely by display based applications.

If a window system specific extension provides a way for a PostScript language program to receive input events directly, the program can perform operations such as mouse tracking and hit detection itself. With some window systems, however, input events are always received by the application. In that case, the application must either perform such computations itself or issue queries to the Display PostScript system. This decision involves a tradeoff between performance and application complexity. One possible approach is for the application to perform hit detection itself for simple shapes but to query the Display PostScript system for more complex shapes.

A program may require information about certain properties of the raster output device, such as whether or not it supports color and how many distinguishable color or gray values it can reproduce. A PostScript language program that is a page description should not need such information; using it compromises device independence. However, an interactive application using the Display PostScript system may desire to vary its behavior

according to the available display technology. For example, a CAD application may use stipple patterns on a binary black-and-white display but separate colors on a color display.

The **deviceinfo** operator returns a dictionary whose entries describe static information about the device. (Dynamic information must be read from the graphics state or obtained through operators such as **wtranslation**.) Some of the entries in this dictionary have standard names that are described in the table below; others may have meanings that are device dependent. Most entries are optional and are present only if they are relevant for that type of device.

Key	*Type*	*Semantics*
Colors	integer	number of independent color components: 1 indicates black-and-white or gray scale only; 3 indicates red, green, blue; 4 indicates red, green, blue, gray (or their complements: cyan, magenta, yellow, black, as typically used in printers).
GrayValues	integer	number of different gray values that individual pixels can reproduce (without halftoning). For example, 2 indicates a binary black-and-white device; 256 indicates an 8 bits-per-pixel gray scale device.
RedValues	integer	number of different red values that individual pixels can reproduce, independent of other colors.
GreenValues	integer	analogous to **RedValues**.
BlueValues	integer	analogous to **RedValues**.
ColorValues	integer	total number of different color values that each pixel can reproduce. If this entry is present and the entries for gray, red, green, and blue are absent, this means that the color components cannot be varied independently but only in combination.

15 MISCELLANEOUS CHANGES

This section contains miscellaneous language changes that have not been documented in earlier sections.

Additions to **statusdict**

As described in the *PostScript Language Reference Manual*, the standard dictionary **statusdict** is the repository for information

and facilities that are specific to individual products. The set of keys and values contained in **statusdict** is product dependent. However, every product's **statusdict** contains a **product** (product name string) and **revision** (product revision number).

In the Display PostScript system, the standard set of **statusdict** entries is extended to include the following:

Key	*Type*	*Semantics*
buildtime	integer	uniquely identifies a specific generation of this product. Its main purpose is to distinguish among various alpha- and beta-test versions of a product prior to its formal release; the value of **revision** is changed only for formal releases. (The integer value of **buildtime** actually represents a date and time in the format used in the machine on which the PostScript interpreter was constructed; this meaning, however, is not of any use to a PostScript language program.)
byteorder	boolean	describes the native (preferred) order of bytes in multiple-byte numbers appearing in binary tokens and binary object sequences (see Section 2). The value *false* indicates high-order byte first; *true* indicates low-order byte first. Although the interpreter will accept numbers in either order, it will process numbers in native order somewhat more efficiently.
realformat	string	identifies the native format for real (floating-point) numbers appearing in binary tokens and binary object sequences (see Section 2). If the native format is IEEE standard, the value of this string is 'IEEE'; otherwise, the value describes a specific native format, e.g., 'VAX'. The interpreter will always accept real numbers in IEEE format, but it may process numbers in native format more efficiently. An application program can query **realformat** to determine whether the interpreter's native format is the same as the application's; if so, translation to and from IEEE format can be avoided.

Syntax and scanner changes

As described in Section 2, the PostScript language syntax has been augmented to introduce binary tokens and binary object sequences. In the course of altering the PostScript interpreter's input scanner to accept the augmented language, we have taken the opportunity to eliminate several anomalies in the existing scanner. These anomalies are obscure; for some, the *PostScript Language Reference Manual* does not give a clear specification of what the correct behavior should be.

The principal change has to do with execution of string objects. A program to be executed by the PostScript interpreter can come

from either a file object or a string object. In the normal case, the interpreter reads from a file object, such as the one for the standard input file. However, as described in Section 3.6 of the *PostScript Language Reference Manual*, the interpreter can also read from an executable string object; this is accomplished by applying **exec** (or other execution operators) to the string. The **token** operator, which invokes the PostScript language scanner only, also accepts a string operand.

The syntax and semantics of a program should be the same whether the program is read from a file or from a string. However, in previous versions of the PostScript interpreter, there has been one difference in the treatment of string literals, enclosed in '(' and ')', which appear in the program being executed. If the program is read from a file, '\' (back-slash) escape sequences have special meanings (see *PostScript Language Reference Manual*, Section 3.3); if the program is read from a string, '\' escape sequences are not recognized and the characters are treated literally.

In the Display PostScript system and in future products based on the same software technology, this distinction between file and string execution semantics is eliminated. '\' escape sequences are now recognized in string literals always, regardless of whether the program is being read from a file or a string.

This change is relatively obscure and is unlikely to affect real programs. A contrived example illustrates the effect of the change:

```
(/a (\\n) def) cvx exec
```

When the outer string is scanned, the '\\' is treated as an escape sequence and replaced by a single '\'; this is true under both old and new conventions. The difference lies in what happens when the outer string is executed – specifically, in the contents of the inner string that is defined to be the value of 'a'. Under the old convention, the '\' in this string is not recognized as an escape; consequently, the string consists of the two characters '\' and 'n'. Under the new convention, the '\' is recognized as an escape; the resulting string consists of a single newline character produced from the escape sequence '\n'.

Note that escape sequences apply only in ASCII encoded string literals. A string appearing in a binary token or binary object sequence is always treated literally (see Section 2). The interpreter can consume a binary encoded program from a string just the same as from a file; the syntax accepted by the interpreter does not depend on the source of the characters being interpreted.

Apart from the change in string execution, there are several other differences between the scanner in the Display PostScript system and that of previous interpreters:

- Outside of a string literal, the old scanner sometimes treats '\' as a self-delimiting special character, depending on context. The new scanner always treats '\' as a regular character except within a string literal. This is consistent with the language specification.
- The characters FF (ASCII \014) and NUL (ASCII \000) are treated as white space characters. Of these, FF will terminate a comment; NUL will not. This is a documentation change; the old and new scanners behave the same in this regard.
- Certain tokens that are syntactically legal numbers but that exceed implementation limits are converted to name objects by the old scanner; the new scanner generates a **limitcheck** error in such cases.
- With the old scanner, all erroneous radix numbers of the form *base#number* are treated as names. With the new scanner, a base value not in the range 2 to 36 inclusive or a number digit not valid for the base causes the token to be treated as a name. However, if the number is syntactically valid but is simply too large to represent, a **limitcheck** occurs.

File system extensions

The Display PostScript system optionally provides access to named files in secondary storage. The file access capabilities are provided as part of the integration of the Display PostScript system with an underlying operating system; there are variations from one such integration to another. Not all the file system capabilities of the underlying operating system are necessarily made available at the PostScript language level.

The PostScript language provides a standard set of operators for accessing files. These consist of **file**, originally described in the *PostScript Language Reference Manual*, and several new operators: **deletefile**, **renamefile**, **filenameforall**, **setfileposition**, and **fileposition**. Although the language defines a standard framework for dealing with files, the detailed semantics of the file system operators (particularly file naming conventions) are operating system dependent.

Files are contained within one or more 'secondary storage devices', hereafter referred to simply as *devices* (but not to be confused with the 'current device', which is a display device in the graphics state). The PostScript language defines a uniform convention for naming devices, but it says nothing about how files in a given device are named. Different devices have different properties, and not all devices support all operations.

A complete file name is in the form '*%device%file*', where *device* identifies the secondary storage device and *file* is the name of the file within the device. When a complete file name is presented to a file system operator, the *device* portion selects the device; the *file* portion is in turn presented to the implementation of that device, which is operating system and environment dependent.

When a file name is presented without a '*%device%*' prefix, a search rule determines which device is selected. The available storage devices are consulted in order; the requested operation is performed for each device until it succeeds. The number of available devices, their names, and the order in which they are searched is environment dependent. Not all devices necessarily participate in such searches; some devices can be accessed only by naming them explicitly.

Normally, there is a device that represents the complete file system provided by the underlying operating system.[19] If so, by convention that device's name is 'os'; thus, complete file names are in the form '*%os%file*', where *file* conforms to underlying file system conventions. This device always participates in

[19]However, this device may impose some restrictions on the set of files that can be accessed. The need for restrictions arises when the PostScript interpreter executes with a user identity different from that of the user running the application program.

searches, as described above; thus, a program can access ordinary files without specifying the '%os%' prefix. There may be more than one device that behaves in this way.

Additionally, there is normally a device that represents font definitions that can be loaded dynamically by the **findfont** operator. If so, by convention that device's name is 'font'; thus, complete file names are in the form '%font%*file*', where *file* is a specific font name such as 'Palatino-BoldItalic'. Note that this naming convention does not necessarily have anything to do with how font files are actually named in the underlying operating system; the 'font' device is logically decoupled from the 'os' device. This device never participates in searches; accessing font files requires specifying the '%font%' prefix. If a 'font' device exists, the built-in definition of **findfont** will attempt to **run** the named font from that device; the program in the font file should create a font dictionary and execute a **definefont** with the same name.

For the operators **file**, **deletefile**, **renamefile**, **status**, and **filenameforall**, a *filename* is a string object that identifies a file. The file name can be in one of three forms:

%*device*%*file*	identifies a file on a specific *device*, as described above.
%*device*	identifies one of the special files '%stdin', '%stdout', '%lineedit', or '%statementedit', described in Section 3.8 of the *PostScript Language Reference Manual.*
file	(first character not '%') identifies a file on an unspecified device; the device is selected by an environment specific search rule, as described above.

An *access* is a string object that specifies how a file is to be accessed. File access conventions are operating system specific. The following access specifications are typical of the UNIX® operating system and are supported by many others. The access string always begins with 'r', 'w', or 'a', possibly followed by '+'; any additional characters supply operating system specific information.

r	open for reading only; error if file doesn't already exist.
w	open for writing only; create file if it doesn't already exists; truncate it if it does.
a	open for writing only; create file if it doesn't already exist; append to it if it does.
r+	open for reading and writing; error if file doesn't already exist.
w+	open for reading and writing; create file if it doesn't already exist; truncate it if it does.
a+	open for reading and writing; create file if it doesn't already exist; append to it if it does.

Timekeeping

The **usertime** operator, which is specified as returning execution time of the PostScript interpreter, now reports interpretation time on behalf of the current context only. The ability to perform per-context timekeeping accurately depends on the underlying operating system; in some environments, it may not be possible to separate execution time of the PostScript interpreter from that of other programs executing concurrently.

A new standard operator, **realtime**, returns elapsed real time, independent of the activities of the PostScript interpreter or other programs.

Standard Error Handlers

As described in Section 3.6 of the *PostScript Language Reference Manual*, when an error occurs, the PostScript interpreter looks up the error's name in **errordict** and executes the associated procedure. That procedure is expected to handle the error in some appropriate way.

The **errordict** present in the initial state of the VM provides standard handlers for all errors. However, **errordict** is a writable dictionary; a program can therefore replace individual error-handlers selectively. Since **errordict** is in the private VM, such changes are visible only to the context that made them (or to other contexts sharing the same space).

The standard error handlers in the Display PostScript system behave slightly differently from the ones described in the *PostScript Language Reference Manual*, Section 3.8. They operate as follows:

- Execute 'false setshared', thereby reverting to private VM allocation mode
- Record information about the error in the special dictionary, **$error**; in the Display PostScript system, **$error** is located in private VM.
- Execute **stop**, thereby exiting the innermost enclosing context established by **stopped**.

The information recorded in the **$error** dictionary is shown in the table in Section 3.8 of the *PostScript Language Reference Manual*. In particular, the entries **newerror**, **errorname**, and **command** are always stored. However, the **ostack**, **estack**, and **dstack** arrays, which record snapshots of the operand, execution, and dictionary stacks, are generated only if the entry **recordstacks** has been previously set to the boolean value *true*; its normal value is *false*.[20]

The procedure **handleerror** is invoked if a program loses control due to an error. In the Display PostScript system, the standard definition of **handleerror** generates a special type of binary object sequence, not a text message. This is described in Section 3.

Font Cache Size

The total size of the font cache can be adjusted dynamically. This enables one to tune the amount of memory consumed by the font cache according to the needs of applications and output devices. With undemanding applications and low-resolution devices, a relatively small font cache suffices. When applications use many fonts in many sizes or output to high-resolution devices, a large font cache is required for good performance.

Adjusting the font cache size is accomplished by an extension to the existing **setcacheparams** operator, which takes a variable

[20]The error handler for **VMerror** never snapshots the stacks, regardless of the value of **recordstacks**. This prevents an attempt to allocate more VM at a time when VM is already exhausted.

number of operands. **currentcacheparams** returns the font cache parameters as described in the *PostScript Language Reference Manual*, with the addition of the result *size*. See the operator description in Section 16.

Permanent Entries on Dictionary Stack

There are three permanent entries on the dictionary stack for the Display PostScript system. In order, starting from the bottom, they are: **systemdict**, **shareddict**, and **userdict**. A new operator, **cleardictstack**, has been added so that a program may clear all nonpermanent entries from the dictionary stack without having to know how many permanent entries there are.

16 OPERATORS

Conventions

This chapter contains detailed descriptions of all the extensions to the PostScript language that implement the Display PostScript system. The operators are organized alphabetically by operator name. Each operator description is presented in the following format:

operator $operand_1$ $operand_2$ ... $operand_n$ **operator** $result_1$ ... $result_m$

Detailed explanation of the operator

EXAMPLE:

An example of the use of this operator. The symbol '⇒' designates values left on the operand stack by the example.

ERRORS:

A list of the errors that this operator might execute.

At the head of an operator description, $operand_1$ through $operand_n$ are the operands that the operator requires, with $operand_n$ being the topmost element on the operand stack. The operator pops these objects from the operand stack and consumes them. After executing, the operator leaves the objects $result_1$ through $result_m$ on the stack, with $result_m$ being the topmost element.

Normally the operand and result names suggest their types. The following table lists most of the operand and result names and their use.

name	*see section*	*description*
filename	15	is a file name string.
font	5.3 of *PostScript Language Reference Manual*	is a dictionary constructed according to the rule for font dictionaries.
halftone	11	is a dictionary constructed according to the rule for halftone dictionaries.
int	3.4 of *PostScript Language Reference Manual*	indicates an integer number.
matrix	4.4 of *PostScript Language Reference Manual*	is an array of six numbers describing a transformation matrix.
num	3.4 of *PostScript Language Reference Manual*	indicates that the operand or result is a number (integer or real).
numstring	2	is an encoded number string.
proc	3.4 of *PostScript Language Reference Manual*	indicates a PostScript procedure (i.e., an executable array or executable packed array).
userpath	8	is an array of path construction operators and their operands or an array of two strings comprising an encoded user path.

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

The documented effects on the operand stack and the possible errors are those produced directly by the operator itself. Many operators cause arbitrary PostScript procedures to be invoked. Obviously, such procedures can have arbitrary effects that are not mentioned in the operator description.

Operator Summary

Structured Output Operators

–	**currentobjectformat** int	return binary object format *83*
obj int	**printobject** –	write binary object to standard output file, using *int* as tag *101*
int	**setobjectformat** –	set binary object format (0=disable, 1=IEEE high, 2=low, 3=native high, 4=low) *113*
file obj int	**writeobject** –	write binary object to *file*, using *int* as tag *131*

Memory Management Operators

–	**currentshared** bool	return current VM allocation mode *84*
key font	**definefont** font	register a font as a font dictionary *84*
any	**scheck** bool	true if *any* is simple or in shared VM, false otherwise *107*
bool	**setshared** –	set VM allocation mode (*false*=private, *true*=shared *115*
int	**setvmthreshold** –	set the allocation threshold for garbage collection *117*
dict key	**undef** –	remove *key* and its value from *dict* *122*
key	**undefinefont** –	remove font definition *123*
int	**vmreclaim** –	control garbage collector *129*
–	**vmstatus** level used maximum	report VM status *130*

Multiple Execution Context Operators

–	**condition** condition	create condition object *81*
–	**currentcontext** context	return current context identifier *82*
context	**detach** –	enable context to terminate immediately when done *87*
mark obj_1 .. obj_n proc	**fork** context	create context executing *proc* with obj_1 .. obj_n as operands *92*
context	**join** mark obj_1 .. obj_n	await context termination and return its results *98*
–	**lock** lock	create lock object *99*
lock proc	**monitor** –	execute *proc* while holding *lock* *99*
condition	**notify** –	resume contexts waiting for *condition* *100*
–	**quit** –	terminates the context *102*
lock condition	**wait** –	release *lock*, wait for *condition*, reacquire *lock* *131*
–	**yield** –	suspend current context momentarily *133*

User Object Operators

–	**UserObjects** array	return **UserObjects** array in **userdict** *125*
index any	**defineuserobject** –	associate *index* with *any* in **UserObjects** array *86*
index	**execuserobject** –	execute *index* element in **UserObjects** array *88*
index	**undefineuserobject** –	remove *index* element from **UserObjects** array *123*

Graphics State Object Operators

gstate	**currentgstate** gstate	read current graphics state into *gstate* *82*
–	**gstate** gstate	create graphics state object *93*
gstate	**setgstate** –	set graphics state from *gstate* *112*

User Path Operators

x_1 y_1 x_2 y_2 r	**arct** –	append tangent arc *81*
ll_x ll_y ur_x ur_y	**setbbox** –	set bounding box for current path *109*
mark blimit	**setucacheparams** –	set user path cache parameters *116*
userpath	**uappend** –	interpret *userpath* and append to current path *120*
–	**ucache** –	declare that user path is to be cached *121*
–	**ucachestatus** mark bsize bmax rsize rmax blimit	return user path cache status and parameters *121*
userpath	**ueofill** –	fill using even-odd rule *121*
userpath	**ufill** –	interpret and fill *userpath* *122*
bool	**upath** userpath	create *userpath* for current path; include **ucache** if *bool* is true *124*
userpath	**ustroke** –	interpret and stroke *userpath* *126*
userpath matrix	**ustroke** –	interpret *userpath*, concatenate *matrix*, and stroke *126*
userpath	**ustrokepath** –	compute outline of stroked *userpath* *127*
userpath matrix	**ustrokepath** –	compute outline of stroked *userpath* *127*

Rectangle Operators

x y width height	**rectclip** –	clip with rectangular path *103*
numarray|numstring	**rectclip** –	clip with rectangular paths *103*
x y width height	**rectfill** –	fill rectangular path *104*
numarray|numstring	**rectfill** –	fill rectangular paths *104*
x y width height	**rectstroke** –	stroke rectangular path *105*

x y width height matrix	**rectstroke** –	stroke rectangular path *105*
numarray|numstring	**rectstroke** –	stroke rectangular paths *105*
numarray|numstring matrix	**rectstroke** –	stroke rectangular paths *105*

Font Operators

font matrix	**makefont** font'	produces new font *99*
key scale|matrix	**selectfont** –	set font dictionary given name and transform *108*
text numarray|numstring	**xshow** –	print characters of *text* using *x* widths in *numarray|numstring* *132*
text numarray|numstring	**xyshow** –	print characters of *text* using *x* and *y* widths in *numarray|numstring* *133*
text numarray|numstring	**yshow** –	print characters of *text* using *y* widths in *numarray|numstring* *134*

Halftone Definition Operators

–	**currenthalftone** dict	return current halftone dictionary *83*
–	**currenthalftonephase** x y	return current halftone phase *83*
–	**currentscreen** frequency angle proc	return current halftone screen *83*
–	**currentscreen** 60 0 halftone	return current halftone dictionary (**sethalftone** was used) *83*
dict	**sethalftone** –	set halftone dictionary *112*
x y	**sethalftonephase** –	set halftone phase *113*
frequency angle proc	**setscreen** –	set halftone screen *114*
num_1 num_2 halftone	**setscreen** –	set halftone screen using halftone dictionary *114*

Scan Conversion Operators

–	**currentstrokeadjust** bool	return current stroke adjust *84*
bool	**setstrokeadjust** –	set stroke adjust (*false*=disable, *true*=enable) *116*

View Clip Operators

–	**eoviewclip** –	view clip using even-odd rule *87*
–	**initviewclip** –	reset view clip *94*
x y width height	**rectviewclip** –	set rectangular view clipping path *106*
numarray|numstring	**rectviewclip** –	set rectangular view clipping paths *106*
–	**viewclip** –	set view clip from current path *128*
–	**viewclippath** –	set current path from view clip *128*

Window System Support Operators

–	**deviceinfo** dict	return dictionary containing information about current device *87*
x y	**infill** bool	test whether point (*x*, *y*) would be painted by **fill** *94*
userpath	**infill** bool	test whether pixels in *userpath* would be painted by **fill** *94*
x y	**ineofill** bool	test whether point (*x*, *y*) would be painted by **eofill** *93*
userpath	**ineofill** bool	test whether pixels in *userpath* would be painted by **eofill** *93*
x y userpath	**inueofill** bool	test whether point (*x*, *y*) would be painted by **ueofill** *95*
$userpath_1$, $userpath_2$	**inueofill** bool	test whether pixels in $userpath_1$ would be painted by **ueofill** of $userpath_2$ *95*
x y userpath	**inufill** bool	test whether point (*x*, *y*) would be painted by **ufill** *96*
$userpath_1$ $userpath_2$	**inufill** bool	test whether pixels in $userpath_1$ would be painted by **ufill** of $userpath_2$ *96*
x y userpath	**inustroke** bool	test whether point (*x*, *y*) would be painted by **ustroke** *97*
x y userpath matrix	**inustroke** bool	test whether point (*x*, *y*) would be painted by **ustroke** *97*
$userpath_1$ $userpath_2$	**inustroke** .bool	test whether pixels in $userpath_1$ would be painted by **ustroke** of $userpath_2$ *97*
$userpath_1$ $userpath_2$ matrix	**inustroke** bool	test whether pixels in $userpath_1$ would be painted by **ustroke** of $userpath_2$ *97*
–	**wtranslation** x y	return translation from window origin to device space origin *132*

File System Operators

string	**deletefile** –	delete named file *86*
pattern proc scratch	**filenameforall** –	execute *proc* for each file name matching *pattern* *90*
file	**fileposition** int	return current position in *file* *91*
$string_1$ $string_2$	**renamefile** –	rename file $string_1$ to $string_2$ *106*
file int	**setfileposition** –	set *file* to specified position *112*
string	**status** pages bytes referenced created true *or* false	return information about named file *118*

Miscellaneous Operators

–	**cleardictstack**	–	pop all nonpermanent dictionaries off dictionary stack *81*
–	**currentcacheparams**	mark size lower upper	return current characteristics of font cache *82*
index name	**defineusername**	–	define encoded name index *85*
–	**realtime**	int	return real time in milliseconds *102*
mark size lower upper	**setcacheparams**	–	change characteristics of font cache *111*
–	**usertime**	int	return context execution time in milliseconds *125*

Errors

invalidcontext	improper use of context operation *97*
invalidid	invalid identifier for window-system-specific operator *98*

arct $x_1\ y_1\ x_2\ y_2\ r$ **arct**

appends an arc of a circle, defined by two tangent lines, to the current path. This operator is identical to **arcto** except that it does not push any results on the operand stack, whereas **arcto** pushes four numbers. That is, **arct** is equivalent to:

```
arcto pop pop pop pop
```

arct can be used as an element of a user path definition, whereas **arcto** is not allowed.

ERRORS:
limitcheck, nocurrentpoint, stackunderflow, typecheck, undefinedresult

cleardictstack – **cleardictstack** –

pops all dictionaries off the dictionary stack except for the three permanent entries, **systemdict**, **shareddict**, and **userdict**.

ERRORS:
(none)

condition – **condition** condition

creates a new condition object, unequal to any condition object already in existence, and pushes it on the operand stack. The condition initially has no contexts waiting on it.

Since a condition is a composite object, creating one consumes VM. The condition's value is allocated either in the current context's space (private VM) or in shared VM according to the current VM allocation mode (see **setshared**).

ERRORS:
stackoverflow, VMerror

copy $gstate_1$ $gstate_2$ **copy** $gstate_2$

copies the value of $gstate_1$ to $gstate_2$, entirely replacing $gstate_2$'s former value, then pushes $gstate_2$ back on the operand stack. (The **copy** operator is thus extended to operate on gstate objects in addition to the types it already deals with.)

ERRORS:
invalidaccess, stackunderflow, typecheck

currentcacheparams – **currentcacheparams** mark size lower upper

pushes a mark object followed by the current cache parameters on the operand stack. The number of cache parameters returned is variable (see **setcacheparams**).

ERRORS:
stackoverflow

currentcontext – **currentcontext** context

returns an integer that identifies the current context.

ERRORS:
stackoverflow

currentgstate gstate **currentgstate** gstate

replaces the value of the *gstate* object by a copy of the current graphics state and pushes *gstate* back on the operand stack.

If *gstate* is in shared VM (see Section 4), **currentgstate** will generate an **invalidaccess** error if any of the composite objects in the current graphics state are in private VM. Such objects might include the current font, screen function, halftone dictionary, transfer function, or dash pattern. In general, allocating gstate objects in shared VM is risky and should be avoided.

ERRORS:
invalidaccess, stackunderflow, typecheck

currenthalftone – **currenthalftone** halftone

If the current halftone was defined by **sethalftone** or by **setscreen** using a halftone dictionary, **currenthalftone** returns the current halftone dictionary in the graphics state.

If the current halftone was defined by **setscreen** using a spot function, **currenthalftone** returns a null object.

ERRORS:
stackoverflow

currenthalftonephase – **currenthalftonephase** x y

returns the current values of the halftone phase parameters in the graphics state. If **sethalftonephase** has not been executed, zero is returned for both values.

ERRORS:
stackoverflow

currentobjectformat – **currentobjectformat** int

returns the current object format parameter (see **setobjectformat**).

ERRORS:
stackoverflow

currentscreen – **currentscreen** frequency angle proc
– **currentscreen** 60 0 halftone

returns the current halftone screen parameters (*frequency*, *angle*, and *proc*) in the graphics state if the current halftone screen was established by **setscreen**. If **sethalftone** was executed, **currentscreen** returns a frequency of 60, an angle of 0, and the halftone dictionary. (See Section 11.)

ERRORS:
stackoverflow

currentshared – **currentshared** bool

returns the current value of the VM allocation mode (see **setshared**).

ERRORS:
stackoverflow

currentstrokeadjust – **currentstrokeadjust** bool

returns the current stroke adjust parameter in the graphics state.

ERRORS:
stackoverflow

definefont key font **definefont** font

has its normal effects on **FontDirectory** and on the font machinery, as documented in the *PostScript Language Reference Manual.*

Note that **FontDirectory** normally refers to the font directory in private VM; **definefont** operates only on that directory and not on **SharedFontDirectory**. However, when shared VM allocation mode is in effect, the name **FontDirectory** refers to the font directory in shared VM; **definefont** operates on it. In the latter case, the value of *font* must itself be allocated in shared VM.

ERRORS:
dictfull, invalidaccess, invalidfont, stackunderflow, typecheck

defineusername index name **defineusername** –

establishes an association between the non-negative integer *index* and the name object *name* in the user name table. Subsequently, the scanner will substitute *name* when it encounters any binary encoded name token or object that refers to the specified user name *index*. (Since binary encoded names specify their own literal or executable attributes, it does not matter whether *name* is literal or executable.)

The user name table is an adjunct to the current context's private VM or *space* (see Section 5). The effect of adding an entry to the table is immediately visible to all contexts that share the same space. Additions to the table are not affected by **save** and **restore**; the association between *index* and *name* persists for the remaining lifetime of the space.

The specified *index* must previously be unused in the name table or must already be associated with the same *name*; changing an existing association is not permitted (an **invalidaccess** error will occur). There may be an implementation limit on *index* values; assigning index values sequentially starting at zero is strongly recommended.

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

defineuserobject index any **defineuserobject** –

establishes an association between the non-negative integer *index* and the object *any* in the **UserObjects** array. First, it creates a **UserObjects** array in **userdict** if one is not already present; it extends an existing **UserObjects** array if necessary. It then executes:

```
userdict /UserObjects get
3 -1 roll put
```

In other words, it simply stores *any* into the array at the position specified by *index*.

If **defineuserobject** creates or extends the **UserObjects** array, it allocates the array in private VM regardless of the current VM allocation mode. (See Section 6.)

The behavior of **defineuserobject** obeys normal PostScript language semantics in all respects. In particular, the modification to the **UserObjects** array (and to **userdict**, if any) is immediately visible to all contexts that share the same space. It can be undone by a subsequent **restore** according to the usual VM rules. *index* values must be within the range permitted for arrays; a large *index* value may cause allocation of an array that would exhaust VM resources. Assigning *index* values sequentially starting at zero is strongly recommended.

ERRORS:
limitcheck, rangecheck, stackunderflow, typecheck, VMerror

deletefile filename **deletefile** –

removes the specified file from the device. If no such file exists, an **undefinedfilename** error occurs. If this operation is not allowed by the device, an **invalidfileaccess** error occurs. If an environment dependent error is detected, an **ioerror** occurs.

ERRORS:
invalidfileaccess, ioerror, stackunderflow, typecheck, undefinedfilename

detach context **detach** –

specifies that the context identified by the integer *context* is to terminate immediately when it finishes executing its top-level procedure *proc*, whereas ordinarily it would wait for a **join**. (If the context is already waiting for a **join**, **detach** causes it to terminate immediately.)

detach executes an **invalidcontext** error if *context* is not a valid context identifier or if the context has already been joined or detached. It is permissible for *context* to identify the current context.

ERRORS:
invalidcontext, stackunderflow, typecheck

deviceinfo – **deviceinfo** dict

returns a read-only dictionary containing static information about the current device. The composition of this dictionary varies according to the properties of the device; typical entries are given in the table in section 14.

The use of **deviceinfo** after a **setcachedevice**operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
stackoverflow

eoviewclip – **eoviewclip**

is similar to **viewclip** except that it uses the even-odd rule to determine the inside of the current path.

ERRORS:
limitcheck

execuserobject index **execuserobject** –

executes the object associated with the non-negative integer *index* in the **UserObjects** array. **execuserobject** is equivalent to:

```
userdict /UserObjects get
exch get exec
```

execuserobject's semantics are similar to those of **exec** or other explicit execution operators. That is, if the object is executable, it is executed; otherwise, it is pushed on the operand stack. See Section 3.6 of the *PostScript Language Reference Manual.*

If **UserObjects** is not defined in **userdict** (because **defineuserobject** has never been executed), an **undefined** error occurs. If *index* is not a valid index for the existing **UserObjects** array, a **rangecheck** error occurs. If *index* is a valid index but **defineuserobject** has not been executed previously for that index, a null object is returned. (See Section 6.)

ERRORS:
invalidaccess, rangecheck, stackunderflow, typecheck, undefined

file filename access **file** file

creates a file object for the file identified by *filename*, accessing it as specified by *access*. The interpretation of the two string operands is described in Section 15. See the *PostScript Language Reference Manual* for a description of file objects in general and the **file** operator in particular.

Once opened, the *file* object remains valid until closed or invalidated. It can be closed explicitly by **closefile** or implicitly by reading to end of file. It can be invalidated by a **restore**, by garbage collection, or by termination of the current context.

The lifetime of a *file* object is based on the VM allocation mode in effect at the time the **file** operator is executed. A **restore** can destroy a *file* object in private VM but not one in shared VM.

If the specified *filename* is malformed or if the file doesn't exist and *access* does not permit creating a new file, **file** executes an **undefinedfilename** error. If *access* is malformed or the requested access is not permitted by the device, an **invalidfileaccess** error occurs. If the number of files opened by the current context exceeds an implementation limit, a **limitcheck** error occurs. If an environment dependent error is detected, an **ioerror** occurs.

ERRORS:
invalidfileaccess, ioerror, limitcheck, stackunderflow, typecheck, undefinedfilename

filenameforall pattern proc scratch **filenameforall** –

enumerates all files whose names match the specified *pattern* string. For each matching file, *filenameforall* copies the file's name into the supplied *scratch* string, pushes a string object designating the substring of *scratch* actually used, and calls *proc*. **filenameforall** does not return any results of its own, but *proc* may do so.

The details of pattern matching are device dependent, but the following convention is typical. All characters in the pattern are treated literally (and are case sensitive), except the following special characters:

*	matches zero or more consecutive characters.
?	matches exactly one character.
\	causes the next character of the pattern to be treated literally, even if it is '*', '?', or '\'.

If *pattern* does not begin with '%', it is matched against device relative file names of all devices in the search order (see the description above). When a match occurs, the file name passed to *proc* is likewise device relative, i.e., it does not have a '*%device%*' prefix.

If *pattern* does begin with '%', it is matched against complete file names in the form '*%device%file*'; pattern matching can be performed on the *device*, the *file*, or both parts of the name. When a match occurs, the file name passed to *proc* is likewise in the complete form '*%device%file*'.

The order of enumeration is unspecified and device dependent. There are no restrictions on what *proc* can do. However, if *proc* causes new files to be created, it is unspecified whether or not those files will be encountered later in the same enumeration. Likewise, the set of file names considered for pattern matching is device dependent. For example, the 'font' device might consider all font names whereas the 'os' general file system device might consider only names in the current working directory.

ERRORS:
ioerror, rangecheck, stackoverflow, stackunderflow, typecheck

fileposition file **fileposition** position

returns the current position in an existing open file. The result is a non-negative integer interpreted as number of bytes from the beginning of the file. If the file object is not valid or the underlying file is not positionable, an **ioerror** occurs.

ERRORS:
ioerror, stackunderflow, typecheck, undefinedfilename

findfont key **findfont** font

obtains a font dictionary, as documented in the *PostScript Language Reference Manual*. It looks for *key* first in **FontDirectory**, then in **SharedFontDirectory**; thus, fonts defined in private VM take precedence over ones defined in shared VM. Only if *key* is not present in either dictionary does **findfont** perform its environment dependent action to locate the font elsewhere.

Note that when shared VM allocation mode is in effect, the name **FontDirectory** refers to the font directory in shared VM. In this situation, **findfont** looks for *key* only in the shared font directory. Additionally, any action that **findfont** takes to obtain a font definition from the external environment must cause that definition to be created in shared VM.

In the Display PostScript system, when the font being sought is not already present in **FontDirectory** or **SharedFontDirectory**, **findfont** attempts to obtain a font definition from the execution environment. If this succeeds, the font is loaded into shared VM and defined in **SharedFontDirectory**, regardless of the current VM allocation mode. This portion of **findfont** is approximately equivalent to:

```
currentshared
true setshared
(%font%name) run          % load font into shared VM
setshared                 % restore old shared mode
```

where *name* is the text of the requested font name (without leading '/'). Since the font definition is shared, it is immediately visible to all contexts and it persists until explicitly removed by **undefinefont**.

ERRORS:
invalidfont, stackoverflow, typecheck

fork mark obj_1 ... obj_n proc **fork** context

creates a new context using the same space (private VM) as the current context. The new context begins execution concurrent with continued execution of the current context; which context executes first is unpredictable.

The new context's environment is formed by copying the dictionary and graphics state stacks of the current context. The initial operand stack consists of obj_1 through obj_n, pushed in the same order (obj_1 through obj_n are objects of any type other than mark). **fork** consumes all operands down to and including the topmost mark. It then pushes an integer that uniquely identifies the new context. The forked context inherits its object format from the current context; all other miscellaneous state variables for the context (see Section 5) are initialized to default values.

When the new context begins execution, it executes the procedure *proc*. If *proc* runs to completion and returns, the context ordinarily will suspend until some other context executes a **join** on *context*; however, if the context has been detached, it will terminate immediately (see **join** and **detach**).

If *proc* executes a **stop** that causes the execution of *proc* to end prematurely, the context will terminate immediately. *proc* is effectively called as follows:

```
proc stopped {handleerror quit} if
% wait for join or detach
quit
```

In other words, if *proc* stops due to an error, the context invokes the error handler in the usual way to report the error; then it terminates, regardless of whether or not it has been detached.

It is illegal to execute **fork** if there has been any previous **save** not yet matched by a **restore**; attempting to do so will cause an **invalidcontext** error.

ERRORS:
invalidaccess, invalidcontext, limitcheck, stackunderflow, typecheck, unmatchedmark

gstate – **gstate** gstate

creates a new graphics state object and pushes it on the operand stack. Its initial value is a copy of the current graphics state.

This operator consumes VM; it is the only graphics state operator that does so. The gstate is allocated in either private or shared VM according to the current VM allocation mode (see Section 4). Allocating a gstate in shared VM is risky, for reasons described under **currentgstate**.

ERRORS:
invalidaccess, stackoverflow, VMerror

ineofill

x y **ineofill** bool
userpath **ineofill** bool

is similar to **infill**, but its 'insideness' test is based on **eofill** instead of **fill**.

ERRORS:
stackunderflow, typecheck

infill

x y **infill** bool
userpath **infill** bool

The first form returns *true* if the device pixel containing the point (*x*, *y*) in user space would be painted by a **fill** of the current path in the graphics state; otherwise, it returns *false*.

In the second form, the device pixels that would be painted by filling the userpath become an 'aperture.' This form of the operator returns *true* if any of the pixels in the aperture would be painted by a **fill** of the current path in the graphics state; otherwise, it returns *false*.

Both forms of this operator ignore the current clipping path and current view clip; that is, they detect a 'hit' anywhere within the current path, even if filling that path would not mark the current page due to clipping. They do not actually place any marks on the current page, nor do they disturb the current path. The following program fragment takes the current clipping path into account:

```
gsave clippath x y infill grestore
x y infill and
```

ERRORS:
stackunderflow, typecheck

initviewclip – **initviewclip** –

replaces the current view clipping path by one that encloses the entire imageable area of the output device. (It can enclose a larger area than that; the actual size and shape of the initial view clip is device dependent.)

ERRORS: (none)

instroke

x y **instroke** bool
userpath **instroke** bool

returns *true* if the device pixel containing the point (*x*, *y*) in user space would be painted by a **stroke** of the current path in the graphics state; otherwise, it returns *false*. It does not actually place any marks on the current page, nor does it disturb the current path.

In the second form of the operator, the device pixels that would be painted by filling the userpath become an 'aperture.' **instroke** returns *true* if any of the pixels in the aperture would be painted by a **stroke** of the current path in the graphics state; otherwise, it returns *false*. It does not actually place any marks on the current page, not does it disturb the current path.

As with **infill**, this operator ignores the current clip path and current view clip; that is, it detects a 'hit' on any pixel that lies beneath a stroke drawn along the current path, even if stroking that path would not mark the current page due to clipping.

The shape against which the point (*x*, *y*) or the aperture, *userpath*, is tested is computed according to the current stroke-related parameters in the graphics state: line width, line cap, line join, miter limit, and dash pattern. It is also affected by the stroke adjust parameter (see Section 12). If the current line width is zero, the set of pixels considered to be part of the stroke is device dependent.

ERRORS:
stackunderflow, typecheck

inueofill

x y userpath **inueofill** bool
$userpath_1$ $userpath_2$ **inueofill** bool

is similar to **inufill**, but its 'insideness' test is based on **ueofill** instead of **ufill**.

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

inufill x y userpath **inufill** bool
$userpath_1$ $userpath_2$ **inufill** bool

returns *true* if the device pixel containing the point (*x*, *y*) in user space would be painted by a **ufill** of the specified *userpath* (see Section 8); otherwise, it returns *false*.

In the second form, the device pixels that would be painted by filling $userpath_1$ become an 'aperture.' **inufill** returns *true* if any of the pixels in the aperture would be painted by a **ufill** of $userpath_2$; otherwise, it returns *false*.

This operator does not actually place any marks on the current page, nor does it disturb the current path in the graphics state. Except for the manner in which the path is specified, **inufill** behaves the same as **infill**.

By itself, this operator is seemingly a trivial composition of several other operators:

```
gsave
newpath uappend
infill
grestore
```

However, when used in conjunction with **ucache**, it can access the user path cache, potentially resulting in improved performance.

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

inustroke

x y userpath **inustroke** bool
x y userpath matrix **inustroke** bool
$userpath_1$ $userpath_2$ **inustroke** bool
$userpath_1$ $userpath_2$ matrix **inustroke** bool

returns *true* if the device pixel containing the point (*x*, *y*) in user space would be painted by a **ustroke** applied to the same operands (see Section 8); otherwise it returns *false*.

In the second form, **inustroke** concatenates *matrix* to the CTM before executing **ustroke** (see **ustroke** operator).

In the third and fourth forms, the device pixels that would be painted by filling $userpath_1$ become an 'aperture.' **inustroke** returns *true* if any of the pixels in the aperture would be painted by a **ustroke** of $userpath_2$; otherwise it returns *false*.

This operator does not actually place any marks on the current page, nor does it disturb the current path in the graphics state. Except for the manner in which the path is specified, **inustroke** behaves the same as **instroke**.

As with **inufill**, if *userpath* is already present in the user path cache, **inustroke** can take advantage of the cached information to optimize execution.

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

invalidcontext (error)

indicates that an invalid use of the context synchronization facilities has been detected. Possible causes include:

- presenting an invalid context identifier to **join** or **detach**;
- executing **monitor** on a lock already held by the current context;
- executing **wait** on a lock not held by the current context;
- executing any of several synchronization operators when an unmatched **save** is pending if the result would be a deadlock.

The PostScript interpreter detects only the simplest types of deadlock. It is possible to encounter deadlocks for which no **invalidcontext** error is generated.

invalidid (error)

indicates that an invalid identifier has been presented to a window-system-specific operator. In each integration of the Display PostScript system with a window system, there exists a collection of window-system-specific operators. The operands of such operators are usually integers that identify windows and other objects that exist outside the PostScript language. This error occurs when the operand does not identify a valid object. It is generated only by window-system-specific operators and not by any standard operator.

join context **join** mark obj_1 ... obj_n

waits for the context identified by the integer *context* to finish executing its top-level procedure *proc*. It then pushes a mark followed by the entire contents of that context's operand stack onto the current context's operand stack. Finally, it causes the other context to terminate.

The objects obj_1 through obj_n are those left on the operand stack by the context that is terminating. Ordinarily there should not be a mark among those objects, since its presence might cause confusion in the context that executes **join**.

If *context* is not a valid context identifier, perhaps because the context has terminated prematurely due to an error, **join** executes an **invalidcontext** error. This also occurs if the context has already been joined or detached, if *context* identifies the current context, or if the context does not share the current context's space.

It is illegal to execute **join** if there has been any previous **save** not yet matched by **restore**; attempting to do so will cause an **invalidcontext** error.

ERRORS:
invalidcontext, stackunderflow, stackoverflow, typecheck

lock – **lock** lock

creates a new lock object, unequal to any lock object already in existence, and pushes it on the operand stack. The state of the lock is initially free.

Since a lock is a composite object, creating one consumes VM. The lock's value is allocated either in the current context's space (private VM) or in shared VM according to the current VM allocation mode (see **setshared**).

ERRORS:
stackoverflow, VMerror

makefont font matrix **makefont** font'

applies *matrix* to *font* producing a new *font'* whose characters are transformed by *matrix* when they are printed as described in the *PostScript Language Reference Manual*. The **makefont**, **scalefont**, and **selectfont** operators produce a font dictionary derived from an original font dictionary but with the **FontMatrix** entry altered. The derived font dictionary is allocated in private or shared VM according to whether the original font dictionary is in private or shared VM; this is independent of the current VM allocation mode.

ERRORS:
stackunderflow, typecheck, VMerror

monitor lock proc **monitor** –

acquires *lock*, first waiting if necessary for it to become free, then executes *proc*, and finally releases *lock* again. The release of *lock* occurs whether *proc* runs to completion or terminates prematurely for any reason.

If *lock* is already held by the current context, **monitor** executes an **invalidcontext** error without disturbing the lock. If the current context has previously executed a **save** not yet matched by a **restore** and *lock* is already held by another context sharing the same space as the current context, an **invalidcontext** error results. These restrictions prevent the most straightforward cases of a context deadlocking with itself.

ERRORS:
invalidcontext, stackunderflow, typecheck

notify condition **notify** –

resumes execution of all contexts (if any) that are suspended in a **wait** for *condition*.

Ordinarily, **notify** should be invoked only within the execution of a **monitor** that references the same *lock* used in the **wait** for *condition*. This ensures that notifications cannot be lost due to a race between a context executing **notify** and one executing **wait**. However, this recommendation is not enforced by the language.

ERRORS:
stackunderflow, typecheck

printobject obj tag **printobject** –

writes a binary object sequence to the standard output file. The binary object sequence contains a top-level array whose length is one; its single element is an encoding of *obj*. If *obj* is composite, the binary object sequence also includes subsidiary array and string values for the components of *obj*.

The *tag* operand, which must be an integer in the range 0 to 255, is used to tag the top-level object; it is used as the second character of the object's representation. As discussed in Section 3, tag values 0 through 249 are available for general use; tag values 250 through 255 are reserved for special purposes such as reporting errors.

The binary object sequence uses the number representation established by the most recent execution of **setobjectformat**. The token type given as the first character of the binary object sequence reflects the number representation that was used. If the object format parameter has been set to zero, **printobject** executes an **undefined** error.

The object *obj* and its components must be of type null, integer, real, name, boolean, string, array, or mark (see Section 2); appearance of an object of any other type (including packed array) will result in a **typecheck** error.

printobject always encodes a name object as a reference to a text name in the string value portion of the binary object sequence, never as a system or user name index.

As is the case for all operators that write to files, the output produced by **printobject** may accumulate in a buffer instead of being transmitted immediately. To ensure immediate transmission, a **flush** is required. This is particularly important in situations where the output produced by **printobject** is the response to a query from the application.

ERRORS:
invalidaccess, ioerror, limitcheck, rangecheck, stackunderflow, typecheck, undefined

quit – **quit** –

causes termination of the execution context that issued the **quit** operator. A snaphot VM file is not produced, even on computers with operating systems and file systems. (This differs from the description in the *PostScript Language Reference Manual*, where the **quit** operator terminates the PostScript interpreter.)

Instead of waiting for the **join** operator to be executed, the context terminates immediately as if the **detach** operator had been executed. Any context attempting to join a context that has executed **quit** will receive an **invalidcontext** error.

ERRORS:
(none)

realtime – **realtime** int

returns the value of a clock that counts in real time, independent of the execution of the PostScript interpreter. The clock's starting value is arbitrary; it has no defined meaning in terms of calendar time. The unit of time represented by the **realtime** value is one millisecond; however, the rate at which it actually changes is implementation dependent.

ERRORS:
stackoverflow

rectclip x y width height **rectclip** –
numarray **rectclip** –
numstring **rectclip** –

intersects the inside of the current clipping path with a path described by the operands. In the first form, the operands are four numbers that describe a single rectangle. In the other two forms, the operand is an array or an encoded number string that describes an arbitrary number of rectangles. After computing the new clipping path, **rectclip** resets the current path to empty, as if by **newpath**.

In the first form, assuming *width* and *height* are positive, **rectclip** is equivalent to:

```
newpath
x y moveto
width 0 rlineto
0 height rlineto
width neg 0 rlineto
closepath
clip
newpath
```

Note that if the second or third form is used to specify multiple rectangles, the rectangles are treated together as a single path and used for a single **clip** operation. Thus, the 'inside' of this combined path is the union of all the rectangular subpaths, since the paths are all drawn in the same direction and the non-zero winding number rule is used.

ERRORS:
limitcheck, stackunderflow, typecheck

rectfill x y width height **rectfill** –
numarray **rectfill** –
numstring **rectfill** –

fills a path consisting of one or more rectangles described by the operands. In the first form, the operands are four numbers that describe a single rectangle. In the other two forms, the operand is an array or an encoded number string that describes an arbitrary number of rectangles. **rectfill** neither reads nor alters the current path in the graphics state.

In the first form, assuming *width* and *height* are positive, **rectfill** is equivalent to:

```
gsave
newpath
x y moveto
width 0 rlineto
0 height rlineto
width neg 0 rlineto
closepath
fill
grestore
```

ERRORS:
stackunderflow, typecheck

rectstroke

x y width height **rectstroke** –
x y width height matrix **rectstroke** –
numarray **rectstroke** –
numarray matrix **rectstroke** –
numstring **rectstroke** –
numstring matrix **rectstroke** –

strokes a path consisting of one or more rectangles described by the operands. In the first two forms, the operands are four numbers that describe a single rectangle. In the remaining forms, the operand is an array or an encoded number string that describes an arbitrary number of rectangles. In any event, if the *matrix* operand is present, **rectstroke** appends it to the CTM before stroking the path. Thus the matrix applies to the line width and dash pattern (if any), but not to the path itself. **rectstroke** neither reads nor alters the current path in the graphics state.

The following example of **rectstroke**, using *x y width height* and *matrix*, is equivalent to:

```
gsave
newpath
x y moveto
width 0 rlineto
0 height rlineto
width neg 0 rlineto
closepath
matrix concat                    % only if matrix operand is present
stroke
grestore
```

ERRORS:
limitcheck, stackunderflow, typecheck

rectviewclip x y width height **rectviewclip** –
numarray **rectviewclip** –
numstring **rectviewclip** –

replaces the current view clip by a rectangular path described by the operands (see Section 9). In the first form, the operands are four numbers that describe a single rectangle. In the other two forms, the operand is an array or an encoded number string that describes an arbitrary number of rectangles. After computing the new view clipping path, **rectviewclip** resets the current path to empty, as if by **newpath**.

Except for the manner in which the path is defined, **rectviewclip** behaves the same as **viewclip**.

Note that if the second or third form is used to specify multiple rectangles, the rectangles are treated together as a single path and used for a single **viewclip** operation. Thus, the 'inside' of this combined path is the union of all the rectangular subpaths, since the paths are all drawn in the same direction and the non-zero winding number rule is used.

ERRORS:
stackunderflow, typecheck

renamefile old new **renamefile** –

changes the name of a file from *old* to *new*, where *old* and *new* are strings that specify file names on the same device. If a file named *old* does not exist, an **undefinedfilename** error occurs. If a renaming operation is not allowed by the device, an **invalidfileaccess** error occurs. If an environment dependent error is detected, an **ioerror** occurs. Whether or not an error occurs if a file named *new* already exists is environment dependent.

ERRORS:
invalidfileaccess, ioerror, stackunderflow, typecheck, undefinedfilename

scheck any **scheck** bool

returns *true* if the operand is simple or if its value is located in shared VM, *false* otherwise. In other words, **scheck** returns *true* if one could legally store its operand as an element of a shared object.

ERRORS:
stackunderflow

selectfont key scale **selectfont** –
key matrix **selectfont** –

obtains a font whose name is *key*, transforms it according to *scale* or *matrix*, and establishes it as the current font dictionary in the graphics state. **selectfont** is equivalent to one of the following, according to whether the second operand is a number or a matrix:

```
exch findfont exch scalefont setfont
exch findfont exch makefont setfont
```

If *key* is present in **FontDirectory**, **selectfont** obtains the font dictionary directly and does not call the **findfont** procedure. However, if *key* is not present, **selectfont** invokes **findfont** in the normal way. In the latter case, it actually executes the name object 'findfont', so it uses the current definition of that name in the context of the dictionary stack. (On the other hand, redefining **exch**, **scalefont**, **makefont**, or **setfont** would not alter the behavior of **selectfont**.)

In the Display PostScript system, fonts can be defined in either **FontDirectory** or **SharedFontDirectory** (see Section 4). **selectfont** looks in both of those places before calling **findfont**.

selectfont can give rise to any of the errors possible for the component operations, including arbitrary errors from a user-defined **findfont** procedure.

EXAMPLE:

```
/Helvetica 10 selectfont
/Helvetica findfont 10 scalefont setfont
```

The two lines of the example have the same effect, but the first one is almost always more efficient.

In a program represented using the binary token or binary object sequence encoding (see Section 2), it may be advantageous to predefine *key* in the user name table so that it can be referenced by a user name index instead of a name string.

ERRORS:
invalidfont, rangecheck, stackunderflow, typecheck

setbbox $ll_x\ ll_y\ ur_x\ ur_y$ **setbbox** –

establishes an explicit bounding box for the current path. The bounding box established by **setbbox** is the smallest rectangle that contains both the existing bounding box, if any, and the bounding box requested by the **setbbox** arguments. These arguments define a rectangle expressed as two pairs of coordinates in user space, oriented with the user-space coordinate-system axes: ll_x and ll_y specify the lower left corner; ur_x and ur_y specify the upper right corner. The upper right coordinate values must be greater than or equal to the lower left values; otherwise a **rangecheck** error will occur.

The coordinates of all subsequent path construction operators must fall within the resulting bounding box. This bounding box remains in effect for the lifetime of the current path — that is, until the next **newpath** or operator that resets the path implicitly, such as **stroke**, is executed — or until it is enlarged by a subsequent **setbbox**.

Once **setbbox** is executed, an attempt to append a path element with a coordinate lying outside the bounding box will give rise to a **rangecheck** error.[21] Bounds checking applies only to the path itself, not to the result of rendering the path. For example, stroking the path may place marks outside the bounding box; this does not cause an error.

Although the **setbbox** operator can be used when defining any path, its main use is in the definition of a user path, where it is mandatory. That is, a user path passed to one of the user-path-rendering operators, such as **ufill**, must begin with a **setbbox** (optionally preceded by a **ucache**). The information passed to **setbbox** enables the user-path-rendering operator to optimize execution. The user path may contain only one **setbbox**. However, multiple executions of **uappend** during the construction of a current path will result in multiple executions of the **setbbox** operator. In this case, each execution of **setbbox** has the potential to enlarge the bounding box.

When a path is constructed without an explicit **setbbox** request, an implicit bounding box for the path is maintained dynamically. Each path construction operator (**moveto**, **lineto**, **curveto**, and so on) enlarges the bounding box as necessary to enclose the elements being appended to the path. In this case the **rangecheck** error is not raised because the implicit bounding box is automatically adjusted to accommodate the growing path. If **setbbox** is executed when such a path

[21]Note that arcs are converted to sequences of **curveto** operations. The coordinates computed as control points for those **curvetos** must also fall within the bounding box. Effectively, this means that the figure of the arc must be entirely enclosed by the bounding box.

exists, the resulting bounding box is enlarged if necessary to enclose the implicit bounding box of the path.
If a bounding box has been established by **setbbox**, execution of **pathbbox** returns a result derived from that bounding box instead of from the implicit bounding box of the path.

ERRORS:
rangecheck, stackunderflow, typecheck

setcacheparams mark size lower upper **setcacheparams** –

sets cache parameters as specified by the integer objects above the topmost mark on the stack, then removes all operands and the mark object as if by **cleartomark**.

The number of cache parameters is variable.[22] If more operands are supplied to **setcacheparams** than are needed, the topmost ones are used and the remainder ignored; if fewer are supplied than are needed, **setcacheparams** implicitly inserts default values between the mark and the first supplied operand.

The *upper* operand specifies the maximum number of bytes that may be occupied by the pixel array of a single cached character, as determined from the information presented by the **setcachedevice** operator. This is the same parameter as is set by **setcachelimit**; see the description of that operator in the *PostScript Language Reference Manual.*

The *lower* operand specifies the threshold at which characters may be stored in compressed form rather than as full pixel arrays. If a character's pixel array requires more than *lower* bytes to represent, it may be compressed in the cache and reconstituted from the compressed representation each time it is needed. Some devices do not support compression of characters.

Setting *lower* to zero forces all characters to be compressed, permitting more characters to be stored in the cache but increasing the work required to print them. Setting *lower* to a value greater than or equal to *upper* disables compression altogether.

The *size* operand specifies the new size of the font cache in bytes (the *bsize* value returned by **cachestatus**). If *size* is not specified, the font cache size is unchanged. If *size* lies outside the range of font cache sizes permitted by the implementation, the nearest permissible size is substituted, with no error indication. Reducing the font cache size can cause some existing cached characters to be discarded, increasing execution time when those characters are next shown.

ERRORS:
limitcheck, rangecheck, typecheck, unmatchedmark

[22]In future versions of the PostScript interpreter there may be more than three cache parameters defined.

setfileposition file position **setfileposition** –

repositions an existing open file to a new *position*, such that the next read or write operation will commence at that position. The *position* operand is a non-negative integer interpreted as number of bytes from the beginning of the file. For an output file, **setfileposition** first performs an implicit **flushfile**.

The result of positioning beyond the existing end of file depends on the behavior of the underlying file system.

Possible causes of an **ioerror** are: the file object is not valid; the underlying file is not positionable; the specified position is invalid for the file; a device dependent error condition is detected.

ERRORS:
ioerror, stackunderflow, typecheck, undefinedfilename

setgstate gstate **setgstate** –

replaces the current graphics state by the value of the *gstate* object. This is a copying operation, so subsequent modifications to the value of *gstate* will not affect the current graphics state or vice versa. Note that this is a wholesale replacement of all components of the graphics state; in particular, the current clipping path is replaced by the value in *gstate*, not intersected with it.

ERRORS:
invalidaccess, stackunderflow, typecheck, undefined

sethalftone halftone **sethalftone** –

establishes *halftone* as the current halftone dictionary in the graphics state. This must be a dictionary constructed according to the rules in Section 11. If *halftone* is a null object instead of a dictionary, **sethalftone** substitutes the default halftone definition for the current device (however it was defined). If the halftone dictionary's **HalftoneType** value is out of bounds or is not supported by the PostScript interpreter, a **rangecheck** error occurs. If a required entry is missing or its value is of the wrong type, a **typecheck** error occurs. If the **Frequency** entry in the halftone dictionary is less than or equal to zero, an **undefinedresult** error occurs.

ERRORS:
limitcheck, rangecheck, stackunderflow, typecheck, undefinedresult

sethalftonephase x y **sethalftonephase** –

sets the current halftone phase parameters in the graphics state. *x* and *y* are integers specifying the new halftone phase, interpreted in device space.

ERRORS:
stackunderflow, typecheck

setobjectformat int **setobjectformat** –

establishes the number representation to be used in object sequences written by subsequent execution of **printobject** and **writeobject**. Output produced by those operators will have a token type that identifies the representation used. The *int* operand is one of the following (see Section 2):

0 disable binary encodings (see below)
1 high-order byte first; IEEE standard real format
2 low-order byte first; IEEE standard real format
3 high-order byte first; native real format
4 low-order byte first; native real format

Note that any of the latter four values specifies the number representation only for output. Incoming binary encoded numbers use a representation that is specified as part of each token (in the initial token type character).

The value 0 disables all binary encodings for both input and output. That is, the PostScript language scanner treats all incoming characters as part of the ASCII encoding, even if a token starts with a character code in the range 128 to 159. The **printobject** and **writeobject** operators are disabled; executing them will cause an **undefined** error. This mode is provided for compatibility with certain existing PostScript language programs.

Each PostScript execution context has its own object format parameter; modifications to this parameter obey the normal **save/restore** discipline. When a context is created by **fork**, the new context inherits its object format from the current context. For other contexts, the initial value of of the object format parameter is implementation dependent; the program must execute **setobjectformat** in order to generate output with a predictable number representation.

ERRORS:
rangecheck, stackunderflow, typecheck

setscreen frequency angle proc **setscreen** –
num_1 num_2 halftone **setscreen** –

sets the current halftone screen definition in the graphics state, as described in the *PostScript Language Reference Manual.*

For compatibility with existing applications, **setscreen** has been extended to take a halftone dictionary instead of the *proc* defining the spot function (see Section 11). In this case, the num_1 and num_2 operands are ignored.

ERRORS:
limitcheck, rangecheck, stackunderflow, typecheck

setshared bool **setshared** –

changes the VM allocation mode. The value *false* denotes private VM allocation; *true* denotes shared VM allocation.

In the normal private VM allocation mode, the values of new composite objects are allocated in the execution context's private VM. This applies both to objects created implicitly by the scanner and ones created explicitly by PostScript operators. Private objects cannot be stored as components of shared objects.

In shared VM allocation mode, the values of new composite objects are allocated in shared VM. Such objects may be stored as components of other shared objects (e.g., **shareddict, SharedFontDirectory**), thereby becoming visible to all contexts.

Creation and modification of shared objects is unaffected by the **save/restore** facility, whose actions are confined to the private VM of the context that executes them. Note that this selective disabling of **save/restore** semantics is based on where each object's value is located; it has nothing to do with the VM allocation mode in effect at the time of the **save** or the **restore**.

While shared VM allocation mode is in effect, the name **FontDirectory** refers to the value of **SharedFontDirectory**, located in shared VM, instead of to the normal private font directory. This affects the behavior of the **definefont** and **undefinefont** operators and the **findfont** procedure.

The standard error handlers in **errordict** execute 'false setshared', thus reverting to private allocation mode if an error occurs.

ERRORS:
stackunderflow, typecheck

setstrokeadjust bool **setstrokeadjust** –

sets the stroke adjust parameter in the current graphics state to *bool*. If *bool* is *true*, automatic stroke adjustment will be performed during subsequent execution of **stroke** and related operators (including **strokepath**; see Section 12). If *bool* is *false*, stroke adjustment will not be performed.

The initial value of the stroke adjustment parameter is device dependent; typically it is *true* for displays and *false* for printers. It is not altered by **initgraphics**.

ERRORS:
stackunderflow, typecheck

setucacheparams mark blimit **setucacheparams** –

sets user path cache parameters as specified by the integer objects above the topmost mark on the stack, then removes all operands and the mark object as if by **cleartomark**. The number of cache parameters is variable and may increase in future versions of the PostScript interpreter. If more operands are supplied to **setucacheparams** than are needed, the topmost ones are used and the remainder ignored; if too few are supplied, **setucacheparams** implicitly inserts default values between the mark and the first supplied operand.

blimit specifies the maximum number of bytes that can be occupied by the reduced representation of a single path in the user path cache. Any reduced path larger than this is not saved in the cache. Changing *blimit* does not disturb any paths that are already in the cache. (A *blimit* value that is too large is automatically reduced to the maximum permissible value without error indication.)

ERRORS:
typecheck, unmatchedmark

setvmthreshold int **setvmthreshold** –

sets the allocation threshold to the specified value. The allocation threshold for a VM is the amount of memory use that will trigger automatic garbage collection for that VM (if automatic garbage collection is enabled; see **vmreclaim**). The system keeps a separate accounting of memory used by each VM.

The allocation threshold for a VM defaults to a system-specific value. The value for a private VM can be changed; the new value must fall within the limits of a system-defined minimum and maximum (see example below). The value for the shared VM cannot be changed. When the allocation threshold for a private VM is exceeded, automatic garbage collection is triggered for that VM. When the allocation threshold for shared VM is exceeded, automatic garbage collection is triggered for the shared VM.

This operation applies only to the VM of the current context. If the specified value is less than the implementation-dependent minimum value, the threshold is set to that minimum value. If the specified value is greater than the implementation-dependent maximum value, the threshold is set to that maximum value. If the value specified is –1, then the threshold is set to the implementation dependent default value. All the other negative values result in a **rangecheck** error.

setvmthreshold never affects the allocation threshold associated with shared VM.

Example: Assuming a default threshold of 40,000, a minimum allowable value of 10,000, and a maximum allowable value of 500,000, the operation in the first column below produces the result shown in the second column.

20000 setvmthreshold	Threshold set to 20,000.
–1 setvmthreshold	Threshold set to 40,000.
20 setvmthreshold	Threshold set to 10,000.
1000000 setvmthreshold	Threshold set to 500,000.
–5 setvmthreshold	**rangecheck** error.

ERRORS:
rangecheck

status

file **status** bool
string **status** *if found*: pages bytes referenced created true
if not found: false

If the operand is a file object, **status** returns *true* if it is still valid (i.e., is associated with an open file), *false* otherwise. This behavior of **status** is as described in the *PostScript Language Reference Manual.*

If the operand is a string, **status** treats it as a file name according to the conventions described above. If there exists a file by that name, **status** pushes four integers of status information followed by the value *true*; otherwise it pushes *false*. The four integer values are:

pages — storage space actually occupied by the file, in implementation dependent units.

bytes — length of file in characters.

referenced — date and time at which the file was last referenced for either reading or writing. The interpretation of the value is according to the conventions of the underlying operating system; the only assumption that a program can make is that larger values indicate later times.

created — date and time at which the information in the file was created.

ERRORS:
stackoverflow, stackunderflow, typecheck

type any **type** name

returns a name object that identifies the type of the object *any*, as documented in the *PostScript Language Reference Manual.* The **type** operator is extended to operate on gstate, lock, and condition objects in addition to the types it already deals with. The possible names that **type** can return are now as follows

arraytype	marktype
booleantype	nametype
conditiontype	nulltype
dicttype	operatortype
filetype	packedarraytype
fonttype	realtype
gstatetype	savetype
integertype	stringtype
locktype	

ERRORS:
stackunderflow

uappend userpath **uappend** –

interprets a user path definition and appends the result to the current path in the graphics state. If *userpath* is an ordinary user path (i.e., an array or packed array whose length is at least 5), **uappend** is equivalent to:

```
systemdict begin        % ensure standard operator meanings
cvx exec                % interpret userpath
end
```

If *userpath* is an encoded user path, **uappend** interprets it and performs the encoded operations. It does not matter whether the *userpath* object is literal or executable.

Note that **uappend** uses the standard definitions of all operator names mentioned in the user path, unaffected by any name redefinition that may have occurred.

A **ucache** appearing in *userpath* may or may not have an effect, depending on the context in which **uappend** is executed. If the current path is initially empty and no path construction operators are executed after **uappend**, a subsequent rendering operator *may* access the user path cache; otherwise it definitely will not. This is particularly useful in the case of **clip** and **viewclip**.

uappend performs a temporary adjustment to the current transformation matrix as part of its execution. This adjustment consists of rounding the t_x and t_y components of the CTM to the nearest integer values. The reason for this is discussed in Section 8.

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

ucache – **ucache** –

notifies the PostScript interpreter that the user path in which the **ucache** operator appears is to be placed in the cache if it is not already there. If present, this operator must appear as the first element of a user path definition (before the mandatory **setbbox**).

The **ucache** operator has no effect of its own when executed; if executed outside a user path definition, it does nothing. It is useful only in conjunction with a user path rendering operator, such as **ufill** or **ustroke**, that takes the user path as an operand. If the user path is not already in the cache, the rendering operator performs the path construction operations specified in the user path and places the results (referred to as the *reduced path*) in the cache. If the user path is already present in the cache, the rendering operator does not interpret the user path but obtains the reduced path from the cache.

ERRORS: (none)

ucachestatus – **ucachestatus** mark bsize bmax rsize rmax blimit

reports the current consumption and limit for two user path cache resources: bytes of reduced path storage (*bsize* and *bmax*) and total number of cached reduced paths (*rsize* and *rmax*). Additionally, it reports the limit on the number of bytes occupied by a single reduced path (*blimit*) – reduced paths that are larger than this are not cached. All **ucachestatus** results except *blimit* are for information only; a PostScript language program can change *blimit* (see **setucacheparams**).

The number of values pushed on the operand stack is variable; future versions of the PostScript interpreter can push additional values between *mark* and *bsize*. The purpose of the *mark* is to delimit the values returned by **ucachestatus**; this enables a program to determine how many values were returned (by **counttomark**) and to discard any unused ones (by **cleartomark**).

ERRORS:
stackoverflow

ueofill userpath **ueofill** –

is similar to **ufill**, but does **eofill** instead of **fill**.

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

ufill userpath **ufill** –

interprets a user path definition and fills the resulting path as if by **fill**. The entire operation is effectively enclosed by **gsave** and **grestore**, so **ufill** has no lasting effect on the graphics state. **ufill** is equivalent to:

```
gsave
newpath
uappend
fill
grestore
```

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

undef dict key **undef** –

removes *key* and its associated value from the dictionary *dict*. *dict* does not need to be on the dictionary stack.

Note that the effect of **undef** can be undone by a subsequent **restore**. That is, if *key* was present in *dict* at the time of the matching **save**, **restore** will reinstate *key* and its former value. (Remember, however, that **restore** has no effect if *dict* is in shared VM; in that case, the effect of **undef** is permanent.)

An **undef** on a dictionary inside a 'forall' on that dictionary will give undefined results. The following example does *not* delete all keys in the dictionary:

```
mydict { pop mydict exch undef } forall
```

The dictionary must first be enumerated into another object and that object must be enumerated to remove the keys:

```
[ mydict { pop } forall ] { mydict exch undef } forall
```

Note that this technique is more memory-efficient than assigning a new dictionary to 'mydict'.

ERRORS:
invalidaccess, stackunderflow, typecheck, undefined

undefinefont key **undefinefont** –

removes *key* and its associated value (a font dictionary) from the **FontDirectory** dictionary. The effect of this is similar to **undef**; a special operator is needed because **FontDirectory** is read-only.

Note that **FontDirectory** normally refers to the font directory in private VM; **undefinefont** operates only on that directory and not on **SharedFontDirectory**. However, when shared VM allocation mode is in effect, the name **FontDirectory** refers to the font directory in shared VM; **undefinefont** operates on it.

ERRORS:
stackunderflow, typecheck, undefined

undefineuserobject index **undefineuserobject** –

breaks the association between the non-negative integer *index* and an object established by some previous execution of **defineuserobject**. It does so simply by replacing the specified **UserObjects** array element by the null object; this is equivalent to:

```
userdict /UserObjects get
exch null put
```

undefineuserobject does not take any other actions such as shrinking the **UserObjects** array. If *index* is not a valid index for the existing **UserObjects** array, a **rangecheck** error occurs.

There is no need to execute **undefineuserobject** prior to executing a **defineuserobject** that reuses the same index. The purpose of **undefineuserobject** is to eliminate references to objects that are no longer needed. This may enable such objects to be reclaimed by the garbage collector.

ERRORS:
rangecheck, stackunderflow, typecheck

upath bool **upath** userpath

creates a new user path object that is equivalent to the current path in the graphics state. **upath** creates a new executable array object of the appropriate length and fills it with the operands and operators needed to describe the current path. **upath** produces only an ordinary user path procedure, not an encoded user path. It does not disturb the current path in the graphics state.

The *bool* operand determines whether or not the resulting user path is to include **ucache** as its first element.

Since the current path's coordinates are maintained in device space, **upath** transforms them to user space using the inverse of the CTM while constructing the user path. Applying **uappend** to the resulting user path will reproduce the same current path in the graphics state, but only if the same CTM is in effect at that time.

upath is equivalent to:[23]

```
[
exch {/ucache cvx} if
pathbbox /setbbox cvx
{/moveto cvx} {/lineto cvx} {/curveto cvx}
   {/closepath cvx} pathforall
] cvx
```

If **charpath** was used to construct any portion of the current path, **upath** is not allowed; its execution will produce an **invalidaccess** error.

ERRORS:
invalidaccess, stackunderflow, typecheck, VMerror

[23]A perfect emulation of **upath** may need to be more complex than this in order to avoid exceeding the implementation limit on depth of the operand stack.

UserObjects -- **UserObjects** array

returns the current **UserObjects** array defined in **userdict**. **UserObjects** is not an operator; it is simply a name associated with an array in **userdict**. This array is created and managed by the operators **defineuserobject**, **undefineuserobjects**, and **execuserobject**. It defines a mapping from small integers (used as array indices) to arbitrary objects (the elements of the array).

The **UserObjects** entry in **userdict** is present only if **defineuserobject** has been executed at least once by the current context (or a context that shares the same space). The length of the array depends on the index operands of all previous executions of **defineuserobject**.

Note that **defineuserobject**, **undefineuserobjects**, and **execuserobject** operate on the value of **UserObjects** in **userdict**, without regard to the dictionaries currently on the dictionary stack. Defining **UserObjects** in some other dictionary on the dictionary stack changes the value returned by executing the name object **UserObjects** but does not alter the behavior of the user object operators.

Although **UserObjects** is an ordinary array object, it should be manipulated only by the user object operators. Improper direct alteration of **UserObjects** can subsequently cause the user object operators to malfunction.

ERRORS:
stackoverflow, undefined

usertime – **usertime** int

returns PostScript interpreter execution time, as described in the *PostScript Language Reference Manual*. In a Display PostScript system that supports multiple execution contexts, the value returned by **usertime** reports execution time on behalf of the current context only.[24] As before, the value has no defined starting point, so **usertime** is useful only for interval timing.

ERRORS:
stackoverflow

[24]A context that executes **usertime** can subsequently execute with reduced efficiency, because in order to perform user time accounting, the PostScript interpreter must perform an operating system call whenever it switches control to and from that context. Therefore, one should not execute **usertime** gratuitously.

ustroke userpath **ustroke** –
userpath matrix **ustroke** –

interprets a user path definition and strokes the resulting path as if by **stroke**. The entire operation is effectively enclosed by **gsave** and **grestore**, so **ustroke** has no lasting effect on the graphics state.

In the first form (with no *matrix* operand), **ustroke** is equivalent to:

```
gsave
newpath
uappend
stroke
grestore
```

In the second form, **ustroke** concatenates *matrix* to the CTM before executing **stroke**. Thus the matrix applies to the line width and the dash pattern (if any) but not to the path itself. This form of **ustroke** is equivalent to:

```
gsave
newpath
exch uappend                % interpret userpath
concat                      % concat matrix to CTM
stroke
grestore
```

The main use of this operation is to compensate for variations in line width and dash pattern that occur if the CTM has been scaled by different amounts in x and y. This is accomplished by defining *matrix* to be the inverse of the unequal scaling transformation.

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

ustrokepath userpath **ustrokepath** –
userpath matrix **ustrokepath** –

replaces the current path with one enclosing the shape that would result if the **ustroke** operator were applied to the same operands. The path resulting from **ustrokepath** is suitable as the implicit operand to a subsequent **fill, clip,** or **pathbbox**. In general, this path is not suitable for **stroke**, as it may contain interior segments or disconnected subpaths produced by **ustrokepath**'s stroke to outline conversion process.

In the first form, **ustrokepath** is equivalent to:

```
newpath
uappend
strokepath
```

In the second form, **ustrokepath** is equivalent to:[25]

```
newpath
exch uappend                % interpret userpath
matrix currentmatrix        % save CTM
exch concat                 % concat matrix to CTM
strokepath
setmatrix                   % restore original CTM
```

ERRORS:
invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck

[25]A more satisfactory emulation of **ustrokepath** would not create a new matrix each time but would define one temporary matrix that it reuses.

viewclip – viewclip –

replaces the current view clipping path by a copy of the current path in the graphics state. The inside of the current path is determined by the normal non-zero winding number rule. **viewclip** implicitly closes any open subpaths of the view clipping path. After setting the view clip, **viewclip** resets the current path to empty, as if by **newpath**.

viewclip is similar to **clip** in that it causes subsequent painting operations to affect only those areas of the current page that lie inside the new view clip path. However, it differs from **clip** in three important respects:

- The view clipping path is independent of the current clipping path. The current clipping path is unaffected; a subsequent **clippath** returns the current clipping path, uninfluenced by the additional clipping imposed by the view clip.
- **viewclip** entirely replaces the current view clipping path, whereas **clip** computes the intersection of the current and new clipping paths.
- **viewclip** performs an implicit **newpath** at the end of its execution, whereas **clip** leaves the current path unchanged.

The view clipping path can be described by a user path (see Section 8); this is accomplished by:

newpath *userpath* uappend viewclip

If *userpath* specifies **ucache**, this operation may take advantage of information in the user path cache.

ERRORS:
limitcheck

viewclippath – viewclippath –

replaces the current path by a copy of the current view clip path. If no view clipping path has been set, **viewclippath** replaces the current path by one that encloses the entire imageable area of the output device (see **initviewclip**).

ERRORS: (none)

vmreclaim int **vmreclaim** –

controls the garbage collection machinery as specified by *int*:

–2	disable automatic collection in both private and shared VM.
–1	disable automatic collection in private VM.
0	enable automatic collection.
1	perform immediate collection in private VM.
2	perform immediate collection in both private and shared VM. This can take a long time, since it must consult the private VMs of all contexts.

Garbage collection causes the memory occupied by the values of inaccessible objects to be reclaimed and made available for re-use. It does not have any effects that are visible to the PostScript language program. There is normally no need to execute the **vmreclaim** operator, since garbage collection is invoked automatically when necessary. However, there are a few situations in which this operator may be useful:

- In an interactive application that is temporarily idle, the idle time can be put to good use by invoking an immediate garbage collection; this defers the need to perform an automatic collection subsequently.
- When monitoring the VM consumption of a program, one must invoke garbage collection before executing **vmstatus** in order to obtain meaningful results.
- When measuring the execution time of a program, one must disable automatic garbage collection in order to obtain repeatable results.

The negative values that disable garbage collection apply only to the current context; that is, they do not prevent collection from occurring during execution of other contexts. Note that disabling garbage collection for too long may eventually cause a program to run out of memory and fail with a **VMerror**.

ERRORS:
rangecheck, stackunderflow, typecheck

vmstatus – **vmstatus** level used maximum

returns information about the state of the VM, as described in the *PostScript Language Reference Manual.* However, in the Display PostScript system, the returned values have more complex interpretations.

VM consumption is monitored separately for private and shared VM. The *used* and *maximum* values apply to either private or shared VM according to the current VM allocation mode (see **setshared**). Additionally, since **save** and **restore** do not have any effect on shared VM, the *level* value is meaningless if the current VM allocation mode is shared.

The *used* value is meaningful only immediately after a garbage collection has taken place (see **vmreclaim**). At other times, it may be too large because it includes memory occupied by objects that have become inaccessible but have not yet been reclaimed.

The *maximum* value reflects the maximum prior memory consumption by the VM region in question. It is not necessarily a limit, since the Display PostScript system can usually obtain more memory dynamically from the underlying operating system. However, because of memory fragmentation it may not be possible to allocate an array or string whose size is *maximum* – *used.* In an environment that supports multiple PostScript execution contexts, available memory can be reallocated from one context's VM to another.

ERRORS:
stackoverflow

wait lock condition **wait** –

releases *lock*, waits for *condition* to be notified (by some other context), and finally reacquires *lock*. The *lock* must originally have been acquired by the current context, which means that **wait** can be invoked only within the execution of a **monitor** that references the same *lock*.

If *lock* is initially held by some other context or is not held by any context, **wait** executes an **invalidcontext** error. On the other hand, during the wait for *condition*, the *lock* can be acquired by some other context. After *condition* is notified, **wait** will wait arbitrarily long to reacquire *lock*.

If the current context has previously executed a **save** not yet matched by a **restore**, **wait** executes **invalidcontext** unless both **lock** and **condition**are in shared VM. The latter case is permitted under the assumption that the **wait** is synchronizing with some context whose space is different from that of the current context.

ERRORS:
invalidcontext, stackunderflow, typecheck

writeobject file obj tag **writeobject** –

writes a binary object sequence to *file*. Except for taking an explicit *file* operand, **writeobject** is identical to **printobject** in all respects.

As is the case for all operators that write to files, the output produced by **writeobject** may accumulate in a buffer instead of being transmitted immediately. To ensure immediate transmission, a **flushfile** is required.

ERRORS:
invalidaccess, ioerror, limitcheck, rangecheck, stackunderflow, typecheck, undefined

wtranslation – **wtranslation** x y

returns the translation from the window origin to the PostScript device space origin. The integers *x* and *y* are the amounts that need to be added to a window system coordinate to produce the PostScript device space coordinate for the same position. That coordinate may in turn be transformed to user space by the **itransform** operator.

Window system and device space coordinates always correspond in resolution and orientation; they differ only in the positions of their origins. The translation from one origin to the other may change as windows are moved and resized; the precise behavior is window system specific.

ERRORS:
stackoverflow

xshow text numarray **xshow** –
text numstring **xshow** –

is similar to **xyshow**. However, for each character shown, **xshow** extracts only one number from *numarray* or *numstring*; it uses that number as the *x* displacement and the value zero as the *y* displacement. In all other repects, **xshow** behaves the same as **xyshow**.

ERRORS:
invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck

xyshow text numarray **xyshow** –
text numstring **xyshow** –

prints successive characters of *text* in a manner similar to **show**. After rendering each character, it extracts two successive numbers from the array *numarray* or the encoded number string *numstring*. These two numbers, interpreted in user space, determine the position of the origin of the next character relative to the origin of the character just shown. The first number is the *x* displacement and the second number is the *y* displacement. In other words, the two numbers override the character's normal width.

If *numarray* or *numstring* is exhausted before all the characters of *text* have been shown, a **rangecheck** error will occur.

ERRORS:
invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck

yield – **yield** –

suspends the current context until all other contexts sharing the same space have had a chance to execute. This should not be used as a synchronization primitive, since there is no way to predict how much execution the other contexts will be able to accomplish. The purpose of **yield** is to break up long-running computations that might lock out other contexts.

ERRORS:
(none)

yshow text numarray **yshow** –
text numstring **yshow** –

is similar to **xyshow**. However, for each character shown, **yshow** extracts only one number from *numarray* or *numstring*; it uses that number as the *y* displacement and the value zero as the *x* displacement. In all repects, it behaves the same as **xyshow**.

text is the string that specifies what characters are to be shown (as in **show**). *numarry* is an array whose elements are all numbers. *numstring* is an encoded number string, constructed as described in Section 2.

ERRORS:
invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck

A CHANGES SINCE LAST PUBLICATION OF THIS DOCUMENT

The changes to *PostScript Language Extensions for the Display PostScript System* from the document dated October 25, 1989, are noted in the paragraphs below.

The **definefont** operator, which was missing from the Operator Summary, has been added.

Minor amplifications and corrections, including changes in possible errors, have been made to the following operators: **definefont**, **currenthalftone**, **rectstroke**, **setcacheparams**, **setgstate**, **setfileposition**, **sethalftone**, **setucacheparams**, **ucache**, **upath**.

There are similar amplifications and corrections throughout the manual.

The index has been enhanced.

The changes to *PostScript Language Extensions for the Display PostScript System* from the document dated May 30, 1989, are noted in the paragraphs below.

The **quit** operator has been added to Section 16, and differences from its description in the *PostScript Language Reference Manual* have been provided.

The **detach** and **vmstatus** operators, which were missing from the Operator Summary, have been added.

The **setbbox** operator description has been amplified.

The changes to *PostScript Language Extensions for the Display PostScript System* from the document dated October 6, 1988, are noted in the paragraphs below.

The header of binary object sequences has been extended to allow binary object sequences with more than 255 top-level objects. The header may have four or eight characters of information. See Binary Object Sequences in Chapter 2, Alternative Language Encodings.

Management of user objects is system specific and may not be performed by the Client Library.

Three keys have been added to the top level font dictionary to control the difference in appearance between hand-tuned bitmaps and scan-converted outlines for characters (**ExactSize**, **InBetweenSize**, and **TransformedChar**). See Outline and Bitmap Font Coordination in Chapter 10, Font Related Extensions.

The dictionary stack has been changed to add a third permanent entry: **shareddict**. Originally, this stack contained only two permanent entries, **systemdict** and **userdict**.

In Appendix C, System Name Encodings, values 372-428 have been changed and values 429-603 have been deleted.

Changes to operators summarized below are documented in full in Chapter 16, Operators.

cleardictstack is a new operator that pops all nonpermanent dictionaries off the dictionary stack.

currentscreen has been extended to return the current halftone screen unless **sethalftone** was used, in which case it returns the current halftone dictionary. Previously, **currentscreen** returned a result that depended upon the halftone dictionary type.

fork gives a forked context the *objectformat* of the current context.

ineofill, given a *userpath* argument, tests whether pixels in *userpath* would be painted by **eofill**.

infill, given a *userpath* argument, tests whether pixels in *userpath* would be painted by **fill**.

instroke, given a *userpath* argument, tests whether pixels in *userpath* would be painted by **stroke**.

inueofill now has a form that tests whether pixels in $userpath_1$ would be painted by **ueofill** of $userpath_2$.

inufill now has a form that tests whether pixels in $userpath_1$ would be painted by **ufill** of $userpath_2$.

inustroke now has a form that tests whether pixels in $userpath_1$ would be painted by **ustroke** of $userpath_2$.

printobject can now write an array that contains an element of type array. Previously, this was not permissible.

rectstroke has an optional *matrix* argument.

setcacheparams now takes a *size* argument.

setscreen has been extended to take either a halftone dictionary or a spot function procedure as its third argument.

writeobject can now write an array that contains an element of type array. Previously, this was not permissible.

B POSTSCRIPT LANGUAGE CHANGES

Several additions have been made to the standard PostScript language. These additions are upward-compatible and do not affect the function of any existing PostScript language programs. The changes are included in all PostScript language implementations with version number 25.0 or higher; they are documented in editions of the *PostScript Language Reference Manual* copyright 1986 or later.

In general, PostScript language programs that are intended to be compatible with all PostScript printers should not make use of the new features. However, it is possible for a program to determine whether or not the new features are present and to invoke them conditionally. The descriptions below suggest how to determine whether a particular feature is present or absent.

Packed arrays

PostScript language procedures are represented as executable arrays which, until now, have been stored in the same fashion as literal data arrays. This representation, while offering maximum flexibility, is very costly in space (8 bytes per element). Large PostScript language programs, such as the built-in server program and downloaded preambles, consume considerable amounts of VM.

Since most programs do not require the ability to be treated as data but only the ability to be executed, a more compact representation has been introduced: the *packed array*. Programs represented as packed arrays are typically 50 to 75 percent smalier than the same programs represented as ordinary arrays.

A packed array object has a type different from an ordinary array object ('packedarraytype' versus 'arraytype'); but in most respects it behaves the same as an ordinary array. You can execute a packed array; you can extract elements (using **get**) or subarrays (using **getinterval**); you can enumerate it (using **forall**); and so forth. Individual elements extracted from a packed array are ordinary PostScript objects; a subarray of a packed array is also a packed array.

The differences between packed arrays and ordinary arrays are:

- Packed arrays are always read-only: you can't use **put**, **putinterval**, etc., to store into one.
- Packed arrays are created differently from ordinary arrays (see below).
- Accessing arbitrary elements of a packed array can be quite slow; however, accessing the elements sequentially (as is done by the PostScript interpreter and by the **forall** operator) is approximately as efficient as accessing an ordinary array.
- The **copy** operator cannot copy into a packed array (since it is read-only); however, it can copy the value of a packed array to an ordinary array of at least the packed array's length.

There are two ways in which packed arrays come into existence. The first and more common way is for the PostScript input scanner to create packed arrays automatically for all executable arrays that it reads. That is, whenever the scanner encounters a '{' while reading a file or string, it accumulates all tokens up to the matching '}' and turns them into a packed array instead of an ordinary array.

The choice of array type is controlled by a mode setting, manipulated by the new operators **setpacking** and **currentpacking** (described at the end of this section). If the array packing mode is *true*, PostScript language procedures encountered subsequently by the scanner are created as packed arrays; if the mode is *false*, procedures are created as ordinary arrays. The default value is *false* (i.e., create ordinary arrays), for compatibility with existing programs.

The other way to create a packed array is to build it explicitly by invoking the **packedarray** operator with a list of operands to be incorporated into a new packed array.

Immediately evaluated names

The language syntax has been extended to include a new kind of name token, the *immediately evaluated name*. When the scanner encounters the token '*//name*' (a name preceded by two slashes

with no intervening spaces), it immediately looks up the name in the context of the current dictionary stack and substitutes the corresponding value for the name. If the name is not found, an **undefined** error occurs.

The substitution occurs *immediately*, regardless of whether or not the token appears inside an executable array delimited by '{...}'. Note that this process is a substitution and not an execution; that is, the name's value is not executed but rather is substituted for the name itself, just as if the **load** operator had been applied to the name. This action is related to the action performed by the **bind** operator (see the *PostScript Language Reference Manual*); but whereas **bind** performs substitution only for names whose values are operators, each occurrence of the '*//name*' syntax is replaced by the value associated with *name* regardless of the value's type. The following examples illustrate this:

```
/a 3 def
/b {(test) print} def
//a ⇒ 3
//b ⇒ {(test) print}
{//a //b a /b} ⇒ {3 {(test) print} a /b}
```

The purpose of using immediately evaluated names is similar to that of using the **bind** operator: to cause names in procedures to become 'tightly bound' to their values. However, a word of caution is in order: indiscriminate use of immediately evaluated names may change the semantics of a program. In particular, recall that when the interpreter encounters a procedure object *directly* it simply pushes it on the operand stack; but when it encounters a procedure object *indirectly* (by looking up an executable name) it executes the procedure. (See Section 3.6 of the *PostScript Language Reference Manual.*) Therefore, execution of the program fragments:

```
{... b ...}
{... //b ...}
```

may have different effects if the value of the name 'b' is a procedure.

The immediately evaluated name facility is present in all versions of the PostScript interpreter since version 25.0 (as reported

by the **version** operator). Earlier versions of the interpreter will scan '*//name*' as two distinct tokens: '*/*', a literal name with no text at all, and '*/name*', a literal name whose text is *name*.

New Operators

setpacking

bool **setpacking** –

sets the array packing mode to the specified boolean value. This determines the type of executable arrays subsequently created by the scanner. The value *true* selects packed arrays; *false* selects ordinary arrays.

The packing mode affects only the creation of procedures by the scanner when it encounters program text bracketed by '{' and '}' during interpretation of an executable file or string object or during execution of the **token** operator. It does not affect the creation of literal arrays by the '[' and ']' operators or by the **array** operator.

The array packing mode setting persists until overridden by another execution of **setpacking** or until undone by a **restore**.

EXAMPLE:

```
systemdict /setpacking known
  {/savepacking currentpacking def
   true setpacking
  } if

... arbitrary procedure definitions ...

systemdict /setpacking known {savepacking setpacking} if
```

If the packed array facility is available, the procedures represented by 'arbitrary procedure definitions' are defined as packed arrays; otherwise they are defined as ordinary arrays. This example is careful to preserve the array packing mode in effect before its execution.

ERRORS:
stackunderflow, typecheck

currentpacking

– **currentpacking** bool

returns the array packing mode currently in effect.

STANDARD VALUE: false

ERRORS:
stackoverflow

packedarray any_0 ... any_{n-1} n **packedarray** packedarray

creates a packed array object of length *n* containing the objects any_0 through any_{n-1} as elements. **packedarray** first removes the non-negative integer *n* from the operand stack. It then removes that number of objects from the operand stack, creates a packed array containing those objects as elements, and finally pushes the resulting packed array object on the operand stack.

The resulting object has a type of 'packedarraytype', a literal attribute, and read-only access. In all other respects, its behavior is identical to that of an ordinary array object.

STANDARD VALUE: false

ERRORS:
rangecheck, stackunderflow, typecheck, VMerror

showpage and **copypage**

The correct use of **showpage** versus **copypage** is a matter requiring some clarification. Inappropriate use of **copypage** can result in significant performance degradation in new PostScript printers.

showpage is the normal operator for causing pages to be output. It has three effects: it prints the current page, it erases the current page, and it reinitializes the graphics state.

copypage is a somewhat more specialized operator that just prints the current page but does not erase it or reset the graphics state. Its main intended use is to permit adding new marks to an existing page, e.g., when building up a page incrementally.

showpage is logically equivalent to the sequence:

```
copypage erasepage initgraphics
```

However, use of **copypage** for printing pages can degrade page throughput significantly. One reason for this is that **showpage** performs the printing and the erasing in parallel whereas the **copypage erasepage** method performs them serially; there are other reasons as well.

copypage should also not be used to defeat the automatic **initgraphics** of **showpage**.[1] That is, to print and erase the current page but leave the graphics state unchanged, you should *not* say:

```
copypage erasepage
```

Instead you should say:

```
gsave showpage grestore
```

Please also note that the correct way to print multiple copies of a page is to associate the desired number of copies with the name **#copies** prior to invoking **showpage**, as discussed under **showpage** in the *PostScript Language Reference Manual*. The **#copies** convention now applies uniformly to both **showpage** and **copypage**, whereas formerly it applied only to **showpage**.

[1]Unfortunately, the current *PostScript Language Tutorial and Cookbook* includes an example that uses this technique.

C SYSTEM NAME ENCODINGS

index	*name*	*index*	*name*	*index*	*name*
0	abs	41	currentrgbcolor	82	idiv
1	add	42	currentshared	83	idtransform
2	aload	43	curveto	84	if
3	anchorsearch	44	cvi	85	ifelse
4	and	45	cvlit	86	image
5	arc	46	cvn	87	imagemask
6	arcn	47	cvr	88	index
7	arct	48	cvrs	89	ineofill
8	arcto	49	cvs	90	infill
9	array	50	cvx	91	initviewclip
10	ashow	51	def	92	inueofill
11	astore	52	defineusername	93	inufill
12	awidthshow	53	dict	94	invertmatrix
13	begin	54	div	95	itransform
14	bind	55	dtransform	96	known
15	bitshift	56	dup	97	le
16	ceiling	57	end	98	length
17	charpath	58	eoclip	99	lineto
18	clear	59	eofill	100	load
19	cleartomark	60	eoviewclip	101	loop
20	clip	61	eq	102	lt
21	clippath	62	exch	103	makefont
22	closepath	63	exec	104	matrix
23	concat	64	exit	105	maxlength
24	concatmatrix	65	file	106	mod
25	copy	66	fill	107	moveto
26	count	67	findfont	108	mul
27	counttomark	68	flattenpath	109	ne
28	currentcmykcolor	69	floor	110	neg
29	currentdash	70	flush	111	newpath
30	currentdict	71	flushfile	112	not
31	currentfile	72	for	113	null
32	currentfont	73	forall	114	or
33	currentgray	74	ge	115	pathbbox
34	currentgstate	75	get	116	pathforall
35	currenthsbcolor	76	getinterval	117	pop
36	currentlinecap	77	grestore	118	print
37	currentlinejoin	78	gsave	119	printobject
38	currentlinewidth	79	gstate	120	put
39	currentmatrix	80	gt	121	putinterval
40	currentpoint	81	identmatrix	122	rcurveto

123	read	167	stroke	211	Times-Roman
124	readhexstring	168	strokepath	212	execuserobject
125	readline	169	sub	256	=
126	readstring	170	systemdict	257	==
127	rectclip	171	token	258	ISOLatin1Encoding
128	rectfill	172	transform	259	StandardEncoding
129	rectstroke	173	translate	260	[
130	rectviewclip	174	truncate	261	]
131	repeat	175	type	262	atan
132	restore	176	uappend	263	banddevice
133	rlineto	177	ucache	264	bytesavailable
134	rmoveto	178	ueofill	265	cachestatus
135	roll	179	ufill	266	closefile
136	rotate	180	undef	267	colorimage
137	round	181	upath	268	condition
138	save	182	userdict	269	copypage
139	scale	183	ustroke	270	cos
140	scalefont	184	viewclip	271	countdictstack
141	search	185	viewclippath	272	countexecstack
142	selectfont	186	where	273	cshow
143	setbbox	187	widthshow	274	currentblackgeneration
144	setcachedevice	188	write	275	currentcacheparams
145	setcachedevice2	189	writehexstring	276	currentcolorscreen
146	setcharwidth	190	writeobject	277	currentcolortransfer
147	setcmykcolor	191	writestring	278	currentcontext
148	setdash	192	wtranslation	279	currentflat
149	setfont	193	xor	280	currenthalftone
150	setgray	194	xshow	281	currenthalftonephase
151	setgstate	195	xyshow	282	currentmiterlimit
152	sethsbcolor	196	yshow	283	currentobjectformat
153	setlinecap	197	FontDirectory	284	currentpacking
154	setlinejoin	198	SharedFontDirectory	285	currentscreen
155	setlinewidth	199	Courier	286	currentstrokeadjust
156	setmatrix	200	Courier-Bold	287	currenttransfer
157	setrgbcolor	201	Courier-BoldOblique	288	currentundercolorremoval
158	setshared	202	Courier-Oblique	289	defaultmatrix
159	shareddict	203	Helvetica	290	definefont
160	show	204	Helvetica-Bold	291	deletefile
161	showpage	205	Helvetica-BoldOblique	292	detach
162	stop	206	Helvetica-Oblique	293	deviceinfo
163	stopped	207	Symbol	294	dictstack
164	store	208	Times-Bold	295	echo
165	string	209	Times-BoldItalic	296	erasepage
166	stringwidth	210	Times-Italic	297	errordict

298	execstack	342	setfileposition	386	K
299	executeonly	343	setflat	387	L
300	exp	344	sethalftone	388	M
301	false	345	sethalftonephase	389	N
302	filenameforall	346	setmiterlimit	390	O
303	fileposition	347	setobjectformat	391	P
304	fork	348	setpacking	392	Q
305	framedevice	349	setscreen	393	R
306	grestoreall	350	setstrokeadjust	394	S
307	handleerror	351	settransfer	395	T
308	initclip	352	setucacheparams	396	U
309	initgraphics	353	setundercolorremoval	397	V
310	initmatrix	354	sin	398	W
311	instroke	355	sqrt	399	X
312	inustroke	356	srand	400	Y
313	join	357	stack	401	Z
314	kshow	358	status	402	a
315	ln	359	statusdict	403	b
316	lock	360	true	404	c
317	log	361	ucachestatus	405	d
318	mark	362	undefinefont	406	e
319	monitor	363	usertime	407	f
320	noaccess	364	ustrokepath	408	g
321	notify	365	version	409	h
322	nulldevice	366	vmreclaim	410	i
323	packedarray	367	vmstatus	411	j
324	quit	368	wait	412	k
325	rand	369	wcheck	413	l
326	rcheck	370	xcheck	414	m
327	readonly	371	yield	415	n
328	realtime	372	defineuserobject	416	o
329	renamefile	373	undefineuserobject	417	p
330	renderbands	374	UserObjects	418	q
331	resetfile	375	cleardictstack	419	r
332	reversepath	376	A	420	s
333	rootfont	377	B	421	t
334	rrand	378	C	422	u
335	run	379	D	423	v
336	scheck	380	E	424	w
337	setblackgeneration	381	F	425	x
338	setcachelimit	382	G	426	y
339	setcacheparams	383	H	427	z
340	setcolorscreen	384	I	428	setvmthreshold
341	setcolortransfer	385	J		

Index

POSTSCRIPT®
LANGUAGE

COLOR EXTENSIONS

ADOBE SYSTEMS
INCORPORATED

PostScript Language Color Extensions

January 23, 1990

Revised by Amy Davidson.

Contents

1 ABOUT THIS MANUAL

This document contains:

- A description of the extensions to the PostScript® language that support new color functionality.
- Detailed information on the associated color operators.

Section 2 provides a general introduction to color functionality in the PostScript language.

Section 3 discusses the extended color capabilities of the PostScript language.

Section 4 contains an alphabetical listing of descriptions for all color operators that have been added to the PostScript language.

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

2 ABOUT THE POSTSCRIPT LANGUAGE COLOR EXTENSIONS

The PostScript language has been extended to provide more complete color functionality. This includes cyan-magenta-yellow-black (CMYK) color specification, black generation and undercolor removal functions, screen and transfer functions for four separate color components, and extension of the **image** concept to a **colorimage** operator that accepts multiple color components.

Earlier versions of the PostScript language support color using the **setrgbcolor** and **sethsbcolor** operators, which enable the PostScript interpreter to paint filled regions, strokes, image masks, and characters in color. On black-and-white machines, these operators generate an equivalent gray shade, which is printed or displayed.

To support color more fully, the PostScript language has been extended to provide the following functions:

- Most significantly, multiple color images: the **colorimage** operator renders a multiple color image; it functions

analogously to the the **image** operator, but uses red-green-blue (RGB) or cyan-magenta-yellow-black (CMYK) color input and generates full-color output.

- Halftone screen definitions: the **setcolorscreen** operator specifies halftone screen definitions for red, green, blue, and gray, or cyan, magenta, yellow, and black concurrently; it is the logical expansion of **setscreen** as it takes the same three arguments to define each screen for each printing ink.
- Color correction: the **setcolortransfer** operator sets the transfer function parameters for red, green, blue, and gray; it is an expansion of **settransfer** to four color components. The **setblackgeneration** operator provides a black generation function that establishes a black component from a cyan, magenta, and yellow specification. The **setundercolorremoval** operator provides undercolor removal from the cyan, magenta, and yellow components to compensate for the addition of black by the black generation function.
- CMYK color specification: the **setcmykcolor** operator allows the user to set the current color in the graphics state to a cyan-magenta-yellow-black color directly, bypassing the color correction operators.

The PostScript language supports one-color, three-color, and four-color output devices. The color devices can be binary (one-bit-per-pixel per color component) or grayscale (multiple-bits-per-pixel per color component, representing a range of intensities of each color component). A binary device uses halftoning to produce intermediate shades of its color components. A device that has eight-bits-per-pixel per component, called a *full grayscale* device, does not use halftoning. Devices with more than one and fewer than eight bits per pixel use a combination of built-in intensities and halftoning to produce the full range of desired shades of their color components. Three-color devices may be either red-green-blue (RGB), typically for displays and film recorders, or cyan-magenta-yellow (CMY) for printers. Four-color devices are cyan-magenta-yellow-black (CMYK) for color printers and color separation making devices.

The color operators described in this document are available in two forms. Some new versions of the PostScript interpreter have these operators built in. For other versions of the PostScript in-

terpreter, a package of PostScript language programs that emulate these operators is available.

3 NEW FEATURES

The color extensions to the PostScript language include CMYK color specification, black generation, and undercolor removal. Because black generation and undercolor removal use procedures, they allow you to adjust the conversion from RGB to CMYK values, for example, by producing less black in black generation.

Operators are provided that allow full color input and output for screens, transfer functions, and images. The new color features are described conceptually in Sections 3.1 through 3.6. The corresponding operators are documented in Section 4. The new operators are identified on first mention by an asterisk; for example, **colorimage***.

3.1 CONVERSION OF RGB VALUES TO CMYK

Colors are formed either by adding light to black or by subtracting light from white. Computer displays and film recorders typically add colors, while printing inks typically subtract colors. These two methods for forming colors give rise to the two major complementary color specifications, the additive RGB specification and the subtractive CMYK specification.

Accordingly, a color component in these specifications either indicates the amount of light it reflects or the amount of light it absorbs. Each one of three standard printing process colors, *cyan*, *magenta*, and *yellow*, absorb one of the standard light components, *red*, *green*, and *blue*, respectively. *Black*, a fourth standard printing process color, absorbs all components of light. In the red-green-blue (RGB) color specification, each of its red, green, and blue components is associated with a real number between 0.0 and 1.0, inclusive, where 0.0 represents dark (no light) and 1.0 represents full light. In the cyan-magenta-yellow-black (CMYK) color specification, each of the four components is associated with real numbers between 0.0 and 1.0 inclusive, where

0.0 represents full light (no ink), and 1.0 represents dark (full ink).

The following equations demonstrate the relationship between the RGB and CMYK color specifications. Since cyan is the absence of red light, magenta is the absence of green light, and yellow is the absence of blue light,

cyan = 1.0 – red
magenta = 1.0 – green
yellow = 1.0 – blue

A color that is 0.2 red, 0.7 green, and 0.4 blue can also be expressed as 1.0 – 0.2 = 0.8 cyan, 1.0 – 0.7 = 0.3 magenta, and 1.0 – 0.4 = 0.6 yellow. To improve the fidelity of blacks and grays, a fourth process color, black, is often available on color printers. Just as red in the RGB specification is the opposite of cyan in the CMYK specification, a black value is the opposite to a PostScript language gray value; that is,

black = 1.0 – gray.

3.2 BLACK GENERATION AND UNDERCOLOR REMOVAL

Logically, cyan, magenta, and yellow are all that are needed to generate a printing color completely. Thus an equal percentage of cyan, magenta, and yellow should create the equivalent percentage of black. In reality, colored printing inks do not mix perfectly, and such combinations often form dark brown shades instead. Thus, it is often desirable to substitute real black ink for the mixed-black portion of a color to obtain a truer color rendition on a printer.

Black generation is the process of calculating the amount of black to be used when trying to print a particular color. Undercolor removal is the process of reducing the amount of cyan, magenta, and yellow components to compensate for the amount of black that was added by the black generation. Flexibility in performing these functions is important for achieving good results under different printing conditions.

The **setblackgeneration*** operator provides the functionality to generate extra black, no black, or a black value equal to all or a

fraction of the minimum values of cyan, magenta and yellow. Its argument is a procedure that takes one numeric argument, the minimum value of *user cyan*, *magenta*, and *yellow* color components, and returns a single numeric result, the *user black* value (where a user color component value is that specified in the PostScript language program before application of the corresponding transfer function). This procedure is automatically applied whenever **setrgbcolor**, the three-color case of **colorimage***, or **sethsbcolor** specifies a color. This *user black* value is then mapped to a *device black* value by applying the gray transfer function to its difference from 1.0 and subtracting the result from 1.0 (see **setcolortransfer***). The black generation function is not applied when **setgray**, **setcmykcolor***, or the one- or four-color cases of **colorimage** specifies colors. This computed black value is used only when producing output on four-color devices.

The **setundercolorremoval*** operator provides functionality to remove some amount of color from each of the cyan, magenta, and yellow components. This amount could be exactly the same amount as was generated to make the black component, zero (so no color is removed from the cyan, magenta and yellow components), some fraction of the black amount, or even a negative amount. Like **setblackgeneration**, this operator permits considerable flexibility in color correction.

The argument to **setundercolorremoval** is a procedure that takes one numeric argument, the minimum value of *user cyan*, *magenta*, and *yellow* color components, and returns a single numeric result that is subtracted from each of these original user color components. This procedure is applied whenever **setrgbcolor**, the three-color case of **colorimage**, or **sethsbcolor** specifies a color. After subtracting the value generated in the above mapping from the color components and resetting negative values to 0.0 and values greater than 1.0 to 1.0, each component is subtracted from 1.0 to yield *red*, *green*, and *blue* components. Each of these components is mapped into a *device color component* using its respective transfer function (see **setcolortransfer**). Undercolor removal is not applied when **setgray**, **setcmykcolor**, or the one- or four-color cases of **colorimage** specifies a color. Undercolor removal is used only when outputting on four-color devices.

The following three sets of equations define the complete color transformation process from RGB to CMYK. In the first set of equations, the values red_u, grn_u, and blu_u are supplied by the user. gry_u and blk_u are assigned the values 1.0 and 0.0, respectively.

$$cyn_u = 1.0 - red_u$$
$$mag_u = 1.0 - grn_u$$
$$yel_u = 1.0 - blu_u$$
$$gry_u = 1.0$$
$$blk_u = 0.0$$

In the second set of equations, k is the minimum value of cyn_u, mag_u, and yel_u. u is the amount of undercolor removal; it is determined by the applying the undercolor removal procedure to the value of k. *UCR()* is the undercolor removal procedure.

$$k = Min(cyn_u, mag_u, yel_u)$$
$$u = UCR(k)$$

In the third set of equations, *BG()* is the black generation procedure. *RedT*, *GrnT*, *BluT*, and *GryT* are the red, green, blue, and gray transfer functions (see **setcolortransfer**), respectively. The result of applying these equations is to produce CMYK output values for the device.

The values red_d, grn_d, blu_d, and gry_d are intermediate values used to compute cyn_d, mag_d, yel_d, and blk_d, which can be sent to a CMYK device.

$$red_d = RedT(1.0 - Min(1.0, Max(0.0, cyn_u - u)))$$
$$grn_d = GrnT(1.0 - Min(1.0, Max(0.0, mag_u - u)))$$
$$blu_d = BluT(1.0 - Min(1.0, Max(0.0, yel_u - u)))$$
$$gry_d = GryT(1.0 - BG(k))$$
$$cyn_d = 1.0 - red_d$$
$$mag_d = 1.0 - grn_d$$
$$yel_d = 1.0 - blu_d$$
$$blk_d = 1.0 - gry_d$$

3.3 DIRECT CMYK COLOR SPECIFICATION

For the most demanding cases, color matching can require more complicated methods than those described above. The

setcmykcolor operator and the four-color case of the **colorimage** operator bypass the black generation and undercolor removal operations, allowing the knowledgeable user to specify the cyan, magenta, yellow, and black color components for a particular device. These operators do not provide correction other than the transfer functions that **setcolortransfer** specifies; the results are device dependent.

The following equations define the complete color transformation process for the **setcmykcolor** operator and the four-color case of the **colorimage** operator. They are equivalent to those given in Section 3.2 except for the omission of the black generation and undercolor removal steps. The values are the same as those defined in Section 3.2.

$$red_u = 1.0 - cyn_u$$
$$grn_u = 1.0 - mag_u$$
$$blu_u = 1.0 - yel_u$$
$$gry_u = 1.0 - blk_u$$
$$red_d = RedT(red_u)$$
$$grn_d = GrnT(grn_u)$$
$$blu_d = BluT(blu_u)$$
$$gry_d = GryT(gry_u)$$
$$cyn_d = 1.0 - red_d$$
$$mag_d = 1.0 - grn_d$$
$$yel_d = 1.0 - blu_d$$
$$blk_d = 1.0 - gry_d$$

3.4 COLOR SCREENS, TRANSFER FUNCTIONS, AND IMAGES

The operators **setcolorscreen**, **currentcolorscreen**, **setcolortransfer**, and **currentcolortransfer** provide an expansion of the operators **setscreen**, **currentscreen**, **settransfer**, and **currenttransfer**, respectively, by setting up a screen and a transfer function for each color component. The **colorimage** operator provides an expansion of the **image** operator to allow samples of one, three, or four color components.

3.5 THE COLORIMAGE OPERATOR

colorimage is the logical expansion of **image** to handle sampled images whose samples are composed of color components rather than gray values. The initial arguments to **colorimage** are the same as those for **image**. The final arguments differ according to the number of color components per sample and according to the encoding method.

The arguments to **colorimage** are shown below; see Section 4 for precise definitions of these arguments:

width height bits/component matrix $proc_0$ [...$proc_{ncolors-1}$] multiproc ncolors

ncolors describes the number of color components in each sample. Legal values for *ncolors* are 1 (gray-level samples only), 3 (RGB samples), or 4 (CMYK samples). *multiproc* is a boolean that distinguishes between encoding methods. When *multiproc* is *false*, there is a single procedure and color components are bunched together; when *multiproc* is *true*, there are multiple procedures, one per color, and components are separated into strings of like colors.

The legal variations of *ncolors* and *multiproc* allow the following possibilities (where *proc* subscripts have been changed to words to indicate the purpose of each procedure):

w h b/c matrix $proc_{gray}$ false 1
w h b/c matrix $proc_{gray}$ true 1
w h b/c matrix $proc_{rgb}$ false 3
w h b/c matrix $proc_{red}$ $proc_{green}$ $proc_{blue}$ true 3
w h b/c matrix $proc_{cmyk}$ false 4
w h b/c matrix $proc_{cyan}$ $proc_{magenta}$ $proc_{yellow}$ $proc_{black}$ true 4

The first two variations here are both equivalent to

w h b/s matrix $proc_{gray}$ **image**

Data formats for colorimage operator. As indicated above, the **colorimage** operator has two forms, distinguished by its *multiproc* argument.

The single-procedure form is most useful if sample input is taken from a source that has already merged the color components. This form provides samples for which each RGB triple or

CMYK quadruple is packed together in the string result of the procedure, using one of the following bit formats (where the high-order bit is shown on the left):

Bits/ comp.	RGB Format				
1	RGBRGBRG	BRGBRGBR	GBRGBRGB	RGBRGBRG	....
2	RRGGBBRR	GGBBRRGG	BBRRGGBB	RRGGBBRR	....
4	RRRRGGGG	BBBBRRRR	GGGGBBBB	RRRRGGGG	....
8	RRRRRRRR	GGGGGGGG	BBBBBBBB	RRRRRRRR	....

or

Bits/ comp.	CMYK Format				
1	CMYKCMYK	CMYKCMYK	CMYKCMYK	CMYKCMYK	....
2	CCMMYYKK	CCMMYYKK	CCMMYYKK	CCMMYYKK	....
4	CCCCMMMM	YYYYKKKK	CCCCMMMM	YYYYKKKK	....
8	CCCCCCCC	MMMMMMMM	YYYYYYYY	KKKKKKKK	....

The multiple-procedure form expects each procedure to return a string of values for only one color component per sample, using the same format as strings returned by the *proc* argument of the **image** operator. For a three-color image, $proc_0$ returns red values, $proc_1$ returns green values, and $proc_2$ returns blue values. For a four-color image, $proc_0$ returns cyan values, $proc_1$ returns magenta values, $proc_2$ returns yellow values, and $proc_3$ returns black values. The **colorimage** operator calls each of these procedures in turn, starting with $proc_0$ and continuing with $proc_1$, $proc_2$, and, if available, $proc_3$. When the **colorimage** operator needs more samples, it calls these procedures again in the same order. The color procedures must use separate strings for the three or four results of the three or four procedures; that is, reusing the red string for the green values may cause some of the red values to be lost. Also, the three or four procedures must return strings of identical lengths within each cycle of three or four calls.

The multiple-procedure form is most useful when color sample data are taken from separate color scanner passes. The **colorimage** operator requires the color data to be interleaved, since the operator requires all three or four components of any sample at the same time in order to do its work. The single-procedure form interleaves the data at the sample level; this may be convenient only if the data are already in that form when preparing the PostScript language page description. The

multiple-procedure form allows interleaving at a much coarser level. Typically, each procedure of the multiple-procedure form returns components for some number of scan lines of samples, where the number of components returned at each call is limited by the string storage available in the PostScript interpreter.

Examples of colorimage operator. The following examples illustrate the use of the **colorimage** operator:

```
EXAMPLE 1:
    /rgbstr 192 string def      % string to hold 256 two-bit samples
                                %   each of red, green, and blue data
    45 140 translate            % locate lower left corner of image
    132 132 scale               % map image to 132 point square
    256 256 2                   % dimensions of source image
    [256 0 0 -256 0 256]        % map unit square to source
    {currentfile                % read image data from program file
      rgbstr readhexstring pop}
    false 3                     % single proc, 3 colors, bit format:
                                %   rrggbbrr ggbbrrgg bbrrggbb ...
    colorimage
    94a1bec8c0b371a3a5c4d281 ...     (98304 hex digits of image data)
```

The code fragment above shows a one-procedure, 2-bit RGB image. The base-4 representation of the hexadecimal data is

```
2110 2201 2332 ...
```

which is composed of the following color samples:

```
r=2 g=1 b=1 r=0   g=2 b=2 r=0 g=1   b=2 r=3 g=3 b=2 ...
```

EXAMPLE 2:

```
/rstr 256 string def        % string to hold 256 8-bit red samples
/gstr 256 string def        % string to hold 256 8-bit green samples
                            %   (distinct from rstr)
/bstr 256 string def        % string to hold 256 8-bit blue samples
                            %   (distinct from rstr and bstr)
45 140 translate            % locate lower left corner of image
132 132 scale               % map image to 132 point square
256 256 8                   % dimensions of source image
[256 0 0 -256 0 256]        % map unit square to source
{currentfile rstr readhexstring pop}
                            % read red data from program file
{currentfile gstr readhexstring pop}
                            % read green data from program file
{currentfile bstr readhexstring pop}
                            % read blue data from program file
true 3                      % multiple proc, 3 colors
colorimage
7b5e606969615365556a6a66 ...    (512 hex digits of red data)
88868d848a92878578787a82 ...    (512 hex digits of green data)
62717c7b736e707d7b6a7c79 ...    (512 hex digits of blue data)
7d8b8d8c837d8b8e9284878e ...    (512 hex digits of red data)
2788b838b8e8e86868988908 ...    (512 hex digits of green data)
81817d857f85858290949487 ...    (512 hex digits of blue data)
... (390144 more hex digits of RGB data, cycling as above)
```

The code fragment above shows a three-procedure, 8-bit RGB image. The initial samples for each color, in hexadecimal representation, are

```
red:    7b  5e  60  69  ...
green:  88  86  8d  84  ...
blue:   62  71  7c  7b  ...
```

EXAMPLE 3:

```
/cstr 128 string def        % string to hold 256 4-bit cyan samples
/mstr 128 string def        % string to hold 256 4-bit magenta samples
                            %   (distinct from cstr)
/ystr 128 string def        % string to hold 256 4-bit yellow samples
                            %   (distinct from cstr and mstr)
/kstr 128 string def        % string to hold 256 4-bit black samples
                            %   (distinct from cstr, mstr, and ystr)
45 140 translate            % locate lower left corner of image
132 132 scale               % map image to 132 point square
256 256 4                   % dimensions of source image
[256 0 0 -256 0 256]        % map unit square to source
{currentfile cstr readhexstring pop}
                            % read cyan data from program file
{currentfile mstr readhexstring pop}
                            % read magenta data from program file
{currentfile ystr readhexstring pop}
                            % read yellow data from program file
{currentfile kstr readhexstring pop}
                            % read black data from program file
true 4                      % multiple proc, 4 colors
colorimage
e1d8caa57b655b6779606b72 ...    (256 hex digits of cyan data)
6bdbb867b9fb6a4859569989 ...    (256 hex digits of magenta data)
996796e639cc0b29f94736c7 ...    (256 hex digits of yellow data)
c9c0cad0d3cad2b7c9e2d7d8 ...    (256 hex digits of black data)
5d2d6d7d4d3d1d4d6c9d4d9d ...    (256 hex digits of cyan data)
4cdcfd4d6d1d8d7d5d4d2d2d ...    (256 hex digits of magenta data)
d2b7c9e2d7d8d8cbbac2d9d8 ...    (256 hex digits of yellow data)
88ae96632a70f6f4d8d9d9d8 ...    (256 hex digits of black data)
... (260096 more hex digits of CMYK data, cycling as above)
```

The code fragment above shows a four-procedure, four-bit CMYK image. The initial samples for each color, in hexadecimal representation, are

```
cyan:       e 1 d 8 c a a 5 ...
magenta:    6 b d b b 8 6 7 ...
yellow:     9 9 6 7 9 6 e 6 ...
black:      c 9 c 0 c a d 0 ...
```

EXAMPLE 4:

```
/cstr 1024 string def      % string to hold 1024 8-bit cyan samples
/mstr 1024 string def      % string to hold 1024 8-bit magenta samples
                           %   (distinct from cstr)
/ystr 1024 string def      % string to hold 1024 8-bit yellow samples
                           %   (distinct from cstr and mstr)
/kstr 1024 string def      % string to hold 1024 8-bit black samples
                           %   (distinct from cstr, mstr, and ystr)
/cfile (img/smp.c) (r) file def    % binary file containing 1048576 8-bit
                                   %   cyan samples
/mfile (img/smp.m) (r) file def    % binary file containing 1048576 8-bit
                                   %   magenta samples
/yfile (img/smp.y) (r) file def    % binary file containing 1048576 8-bit
                                   %   yellow samples
/kfile (img/smp.k) (r) file def    % binary file containing 1048576 8-bit
                                   %   black samples
36 126 translate           % locate lower left corner of image
540 540 scale              % map image to 540 point square
1024 1024 8                % dimensions of source image
[1024 0 0 -1024 0 1024]        % map unit square to source
{cfile cstr readstring pop}    % read cyan data from img/smp.c
{mfile mstr readstring pop}    % read magenta data from img/smp.m
{yfile ystr readstring pop}    % read yellow data from img/smp.y
{kfile kstr readstring pop}    % read black data from img/smp.k
true 4                     % multiple proc, 4 colors
colorimage
```

The code fragment above shows a four-procedure, 8-bit CMYK image, with cyan, magenta, yellow, and black samples taken from the files *img/smp.c*, *img/smp.m*, *img/smp.y*, and *img/smp.k*, respectively. This example only applies to a PostScript interpreter that has a file system.

3.6 COLOR IMPLEMENTATIONS

Each PostScript interpreter uses default color output methods that correspond to its target printer. If the printer is a direct-color binary device, the standard output method produces three- or four-color output. If the printer is a grayscale color device, the PostScript interpreter uses a grayscale three- or four-color output method. If the printer is a black-and-white device, the default output method produces a single black-and-white rendition of each page described.

Some printers can also be used to produce color separations. A

color separation consists of three or four black-and-white component pages for each color page to be described, with each black-and-white page corresponding to the output for one color component. Separations are normally prepared for use in a subsequent printing process in which a single page is overprinted three or four times to form the intended full-color output, each time using a different black-and-white component page and the ink color associated with it.

4 OPERATORS

The following pages contain an alphabetical listing of the new PostScript color operators.

colorimage width height bits/component matrix proc$_0$ [...proc$_{ncolors-1}$] multiproc
ncolors **colorimage** –

renders a sampled image on the current page. The samples can contain one, three, or four color components. The first four arguments are the same as those for the **image** operator. The *bits/component* argument applies equally to all color components. **colorimage** permits its *proc$_i$* arguments to return RGB or CMYK sample values rather than the single-color (gray) values returned by the *proc* argument of the **image** operator.

The *ncolors* argument (1, 3, or 4) is the number of color components represented in the samples. If *ncolors* is 1, the samples have only one component, a gray component, and the operation of **colorimage** is equivalent to that of **image** with the same five initial arguments. If the *ncolors* argument is 3, the **colorimage** operator takes RGB (light-high) samples. If the *ncolors* argument is 4, the **colorimage** operator takes CMYK (dark-high) samples. On a four-color (CMYK) machine, the PostScript interpreter converts a three-color (RGB) image to CMYK using the black generation and undercolor removal procedures; a four-color (CMYK) image bypasses these operations.

The *multiproc* argument is a boolean that distinguishes between two forms of the **colorimage** operator: *false* indicates the single-procedure form, which requires one procedure argument (*proc$_0$*); *true* indicates the multiple-procedure form, which requires one procedure argument per sample color (*proc$_0$ ... proc$_{ncolors-1}$*) – three procedure arguments for RGB samples or four-procedure arguments for CMYK samples. If the *ncolors* argument is 1, there is only one procedure argument, *proc$_0$*, regardless of the value of the *multiproc* argument. For a detailed description of the data formats and how the *proc$_i$* procedures are called, see "Data formats for colorimage operator" on page 8.

Use of **setcolorscreen**, **setcolortransfer**, **setscreen**, or **settransfer** by any of the *proc$_i$* procedures causes unpredictable results. Use of the **colorimage** operator after a **setcachedevice** within the context of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
limitcheck, rangecheck, stackunderflow, typecheck, undefined, undefinedresult

currentblackgeneration – **currentblackgeneration** proc

returns the current black generation function in the graphics state (see **setblackgeneration**).

ERRORS:
stackoverflow

currentcmykcolor – **currentcmykcolor** cyan magenta yellow black

returns the four components of the current color in the graphics state according to the cyan-magenta-yellow-black color model (see **setcmykcolor**).

Note that the **currentgray** operator returns a weighted average of all four color components. Applying it is the equivalent of the following use of **currentcmykcolor**:

```
1.0 currentcmykcolor 4 1 roll 0.11 mul 3 1 roll 0.59 mul
exch 0.30 mul add add add sub dup 0.0 lt {pop 0.0} if
```

ERRORS:
stackoverflow

currentcolorscreen – **currentcolorscreen** r/c-frequency r/c-angle r/c-proc g/m-frequency g/m-angle g/m-proc b/y-frequency b/y-angle b/y-proc g/k-frequency g/k-angle g/k-proc

returns all 12 current halftone screen parameters in the graphics state (see **setcolorscreen**). In the notation used here and in **setcolorscreen**, r/c is red/cyan, g/m is green/magenta, b/y is blue/yellow, and g/k is gray/black.

The **currentcolorscreen** operator is the logical expansion of **currentscreen** to four color components. Applying the **currentscreen** operator returns the three parameters describing the gray/black screen. It is the equivalent of the following use of **currentcolorscreen**:

```
currentcolorscreen 12 3 roll 9 {pop} repeat
```

ERRORS:
stackoverflow

currentcolortransfer – **currentcolortransfer** redproc greenproc blueproc grayproc

returns the current transfer functions in the graphics state for each of the four color components (see **setcolortransfer**).

The **currentcolortransfer** operator is the logical expansion of **currenttransfer** to four color components. Applying the **currenttransfer** operator returns the gray transfer function. It is the equivalent of the following use of **currentcolortransfer**:

```
currentcolortransfer 4 1 roll pop pop pop
```

ERRORS:
stackoverflow

currentundercolorremoval – **currentundercolorremoval** proc

returns the current undercolor removal function in the graphics state (see **setundercolorremoval**).

ERRORS:
stackoverflow

setblackgeneration proc **setblackgeneration** –

sets the current black generation function parameter in the graphics state. The *proc* operand must be a PostScript language procedure that can be called with a number in the range 0.0 to 1.0 (inclusive) on the operand stack and that returns a number in the same range. This procedure maps the minimum of the *user cyan*, *magenta*, and *yellow* color components to *user black* values.

For additional information, see Section 3.2.

EXAMPLE:

```
{dup .75 le {pop 0.0} {.75 sub 4.0 mul} ifelse} setblackgeneration
```

This PostScript language code fragment sets the black component to zero when the minimum of cyan, magenta, and yellow is less than or equal to .75. Minima greater than .75 produce a black component that increases linearly from 0.0 (at a minimum of .75) to 1.0 (when user cyan, magenta, and yellow all have values of 1.0).

The use of **setblackgeneration** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
stackunderflow, typecheck

setcmykcolor cyan magenta yellow black **setcmykcolor** –

sets the current color parameter in the graphics state to a color described by the parameters *cyan*, *magenta*, *yellow*, and *black*, each of which must be a number in the range 0.0 to 1.0 inclusive. This establishes the color used subsequently to paint shapes such as lines, areas, and characters on the current page. This operator bypasses the black generation and undercolor removal operations.

For additional information, see Section 3.3.

Note that applying the **setgray** operator sets the gray color component to its single argument value and the red, green, and blue color components to 1.0. It is the equivalent of the following use of **setcmykcolor**:

```
0.0 0.0 0.0 1.0 5 -1 roll sub setcmykcolor
```

The use of **setcmykcolor** after a **setcachedevice**operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
stackunderflow, typecheck

setcolorscreen r/c-frequency r/c-angle r/c-proc g/m-frequency g/m-angle g/m-proc b/y-frequency b/y-angle b/y-proc g/k-frequency g/k-angle g/k-proc **setcolorscreen** –

sets the current halftone screen definitions for red/cyan (r/c), green/magenta (g/m), blue/yellow (b/y), and gray/black (g/k) output color components in the graphics state. Each of the *r/c-frequency*, *g/m-frequency*, *b/y-frequency*, and *g/k-frequency* operands is a number that specifies the screen frequency for one output color component, measured in halftone cells per inch in device space. The *r/c-angle*, *g/m-angle*, *b/y-angle*, and *g/k-angle* operands specify the number of degrees by which their respective halftone screens are rotated with respect to the device coordinate system. Each of the *r/c-proc*, *g/m-proc*, *b/y-proc*, and *g/k-proc* operands is a PostScript language procedure (as for **setscreen**). Each procedure defines one color component's spot function. (For more information on spot functions, see the section on halftone screens in the *PostScript Language Reference Manual*.) The red/cyan, green/magenta, and blue/yellow screens have no effect on a black-and-white device, the gray/black screen has no effect on an RGB or CMY device, and no screens have any effect on a full (8-bits-per-pixel) grayscale device.

Color printers that use halftoning may require a different angle for each color component in order to produce attractive output. Each color printer containing a PostScript interpreter has a default color screen chosen to look good on that printer.

The **setcolorscreen** operator is the logical expansion of **setscreen** to four color components. It takes the same three types of arguments as **setscreen**, but repeated four times. Applying the **setscreen** operator in an environment with four color components sets all four screens equally. It is the equivalent of the following use of **setcolorscreen**:

```
3 copy 6 copy setcolorscreen
```

EXAMPLE:

```
% 50 line dot screen with 75 degree cyan,
% 15 degree magenta,
% 0 degree yellow, and 45 degree black angled screens,
% which are standard for color printing
/sfreq 50 def              % 50 halftone cells per inch
/sproc {dup mul exch dup mul add 1 exch sub} def
```

```
                                % dot-screen spot function
sfreq 75 /sproc load            % 75 degree red (cyan) screen
sfreq 15 /sproc load            % 15 degree green (magenta) screen
sfreq 0 /sproc load             % 0 degree blue (yellow) screen
sfreq 45 /sproc load            % 45 degree gray (black) screen
setcolorscreen
```

ERRORS:
limitcheck, rangecheck, stackunderflow, typecheck

setcolortransfer redproc greenproc blueproc grayproc **setcolortransfer** –

sets the current transfer function parameters for red, green, blue, and gray in the graphics state. Each operand must be a PostScript language procedure that may be called with a number in the range 0.0 to 1.0 (inclusive) on the operand stack and that will return a number in the same range. These procedures map user values of the color components (that is, those specified by **setrgbcolor** and adjusted by **setblackgeneration** and **setundercolorremoval**, or 1.0 minus those specified by **setcmykcolor**) to *device* color components (for halftones, a weighted average of the lightness of pixels in a halftone cell). Only those transfer functions corresponding to color components supported by a device will have an effect on that device's output. For example, *redproc*, *greenproc*, and *blueproc* will have no effect on a black-and-white device, while *grayproc* will have no effect on an RGB device.

The single-color **settransfer** operator takes a single procedure argument whose purpose is to provide correction for a printer's halftoning response. This operator is useful for a variety of effects beyond its original intention as a gray response correction function, but it is useful only in the context of a single output color, as on black-and-white printers. The **setcolortransfer** operator is the logical expansion of **settransfer** to four color components; it takes four function arguments, each similar in purpose to the function argument of **settransfer**, but each function separately controls the response for each of the red (1.0 minus cyan), green (1.0 minus magenta), blue (1.0 minus yellow) and gray (1.0 minus black) components, respectively.

Applying the **settransfer** operator sets all four transfer functions equally. It is the equivalent of the following use of **setcolortransfer**:

```
dup dup dup setcolortransfer
```

EXAMPLE:

```
{} {} {dup mul} {} setcolortransfer
```

This PostScript language code fragment sets device blue as the square of user blue and leaves the other color components unchanged.

Calling **settransfer** with the argument

```
{1 exch sub}
```

to invert an output image is not guaranteed to work if any of the following operators are used in generating the image: **colorimage**, **setcmykcolor**, **setcolortransfer**, **sethsbcolor**, and **setrgbcolor**. In the case of a device with four color components, inversion can be more complicated than merely inverting all of the components.

The use of **setcolortransfer** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
stackunderflow, typecheck

setundercolorremoval proc **setundercolorremoval** –

sets the current undercolor removal function parameter in the graphics state. The *proc* operand must be a PostScript language procedure that can be called with a number in the range 0.0 to 1.0 (inclusive) on the operand stack and that will return a number in the range –1.0 (to *increase* the color components) to +1.0 (to *decrease* the color components). This procedure maps the minimum of the cyn_u, mag_u, and yel_u color components to a value to be subtracted from each of these same components.

For additional information, see Section 3.2.

EXAMPLE:

```
{currentblackgeneration exec .5 mul} setundercolorremoval
```

This PostScript language code fragment sets the undercolor removal to half the value of the black component from black generation.

The use of **setundercolorremoval** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
stackunderflow, typecheck

A CHANGES SINCE LAST PUBLICATION OF THIS DOCUMENT

Changes to *PostScript Language Color Extensions* from the document dated October 25, 1989, are noted in the paragraphs below.

The formulas used in black generation and undercolor removal are more clearly explained.

Minor amplifications and corrections have been made.

The index has been enhanced.

Changes to *PostScript Language Color Extensions* from the document dated October 7, 1988, are noted in the paragraphs below.

The introduction has been reorganized.

Minor amplifications and corrections have been made.

Index

DISPLAY POSTSCRIPT®
SYSTEM

Client Library Reference Manual

ADOBE SYSTEMS
INCORPORATED

Client Library Reference Manual

January 23, 1990

PostScript and Display PostScript are registered trademarks of Adobe Systems Incorporated.

Macintosh is a registered trademark of Apple Computer Incorporated. UNIX is a registered trademark of AT&T Information Systems. X Window System is a trademark of the Massachusetts Institute of Technology.

Written by Amy Davidson.

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 previous version.

- 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

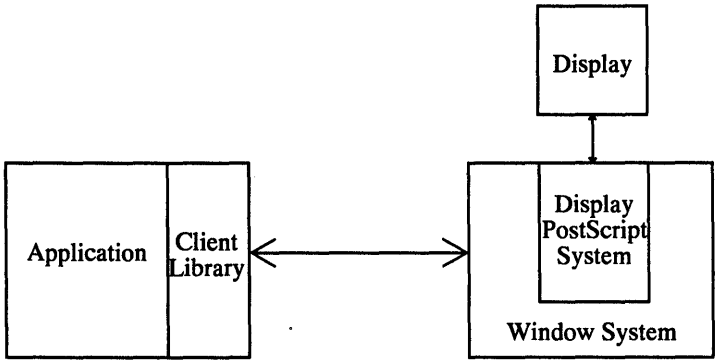
The typographical conventions used in this manual are as follows:

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 programs as ASCII text, as if to a laser printer with a PostScript interpreter. This technique can be used for development and debugging or for displaying PostScript language code imported by the application — for instance, from an EPS file.

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 code 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 con-

texts 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. The 'textProc' should 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 ex-

ercise 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 code 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. Input sent to a child is not passed to its parent. 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 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.)

When the client and the server are separate processes and the buffered code need not be executed immediately, the application can flush the buffers with **flush** rather than synchronizing with the context; synchronizing is described in Section 6.4.

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. To get lines narrower than one point, use fractions of 1 such as 0.3 or 0.25.

8 EXAMPLE APPLICATION PROGRAM

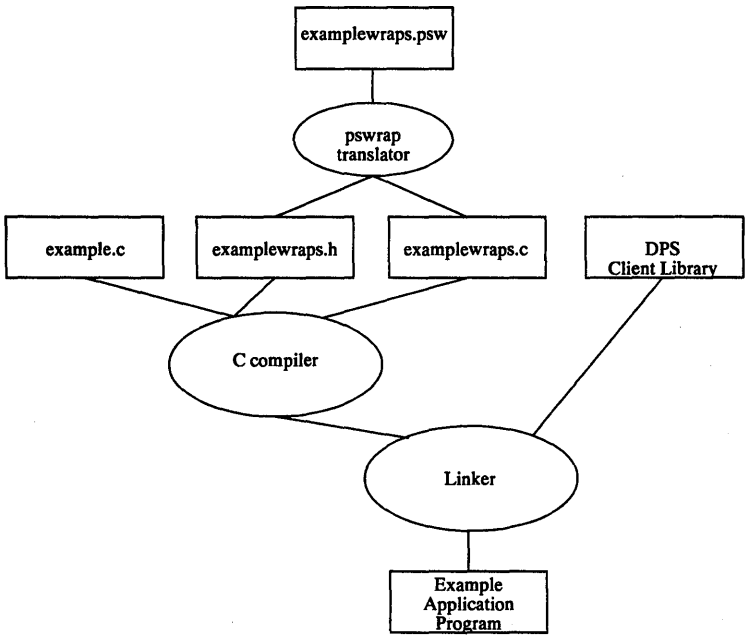
This section provides a simple example of how to use the Display PostScript system through the Client Library. The example:

- 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);
  }

/* Convert the X Window System coordinates at the lower right corner
   of the window to get  the width and height in user space. */

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

/* Locate the box in the middle of the window. */

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

/* Paint an ochre box. */

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 draw-

ing 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 contents of the buffer will *not* be converted according to the context's encoding parameters.

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

An *extended binary object sequence* is required if there are more than 255 top-level objects in the sequence. The extended binary object sequence 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 PUBLICATION OF THIS DOCUMENT

Changes to the *Client Library Reference Manual* from the document dated October 25, 1989, are noted in the paragraphs below.

Input sent to a child context is not passed to its parent.

In calls to *DPSWriteData*, the contents of the buffer will *not* be converted according to the context's encoding parameters.

A few additional minor amplifications and corrections have been made.

Changes to the *Client Library Reference Manual* from the document dated October 7, 1988, are noted in the paragraphs below.

The manual has been completely reorganized and rewritten.

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.

Note: Avoid using exception handling with the X Window System because lower levels of software, such as Xlib, are not prepared to handle exceptions or to have control taken away from them.

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 proce-

dure 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'. When the handler is entered, the first

argument that was passed to *RAISE* get assigned to 'Exception.Code' and the second argument gets assigned to 'Exception.Message'. 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-evaluate-print 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

DISPLAY POSTSCRIPT®
SYSTEM

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

ADOBE SYSTEMS
INCORPORATED

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

January 23, 1990

PostScript and Display PostScript are registered trademarks of Adobe Systems Incorporated.
X Window System is a trademark of the Massachusetts Institute of Technology.
*Helvetica is a trademark of Linotype AG and/or its subsidiaries.

Written by Amy Davidson.

Contents

List of Figures

1 ABOUT THIS MANUAL

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

1.1 DOCUMENTATION

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

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

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

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

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

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

1.2 WHAT THIS MANUAL CONTAINS

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

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

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

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

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

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

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

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

1.3 TYPOGRAPHICAL CONVENTIONS

The typographical conventions used in this manual are as follows:

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

2 ABOUT THE DISPLAY POSTSCRIPT EXTENSION TO X

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

- *The PostScript imaging model* allows the application developer to express graphical displays at a higher level of abstraction than is possible with Xlib. This improves device independence and portability. The integration of the imaging model with X requires consideration of several issues, including coordinate system conversions (see Section 3.3.1), event handling (see Section 4.8), and resource management (see Section 4.7).
- *The PostScript interpreter* allows an application to execute PostScript language code.
- *Wrapped procedures* allow PostScript language programs to be embedded in an application as C-callable procedures.

3 BASIC FACILITIES

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

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

3.1 INITIALIZATION

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

3.2 CREATING A CONTEXT

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

A 'DPSContextRec' is a data structure on the client side that represents all of the state needed by the Client Library to communicate with a context. A pointer of type 'DPSContext' is a handle to this data structure. When the application creates a con-

text in the interpreter, a 'DPSContextRec' is automatically created for use by the client (except for forked contexts; see Section 4.5). The 'DPSContextRec' contains pointers to procedures that implement all of the basic operations that a context can perform.

There are two procedures that create both a context in the server and a 'DPSContextRec' for the client. The first, *XDPSCreateSimpleContext*, uses the default colormap, and is adequate for most applications. The second, *XDPSCreateContext*, is a more general function that allows you to specify colormap information. Other procedures for creating just the 'DPSContextRec' — for contexts that already exist in the server — are covered in Section 4.

3.2.1 Using XDPSCreateSimpleContext

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

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

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

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

- 'plane_mask'
- 'subwindow_mode'
- 'clip_x_origin'
- 'clip_y_origin'
- 'clip_mask'

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

The arguments 'x' and 'y' are offsets that specify where the device space origin is relative to the window origin. To place the device space origin (and thus the user space origin) in the standard location, pass zero for 'x' and the height of the window in pixels for 'y'. See the discussion of coordinate systems in Section 3.3.1.

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

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

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

XDPSCreateSimpleContext may allocate a substantial number of cells in the default colormap. For example, a typical allocation for an 8-plane PseudoColor device is 125 cells for the color cube, representing a 5x5x5 RGB cube. (The gray ramp typically uses the five grays that form the diagonal of the cube.) *XDPSCreateSimpleContext* checks the root window for the RGB_DEFAULT_MAP property. If the property exists, the color cells it specifies are used for the context's color cube. If the property does not exist, color cells are allocated and the property is defined. The allocated cells are typically treated as "read-only retained" so that other DPS/X clients may share the allocated colors. The advantage of using the color allocation facilities provided by *XDPSCreateSimpleContext* is that the application has available a wide range of colors (many more than the number of cells), each with a reasonable rendering, without having to provide for the possibility that colormap allocations may fail. The disadvantage is that a large number of color cells is allocated from the default colormap.

3.2.2 Using XDPSCreateContext

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

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

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

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

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

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

Consider the characteristics of your application when deciding whether to use *XDPSCreateSimpleContext*, with its default allocation of colors, or *XDPSCreateContext*, with 'actual'. An application may allow the end user to define a variety of colors. Such an application — a graphics editor, for example — could use *XDPSCreateSimpleContext*. On the other hand, an application that allows the end user to specify only a few colors — the foreground and background colors of a performance meter, for example — should probably use *XDPSCreateContext* and set 'actual' to the number of colors that can be requested by the end user.

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

3.3 EXECUTION

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

3.3.1 Coordinate Systems

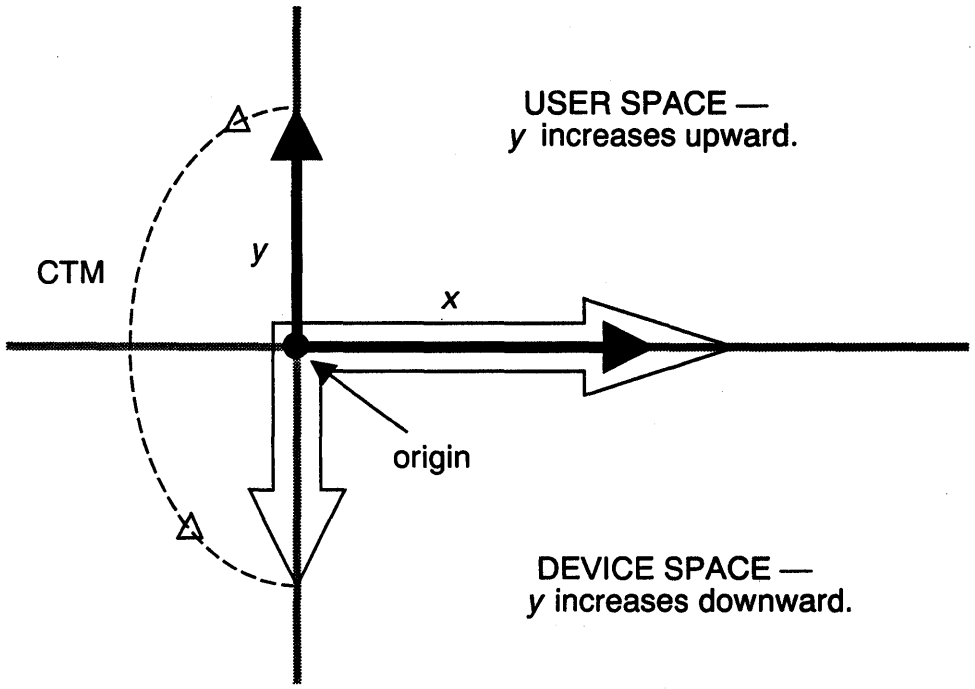
The application must use user space coordinates when communicating with the PostScript interpreter and X coordinates when communicating with other parts of the X Window System. Therefore coordinate conversions may be necessary. This section describes:

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

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

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

Figure 1 *User Space and Device Space*



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

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

As described in *PostScript Language Extensions for the Display PostScript System*, pixel boundaries fall on integer coordinates in device space. A pixel is a half-open region, meaning that it in-

cludes half of its boundary points. For any point *(x, y)* in device space, let i = floor(x) and j = floor(y), where x and y are real numbers and i and j are integers. The pixel that contains this point is the one identified as *(i, j)*, which is equivalent to the X coordinate for that pixel.

To convert user space coordinates to X coordinates:

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

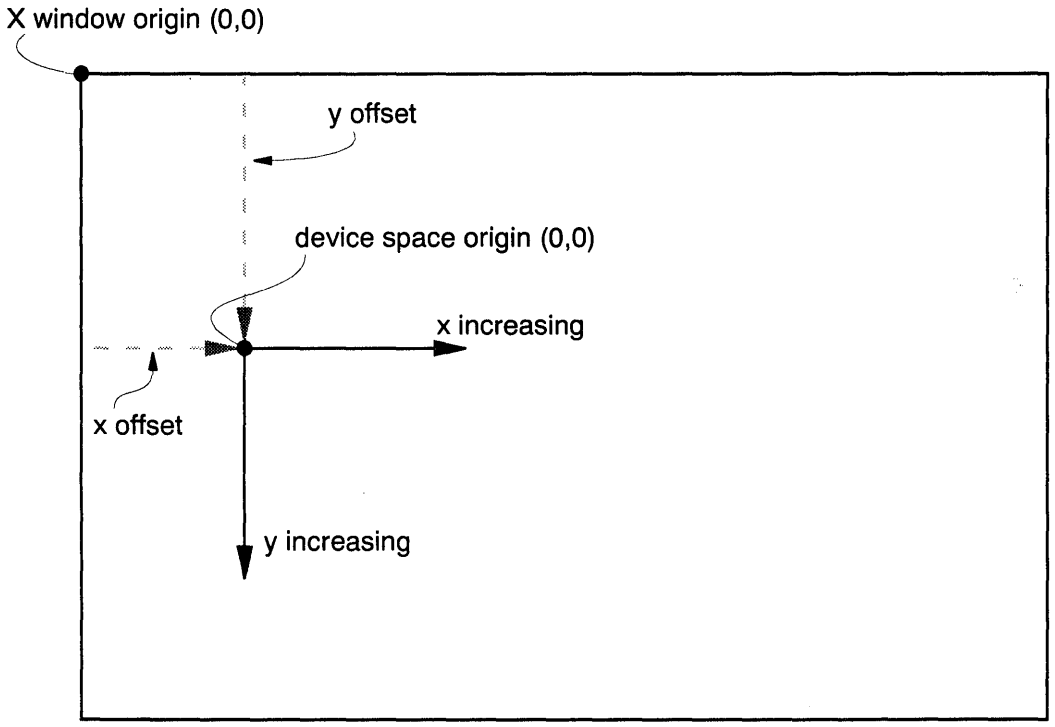
Similarly, to convert X coordinates to user space coordinates:

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

See Section 5.3 for examples of coordinate conversions.

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

Figure 2 *Window Origin and Device Space Origin*



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

Coordinate conversions are required under the following conditions:

- If you use the PostScript imaging model to render graphics using coordinates received from X events, the X coordinates must first be converted into user space coordinates.

For instance, if you allow the user to select a line of text in a text editor, coordinate conversions will be required.

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

Coordinate conversions are not required under the following conditions:

- If you use the PostScript imaging model for output only (rendering text and graphics without user interaction in the display area), no coordinate conversions are required. Simply express coordinates in user space. For example, assuming the default user space, the letter A shown at coordinate 'x=72, y=72' will appear upright 1 inch to the right and 1 inch above the bottom left corner of the window.
- If the only rendering you do in response to X events is with X primitives, you don't have to perform coordinate conversions unless you are altering pixels that were rendered by the PostScript interpreter.

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

Resizing the window may have an effect on the device space origin, and thus the offsets to that origin, depending upon the bit gravity of the window. See Section 3.3.4.

3.3.2 Mixing Display PostScript and X Rendering

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

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

- Copy the pixels that were painted by a PostScript language

program to a pixmap with several *XCopyArea* calls. These pixels will temporarily be obscured by the control points, so they must be preserved.

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

Now suppose a control point is moved. A series of mouse events would be handled as follows:

- Copy the pixels underlying the control point back from the pixmap, effectively erasing the control point at the original location.
- Compute the new position of the control point from the mouse event.
- Copy the pixels at the new location to the pixmap. Call *XFillRectangle* to display the control point at the new location.

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

- Their coordinate systems are different. See Section 3.3.1 for more information on coordinate systems.
- PostScript language programs run asynchronously with respect to other X requests. A PostScript language rendering request is not guaranteed to be complete before a subsequent X request is executed, unless synchronized. See Section 4.9 for more information on synchronization.
- X tends to be pixel and plane oriented; graphics operations that manipulate pixels and planes are necessarily device dependent. The PostScript imaging model deals with abstract graphical representations (paths) and abstract colors. The PostScript interpreter tries to give the best rendering possible for the device. If device independence is important for your application, use X primitives sparingly, preserving device independence as much as possible.

3.3.3 Clipping and Repainting

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

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

- If you're drawing with PostScript language code only, use the clipping mechanism provided by the PostScript imaging model. This is sufficient for nearly all applications.
- If you're also using X primitives and want to clip them as well as draw using DPS/X, use the clipping specified by the X 'GC'.

Exposure events may be handled with a variety of strategies:

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

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

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

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

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

- Create a view clip (see *PostScript Language Extensions for the Display PostScript System*) by converting the coordinates of the list of exposure rectangles to user space coordinates and executing **rectviewclip** with this list.
- Then redisplay all of the text and graphics objects by executing the PostScript language programs that describe them. Only those areas within the the view clip will actually be repainted.

This strategy reduces annoying window flicker, but may do more

work than is necessary since programs describing graphics objects that are completely clipped are executed anyway.

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

- Use a list of rectangles from the exposure events to create a view clip.
- Then, instead of running all of the PostScript language programs, redraw only those graphics objects whose bounding boxes intersect the view clip.

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

3.3.4 Resizing the Window

When the window is resized, the X server moves the window bits according to the bit gravity of the window. If the window is being use for imaging with the PostScript language, the origin of the device space is also moved according to the bit gravity of the window; see Section 3.3.1 for a discussion of coordinate systems. The result of this automatic movement is that the 'x' and 'y' offsets that were specified when the context was created (or that were last changed with the **setXoffset** operator) are changed. The application may need to keep track of these changes.

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

Figure 3 *How Bit Gravity Affects Offsets*

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

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

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

3.3.5 User Object Indices

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

The surrogate objects provided by the Client Library are called *user objects*. A user object is simply an integer ('long int') that represents an actual object (of any type) in the interpreter. To define a new user object, the application must first obtain a *user object index* from the Client Library. The procedure *DPSNewUserObjectIndex* returns a new user object index. The Client Library is the sole allocator of new user object indices in order to guarantee that indices are unique. User object indices are dynamic and should neither be used as arithmetic values — for example, don't add 1 to get the next available index — nor stored in a file or other long-term storage.

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

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

User objects are typically employed under the following circumstances:

- *When graphical objects or other application objects are created dynamically*, such as the user path a graphics editor builds as the user draws an illustration.
- *When a user name should not be employed.* A user object is a convenient and efficient substitute for a dynamically defined user name, which must be passed to a wrap as a string.

See *PostScript Language Extensions for the Display PostScript System* and the *pswrap Reference Manual* for further discussion of user objects.

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

'DPSLastUserObjectIndex', which should be treated as read-only.

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

EXAMPLE

```
/* A wrapped procedure that defines an icon dictionary. */

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

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

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

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

defineps Draw_Icon(userobject icon)
                                % Input Arguments:
                                % icon  user object representing
                                %       and icon dictionary.
                                %       Note: since we are going
```

```
                        %       to execute the object,
                        %       we can declare it as
                        %       userobject to pswrap.

    icon begin              % Gets and execs the user object
                        %   which is a dictionary, begins it.
                        %   Note that there is an implicit
                        %   execuserobject here since icon
                        %   was declared 'userobject'.
    gsave
    icon_x icon_y translate     % Put origin at specified coordinates.

      gsave
      icon_prog             % Draw icon.
      grestore

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

/* C procedure to create and display a new icon */

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

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

3.3.6 Errors and Error Codes

There are two classes of errors that can occur while using DPS/X: protocol errors and context errors.

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

Context errors can arise whenever a 'DPSContext' handle is passed to a DPS/X procedure or wrap. X-specific error codes are discussed in Section 6.1.1 on page 43. See the *Client Library Reference Manual* for a discussion of the standard Display PostScript system error codes.

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

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

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

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

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

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

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

3.4 TERMINATION

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

DPSDestroyContext and *DPSDestroySpace* are provided to allow an application to release these resources without exiting.

This might be needed if, for example, the context and space must be destroyed and recreated from scratch to recover from a PostScript language error. These procedures are described in detail in the *Client Library Reference Manual.* To summarize, *DPSDestroyContext* destroys the context resource in the server and the 'DPSContextRec' in the client. *DPSDestroySpace* destroys the space resource in the server and the 'DPSSpaceRec' in the client as well as all contexts within the space, including their 'DPSContextRec' records.

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

4 ADDITIONAL FACILITIES

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

4.1 IDENTIFIERS

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

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

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

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

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

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

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

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

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

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

4.2 ZOMBIE CONTEXTS

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

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

Any request made to a zombie context will generate a status event of type 'PSZOMBIE'. See Section 4.8 for more information about status events.

4.3 BUFFERS

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

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

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

4.4 ENCODINGS

The *Client Library Reference Manual* discusses the general concept of encodings and conversions. A wrapped procedure always generates a binary object sequence, which is passed to the context for further processing. Typically, the binary object sequence is simply passed to the lowest level of the Client Library to be packaged into a request, without any change to its contents. However, by setting the encoding parameters of the 'DPSContextRec' with the *DPSChangeEncoding* procedure, the binary object sequence can be converted to some other encoding before it is sent or written.

DPS/X supports the conversions shown in Figure 4:

Figure 4 *Encoding Conversions*

Conversion	*Description*
binary object sequence to ASCII	Makes a binary object sequence readable by humans. The output of wrapped procedures may be inspected and analyzed. Also useful for generating page descriptions to be printed. This is the default setting for text contexts. Execution contexts may also be made to convert binary object sequences to ASCII, but there is little purpose in doing this.
binary object sequence to binary-encoded tokens	Binary-encoded token encoding is the most compact encoding for the PostScript language. This conversion is useful for storing code permanently, or for exchanging code with another application. Either a text context or an execution context may perform this conversion, but it is mainly used for text contexts.
binary object sequence with user name indices to binary object sequence with user name strings	This conversion is necessary if the binary object sequence is going to be stored permanently (for example, on a file) or if the binary object sequence is to be used by another client or with a shared context (see Section 4.7). User name indices are created dynamically and are unique only within a single "instance" of the Client Library — for example, in the application's process address space. In this case, user names must be represented by strings if they are to be used outside of the application's process address space.
binary-encoded tokens to ASCII	Binary-encoded tokens read from an external data source such as a file can be converted to ASCII for human inspection, sent to an intepreter, or stored in a page description for printing. After the context's encoding has been set using *DPSChangeEncoding*, buffers of binary-encoded tokens can be read and passed to *DPSWritePostScript* for conversion. Either a text context or an execution context can perform this conversion, but it is used mainly for text contexts.

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

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

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

```
DPSChangeEncoding(context, context->programEncoding, dps_strings);
```

4.5 FORKED CONTEXTS

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

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

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

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

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

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

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

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

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

4.6 MULTIPLE SERVERS

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

In order to support access to multiple servers, DPS/X procedures take a pointer to 'Display' records where appropriate.

4.7 SHARING RESOURCES

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

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

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

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

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

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

- User names in binary encodings of the PostScript language must be sent as strings. This is because the mapping of user name indices are not guaranteed to be unique across clients. The default 'DPSNameEncoding' of the 'DPSContextRec' created for a shared context is 'dps_string'. It cannot be changed to 'dps_indexed'.
- Output from the context, including wrap result values, text, and errors, is sent only to the context's original creator, not to any clients sharing the context. Status events, however, are sent to clients sharing the context, as specified by the status event mask (see Section 4.8).
- When *DPSDestroyContext* or *DPSDestroySpace* is applied to a shared context or space, only the client-side data structures are destroyed. The execution context, the space, and the resources associated with these objects can be destroyed only by the creator.
- If the creator destroys resources, any reference to a

destroyed resource will result in a protocol error, which is sent to the client sharing the resource.

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

4.8 STATUS EVENTS

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

Status events can be used to perform the following tasks:

- Send code, using flow control, from the application to a context.
- Control the suspension and resumption of execution.
- Synchronize PostScript interpreter execution with X rendering requests.
- Monitor a context to determine whether it is runaway, "wedged" (stuck), or zombie.

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

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

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

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

'PSNEEDSINPUT'
: The context is waiting for code to execute, a condition commonly known as "blocked on input."

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

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

Except for 'PSSTATUSERROR', these status events may be disabled (see below).

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

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

- 'PSRUNNINGMASK'
- 'PSNEEDSINPUTMASK'
- 'PSZOMBIEMASK'
- 'PSFROZENMASK'

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

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

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

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

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

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

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

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

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

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

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

See Section 4.9 for details on how the 'PSFROZEN' status event can be used.

4.9 SYNCHRONIZATION

As discussed in Section 3.3.2, X rendering primitives and PostScript language execution may, in most cases, be intermixed freely. However, in some situations PostScript language execution needs to be synchronized with X.

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

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

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

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

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

4.9.1 Waiting

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

- The application has complete control of the code to be executed. That is, it uses wrapped procedures, single-operator procedures, or dynamically generated code fragments such as user path descriptions. No code comes from external sources such as end-user input.
- The application reads external files with a known structure that can be parsed and understood, such as PostScript language page descriptions that are compliant with *Adobe Systems Document Structuring Conventions*.

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

4.9.2 Freezing

Freezing a context is appropriate if the application has insufficient knowledge of the completeness of the PostScript language code to be executed. This can happen if an end user is allowed to enter arbitrary PostScript language programs (for instance, in an interactive interpreter executive) or if an input file lacks a well-defined structure.

In this case, the input must contain an executable name that the application has defined. For example, the **showpage** operator terminates each page in a page description file. The application can take advantage of this, as shown in the following example.

Example: The application has defined **showpage** to execute an operator that will notify the application that the page is done. The **clientsync** operator fulfils this function:

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

When **clientsync** is executed, the context is put into the 'PSFROZEN' state, and a 'PSFROZEN' event is generated. The application must have enabled the 'PSFROZEN' event and registered a handler for that context; see Section 4.8 for more information on status events. The handler may then set a flag indicating that the image in the pixmap is complete. The next time the application goes around its main loop, it can test the flag and call *XCopyArea*.

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

5 PROGRAMMING TIPS

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

5.1 DON'T USE XIFEVENT

Don't call *XIfEvent* in your application. This routine will cause events that were generated and queued by an execution context to be processed repeatedly (once for each call to *XIfEvent*) without being dequeued. This may result in wrap results or text output being erroneously duplicated or may cause false status events to be reported. Use *XCheckIfEvent* instead.

This restriction may not apply to future implementations of Xlib.

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

5.2 INCLUDE FILES

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

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

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

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

5.3 COORDINATE CONVERSIONS

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

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

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

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

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

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

To convert an X coordinate into a user space coordinate, perform the following calculations:

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

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

To convert a user space coordinate into an X coordinate, perform the following calculations:

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

The equations listed above have the following limitations:

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

5.4 FONTS

The **filenameforall** operator can be used to obtain a list of the fonts available to the server. See *PostScript Language Extensions for the Display PostScript System* for a description of **filenameforall**. Use the pattern '(%font%*)' to generate a list of fonts. The font file names may be sent back as ASCII text and processed with a customized text handler, or they may be stored in an array and then accessed one at a time by calling a wrapped procedure.

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

5.5 PORTABILITY ISSUES

Portability issues may arise under any of the following situations:

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

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

5.5.1 Color

Use PostScript operators such as **setrgbcolor** rather than X primitives to draw with color. The PostScript interpreter will provide the best rendering possible for the device. The Display PostScript system can produce a variety of halftone patterns representing gray values or colors, so that one color can be seen against the background of another color even on a monochrome device. Contrast this with the rendering facilities of the X Window System, where a request for any color other than white on a monochrome device will give you black.

DPS/X color rendering is device independent. Here's how DPS/X handles color requests:

- On a monochrome device, you'll get a dithered (halftone) pattern of black and white pixels. For example, if you ask for red by specifying '1 0 0 setrgbcolor', you'll get some halftone gray pattern composed of black and white pixels; this pattern will be distinct from other "colors".
- On a grayscale device you'll get a halftone pattern using gray levels; this offers greater distinction among "colors."
- On a color device (4-plane, 8-plane, and so on), you'll get the requested color if it's one of those predefined for the context; otherwise you'll get a dithered pattern of RGB pixels that approximates the color.
- If you've allocated solid colors beyond those predefined for the context, you'll get a non-dithered color just as you would with X (subject to the same restrictions).
- A color request will never simply fail.

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

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

- On a color device, you'll get the color you requested only if there's space in the colormap or the device is a TrueColor device.
- A color request can fail, and there's no recourse except to try requesting a different color.

5.5.2 Resolution

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

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

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

5.5.3 Fonts

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

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

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

6 X-SPECIFIC DATA AND PROCEDURES

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

6.1 DATA STRUCTURES

Data structures defined in the *dpsXclient.h* header file are described below.

6.1.1 Extended Error Codes

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

'dps_err_invalidAccess'
: An attempt was made to receive output from a context created by another client. Contexts send their output only to the original creator. If the application tries to get output from a context created by another client — for example, by calling a wrap that returns a result — this error is reported.

'dps_err_encodingCheck'
: An attempt was made to change name or program encoding to unacceptable values. This error can occur when changing name encoding for a context created by another client or a context created in a space that was created by another client. Such contexts must have string name encoding ('dps_strings').

'dps_err_closedDisplay'
: An attempt was made to send PostScript language code to a context whose 'Display' is closed.

'dps_err_deadContext'
: An attempt was made to get output from a zombie context (a context that has died in the server but still has its X resources active).

6.1.2 Status Event Masks

The status event types supported in DPS/X are shown in Figure 5. The first column shows the status event type that is reported by the server. The second column shows the associated single-bit status mask values that can be combined with logical OR to set a context's status mask. The third column describes the status event.

Figure 5 *Status Events*

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

For more information on status events, see Section 4.8.

6.1.3 Types and Global Variables

DPSLastUserObjectIndex

```
long int DPSLastUserObjectIndex;
```

'DPSLastUserObjectIndex' is a global variable containing the last user object index assigned for this application. This variable should be treated as read-only. For more information about user object indices, see *DPSNewUserObjectIndex* and Section 3.3.5.

XDPSStatusProc

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

This is a procedure type for defining the call-back procedure that handles status events for the client. The procedure will be called with two parameters: the context it was registered with and the status code derived from the event. For more information about status events, see *XDPSRegisterStatusProc* and Section 6.1.2.

6.2 PROCEDURES

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

DPSChangeEncoding

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

DPSChangeEncoding changes one or both of the context's encoding parameters. See the *Client Library Reference Manual* for definitions of 'DPSNameEncoding' and 'DPSProgramEncoding'. Supported conversions are described in Figure 4 on page 28.

DPSContextFromContextID

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

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

The application must call this procedure before attempting to communicate with a forked context. *DPSContextFromContextID* creates the client-side data structures for the context and associates them with the server-side structures previously created by the **fork** operator. 'cid' is the context identifier (of type 'long int') that is assigned to the forked context by the PostScript interpreter. 'ctxt' is the handle of the context that created the forked context; its 'DPSContextRec' will be used as a model for the 'DPSContextRec' of the forked context, as described below.

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

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

DPSCreateTextContext

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

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

DPSDefaultTextBackstop

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

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

DPSDestroyContext

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

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

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

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

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

DPSDestroySpace

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

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

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

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

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

DPSNewUserObjectIndex

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

DPSNewUserObjectIndex returns a new user object index. The Client Library is the sole allocator of new user object indices. The application should not attempt to compute them from a previously obtained index. Because user object indices are dynamic, they should not be used as numeric values for computation or saved in long-term storage such as a file. See Section 3.3.5 for more information on user object indices.

XDPSContextFromSharedID

```
DPSContext XDPSContextFromSharedID(dpy, cid, textProc, errorProc);
  Display *dpy;
  ContextPSID cid;
  DPSTextProc textProc;
  DPSErrorProc errorProc;
```

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

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

XDPSContextFromXID

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

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

XDPSCreateContext

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

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

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

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

The following restrictions apply:

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

See Section 3.2 for additional information on creating a context.

XDPSCreateSimpleContext

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

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

The procedure creates a context associated with 'dpy', 'drawable' and 'gc'. 'x' and 'y' are offsets from the 'drawable' origin to the PostScript device space origin in pixels.

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

The following restrictions apply:

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

See Section 3.2 for additional information on creating a context.

XDPSFindContext

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

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

XDPSGetContextStatus

```
int XDPSGetContextStatus(ctxt);
  DPSContext ctxt;
```

XDPSGetContextStatus returns the status of 'ctxt'. This procedure does not alter the mask established for 'ctxt' by *XDPSSetStatusMask*. For information on status events, see Sections 4.8 and 6.1.2.

XDPSRegisterStatusProc

```
void XDPSRegisterStatusProc(ctxt, proc);
  DPSContext ctxt;
  XDPSStatusProc proc;
```

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

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

XDPSSetStatusMask

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

XDPSSetStatusMask sets the status mask for the context:

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

A given status event type may be set in only one of the three status masks. If an event is set in more than one mask, a protocol error ('Value') is generated and the context is left unchanged. For more information on status events, see Sections 4.8 and 6.1.2.

XDPSSpaceFromSharedID

```
DPSSpace XDPSSpaceFromSharedID(dpy, sxid);
  Display *dpy;
  SpaceXID sxid;
```

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

XDPSSpaceFromXID

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

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

XDPSUnfreezeContext

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

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

XDPSXIDFromContext

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

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

XDPSXIDFromSpace

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

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

7 X-SPECIFIC POSTSCRIPT OPERATORS

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

operator $operand_1$ $operand_2$ ... $operand_n$ **operator** $result_1$ ... $result_m$

Detailed explanation of the operator.

ERRORS:
A list of the errors that this operator might execute.

At the head of an operator description, $operand_1$ through $operand_n$ are the operands that the operator requires, with $operand_n$ being the topmost element on the operand stack. The operator pops these objects from the operand stack and consumes them. After executing, the operator leaves the objects $result_1$ through $result_m$ on the stack, with $result_m$ being the topmost element.

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

Error conditions include the following:

rangecheck Invalid match: Either the 'drawable' and 'gc' have different depths or they don't have a 'Visual' that matches the colormap associated with the context.

stackunderflow Not enough operands on the operand stack.

typecheck Invalid X resource ID.

undefined The device associated with the context is not a display device.

clientsync – **clientsync** –

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

For an example of the use of **clientsync**, see Section 4.9.2.

currentXgcdrawable – **currentXgcdrawable** gc drawable x y

The **currentXgcdrawable** operator returns the X 'gc', 'drawable', and offset from the origin of the 'drawable' to the device space origin for the current context. Results returned by this operator can be input to **setXgcdrawable**.

ERRORS:
undefined

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

The **currentXgcdrawablecolor** operator is similar to the **currentXgcdrawable** operator, except that it also returns an array of 12 integers describing the color cube, gray ramp, and other color variables used for the context. The 'colorinfo' array, described in Figure 6, has the following form:

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

Figure 6 *The 'colorinfo' Array*

Value	*Description*
'maxgrays'	Maximum number of gray values. Equivalent to 'red_max' field of 'XStandardColormap' for 'GrayScale' colormaps.
'graymult'	Scale factor to compute gray pixel. Equivalent to 'red_mult' field of 'XStandardColormap' for 'GrayScale' colormaps.
'firstgray'	First gray pixel value. Equivalent to 'base_pixel' field of 'XStandardColormap' for 'GrayScale' colormaps.
'maxred'	Maximum number of red values. Equivalent to 'red_max' field of 'XStandardColormap'.
'redmult'	Scale factor to compute color pixel. Equivalent to 'red_mult' field of 'XStandardColormap'.
'maxgreen'	Maximum number of green values. Equivalent to 'green_max' field of 'XStandardColormap'.
'greenmult'	Scale factor to compute color pixel. Equivalent to 'green_mult' field of 'XStandardColormap'.
'maxblue'	Maximum number of blue values. Equivalent to 'blue_max' field of 'XStandardColormap'.
'bluemult'	Scale factor to compute color pixel. Equivalent to 'blue_mult' field of 'XStandardColormap'.
'firstcolor'	First color pixel value. Equivalent to 'base_pixel' field of 'XStandardColormap'.
'colormap'	The colormap that these pixel values are allocated in.
'actual'	The upper limit of additional RGB colors, as in the 'actual' argument to *XDPSCreateContext.*

ERRORS:
undefined

currentXoffset – **currentXoffset** x y

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

ERRORS:
undefined

setXgcdrawable gc drawable x y **setXgcdrawable** –

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

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

ERRORS:
rangecheck stackunderflow typecheck undefined

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

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

ERRORS:
rangecheck stackunderflow typecheck undefined

setXoffset x y **setXoffset** –

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

ERRORS:
stackunderflow undefined

setXrgbactual red green blue **setXrgbactual** bool

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

Executing **setXrgbactual** is a way of ensuring that the color you request is actually allocated, not dithered. Colors specified by **setXrgbactual** do not count against the number of 'actual' colors that are allocated automatically; see Section 3.2.2. **setXrgbactual** may be called even if the context was created with 'actual' set to zero.

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

ERRORS:
stackunderflow typecheck undefined

A CHANGES SINCE LAST PUBLICATION OF THIS DOCUMENT

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

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

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

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

Numerous additional amplifications and corrections have been made.

B ADVANCED EXCEPTION HANDLING

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

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

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

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

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

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

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

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

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

B.1 DEFERRED ERROR HANDLING EXAMPLE

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

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

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

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

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

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

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

/* Forward declarations */

void Error();
int ErrorCallbackProc();

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

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

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

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

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

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

}

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

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

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

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

B.2 ERROR HANDLER INTERFACE

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

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

```
/* errorproc.h */

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

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

/* Queue of deferred error entries */

extern ERRQueueEntry ERRQueueHead;

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

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

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

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

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

The header file is described in the paragraphs that follow.

'ERRQueueEntry' is a structure that contains information about errors of type 'DPSErrorCode'. This structure serves as a header for a potentially larger structure; some errors require additional information for optimal processing. In those cases, the 'arg1' element points to the additional information, which is appended

to the entry header. See *DPSErrorProc* in the *Client Library Reference Manual* for information on the structure of such additional information.

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

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

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

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

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

ERRProcessErrors is called by the application when it is ready to handle any pending errors queued by *ERRDeferredErrorProc*. It removes as many pending error entries from the error queue as is allowed by the call-back procedure; the actual processing of each error entry is left to the call-back procedure passed as an argument to *ERRProcessErrors*. An argument to be passed to the call-back procedure is also provided, allowing the application to specify the disposition of an error without having to manage the error entry queue. If *ERRProcessErrors* is called with a NULL

call-back procedure, *ERRPrintErrorMsg* is substituted. In other words, the default action is to print an error message. If the call-back procedure returns *false*, *ERRProcessErrors* returns immediately to the caller, potentially leaving unprocessed entries still on the error queue. *ERRProcessErrors* returns *true* if any errors were processed; it returns *false* if no error entries were found on the queue.

B.3 ERROR HANDLER IMPLEMENTATION

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

```
/* errprocsample.c */

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

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

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

ERRQueueEntry ERRQueueHead = { NULL };

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

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

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

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

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

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

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

/* Enqueue the new entry */

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

} /* ERRDeferredErrorProc */

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

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

DPSTextProc textProc = DPSGetCurrentTextBackstop();

if (!textProc)
  return(1);

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

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

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

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

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

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

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

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

 } /* ERRPrintErrorMsg */

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

 if (proc == NULL)
   proc = ERRPrintErrorMsg;

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

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

Index

DISPLAY POSTSCRIPT®
SYSTEM

pswrap
Reference Manual

ADOBE SYSTEMS
INCORPORATED

pswrap Reference Manual

January 23, 1990

Written by Amy Davidson.

Contents

1 ABOUT THIS MANUAL

This manual is the programmer's reference manual for the *pswrap* translator. It tells you how to use *pswrap* to create C-callable procedures that contain PostScript® language code.

Section 2 introduces the *pswrap* translator.

Section 3 tells you how to run *pswrap* and documents the options in the *pswrap* command line.

Section 4 tells you how to write wrap definitions for *pswrap*.

Section 5 tells you how to declare input arguments.

Section 6 tells you how to declare output arguments.

Appendix A lists error messages from the *pswrap* translator.

Appendix B describes the syntax used in wrap definitions.

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

This manual does not provide information on the PostScript language, the Display PostScript® system, or the Client Library (the programming interface to the Display PostScript system). For more information regarding these topics, see the following manuals:

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

2 ABOUT *PSWRAP*

The *pswrap* translator provides a natural way for an application developer or toolkit implementor to compose a package of C-callable procedures that send PostScript language code to the PostScript interpreter. These C-callable procedures are known as

wrapped procedures or *wraps*. (A *wrap* is a procedure that consists of a C declaration with a PostScript language body. A *wrap body* is the PostScript language program fragment in a wrap.)

Here's how *pswrap* fits into the Display PostScript system:

- You write the PostScript language programs required by your application, using the *pswrap* syntax described in this manual to define a C-callable procedure and specify input and output arguments.
- You run *pswrap* to translate these PostScript language programs into wrapped procedures.
- You compile and link these wraps with the application program.
- When a wrap is called by the application, it sends encoded PostScript language to the PostScript interpreter and receives the values returned by the interpreter.

A *pswrap* source file associates PostScript language code with declarations of C procedures; *pswrap* writes C source code for the declared procedures, in effect wrapping C code around the PostScript language code. Wrapped procedures can take input and output arguments:

- Input arguments are values a wrap sends to the PostScript interpreter as PostScript objects.
- Output arguments are pointers to variables where the wrap stores values returned by the PostScript interpreter.

Wraps are the most efficient way for an application to communicate with the PostScript interpreter.

3 USING *PSWRAP*

The form of the *pswrap* command line (UNIX- and C-specific) is:

```
pswrap [-ar] [-o outputCfile] [-h outputHfile] [-s maxstring] [inputFile]
```

where square brackets [ ] indicate optional items.

3.1 COMMAND-LINE OPTIONS

The *pswrap* command-line options are described below.

---

inputFile — A file that contains one or more wrap definitions. *pswrap* transforms the definitions in *inputFile* into C procedure definitions. If no input file is specified, the standard input (which can be redirected from a file or pipe) is used. The input file can include text other than procedure definitions. *pswrap* converts procedure definitions to C procedures and passes the other text through unchanged; therefore, it is possible to intersperse C-language source code with wrap definitions in the input file.

Note: Although C code is allowed in a *pswrap* input file, it is *not* allowed within a wrap body. In particular, C '#define' macros cannot be used inside a wrap.

-a — Generates ANSI C procedure prototypes for procedure declarations in *outputCfile* and, optionally, *outputHfile.* (See the **-h** option.) The **-a** option allows compilers that recognize the ANSI C Standard to do more complete typechecking of parameters. To save space, the **-a** option also causes *pswrap* to generate 'const' declarations.

Note: ANSI C procedure prototype syntax is not recognized by most non-ANSI C compilers, including many compilers based on the *Portable C Compiler.* Use the **-a** option only in conjunction with a compiler that conforms to the ANSI C Standard.

-h *outputHFile* — Generates a header file that contains 'extern' declarations for nonstatic wraps. This file may be used in '#include' statements in modules that use wraps. If the **-a** option is specified, the declarations in the header file are ANSI C procedure prototypes. If the **-h** option is omitted, a header file is not produced.

-o *outputCFile* Specifies the file to which the generated wraps and passed-through text are written. If omitted, the standard output is used. If the **-a** option is also specified, the procedure declarations generated by *pswrap* are in ANSI C procedure prototype syntax.

-r Generates reentrant code for wraps that are shared by more than one process (as in shared libraries). Since the **-r** options causes *pswrap* to generate extra code, use it only when necessary.

-s *maxstring* Sets the maximum allowable length of a PostScript string object or PostScript hex string object in the wrap body input. A syntax error will be reported if a string is not terminated with ')' or '>' within *maxstring* characters. *maxstring* cannot be set lower than 80. The default is 200.

3.2 '#LINE' DIRECTIVES

Since the C source code generated for wrapped procedures usually contains more lines than the input wrap body does, *pswrap* inserts '#line' directives into the output wrap. These directives record input line numbers in the output wrap source file so that a source-code debugger can display them. Since a debugger displays C source code, not the PostScript language code in the wrap body, *pswrap* inserts **#line** directives for both the *inputFile* and the *outputCfile*.

Note: Unless both the input and output files are named on the command line, the '#line' directives will be incomplete; in the latter case, they will lack the name of the C source file *pswrap* produces. Use of the standard input and standard output streams is discouraged for this reason.

---

pswrap writes diagnostic output to the standard error if there are errors in the command line or in the input. If *pswrap* encounters errors during processing, it reports the error and exits with a non-zero termination status.

4 WRITING A WRAP

Here is a sample wrap definition. It declares the *PSWGrayCircle* procedure, which creates a solid gray circle with a radius of 5.0 centered at (10.0, 10.0):

Wrap definition:

```
defineps PSWGrayCircle()
    newpath
    10.0 10.0 5.0 0.0 360.0 arc
    closepath
    0.5 setgray
    fill
endps
```

Procedure call:

```
PSWGrayCircle();
```

PostScript language code equivalent:

```
newpath
10.0 10.0 5.0 0.0 360.0 arc
closepath
0.5 setgray
fill
```

The rules for defining a wrapped procedure are given in the next section.

4.1 THE WRAP DEFINITION

Each wrap definition consists of four parts:

'defineps' Begins the definition; must appear at the beginning of a line, without any preceding spaces or tabs.

Declaration of the C-callable procedure
The name of the procedure followed by a list in parentheses of the arguments it takes. The arguments are optional; the parentheses are required even for a procedure without arguments. (Note that wraps do not return values; they are declared 'void'.)

Wrap body	PostScript language program fragment. This fragment is sent to the PostScript interpreter. It includes a series of PostScript operators and operands separated by spaces, tabs, and newline characters.
'endps'	Ends the definition. Like 'defineps', 'endps' must appear at the very beginning of a line.

By default, wrap definitions introduce external (that is, global) names that can be used outside the file in which the definition appears. To introduce private (local) procedures, declare the wrapped procedure as static. For example, the *PSWGrayCircle* wrap shown above can be made static by substituting the following statement for the first line:

```
defineps static PSWGrayCircle()
```

Note: It is helpful for the application to use a naming convention for wraps that identifies them as such; for example, *PSWDrawBox*, *PSWShowTitle*, *PSWDrawSlider*, and so on.

4.2 COMMENTS

C comments can appear anywhere outside a wrap definition. PostScript language comments can appear anywhere after the procedure is declared and before the definition ends. *pswrap* strips PostScript language comments from the wrap body. Comments cannot appear within PostScript string objects:

```
/*This is a C comment*/
defineps PSWNoComment()
   (/*This is not a comment*/)show
   (%Nor is this.)length
   %This is a PS comment
endps
```

Wraps cannot be used to send PostScript language comments that contain structural information (%% and %!). Use another Client Library facility such as *DPSWriteData* for this purpose.

4.3 THE WRAP BODY

pswrap accepts any valid PostScript language code as specified in the *PostScript Language Reference Manual*, *PostScript Language Extensions for the Display PostScript System*, and *PostScript Language Color Extensions*. If the PostScript language code in a wrap body includes any of the following symbols, the opening and closing marks must balance.

'{ }'	Braces (to delimit a procedure)
'[ ]'	Square brackets (to define an array)
'( )'	Parentheses (to enclose a string)
'< >'	Angle brackets (to mark a hexdecimal string)

Parentheses *within* a string body must balance or be quoted with '\' according to standard PostScript language syntax.

Note: *pswrap* does not check a wrap definition for valid or sensible PostScript language code.

---

pswrap attempts to wrap whatever it encounters. Everything between the closing parenthesis of the procedure declaration and the end of the wrap definition is assumed to be an element of the PostScript language unless it is part of a comment or matches one of the wrap arguments.

Note: *pswrap* does not support the // PostScript language syntax for immediately evaluated names. See the *PostScript Language Reference Manual* for more information about immediately evaluated names.

4.4 ARGUMENTS

Argument names in the procedure header are declared using C types. For instance, the following example declares two variables, 'x' and 'y', of type 'long int'.

```
defineps PSWMyFunc(long int x,y)
```

There can be any number of input and output arguments. Input arguments must be listed before output arguments in the wrap header. Precede the output arguments, if any, with a vertical bar '|'. Separate arguments of the same type with a comma. Separate arguments of differing types with a semicolon. A semicolon is optional before a vertical bar or a right parenthesis; the two examples below are equivalent:

```
defineps PSWNewFunc(float x,y; int a | int *i)
defineps PSWNewFunc(float x,y; int a; | int *i;)
```

4.5 INPUT ARGUMENTS

Input arguments describe values that the wrap converts to encoded PostScript objects at run time. When an element within the wrap body matches an input argument, the value that was passed to the wrap replaces the element in the wrap body. Input arguments represent placeholders for values in the wrap body. They are not PostScript language variables (names). Think of them as macro definitions that are substituted at run time.

For example, the *PSWGrayCircle* procedure defined on page 5 can be made more useful by providing input arguments for the radius and center coordinates:

Wrap definition:

```
defineps PSWGrayCircle(float x,y, radius)
   newpath
   x y radius 0.0 360.0 arc
   closepath
   0.5 setgray
   fill
endps
```

Procedure call:

```
PSWGrayCircle(25.4, 17.7, 40.0);
```

PostScript language code equivalent:

```
newpath
25.4 17.7 40.0 0.0 360.0 arc
closepath
0.5 setgray
fill
```

The value of input argument 'x' replaces every occurrence of 'x' in the wrap body. This version of *PSWGrayCircle* draws a circle of a specified size at a specified location.

4.6 OUTPUT ARGUMENTS

Output arguments describe values that PostScript operators return. For example, the standard PostScript operator **currentgray** returns the gray-level setting in the current graphics state. PostScript operators return values by placing them on the top of the operand stack. To return the value to the application, place the name of the output argument in the wrap body at a time when the desired value is on the top of the operand stack. For example, the following wrap gets the value returned by **currentgray**:

Wrap definition:

```
defineps PSWGetGray( | float *level)
   currentgray level
endps
```

Procedure call:

```
float aLevel;
PSWGetGray(&aLevel);
```

PostScript language code equivalent:

```
currentgray
% Pop current gray level off operand stack
%   and store in aLevel. [1]
```

When an element within a wrap body matches an output argument in this way, *pswrap* replaces the output argument with code that returns the top object on the operand stack. For every output argument, the wrap will perform the following operations:

- Pop an object off the operand stack.
- Send it to the application.
- Convert it to the correct C data type.
- Store it at the place designated by the output argument.

Each output argument must be declared as a pointer to the location where the procedure stores the returned value. To get a 'long int' back from a *pswrap*-generated procedure, declare the output argument as 'long int *', as in the following example:

[1]See the "Runtime Support" section of the *Client Library Reference Manual* for a discussion of how *pswrap* uses the **printobject** operator to return results.

Wrap definition:

```
defineps PSWCountExecStack( | long int *n)
   countexecstack n
endps
```

Procedure call:

```
long int aNumber;
PSWCountExecStack(&aNumber);
```

PostScript language code equivalent:

```
countexecstack
% Pop count of objects on exec stack
%   and return in aNumber.
```

To receive information back from the PostScript interpreter, use only the syntax for output arguments described here. Do not use operators that write to the standard output (such as =, ==, **print**, or **pstack**). These operators send ASCII strings to the application that *pswrap*-generated procedures cannot handle.

Warning: For an operator that returns results, the operator description shows the order in which results are placed on the operand stack, reading from left to right. (See the "Operators" chapters of the *PostScript Language Reference Manual* and *PostScript Language Extensions for the Display PostScript System.*) When you specify a result value in a wrap body, the result is taken from the top of the operand stack. Therefore the order in which wrap results are stated must be the reverse of their order in the operator description.

For instance, the PostScript operator description for **currentpoint** returns two values, *x* and *y*:

```
– currentpoint  x y
```

The corresponding wrap definition must be written:

```
defineps PSWcurrentpoint (| float *x, *y)
        currentpoint y x        % Note: y before x.
endps
```

Sections 5 and 6 discuss the details of input and output arguments, respectively.

5 DECLARING INPUT ARGUMENTS

This section defines the data types allowed as input arguments in a wrap. In the following list, square brackets indicate optional elements:

- 'DPSContext'. If the wrap specifies a context, it must appear as the first input argument. ('DPSContext' is a handle to the context record; see the *Client Library Reference Manual* for more information.)
- One of the following *pswrap data types* (equivalent to C data types except for 'boolean' and 'userobject', which are special to *pswrap*):

'boolean'	'userobject'
'int'	'unsigned [int]'
'short [int]'	'unsigned short [int]'
'long [int]'	'unsigned long [int]'
'float'	'double'

- An array of a *pswrap* data type
- A character string ('char *' or 'unsigned char *')
- A character array ('char [ ]' or 'unsigned char [ ]') (The square brackets are part of C syntax.)

A string ('char *') passed as input may not be more than 65,535 characters. An array may not contain more than 65,535 elements.

5.1 SENDING BOOLEAN VALUES

If an input argument is declared as 'boolean', the wrap expects to be passed a variable of type 'int'. If the variable has a value of zero, it is translated to a PostScript boolean object with the value *false*. Otherwise it is translated to a PostScript boolean object with the value *true*.

5.2 SENDING USER OBJECT VALUES

Input parameters declared as type 'userobject' should be passed as type 'long int'. The value of a 'userobject' argument is an index into the **UserObjects** array. See *PostScript Language Extensions for the Display PostScript System* for a description of user objects.

When *pswrap* encounters an argument of type **userobject**, it will generate PostScript language code to obtain the object associated with the index. For example:

Wrap definition:

```
defineps PSWAccessUserObject(userobject x)
   x
endps
```

Procedure call:

```
long int aUserObject;
   ...
/* assume aUserObject = 6 */
PSWAccessUserObject(aUserObject);
```

PostScript language code equivalent:

```
6 execuserobject
```

If the object is executable, it will be executed; if it's not executable, it will be pushed on the operand stack.

If you want to pass the index of a user object without having it translated by *pswrap* as described above, declare the argument to be of type 'long int' rather than type 'userobject'. Here is an example of a wrap that defines a user object:

Wrap definition:

```
defineps PSWDefUserObject(long int d)
   d 10 dict defineuserobject
endps
```

Procedure call:

```
long int anIndex;
   ...
/*  assume anIndex = 12 */
PSWDefUserObject(anIndex);
```

PostScript language code equivalent:

```
12 10 dict defineuserobject
```

5.3 SENDING NUMBERS

An input argument declared as one of the 'int' types is converted to a 32-bit PostScript integer object before it is sent to the interpreter. A 'float' or 'double' input argument is converted to a 32-bit PostScript real object. These conversions follow the usual C conversion rules.[2]

Note: Since the PostScript language does not support unsigned integers, unsigned integer input arguments are converted to signed integers in the body of the wrap.

5.4 SENDING CHARACTERS

An input argument composed of characters is treated as a PostScript name object or string object. The argument can be declared as a character string or as a character array.

pswrap expects arguments that are passed to it as character strings ('char *' or 'unsigned char *') to be null terminated ('\0'). Character arrays are not null terminated. The number of elements in the array must be specified as an integer constant or as an input argument of type 'int'. In either case, the integer value must be positive. See Section 5.5 for an example of this rule.

5.4.1 Text Arguments

An input argument declared as a character string or character array is converted to a single PostScript name object or string object. Such an argument is referred to as a *text argument.*

The PostScript interpreter does not process the characters of text arguments. It assumes that any escape sequences ('\n', '\t', and so on) have been processed before the wrap is called.

To make *pswrap* treat a text argument as a PostScript literal

[2]See *The C Programming Language*, R. W. Kernighan and D. M. Ritchie (Englewood Cliffs, NJ: Prentice-Hall, 1978) or *C: A Reference Manual*, S. P. Harbison and G. L. Steele, Jr. (Englewood Cliffs, NJ: Prentice-Hall, 1984).

name object, precede it with a slash, as in the *PSWReadyFont* wrap definition below. (Only names and text arguments can be preceded by a slash.)

Wrap definition:

```
defineps PSWReadyFont( char *fontname; int size )
   /fontname size selectfont
endps
```

Procedure call:

```
PSWReadyFont("Sonata", 6);
```

PostScript language code equivalent:

```
/Sonata 6 selectfont
```

To make *pswrap* treat a text argument as a PostScript string object, enclose it within parentheses. The *PSWPutString* wrap definition below shows a text argument, '(str)':

Wrap definition:

```
defineps PSWPutString(char *str; float x, y)
   x y moveto
   (str) show
endps
```

Procedure call:

```
PSWPutString("Hello World", 72.0, 72.0);
```

PostScript language code equivalent:

```
72.0 72.0 moveto
(Hello World) show
```

Note: Text arguments are recognized within parentheses only if they appear alone, without any surrounding whitespace or additional elements. In the following wrap definition, only the first string is replaced with the value of the text argument. The second and third strings are sent unchanged to the interpreter.

```
defineps PSWThreeStrings(char *str )
   (str) (  str  ) (a str)
endps
```

If a text argument is not marked by either a slash or parentheses, *pswrap* treats it as an executable PostScript name object. In the following example, 'mydict' is treated as executable:

Wrap definition:

```
defineps PSWDoProcedure(char *mydict )
   mydict /procedure get exec
endps
```

Procedure call:

```
PSWDoProcedure("lexicon");
```

PostScript language code equivalent:

```
lexicon /procedure get exec
```

5.5 SENDING ARRAYS OF NUMBERS OR BOOLEANS

Each element in the wrap body that names an input array argument represents a PostScript literal array object that has the same element values. In the *PSWSetMyMatrix* wrap definition below, the current transformation matrix is set using an array of six floating-point values:

Wrap definition:

```
defineps PSWSetMyMatrix (float mtx[6])
   mtx setmatrix
endps
```

Procedure call:

```
static float anArray[] = {1.0, 0.0, 0.0, -1.0, 0.0, 0.0};
PSWSetMyMatrix(anArray);
```

PostScript language code equivalent:

```
[1.0 0.0 0.0 -1.0 0.0 0.0] setmatrix
```

The *PSWDefineA* wrap below sends an array of variable length to the PostScript interpreter:

Wrap definition:

```
defineps PSWDefineA (int data[x]; int x)
   /A data def
endps
```

Procedure call:

```
static int d1[] = {1, 2, 3};
static int d2[] = {4, 5};
   ...
PSWDefineA(d1, 3);
PSWDefineA(d2, 2);
```

PostScript language code equivalent:

```
/A [1 2 3] def
/A [4 5] def
```

5.6 SENDING A SERIES OF NUMERIC OR BOOLEAN VALUES

Occasionally, it is useful to group several numeric or boolean values into a C array and pass the array to a wrap that will send the individual elements of the array to the PostScript interpreter, as in the following example:

Wrap definition:

```
defineps PSWGrayCircle(float nums[3], gray)
   newpath
   \nums[0]  \nums[1]  \nums[2] 0.0 360.0 arc
   closepath
   gray setgray
   fill
endps
```

Procedure call:

```
static float xyRadius = {40.0, 200.0, 55.0};
PSWGrayCircle(xyRadius, .75);
```

PostScript language code equivalent:

```
newpath
40.0 200.0 55.0 0.0 360.0 arc
closepath
.75 setgray
fill
```

In the example above, '\nums[*i*]' identifies an element of an input array in the wrap body, where 'nums' is the name of an input boolean array or numeric array argument, *i* is a non-negative integer literal, and no whitespace is allowed between '\' and ']'.

5.6.1 Specifying the Size of an Input Array

As the foregoing examples illustrate, you can specify the size of an input array in two ways:

- Give an integer constant as the size when you define the procedure, as in the *PSWGrayCircle* wrap definition.
- Give an input argument that evaluates to an integer at run time as the size, as in the *PSWDefineA* wrap definition on page 18.

In either case, the size of the array must be a positive integer with a value not greater than 65,535.

5.7 SPECIFYING THE CONTEXT

Every wrap communicates with a PostScript execution context. The current context is normally used as the default. The Client Library provides operations for setting and getting the current context for each application. To override the default, declare the first argument as type 'DPSContext' and pass the appropriate context as the first parameter whenever the application calls the wrap. Here is an example of a wrap definition that explicitly declares a context:

Wrap definition:

```
defineps PSWGetGray(DPSContext c | float *level)
    currentgray level
endps
```

Procedure call:

```
DPSContext myContext;
float aLevel;
    ...
PSWGetGray(myContext, &aLevel);
```

PostScript language code equivalent:

```
currentgray
% Pop current gray level off operand stack
%   and store in aLevel
```

Warning: Do not refer to the name of the context in the wrap body.

---

6 DECLARING OUTPUT ARGUMENTS

To receive information back from the PostScript interpreter, the output arguments of a wrap must refer to locations where the information can be stored. An output argument can be declared as one of the following:

- A pointer to one of the *pswrap* data types listed on page 13, except for 'userobject'.

- An array of one of these types.
- A character string ('char *' or 'unsigned char *').
- A character array ('char [ ]' or 'unsigned char [ ]').

If an output argument is declared as a pointer or character string, the procedure writes the returned value at the location pointed to.

For an output argument declared as a pointer, previous return values are overwritten if the output argument is encountered more than once in executing the wrap body. For an output argument declared as a character string ('char *'), the value is stored only the first time it is encountered.

If an output argument is declared as an array of one of the *pswrap* data types (see page 13 for a list) or as a character array, the wrap fills the slots in the array (see Section 6.3).

Note: Whenever an array output argument is encountered in the wrap body, the values on the PostScript operand stack are placed in the array in the order in which they would be popped off the stack. When no empty array elements remain, no further storing of output in the array is done. No error is reported if elements are returned to an array that is full.

You can specify output arguments in the 'defineps' statement in any order that is convenient. The order of the output arguments has no effect on the execution of the PostScript language code in the wrap body.

pswrap does not check whether the wrap definition provides return values for all output arguments, nor does it perform type checking for declared output arguments.

6.1 RECEIVING NUMBERS

PostScript integer objects and real objects are 32 bits long. When returned, these values are assigned to the variable provided by the output argument. On a system where the size of an 'int' or 'float' is 32 bits, pass a *pointer* to an 'int' as the output argument for a PostScript integer object; pass a *pointer* to a 'float' as the output argument for a PostScript real object:

```
defineps PSWMyWrap ( | float *f; int *i)
```

A PostScript integer object or real object can be returned as a 'float' or 'double'. Other type mismatches cause a **typecheck** error (for example, attempting to return a PostScript real object as an 'int').

6.2 RECEIVING BOOLEAN VALUES

A procedure can declare a pointer to a 'boolean' as an output argument:

Wrap definition:

```
defineps PSWKnown(char *Dict, *x | boolean *ans)
   Dict /x known ans
endps
```

Procedure call:

```
int found;
   ...
PSWKnown("statusdict", "duplex", &found);
```

PostScript language code equivalent:

```
statusdict /duplex known found
```

This wrap expects to be passed the address of a variable of type 'int' as its output argument. If the PostScript interpreter returns the value *true*, the wrap places a value of 1 in the variable referenced by the output argument. If the interpreter returns the value *false*, the wrap places a value of zero in the variable.

6.3 RECEIVING A SERIES OF OUTPUT VALUES

To receive a series of output values as an array, declare an array output argument; then write a wrap body in the PostScript language to compute and return its elements, one or more elements at a time. The example below declares a wrap that returns the 256 font widths for a given font name at a given font size:

Wrap definition:

```
defineps PSWGetWidths(char *fn; int size | float wide[256] )
   /fn size selectfont
   0 1 255 {
         (X) dup 0 4 -1 roll put
         stringwidth pop wide
   } for
endps
```

Procedure call:

```
float widths[256];
PSWGetWidths("Serifa", 12, widths);
```

PostScript language code equivalent:

```
/Serifa 12 selectfont
0 1 255 {
   (X) dup 0 4 -1 roll put
   stringwidth pop
   % Pop width for this character and insert width
   %    into widths array at current element;
   %    point to next element.
} for
```

In the above example, the loop counter is used to assign successive ASCII values to the scratch string '(X)'. The **stringwidth** operator then places both the width and height of the string on the PostScript operand stack. (Here it operates on a string just one character long.) The **pop** operator removes the height from the stack, leaving the width at the top. The occurrence of the output argument 'wide' in this position triggers the width to be popped from the stack, returned to the application, and inserted into the output array at the current element. The next element then becomes the current element.

The **for** loop (the procedure enclosed in braces followed by **for**) repeats these operations for each character in the font, beginning with the 0th and ending with 255th element of the font array.

6.3.1 Receiving a Series of Array Elements

A PostScript array object can contain a series of elements to be

stored in an output array. The output array is filled in, one element at a time, until it's full. Therefore the *PSWTest* wrap defined below will return '{1, 2, 3, 4, 5, 6}':

```
defineps PSWTest(| int Array[6])
   [1 2 3] Array
   [4 5 6] Array
endps
```

The *PSWTestMore* wrap defined below will return '{1, 2, 3, 4}':

```
defineps PSWTestMore(| int Array[4])
   [1 2 3] Array
   [4 5 6] Array
endps
```

6.3.2 Specifying the Size of an Output Array

The size of an output array is specified in the same manner as the size of an input array. Use a constant in the wrap definition or an input argument that evaluates to an integer at run time. If more elements are returned than fit in the output array, the additional elements are discarded.

6.4 RECEIVING CHARACTERS

To receive characters back from the PostScript interpreter, declare the output argument either as a character string or as a character array.

If the argument is declared as a character string, the wrap copies the returned string to the location indicated. Be careful to provide enough space for the maximum number of characters that might be returned, including the null character ('\0') that terminates the string. Only the first string encountered will be returned. For example, in the *PSWStrings* procedure defined below, the string '123' will be returned:

```
defineps PSWStrings(| char *str)
   (123) str
   (456) str
endps
```

Character arrays, on the other hand, are treated just like arrays of numbers. In the *PSWStrings2* procedure, the value returned for 'str' will be '123456':

```
defineps PSWStrings2(| char str[6])
   (123) str
   (456) str
endps
```

If the argument is declared as a character array (for example, 'char s'[*num*]), the procedure copies up to *num* characters of the returned string into the array. Additional characters are discarded. The string is not null terminated.

6.5 COMMUNICATION AND SYNCHRONIZATION

The PostScript interpreter can run as a separate process from the application; it can even run on a separate machine. When the application and interpreter processes are separated, the application programmer must take communication into account. This section alerts you to communication and synchronization issues.

A wrap that has no output arguments returns as soon as the wrap body is transferred to the client/server communication channel. In this case, the communication channel is not necessarily flushed. Since the wrap body is not executed by the PostScript interpreter until the communication channel is flushed, errors arising from the execution of the wrap body can be reported long after the wrap returns.

In the case of a wrap that returns a value, the entire wrap body is transferred to the client/server communication channel, which is then flushed. The client-side code awaits the return of output values followed by a special termination value. Only then does the wrap return.

See the *Client Library Reference Manual* for information concerning synchronization, run-time errors, and error handling.

A ERROR MESSAGES FROM THE *PSWRAP* TRANSLATOR

The following is a list of error messages the *pswrap* translator can generate:

input parameter used as a subscript is not an integer

output parameter used as a subscript

char input parameters must be starred or subscripted

hex string too long

invalid characters

invalid characters in definition

invalid characters in hex string

invalid radix number

output arguments must be starred or subscripted

out of storage, try splitting the input file

-s 80 is the minimum

can't allocate char string, try a smaller -s value

can't open file for input

can't open file for output

error in parsing

string too long

usage: pswrap [-s maxstring] [-ar] [-h headerfile] [-o outfile] [infile]

endps without matching defineps

errors in parsing

errors were encountered

size of wrap exceeds 64K

parameter reused

output parameter used as a subscript

non-char input parameter

not an input parameter

not a scalar type

wrong type

parameter index expression empty

parameter index expression error

end of input file/missing endps

B SYNTAX

Square brackets [] mean that the enclosed form is optional. Curly brackets { } mean that the enclosed form is repeated, possibly zero times. A vertical bar | separates choices in a list.

```
Unit =
    ArbitraryText {Definition ArbitraryText}

Definition =
    NLdefineps ["static"] Ident "(" [Args] ["|" Args]")" Body NLendps

Body =
    {Token}

Token =
    Number | PSIdent | SlashPSIdent
    | "("StringLiteral")"
    | "<"StringLiteral">"
    | "{" Body "}"
    | "[" Body "]"
    | Input Element

Args =
    ArgList {";" ArgList} [";"]

ArgList =
     Type ItemList

Type =
    "DPSContext" | "boolean" | "float" | "double"
    | ["unsigned"] "char" | ["unsigned"] ["short" | "long"] "int"

ItemList =
      Item {"," Item}

Item =
    "*" Ident | Ident ["["Subscript"]"]

Subscript =
      Integer | Ident
```

B.1 SEMANTIC RESTRICTIONS

- DPSContext must be the first input argument if it appears at all.
- A simple char argument (char Ident) is never allowed (must be * or [ ]).
- A simple Ident item is not allowed in an output item list (must be * or [ ].

B.2 CLARIFICATIONS

- NLdefineps matches the terminal defineps at the beginning of a new line.
- NLendps matches the terminal endps at the beginning of a new line.
- Ident follows the rules for C names; PSIdent follows the rules for PostScript language names.
- SlashPSIdent is a PostScript language name preceded by a slash.
- StringLiteral tokens follow the PostScript language conventions for string literals.
- Number tokens follow the PostScript language conventions for numbers.
- Integer subscripts follow the C conventions for integer constants.
- Input Element is \n[i] where *n* is the name of an input array argument, *i* is a non-negative integer literal, and no white space is allowed between \ and].

C CHANGES SINCE LAST PUBLICATION OF THIS DOCUMENT

Changes to the *pswrap Reference Manual* from the document dated October 25, 1989, are noted in the paragraphs below.

A string ('char *') passed as input may not be more than 65,535 characters. An array may not contain more than 65,535 elements.

The examples were expanded to include, in each case, the wrap definition, the corresponding procedure call, and the equivalent PostScript language code.

Changes to the *pswrap Reference Manual* from the document dated October 6, 1988, are noted in the paragraphs below.

The manual was rewritten and reorganized. Numerous technical clarifications and corrections were made.

Index

ENCAPSULATED POSTSCRIPT® FILES Specification Version 2.0

June 5, 1989
PostScript® Developer Support Group

Adobe Systems Incorporated
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400

PN LPS5002

POSTSCRIPT®

ENCAPSULATED POSTSCRIPT® FILES Specification Version 2.0

June 5, 1989
PostScript® Developer Support Group
(415) 961-4111

This document specifies the format required for import of Encapsulated PostScript (EPS) Files into an application. This specification suggests a standard for importing PostScript language files in all environments, and contains specific information about both the Macintosh® and MS-DOS environments. This format conforms to Adobe Systems' Document Structuring Conventions, Version 2.0.

The rules that should be followed in creating importable PostScript language files are a subset of the structuring conventions proposed by Adobe Systems Incorporated; refer to the *PostScript Language Reference Manual*, Appendix C, and *Document Structuring Conventions*, version 2.0, available from Adobe Systems. Files must also be "well-behaved" in their use of certain PostScript language operators, manipulation of the graphics state, and manipulation of the PostScript interpreter's stacks and any global dictionaries. These conventions are designed to allow cooperative sharing of files between many systems using the PostScript language.

Fundamentally, an EPS file is a standard PostScript language file with a bitmap screen preview included optionally in the format. The purpose of an EPS file is to be included into other document makeup systems as an illustration, and the screen representation is intended to aid in page composition. The bitmap is normally discarded when printing, and the PostScript language segment of the file is used instead. Typically any manipulation of the screen image that is performed by the user (such as scaling, translating, or rotation on screen) should be tracked by the page layout application and an appropriate transformation should precede the EPS file when it is sent to the printer.

1. EPS FILE FORMAT GUIDELINES

An EPS file should conform to at least Version 2.0 of the Adobe Document Structuring Conventions. This does not explicitly require any of the structuring comments to be employed, but if used, they should be in accordance with that specification. Additionally, an EPS file is required to contain the %%BoundingBox comment, and is required to be "well-behaved" (see pages 3-4). An EPS file may optionally contain a bitmap image suitable for WYSIWYG screen display, as discussed herein.

The structure of an EPS file is marked by PostScript language comments, according to the *PostScript Document Structuring Conventions*. These are covered briefly here for reference. Structuring comment lines must begin with "%!" or "%%" and terminate with a *newline* (either return or linefeed) character. EPS file conventions require that a comment line be no longer than 256 bytes. A comment line may be continued by beginning the continuation line with "%%+". The EPS file should begin with a header of structuring comments, as specified in the PostScript Structuring Conventions.

2. REQUIRED PARTICIPATION

In order to support Encapsulated PostScript files effectively, some cooperation is required on the parts of those who produce EPS files and those who use EPS files (typically by including them into other documents).

2.1 WHEN PRODUCING EPS FILES

There are certain required comments and several recommended ones that must be provided in the EPS file. These are detailed in Section 3. The file must also be a single page (not a multiple-page document) and must be a *conforming* PostScript language document. Conformance requirements are mostly detailed here, but for the full specification, please refer to the *Document Structuring Conventions* from Adobe Systems.

2.2 WHEN READING AND USING EPS FILES

When including an EPS file into your document, you should basically think of that piece of code as having been generated by your program. After all, that is what all programs (and users) who encounter your print file will think. In particular, you must find out enough about the file to intelligently make it part of your document. The only tricky part of this relates to font usage. This is also the most difficult part of this specification to understand. Basically, you just have to figure out what the requirements of the illustration are and incorporate them into your own requirements (pass them downstream). Then all issues of font management are essentially the same as they were before you included the illustration (and are beyond the scope of this document).

As long as you don't remove relevant information from a file, and as long as you update your global view of font usage and resource requirements to reflect those that you just imported, the rest is fairly easy. The intent behind the EPS specification, in fact, is to make the most of cooperation between producers and consumers of PostScript language files so that neither has to do much, but the combined advantage is great.

3. REQUIRED COMMENTS

The first comment in the header (and the first line in the file) should be the version comment:

%!PS-Adobe-2.0 EPSF-2.0

This indicates to an application that the PostScript language file conforms to this standard. The version number following the word **"Adobe-"** indicates the level of adherence to the standard PostScript Document Structuring Conventions. The version number following the word **"EPSF"** indicates the level of EPSF-specific comments.

The following comment must be present in the header; if it is not present then an importing application may issue an error message and abort the import:

%%BoundingBox: LLx LLy URx URy

The values are in the PostScript default user coordinate system, in points (1/72 of an inch, or 0.3527 mm), with the origin at the lower left corner. The bounding box must be expressed in default user coordinate space. This seems to be a big question among implementors of this specification. Regardless of the coordinate system in which your

application operates, here is a foolproof way of determining the correct bounding box: *print* the page, get out a point ruler, and *measure* first to the lower left corner, then to the upper right corner, using the lower-left corner of the physical paper as your origin. This works because it measures the end result (the marks on the page), and none of the computation matters.

4. OPTIONAL COMMENTS

The following header comments are strongly recommended in EPS files. They provide extra information about the file that can be used to identify it on-screen or when printing.

%%Title: included_document_title
%%Creator: creator_name
%%CreationDate: date_and_time

The **%%Creator**, **%%Title**, and **%%CreationDate** comments may be used by an application or spooler to provide human-readable information about a document, or to display the file name and creator on the screen if no bitmapped screen representation was included in the EPS file.

%%EndComments

This comment indicates an explicit end to the header comments, as specified in the structuring conventions.

4.1 HOW TO USE THESE COMMENTS (PHILOSOPHY)

All of the comments in EPS files provide information of some sort or another. Exactly how you use this information is up to you, but you are encouraged not to reduce the amount of information in a file (when you import it or include it, for example) by removing or altering comments. In general, the comments tell you what fonts and files are used, and where. Not everybody cares about these things, but if you do care, then the information is available.

The whole issue of Encapsulated PostScript files is that they are "final form" print files that may be far from the printer that they will actually be imaged on. If they have specific needs, particularly in terms of font usage, these needs must be carefully preserved and passed on downstream, and the program that actually prints the composite document must take pains to make sure the fonts are available at print time.

Any piece of software that generates PostScript language code is potentially both a consumer and a producer of Encapsulated PostScript files. It is probably best not to think that you are at either end of the chain. In particular, if you import an Encapsulated PostScript file, integrate it into your document somehow, and then go to print your document, you are responsible for reading and understanding any of the font needs of the EPS file you imported. These should then be reflected in your *own* font usage comments. If the illustration on page *3* uses the Bodoni font but the rest of your document is set in Times, suddenly your document now also uses the Bodoni font (you included the illustration, after all). This should be reflected in the outermost **%%DocumentFonts** comments and any other appropriate ones.

4.2 FONT MANAGEMENT COMMENTS

If fonts are used, the following two comments (which are defined in version 2.0 of the PostScript Document Structuring Conventions) should be included in the header of the EPS file. The **%%IncludeFont** and **%%Begin/%%EndFont** comments should be thought of as inverses of one another. That is, if you encounter an **%%IncludeFont** comment and actually include a font file at that point, you should enclose the font in **%%BeginFont** and **%%EndFont** comments. Conversely, if you see fit to remove a font from a print file (one that presumably had been delimited with comments), you should always replace it with an **%%IncludeFont** comment rather than completely stripping it. This guarantees the reversibility of your actions.

%%DocumentFonts: font1 font2 .... %%+ font3 font4

The **%%DocumentFonts** comment provides a full list of all fonts used in the file. Font names should refer to *non-reencoded* printer font names and should be the valid PostScript language names (without the leading slashes). An application that imports an EPS file should be responsible for satisfying these font needs, or at least updating its own **%%DocumentFonts** list to reflect any new fonts.

%%DocumentNeededFonts: font1 font2 ....

The **%%DocumentNeededFonts** comment lists all fonts that are to be included at specific points within the EPS file as a result of the **%%IncludeFont** comment. These fonts must also be listed in the **%%DocumentFonts** comment, but an application may or may not pre-load these at the beginning of the job. The responsibility should be taken, however, by any program that thinks it is actually printing the file, to make sure the fonts requested will be available when the file is printed. This may mean that the individual **%%IncludeFont** comments may be satisfied and the fonts placed in-line, or they may simply be ignored, if the fonts are determined to be already available on the printer. As a third possibility, there may be enough memory to download all the fonts in front of the job and avoid processing the individual requests. This **%%DocumentNeededFonts** comment provides foreshadowing of the **%%IncludeFont** comments to follow, to give printing managers enough information to make these choices intelligently.

%%IncludeFont: fontname

The **%%IncludeFont** comment signals to an application that the specified font is to be loaded at that precise location in the file. It is analogous to the familiar **#include** syntax in the C language. An application should load the specified font regardless of whether the same font has been loaded already by other preceding **%%IncludeFont** comments, since the font may have been embedded within a PostScript language **save** and **restore** construct. However, if the font is determined to be available *prior* to the entire included EPS file (for instance, it may be in ROM in the printer or might have been downloaded prior to the entire print job) the **%%IncludeFont** comment may be ignored by printing manager software.

When an application satisfies an **%%IncludeFont** request, it should *always* bracket the font itself with the **%%BeginFont** and **%%EndFont** comments.

A font that is wholly contained, defined, and used within the EPS file (a downloaded font) should be noted in the **%%DocumentFonts** comment, but *not* the **%%DocumentNeededFonts** comment. The font should follow conventions listed in the Document Structuring Conventions in order to retain full compatibility with print spoolers.

%%BeginFont: fontname
%%EndFont

The **%%BeginFont** and **%%EndFont** comments bracket an included downloadable font. The **fontname** is the simple PostScript language name for the font. These fonts may be stripped from the included file if they are determined to be available (but should be replaced by an **%%IncludeFont** comment).

4.3 FILE MANAGEMENT COMMENTS

%%IncludeFile: filename

This comment, which can occur only in the body of an EPS file, allows a separate file to be inserted at any point within the EPS file. The file might not be searched for or inserted until printing actually occurs, so user care is required to ensure its availability. If it is used, the %%DocumentFiles comment should be used as well. See the Structuring Conventions for more information.

%%BeginFile: filename
%%EndFile

The **%%BeginFile** and **%%EndFile** comments bracket an included file. They are the "inverse" of the **%%IncludeFile** comment. The **filename** is evaluated in the context of the local file system. These files may *not* be stripped from the included file at print time, because they undoubtedly contain executable code. However, they may be temporarily removed, or "factored out" to save space during storage. They should always be replaced by the **%%IncludeFile** comment.

4.4 COLOR COMMENTS

%%DocumentProcessColors: keyword keyword ...

This comment marks the use of process colors within the document. Process colors are defined to be *cyan*, *magenta*, *yellow*, and *black*. These four colors are indicated in this comment by the keywords **Cyan**, **Magenta**, **Yellow**, and **Black**. This comment is used primarily when producing color separations. The **(atend)** conventions is allowed.

%%DocumentCustomColors: name name ...

This indicates the use of custom colors within a document. These colors are arbitrarily named by an application, and their CMYK or RGB approximations are provided through the **%%CMYKCustomColor** or **%%RGBCustomColor** comments within the body of the document. The names are specified to be any arbitrary PostScript language string except (Process Cyan), (Process Magenta), (Process Yellow), and (Process Black), which need to be reserved for custom color implementation by applications. The **(atend)** specification is permitted.

%%BeginProcessColor: keyword
%%EndProcessColor

The **keyword** here is either **Cyan**, **Magenta**, **Yellow**, or **Black**. During color separation, the code between these comments should only be downloaded during the appropriate pass for that process color. Intelligent printing managers can save considerable time by omitting code within these bracketing comments on the other three separations. Extreme care must be taken by the document composition software to correctly control overprinting and "knockouts" if these comments are employed, since the code may or may not actually be executed.

%%BeginCustomColor: keyword
%%EndCustomColor

The **keyword** here is any PostScript language string except (Process Cyan), (Process Magenta), (Process Yellow), and (Process Black). During color separation, the code between these comments should only be downloaded during the appropriate pass for that custom color. Intelligent printing managers can save considerable time by omitting code within these bracketing comments on the other three separations. Extreme care must be taken by the document composition software to correctly control overprinting and knockouts if these comments are employed, since the code may or may not be executed.

%%CMYKCustomColor: cyan magenta yellow black *keyword*

This provides an *approximation* to the custom color specified by ***keyword.*** The four components of **cyan**, **magenta**, **yellow**, and **black** must be specified as numbers from 0 to 1 representing the percentage of that process color. These numbers are exactly analogous to the arguments to the **setcmykcolor** PostScript language operator. The ***keyword*** follows the same custom color naming conventions for the **%%DocumentCustomColors** comment.

%%RGBCustomColor: red green blue *keyword*

This provides an *approximation* to the custom color specified by ***keyword.*** The three components of **red**, **green**, and **blue** must be specified as numbers from 0 to 1 representing the percentage of that process color. These numbers are exactly analogous to the arguments to the **setrgbcolor** PostScript language operator. The ***keyword*** follows the same custom color naming conventions for the **%%DocumentCustomColors** comment.

5. "WELL-BEHAVED" RULES

An application should encapsulate the imported EPS code in a save / restore construct, which will allow all printer VM (memory) to be recovered and all graphics state restored. Since the code in the imported EPS file will be embedded within the PostScript language that an application will generate for the current page, it is necessary that it obey the following rules, in order to keep from disrupting the enclosing document:

5.1 OPERATORS TO AVOID

The following PostScript operators should not be included in a PostScript language file for import; the result of executing any of these is not guaranteed (see the PostScript Document Structuring Conventions for more on this):

grestoreall	initgraphics	initmatrix	initclip
erasepage	copypage	banddevice	framedevice
nulldevice	renderbands	setpageparams	note
exitserver	setscreen*	settransfer*	

5.2 THE 'SETSCREEN' AND 'SETTRANSFER' OPERATORS

The **setscreen** operator is troublesome when one file is included within another. **setscreen** is a system-level command that is appropriate for changing the halftone machinery to compensate for marking engine tendencies, but when used for "special effects" can cause problems. For EPS files, the **setscreen** and **settransfer** operators are permitted only under restricted terms.

THE 'SETTRANSFER' AND 'SETCOLORTRANSFER' OPERATORS

The **settransfer** operator changes the gray-level and color response curves over the interval from **0** to **1**. There are two basic uses of it: to invert an image (typically flipping blacks and whites, less often colors), or to adjust the response curve for a particular output device.

The best (and required) approach for using **settransfer** is to combine your function with the existing one. Here is the recommended way to do this:

```
{ dummy exec  1 exch sub } dup 0 currentransfer put settransfer
```

In this example, the desired transfer function is the code **1 exch sub**. The **dummy exec** essentially executes the existing transfer function before executing the new code. The name **dummy** is replaced by the actual procedure body from the existing transfer function through the **put** instruction. The result is conceptually equivalent to this:

```
{ { original proc } exec 1 exch sub } settransfer
```

This approach is better than "concatenating" procedures because it does not require the existing transfer function to be duplicated (consuming memory).

5.3 THE 'SHOWPAGE' OPERATOR

The **showpage** operator is permitted in EPS files primarily because it is present in so many PostScript language files. It is reasonable for an EPS file to use the **showpage** operator if needed (although it is not necessary if the file is truly imported into another document). It is the including application's responsibility to *disable* showpage if needed. The recommended method to accomplish this is as follows:

TEMPORARILY DISABLING 'SHOWPAGE'

```
/BEGINEPSFILE { %def
        /EPSFsave save def
        0 setgray 0 setlinecap 1 setlinewidth 0 setlinejoin 10 setmiterlimit [] 0 setdash
        newpath
        /showpage { } def
} bind def
/ENDEPSFILE { %def
        EPSFsave restore
} bind def

BEGINEPSFILE
 100 300 translate
 .5 .5 scale
 % include the EPS file here, which may execute showpage with no effect

ENDEPSFILE      % restore state and continue
```

This method will only disable the **showpage** operator during the execution of the EPS file, and will restore the previous semantics of **showpage** afterward. It is the responsibility of the EPS file itself to avoid the operators listed in the previous section that might cause unexpected behavior when imported. They need not be redefined along with **showpage**, although it is permissible to do so.

5.4 STACKS AND DICTIONARIES

All of the PostScript interpreter's stacks (including the dictionary stack) should be left in the state that they were in before the imported PostScript language code was executed. This is normally the case for well-written PostScript language programs, and this is still the best way to keep unanticipated side-effects to a minimum. Please avoid unnecessary **clear** and **"countdictstack 2 sub { end } repeat"** cleanup techniques. If you have accidentally left something on one of the stacks, it is best to understand your program well enough to get rid of it, rather than issuing a wholesale cleanup instruction at the end, which will not only clear your operands from the stack, but perhaps will clear other objects as well.

It is recommended that the imported EPS file create its own dictionary instead of writing into whatever the current dictionary might be. Make sure that this dictionary is removed from the dictionary stack when through (using the PostScript language **end** operator) to avoid the possibility of an **invalidrestore** error. Also, no global string bodies should be changed (with either **put** or **putinterval**).

If a special dictionary (like **statusdict**) is required in order for the imported PostScript language code to execute properly, then it should be included as part of the EPS file. However, it should be enclosed in very specific **%%BeginFeature** and **%%EndFeature** comments as specified in the Document Structuring Conventions. No dictionary should be assumed to be present in the printer, and fonts should be reencoded as needed by the EPS file itself.

5.5 THE GRAPHICS STATE

When a PostScript language program is imported into the middle of another executing program, the state of the interpreter may not be exactly in its default state. The EPS file should assume that the graphics state is in its default state, even though it may not be. An importing application may choose to scale the coordinate system or to change the transfer function to change the behavior of the EPS file somewhat. If the EPS file makes assumptions about the graphics state (like the clipping path) or explicitly sets something it shouldn't (the transformation matrix), the results may not be what were expected.

The importing application is responsible for returning the color to be black, the current dash pattern, line endings, and other miscellaneous aspects of the graphics state to their default condition (as specified in the PostScript Language Reference Manual). This can be done in either of two ways: the initial graphics state can be restored from variables, or the state can be explicitly set:

```
/BEGINEPSFILE { %def
        /EPSFsave save def
        0 setgray 0 setlinecap 1 setlinewidth 0 setlinejoin 10 setmiterlimit [] 0 setdash
        newpath
        /showpage { } def
} bind def

/ENDEPSFILE { %def
        EPSFsave restore
} bind def
```

6. FILE TYPES AND FILE NAMING

APPLE MACINTOSH FILES

The Macintosh file type for application-created PostScript language files is **EPSF**. Files of type **TEXT** will also be allowed, so that users can create EPS files with standard editors, although the Structuring Conventions must still be strictly followed. A file of type EPSF should contain a PICT resource in the resource fork of the file containing a screen representation of the PostScript language code. The file name itself may follow any naming convention as long as the file type is **EPSF**. If the file type is **TEXT**, the extensions **.epsf** and **.epsi**, respectively, should be used for the Macintosh-specific format and EPSI interchange format.

MS-DOS AND PC-DOS FILES

The recommended file extension is **.EPS**. For EPSI files, the extension should be **.EPI**. Other file extensions also can be used, but it will be assumed that these files are text-only files with no screen metafile included in them.

OTHER FILE SYSTEMS

In general, the extension **.epsf** is the preferred way to name an EPS file, and **.epsi** for the interchange format. In systems where lower-case letters are not recognized or are not significant, all upper-case can be used.

7. SCREEN REPRESENTATIONS

The EPS file will usually have a graphic screen representation so that it can be manipulated and displayed on a workstation's screen prior to printing. The user may position, scale, crop or rotate this screen representation, and the composing software should keep track of these manipulations and reflect them in the PostScript language code that is ultimately sent to the printing device.

The exact format of this screen representation is machine-specific. That is, each computing environment may have its own preferred bitmap format, and that is typically the appropriate screen representation for that environment. An interchange representation is specified that should be implemented by everyone, and any environment-specific formats can be supported in addition, as deemed appropriate.

7.1 APPLE MACINTOSH: PICT RESOURCE

A QuickDraw™ representation of the PostScript language file can be created and stored as a PICT in the resource fork of the file. It should be given resource number 256. If the PICT exists, the importing application may use it for screen display. If the *picframe* is transformed to PostScript language coordinates, it should agree with the **%%BoundingBox** comment.

Given the size limitations on PICT images, this may not always agree for large illustrations. If there is a discrepancy, the **%%BoundingBox** always should be taken as the "truth", since it accurately describes the area that will be imaged by the PostScript language code itself. In this situation, applications *producing* the preview PICT must all take the same action so that the importing application knows what to do.

Since it is more important to have a reasonable facsimile of the image than it is to have any particular part of it be high quality, the PICT image should be *scaled* to fit within the constraints of the PICT format. That is, the picture will all be there (it will not be cropped), but it will actually be *smaller* than the real image. The importing application should then scale the PICT to a size which matches the bounding box as expressed in the **%%BoundingBox** comment.

7.2 PC/DOS: WINDOWS METAFILE OR TIFF FILE

Either a Microsoft® Windows Metafile or a TIFF (Tag Image File Format) section can be included as the screen representation of an EPS file.

The EPS file itself has a binary header added to the beginning that provides a sort of "table of contents" to the file. This is necessary since there is not a second "fork" within the file system as there is in the Macintosh file system.

NOTE:
It is always permissible to have a pure ASCII PostScript language file as an EPS file in the DOS environment, as long as it does not contain the preview section. The importing application should check the first three bytes of the file. If they match the header as shown below, the binary header should be expected. If the first two match %!, it should be taken to be an ASCII PostScript language file.

DOS EPS Binary File Header

Bytes	Description
0-3	Must be hex C5D0D3C6 (byte 0=C5)
4-7	Byte position in file for start of PostScript language code section.
8-11	Byte length of PostScript language section
12-15	Byte position in file for start of Metafile screen representation.
16-19	Byte length of Metafile section (PSize)
20-23	Byte position of TIFF representation
24-27	Byte length of TIFF section
28-29	Checksum of header (XOR of bytes 0-27) NOTE: if Checksum is FFFF then it is to be ignored.

Note:
It is assumed that either the Metafile or the TIFF position and length fields are zero; that is, only one or the other of these two formats is included in the EPS file.

The Metafile should follow the guidelines set forth by the Windows specification. In particular, it should not set the viewport or mapping mode, and it should set the window origin and extent. The application should scale the picture to fit within the **%%BoundingBox** comment specified in the PostScript language file.

8. DEVICE-INDEPENDENT INTERCHANGE FORMAT

This last screen representation is intended as an interchange format between widely varied systems. In particular, the bitmap preview section of the file is very simple and is represented as ASCII hexadecimal in order to be more easily transportable. This format is dubbed Encapsulated PostScript Interchange format, or "EPSI."

This format wins no prizes for compactness, but it should be truly portable and requires no special code for decompressing or otherwise understanding the bitmap portion, other than the ability to understand hexadecimal notation.

It is expected that applications that support EPSF will gradually head toward supporting only two formats: the first is the "native" format for the environment in which the application runs (where the preview section is Macintosh PICT or TIFF or Sun raster files or whatever); the second format should simply be this interchange format. Then files can be interchanged between widely varying systems without each having to know the preferred bitmap representation of the others.

%%BeginPreview: width height depth lines
%%EndPreview

These comments bracket the preview section of an EPS file in Interchange format (EPSI). The **width** and **height** fields provide the number of image samples (pixels) for the preview. The **depth** field provides how many bits of data are used to establish one sample pixel of the preview (1, 2, 4, or 8). An image which is 100 pixels wide will always have 100 in the **width** field, although the number of bytes of hexadecimal needed to build that line will vary if **depth** varies. The **lines** field tells how many lines of hexadecimal are contained in the preview, so that they may be easily skipped by an application that doesn't care. All the arguments are integers.

8.1 SOME RULES AND GUIDELINES FOR "EPSI" FILES

The following guidelines attempt to clarify a few basic assumptions about the EPSI format. It is intended to be extremely simple, since its purpose is interchange. No system should have to do much work to decipher one of these files, and the preview section is mostly just a convenience to begin with. This format is accordingly deliberately kept simple and option-free.

- The preview section must be after the header comment section but before the document prologue definitions. That is, it should immediately follow the **%%EndComments** line in the EPS file.

- In the preview section, **0** is white and **1** is black, in deference to the majority. Arbitrary transfer functions and "flipping" black and white are not supported.

- The Preview image can be of *any* resolution. The size of the image is determined solely by its *bounding box*, and the preview data should be scaled to fit that rectangle. Thus, the **width** and **height** parameters from the image are *not* its measured dimensions, but simply describe the amount of data supplied for the preview. The dimensions are described only by the bounding rectangle.

- The hexadecimal lines must never exceed 255 bytes in length. In cases where the preview is very wide, the lines must be broken. The line breaks can be made at any even number of hex digits, since the dimensions of the finished preview are established by the **width**, **height**, and **depth** values.

- All non-hexadecimal characters should be ignored when collecting the data for the preview, including tabs, spaces, newlines, percent characters, and other stray ASCII characters. This is analogous to the PostScript language **readhexstring** operator.

- Each line of hexadecimal will begin with a percent sign ('%'). This makes the entire preview section into a PostScript language comment, so that the file can be printed without modification.

- If the **%%IncludeFile** or **%%BeginFile / %%EndFile** comments are ever used to extract the preview section from the EPS file, then the **lines** argument to the **%%BeginPreview** comment must be adjusted accordingly. The **lines** value specifies only the number of lines to *skip* if you're not the least bit interested.

- If the **width** of the image is not a multiple of 8 bits, the hexadecimal digits are padded out to the next highest multiple of 8 bits.

EXAMPLE "EPSI" FILE

Here is a sample file showing the EPS Interchange (EPSI) format. The preview section is expressed in user space and the correct comments are included. Remember that there are 8 bits to a byte, and that it requires 2 hexadecimal digits to represent one binary byte. Therefore the 80-pixel width of the image requires 20 bytes of hexadecimal data, which is (80 / 8) * 2. The PostScript language segment itself simply draws a box, as can be seen in the last few lines.

```
%! PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 80 24
%%Pages: 0
%%Creator: Glenn Reid
%%CreationDate: September 19, 1988
%%EndComments
%%BeginPreview: 80 24 1 24
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FF00000000000000000FF
% FF00000000000000000FF
% FF00000000000000000FF
% FF00000000000000000FF
% FF00000000000000000FF
% FF00000000000000000FF
% FF00000000000000000FF
% FF00000000000000000FF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
% FFFFFFFFFFFFFFFFFFFF
%%EndPreview
%%EndProlog
%%Page: "one" 1
 4 4 moveto 72 0 rlineto 0 16 rlineto -72 0 rlineto closepath
 8 setlinewidth stroke
%%Trailer
```

DOCUMENT STRUCTURING CONVENTIONS
Specification
Version 2.1

January 16, 1989
PostScript® Developer Tools & Strategies Group

Adobe Systems Incorporated
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400

PN LPS5001

POSTSCRIPT®

DOCUMENT STRUCTURING CONVENTIONS Specification Version 2.1

January 16, 1989
PostScript® Developer Tools and Strategies Group

1. INTRODUCTION

This document provides an extension to the published PostScript language document structuring conventions (Version 1.0). These extended comment conventions will provide print spoolers, servers and post-processors (known collectively in this document as *document managers*) with additional information about the structural organization and resource requirements of a PostScript language document file.

The Document Structuring Conventions are provided to address issues of PostScript language resource management at a site-wide level, and to allow software systems to maintain print files independent of printer-specific issues such as available paper sizes, fonts, output bins or file systems. The structuring conventions are also designed to work with the Adobe PostScript Printer Description files (PPD files), which provide the PostScript language extensions for specific printer features in a regular, parsable format. For example, these files will include information about how to invoke manual feed or different paper sizes, and they will also include information about the fonts that are built into the ROM of each printer. The structuring conventions will work in tandem with these files to provide a method of specifying and invoking printer features.

The Adobe Document Structuring Conventions allow complete cooperation between document handling mechanisms at all levels. They provide a layer of structural information which is, in a sense, superimposed on a PostScript language document file. This level of structure is used by document handling software without the need for parsing the PostScript language directly. This layer is *cooperative* rather than *enforced*, in that the comments themselves are always ignored by the PostScript interpreter, and they may or may not be interpreted by other software. A document is said to be *conforming* if it observes the Adobe® Document Structuring Conventions, and the conforming document can be expected to adhere to specific structuring constraints in that event.

It is assumed that document managers may be capable of producing prologue, font and disk file resources, although that is not required to make effective use of the structuring conventions. It is also assumed that the reader is mostly familiar with the motivation, goals and design of the existing structuring conventions. For further information, consult *Appendix C* in the *PostScript Language Reference Manual.* All of the existing comment conventions have been included in this document for reference and continued use.

2. HISTORY AND MOTIVATION

One of the most important levels of document structuring in the PostScript language is the distinction between the *Document Prologue* and the *Document Script.* The prologue is typically a set of procedure definitions appropriate for the set of operations required by a document composition system, and the script is the software-generated data that represents a particular document. A *conforming* PostScript language document description will usually have a clearly defined prologue and script (see the **%%EndProlog** structure convention) which should be distinct within a given print file. In other words, there should be nothing "executed" in the prologue, and "no definitions" in the script. Furthermore, a document *script* may be divided again into several *pages*, each of which will be *functionally independent* of the other pages. This means that the individual pages should be able to execute in any order and may be physically rearranged without affecting the resulting document. Pages may be printed in parallel as long as the prologue definitions are made available to each page. Note further that one document description may *wholly contain* another document description as part of its script without disrupting the semantics of the structuring. For instance, a PostScript language document may be placed into another document as an illustration with no effect on the functionality of the containing document as a whole. In fact, this is one of the primary strengths of PostScript language document descriptions: the ability to completely merge documents from different sources at the final *printing* stage.

3. POSTSCRIPT LANGUAGE COMMENTS

The comment conventions are divided roughly into the following three areas:

- Structure Comments
- Resource Requirements
- Query Conventions

Structure Comments are used to delimit the various structural components of a PostScript language page description, including its *prologue*, *script*, and *trailer*, and where the page breaks fall (if there are any). These comments have been expanded in this version to include document and page setup information, and to provide a markup convention for noting the beginning and end of particular pieces of the page description that might need to be identified for further use. In particular, embedded fonts files, and procedure definitions will have begin/end comment constructs around them to facilitate removing them from the print job (if necessary) or re-structuring the document. *Resource Requirements* are comments used to specify resources which are required by the PostScript language page description, but which have not been included within its text (such as prologues, fonts and included files). These requirement conventions may also specify other document requirements, which may vary from a particular stock form to a specific paper color or collating order. Requirements for individual printer features such as paper trays will also fall into this category. *Query comments* are used to delimit parts of a PostScript language program which query the current state or characteristics of a printer, including the availability of prologues, fonts, files, virtual memory and any printer-specific features and enhancements.

Any or all of these conventions may be used within a single document description, but typically some subset of them will be used consistently within a particular printing environment. The Structuring Conventions have been designed with maximum flexibility in mind, and with a minimum amount of interdependency between the conventions themselves. Thus one may employ only Structure Conventions in an environment where the presence of a spooler may not be guaranteed, or may freely use only the Resource Requirement comments on a highly spooled network. See the section on **Using the Document Structuring Conventions** for more information.

4. CONFORMING FILES

A PostScript language document file is said to be *conforming* if it obeys a proper subset of these structuring conventions. In addition, there are several other constraints which apply to conforming PostScript language documents that will allow document handling software to deal reasonably with the document files. They may be thought of as "promises" that the document description observes certain standard usage and structuring constraints. Primarily these boil down to avoidance of "system-level" PostScript language operators which may cause attempts at processing the documents to fail (see **PostScript language operators to avoid**, *at the end of this section).* If these structuring conventions are employed, care should be taken to use them correctly and in accordance with their intended goals. *Failure to do so may result in unexpected behavior of document files within some document handling systems.* A simple example of this is a document file which results in an execution error when the pages are re-ordered by a document spooler. This can result simply from the improper employment of the **%%Page:** comment convention.

Here are some general constraints which apply to all conforming PostScript language documents:

- **prologue and script:** When a document is divided into a prologue and script (with the **%%EndProlog** comment and/or the **%%BeginProcSet** comments), there must be nothing *executed* in the prologue (make only definitions) and *no definitions* in the script. The former of these is the more important; the prologue should always be such that it can be removed from a document and downloaded only once (permanently) into the printer. All subsequent documents that are downloaded with this prologue stripped out should still execute correctly.

- **%%Page:** If this comment is used within the body of a document, then the pages *shall not* have any inter-dependencies. Each page may rely on certain PostScript language operations having been defined by the document prologue, but it is not acceptable to have any "state" set in one page of a document that is relied upon by another page in the document. The reason for this is to allow document managers to physically rearrange the document's pages without affecting the execution of the document description, or even to print different pages in parallel on more than one printer. It is better to *omit* the **%%Page** comment than to employ it without conforming to this constraint.

- **line length:** To provide compatibility with a large body of existing software, a conforming PostScript language document description will not have lines exceeding 255 characters, excluding line termination characters. The intent is to be able to read lines into a 255-character buffer without overflow (Pascal strings are a common example of this sort of buffer). The PostScript interpreter imposes

no constraints as to where line breaks occur (even in string bodies and hexadecimal bitmap representations) and this level of conformance should not pose a problem for software development. Any document structuring comment which needs to be continued on another line to avoid violating this convention should use the "%%+" notation to indicate that a comment line is being continued. This notation may be used after any of the document comment conventions, but may only be necessary in those comments which provide a list of font names (such as **%%DocumentFonts**). Here is an example of its use:

```
%%DocumentFonts: Palatino-Roman Palatino-Bold
%%+ Palatino-Italic Palatino-BoldItalic Courier
%%+ Optima LubalinGraph-DemiOblique
```

- **line endings:** Lines may be terminated with any combination of carriage return or linefeed characters (decimal ASCII 13 and 10, respectively).

- **showpage** If the **showpage** operator is used with **save** and **restore**, the **showpage** shall occur *after* the page-level **restore** operation. The motivation for this is to be able to handily redefine the showpage operator to have side effects in the printer VM such as maintaining page counts for printing *n-up* copies on a single sheet of paper. If the **showpage** is executed within the confines of a page-level **save/restore**, then attempts to redefine showpage to perform extra operations will not work as intended.

- **copypage** If multiple copies of a page are needed, this should be specified by using the #copies convention. The showpage operator in all PostScript interpreter implementations will check for the existence of the name **#copies** in the current dictionary context, and will print as many copies of the page as this name reflects. A typical use of this is as follows:

```
/#copies 3 store
showpage
```

- **PostScript language operators to avoid.** There are some PostScript language operators that are intended for system-level jobs which are not appropriate in the context of a conforming document description. A restriction is placed on their use subject to very careful consideration: a document may be said to be *non-conforming* if any of these operations are used, but if they are used carefully for a very specific reason, they may not actually disturb reasonable handling by document managers. Operations listed with an asterisk (*) should especially be avoided. Others should be used only with extreme care. The risks of using these operators involve either rendering a document *device-dependent* or to unnecessarily inhibit constructive post-processing of document files for different printing needs (for example, embedding one PostScript language document within another). Here is a list of PostScript language operators that should be avoided by conforming document files:

exitserver*	quit*	erasepage*	copypage*
initgraphics*	grestoreall*	initmatrix*	initclip*
framedevice	banddevice	nulldevice	setdevice
setsccbatch	setmatrix	setscreen	settransfer

Note: There is a distinction between a PostScript language *document* and a *print job* (or *session*). A *document* is a data representation that may be transmitted, edited, stored, spooled, or otherwise processed. A print job is an active session with a PostScript printer, which may entail querying the printer, dynamically including (or removing) resources or fonts, even reordering the pages, and may or may not preserve the document exactly as it was originally stored. This is an important distinction, because a *conforming* document is just a document file, and in that context may be asked to conform to use constraints that are not appropriate for an actual PostScript language print job (for instance, the restriction on use of device setup procedures). *This affects primarily programs which act as both document composition programs and as printing managers.* If a job is actually printed (a connection is opened with a printer), then the printing session will need to perform certain document management tasks like querying the printer, but if the PostScript language file is saved to disk instead of being sent to the printer (a useful feature of any document manager) then the file should conform to the Document Structuring conventions (which may mean stripping out all queries, device-dependent PostScript language code, etc.).

4.1 PARSING RULES

Here are a few explicit rules that should apply when parsing these comments (and which should be observed when creating conforming documents). There are also some general rules specified in the previous section on conforming documents that should be followed.

- Many comments have a colon separating the comment from its arguments. This colon is not present in all comments (witness **%%EndProlog**) and should be considered part of the comment itself for parsing purposes. It is *not* an optional character.

- Comments with arguments (like **%%Page**) should have a space separating the colon from the first argument. Unfortunately, due to existing software, this space must be considered optional.

- "white space" characters within comments may be either spaces or tabs (decimal ASCII 32 and 9, respectively).

- Comments are case-sensitive, as are all of the arguments following a comment.

- The character set for comments is limited to printable ASCII characters. The comments themselves will only contain alphabetic characters (with the exception of the **?** used to introduce query comments). The arguments may include any character valid in the PostScript language character set, especially where procedure names, font names, and strings are represented.

- When looking for the **%%Trailer** comment (or any **(atend)** comments), make sure to allow for *nested* documents. Observe **%%BeginDocument** and **%%EndDocument** comments, as well as **%%BeginBinary** and **%%EndBinary**.

- The header comments can be explicitly terminated by an instance of **%%EndComments** or implicitly by any line that does not begin with **%%**.

- Lines should never exceed 255 characters, and may be terminated with any combination of carriage return and linefeed characters (decimal ASCII 13 and 10).

- The order of some comments within the document is significant, but within a section of the document, they may appear in any order (for example, within the header section, **%%DocumentFonts**, **%%Title**, and **%%Creator** may appear in any order).

4.2 USING THE DOCUMENT STRUCTURING CONVENTIONS

The Adobe Document Structuring Conventions are designed to facilitate communication between document composition systems and network document handlers. To some extent this will be installation-dependent. Ideally, a document composition system should be able to compose a document without regard to available resources (for instance, paper sizes and font availability), and should be able to rely on a document management system at printing time to determine the availability of the resources and to provide the user with reasonable alternatives.

In keeping with this, the Structuring Conventions contain a certain amount of redundancy. In particular (and as an example), there are two philosophically distinct ways in which a paper size might be specified: in one, the document composition system will *trust* its environment to handle the resource requirements appropriately, and will merely specify what its particular requirements are. On the other hand, a document composer may not know what the network environment holds, and therefore will *include* all the necessary resources (and perhaps printer-specific PostScript language code), but will delimit these included resources or code segments in such a way that a document manager may recognize and replace them. It is up to the software implementor to determine which of these methods is appropriate for a given environment. In some cases, both may be used.

These two distinct philosophies for using the Structuring Conventions are mirrored in the comments themselves. Many of the new Structure comments provide *BEGIN* and *END* constructs intended to identify any printer-specific elements of a document. The document itself should thus be impervious to the presence (or absence) of a document manager or spooler. Many of the Requirement Conventions provide a mechanism to *specify* a need for some resource or printer-specific feature, but to leave the invocation of that feature up to the document manager. This is an example of complete network cooperation, where a document can forestall some printing decisions and pass them to the next layer of document management. In general, this latter approach is the preferred one.

4.3 MINIMALLY REQUIRED COMMENTS

In the Version 1.0 conventions, there were several comment conventions that were *required* in order to minimally conform to that version of the specification. These were **%%DocumentFonts, %%EndProlog, %%Page,** and **%%Trailer**. In version 2.0, the rules are being changed somewhat. *There are no longer any* ***required*** *comments*. All the comments are "optional," in the sense that they may not be appropriate in a given situation, and therefore need not be employed. However, if a comment is used, then it must be used according to this specification. For example, the **%%BoundingBox** comment should be employed if it is appropriate (for instance, if the document is to be used as an illustration in another document), but it is not required if the PostScript language file does not make marks on the page. Similarly, the **%%DocumentFonts** comment is not required if no fonts are used. There are no comments which are strictly required since there are so many different types of PostScript language documents possible. The rule is to *make sure to use them correctly* if you use them. The secondary rule is that you should use as many of

these structuring conventions as are appropriate, as they will greatly enhance the ability to use the document in many environments, under many document management systems.

4.4 EXISTING VERSION 1.0 STRUCTURE COMMENTS

PostScript language page descriptions conforming to Version 1.0 of the comment conventions look something like the following example:

```
%!PS-Adobe-1.0
%%Title: title
%%Creator: text
%%CreationDate: text
%%For: text
%%DocumentFonts: (atend)
%%Pages: pagecount
%%BoundingBox: LLx LLy URx URy
%%EndComments

%%EndProlog
%%Page: label 1
%%PageFonts: fontname fontname

%%Trailer
```

5. STRUCTURE COMMENTS

What follows in this document is a list of the comment conventions which comprise Version 2.0 of the Adobe Document Structuring Conventions. All of the existing Version 1.0 conventions are included here, both as a reference and because they are technically also part of the full Version 2.0 specification. These extensions are designed to provide the maximum amount of information and flexibility to a document management system, and can be ignored by any software that does not know how to interpret them. The structure comments are grouped into three types of conventions:

- **Header Comments:** these occur once in a document, and may be thought of as a "table of contents."

- **Body Comments:** these may appear anywhere in a document, and generally consist of *BEGIN\END* constructs to delimit specific ingredients of a document file (such as fonts).

- **Page Comments:** these are page-level structure comments which are analogous to the header comments in purpose.

All of the comments are listed in alphabetical order in the Index at the end of this document for reference. Version 1.0 comments are flagged with an asterisk (*).

5.1 HEADER COMMENTS

The first set of Structure Comments are historically called "header" comments. That means that they appear first in a document file, before any of the executable PostScript language code (and before the prologue). In order to simplify a document manager's job in parsing these header comments, there are two rules that apply to them:

- The first rule is that if there is more than one instance of a header comment in a document file, *the first one encountered* is considered to be the truth. The reason for this is to simplify nesting documents one within another without having to remove the header comments.

- The second rule is that *header comments should be contiguous*. That is, if a document manager comes across a line that does not begin with **"%%"**, then it may quit parsing for header comments. This is important, and easy enough to adhere to. The comments may also be *explicitly* ended with the **%%EndComments** convention *(see below)*.

Note that all instances of lines beginning with **"%!"** *after the first instance* are ignored by document managers, although to avoid confusion this notation should not appear twice within the block of header comments.

%!PS-Adobe-2.0

This comment differs from the previous **%!PS-Adobe-1.0** comment only in version number. It indicates that the PostScript language page description fully conforms to Version 2.0 of the Adobe Structuring Conventions. This comment must occur as the *first* line of the PostScript language file.

There are three *keywords* that may follow this comment on the same line. They are designed to flag the entire print job as a particular *type* of job so that document managers may immediately switch gears to some appropriate mode of processing. Currently the following types of jobs are recognized:

- **EPS** files. These are known as "encapsulated PostScript files," and primarily are PostScript language files which produce illustrations. The format is designed to facilitate including these files in other documents. The exact format of these EPS files is described in another document. The version number of the EPS format may also be specified here (see example below).

- **Query** job. This indicates that the entire job consists of PostScript language queries to which replies are expected. See also the discussion of this under the **Query Comments** section of this document.

- **ExitServer** job. This flags a job that will execute the PostScript language **exitserver** command to permanently register the contents in the PostScript language device. This is needed by some document managers to effectively handle these special jobs. See also the discussion under **%%BeginExitServer**.

The format for these comments is as follows:

```
%!PS-Adobe-2.0 EPSF-1.2
%!PS-Adobe-2.0 Query
%!PS-Adobe-2.0 ExitServer
```

%%Title: text*

This comment provides a text title for the document that is useful for printing banner pages and for routing or recognizing documents. This comment is part of the existing Version 1.0 structuring conventions, as the "*" notation indicates.

%%Creator: text*

Indicates the document creator (usually the name of the document composition software).

%%CreationDate: text*

Indicates the time that the document was created. The time is expressed as a text string, and can be in any format. It is meant to be used purely for informational purposes, such as printing on banner pages, and need not be in a standardized form.

%%For: text*

Indicates for whom the document is intended. This is frequently the "user name" of the individual who composed the document (as determined by the document composition software). This can be used for banner pages or for routing the document after printing.

%%Routing: text

This comment provides information about how to route a document back to its owner after printing. It may contain information about mail addresses or office locations, at the discretion of the system administrator.

%%ProofMode: keyword

This comment provides information about the level of accuracy that is required for printing. It is intended to provide guidance to the printing manager for appropriate tactics when error conditions arise or when resource shortages are encountered. There are three modes which are currently defined:

TrustMe
Substitute
NotifyMe

These modes may be thought of as instructions to the printing manager. If the printing manager detects a resource shortage (such as a missing font or paper size), it should take action based on these proof modes.

The **TrustMe** mode means that the printing manager should take *no* special action. The intent is that the document formatting programs (or the user) knows more than the printing manager. For example, fonts may be available on a network font server that the printing manager does not know about. Even with something like **%%IncludeFile**, if the **%%ProofMode** is **TrustMe**, the printing manager should proceed, even if a file cannot be found. The assumption is that the document itself can compensate for the file not being included.

The **Substitute** mode means that the printing manager should do the best it can to supply missing resources with alternatives. This may mean similar fonts (or bitmapped screen fonts), scaling pages (or tiling) when paper sizes are not available, etc.

The last mode, **NotifyMe**, means that the file should not be printed at all if there are any mismatches or resource shortages noted by the printing manager. An example of this might be expensive printing on a color printer. If the correct font is not available, then the user probably does *not* want a default font.

These modes are intended for the printing manager to consider *before* it actually prints the file, based on its own research into available fonts, paper sizes, and other resources. If the file is printed and an error occurred, that is a separate issue.

%%BoundingBox: LLx LLy URx URy*

This specifies the bounding box that encloses all the marks painted on the virtual page that encompasses all pages of a document. That is, it must be a "high water mark" in all directions for marks made on any page. All four values must be integers; (LLx, LLy) and (URx , URy) are the coordinates of the lower left and upper right corners of the bounding box in the *default user coordinate system.* The specification **(atend)** is permitted, in which case the deferred comment must occur after the **%%Trailer** comment. See the *PostScript Language Reference Manual, Appendix C* for more details on the **(atend)** notation.

%%Pages: integer [page order]*

This comment takes an integer argument that represents the total number of pages that the document will print, and optionally an argument to indicate the page order. The page count should correspond to the number of pages that would be produced by the execution of **showpage** or **copypage**. If the document produces *no* pages (for instance, if it represents an included illustration which does not use **showpage**), then the page count should be 0. The page order argument is intended to help document managers to determine the order in which the pages are presented in the document file, which in turn will allow a document manager to optionally reorder the pages. It will consist of any of three potential values: **-1, 0, or 1**. A value of **-1** is taken to mean that the pages are in *descending (or n-1)* order, a value of **1** means *ascending (or 1-n)* order, and a **0** represents a *special* order (for instance, signature order). The distinction between a page order code of **0** and no page order at all is as follows: in the absence of the page order argument, any assumption can be made about the page order, and any reordering of the pages is permitted; however, if the page order argument is present and **0**, then the pages should be left intact in the order given. See also the **%%Page** comment. The specification '**(atend)**' is permitted.

%%Requirements: keyword keyword ...

This comment is used to indicate document requirements such as duplex printing, hole punching, collating, sorting, or other physical document processing needs. The **keyword** parameter may correspond to a specific printer feature, or it may be used to designate processing requiring "human intervention." A recommended initial set of values for this comment follows:

duplex	simplex	punch3	punch5
colorprinter	fold	staple	

These keywords may be used together in any combination, or others may be added to compensate special document processing needs. It is up to the document manager to ensure that these requirements are met.

Note: For documents that require **duplex** printing, some printers will be able to handle this directly, while other printers may require human intervention to feed the paper. The document management system must make a decision based on this keyword as to whether or not a printer can be found that will support duplex printing (based on the information provided in the Printer Description files or through querying the printer). If not, some reordering of the document pages may be required. In this case, the **%%Pages** comment should be consulted to provide more information about the existing ordering of pages.

5.2 FONT AND FILE MANAGEMENT

Please read the following section carefully. There are several comments which seem to "interact" with one another, especially within the font handling area. Keep in mind that the **%%DocumentFonts** comment is always the *superset* of any other font comments, and should always contain a list of all fonts used by the documents. Other comments comprise special *subsets* of this list which may be used for font management. There are analogous comments for file and prologue management, as well.

%%DocumentFonts: fontname fontname ...*

This comment is already present in Version 1.0 of the comment conventions, but is included here for reference. It indicates that all the fonts listed are used by the print job. In particular, there will be at least one invocation of the **findfont** operator for each of the font names listed. The notation **(atend)** is permitted here, as specified in the version 1.0 format. If **(atend)** is used, then the list *must* be supplied in the **%%Trailer** section of the document, in the same format as it would normally have appeared in the header comments. See also **%%DocumentNeededFonts** and **%%DocumentSuppliedFonts**.

%%DocumentNeededFonts: fontname fontname ...

This comment (if present) will provide a list of PostScript font names that are *required* by the document (and *not* contained within the document file). It is assumed that there will be at least one corresponding instance of the **%%IncludeFont** directive for each font listed in this section. The purpose of this comment is to allow spoolers to make a decision as to whether the file needs to be parsed to provide font files within the body of the document. See also **%%DocumentFonts** and **%%DocumentSuppliedFonts**.

%%DocumentSuppliedFonts: fontname fontname ...

This convention will provide a list of all the font files that have been provided *within the document print file* as downloaded fonts. There will be at least one corresponding **%%BeginFont** and **%%EndFont** pair within the document description for each of the listed font names. See also **%%DocumentFonts** and **%%DocumentNeededFonts**.

%%DocumentProcSets: name version revision ...

This comment provides a list of all the ProcSets referenced within the document. It is similar to the **%%DocumentFonts** comment in use. The notation **(atend)** is permitted here, as specified in the version 1.0 format. If **(atend)** is used, then the list *must* be supplied in the **%%Trailer** section of the document, in the same format as it would normally have appeared in the header comments. The revision field

should be taken to be upwardly compatible with procedure sets of the same version number. That is, if "**myprocs 1.0 0**" is requested, then "**myprocs 1.0 2**" should be compatible, although the converse (backward compatibility) is not necessarily true.

%%DocumentNeededProcSets: name version revision ...

This comment indicates that the document requires the ProcSets listed (for more information, see **%%BeginProcSet**. The *name*, *version*, and *revision* fields should uniquely identify the ProcSet. If a version numbering scheme is not used, these fields should still be filled with a "dummy" value (such as **""** or **0**). These comments are always used with the **%%IncludeProcSet** comment.

%%DocumentSuppliedProcSets: name version revision ...

This comment indicates that the document contains (or supplies) the ProcSets listed. This corresponds to later use of the **%%BeginProcSet** and **%%EndProcSet** comments. The *name*, *version*, and *revision* fields should uniquely identify the ProcSet.

%%DocumentNeededFiles: filename filename ...

Lists the included files needed by a document description. Each file mentioned in this list will appear later in the document as the argument of a **%%IncludeFile** comment.

%%DocumentSuppliedFiles: filename filename ...

Lists the files included in a document description. Each file mentioned in this list will appear later in the document in the context of a **%%BeginFile** and **%%EndFile** comment construct.

NOTE:
There are now header comments which make reference to the fonts, files, or procedure sets either needed or supplied by a document. These must be carefully used to provide the correct information to a document management program. In particular, they have a specific relationship to each other. The ***%%DocumentNeeded*** *comments are intended to help a document manager decide whether further parsing of a document file is necessary to provide these included resources. They should only be used if the corresponding* ***%%Include*** *comment is used. The* ***%%Document-Supplied*** *comments are extra information for spoolers which wish to "cache" the resources, and provide helpful directories of the resources contained within the print file. They are optional, while the* ***%%DocumentNeeded*** *comments are not optional if the* ***%%Include*** *mechanism is employed. Notice that the* ***%%DocumentFonts*** *comment is special. It existed in Version 1.0, and still is valid, but is treated somewhat differently than the others. The reason for this is that a document may use a font which is believes is permanently loaded into the printer (or in ROM), and therefore need not be explicitly loaded by a server through the* ***%%Include*** *mechanism, nor need be supplied in the document. If mention is made of a font in the* ***%%DocumentFonts*** *line which is determined not to be present in the printer, it should be provided by the font manager, and may be loaded at any point prior to its use by the document file (typically these will be downloaded fonts which are downloaded prior to the document file, but which are not permanently resident in the printer.*

NOTE also:
These comments which provide lists of font or file names may grow longer than the 256 character line length convention. In this case it is permissible

(and recommended) to employ the %%+ comment to indicate the continuation of a comment on the following line (see above).

These conventions (since they are *document level* comments) should be present in the header of a document, if they are used. That is, they should appear before the first instance of **%%EndComments**, or the first non-comment line (which is interpreted as an implicit **%%EndComments**).

%%DocumentPaperSizes: size size ...

Lists all the paper sizes needed by this print job. These sizes will correspond to the nominal size names as specified in the Printer Description file format keywords (see the discussion under **%%BeginPaperSize** for more information).

%%DocumentPaperForms: formname formname ...

Lists all the forms needed by this print job. These form names will correspond to notions of particular forms as understood by a document manager. They might, for instance, include company letterhead, tax forms, or other special forms. They will be explicitly called for by each page that requires them through the use of the **%%PaperForm** comment *(see)*.

%%DocumentPaperColors: colorname colorname ...

Indicates particular colors of paper required by a print job. For example, colorname might be **"goldenrod."** Again, this comment will be supported by corresponding **%%PaperColor** comments for the particular pages requiring these paper colors.

%%DocumentPaperWeights: integer integer ...

Indicates a weight of paper (in *pounds* or other measure) expressed as an integer value. For example, the value might be **20**. There should be at least one instance of the **%%PaperWeight** comment for each weight listed here. See also **%%PaperWeight**.

%%DocumentPrinterRequired: printer product [version revision]

This directs that the PostScript language page description is intended for a particular printer, identified by its network printer name and/or product string and optionally by its version and revision strings, as defined by the printer's PPD file (and as returned by the **product**, **version**, and **revision** PostScript language operators. This may be used to request a particular printer in a highly networked environment where that printer may be more convenient, to override spooler defaults, or perhaps if the PostScript language file itself contains printer-specific elements. This last case should rarely be necessary, as most documents that require particular features of a PostScript printer can provide Requirement Conventions indicating a need for that feature, rather than requiring a particular brand of printer. Then, if other printers are available which have the necessary features, the page may still be printed as desired. Two examples follow:

```
%%DocumentPrinterRequired: "Local Print" (LaserWriter) 23.0
%%DocumentPrinterRequired: "" "(Linotype)"
```

%%DocumentProcessColors: keyword keyword ...

This comment marks the use of process colors within the document. Process colors are defined to be *cyan*, *magenta*, *yellow*, and *black*. These four colors are indicated in this comment by the keywords **Cyan**, **Magenta**, **Yellow**, and **Black**. This comment is used primarily when producing color separations. See also **%%PageProcessColors**. The **(atend)** conventions is allowed.

%%DocumentCustomColors: name name ...

This indicates the use of custom colors within a document. These colors are arbitrarily named by an application, and their CMYK or RGB approximations are provided through the **%%CMYKCustomColor** or **%%RGBCustomColor** comments within the body of the document. The names are specified to be any arbitrary PostScript language string except (Process Cyan), (Process Magenta), (Process Yellow), and (Process Black), which need to be reserved for custom color implementation by applications. The **(atend)** specification is permitted.

%%EndComments*

This indicates an explicit end to the header comments of the document. Any line that does not begin with "**%%**" also constitutes an implicit end of the header comment block.

%%EndProlog*

This comment marks the boundary between the document's prologue and its script. This convention is already widely observed, and should be included in all documents which have a distinct prologue and script. *Note:* breaking a document into a prologue and a script is conceptually important, although not all document descriptions will fall neatly into this model. If your document represents free form PostScript language fragments which might entirely be considered a "script", you should still include the **%%EndProlog** comment, although there may be nothing in the prologue part of the file. This will effectively make the entire document a "script." Be careful, still, in employing the **%%Page** comment, because document managers may make assumptions about page reversal based on these comments. See the discussion of **%%Page** in the section entitled *Page Structure and Requirement Comments* for more information.

5.3 BODY COMMENTS

Body comments may appear anywhere in a document. They are designed to provide structural information about the document file's organization, and should match any related information provided in the header comments section.

%%BeginSetup
%%EndSetup

These comments delimit the part of the document that does device setup for a particular printer or document. There may be procedures for setting page size, invoking manual feed, establishing a scale factor (or "landscape" mode), or other document-level setup procedures. There may also be general initialization code such as executing **save** or setting some aspects of the graphics state. These delimiters are useful for a document manager to be able to detect an entire section of initialization

procedures. If present, this comment will fall after the **%%EndProlog** directive but before the first **%%Page** (In other words, it is *not part of the prologue.* It should be in the first part of the script, before any pages are specified.

Note: Specification of some of the page-level comments are permitted within this section of the document, such as **%%PaperColor.** If found in the document setup section, then they are taken to be in effect for the entire document, rather than for a single page. See the discussion under **Page Comments** in the **Resource Requirements** section of this document.

%%BeginDocument: name [version] [type]
%%EndDocument

These comments are used to delimit an *entire file* that is imported as part of another PostScript language document or print job. This comment is required to allow multiple occurrences of the **%%EndProlog** and **%%Trailer** comment in the body of a document. Any existing document file that is embedded within another document file *must* be surrounded by these comments. The **version** and **type** fields are optional, and if used should provide extra information for recognizing specific documents.

Note: When a document file is printed, usually a certain amount of PostScript language code is added to the file that may deal with font downloading issues, with paper sizes, or other aspects of printing once a printer has been selected for the document. At that stage, the printing manager should embed the original document (along with all the structuring conventions which may fall within that file) within **%%BeginDocument** and **%%EndDocument** comments.

%%BeginFile: filename
%%EndFile

The enclosed segment resulted from having included a file in the PostScript language stream. The file server component of a document manager may perhaps extract a copy of this file for later use by the **%%DocumentFiles**, **%%PageFiles** or **%%IncludeFile** comments. The file name will usually correspond to the original disk file name on the host system.

%%BeginFont: fontname [printername]
%%EndFont

These comments delimit a downloaded font. The font server component of a document manager may remove the font from the print file (for instance, if the chosen printer already has the font resident), or it may simply keep a copy of it for later use by the **%%IncludeFont** comment. The **fontname** field should be the valid PostScript language name of the font (as used by the **definefont** operator), and the optional **printername** field may contain the network name of the printer, in an environment where fonts may be tied to particular printers.

%%BeginProcSet: name version revision
%%EndProcSet

The PostScript language code enclosed by the **%%BeginProcSet** and **%%EndProcSet** comments typically represents some subset of the document prologue. The prologue may be broken down into many sub-packages, or *procedure sets*, which may define groups of routines appropriate for different imaging

requirements. These individual "ProcSets" are identified by name, version, and revision numbers for reference by a document management system. A document manager may choose to extract these ProcSets from the print file in order to manage them separately for a whole family of documents. Note that an entire document prologue may be an instance of a ProcSet, in that it is a body of procedure definitions used by a document description file. (See also the **%%DocumentProcSets, %%IncludeProcSet** and **%%IncludeFile** comments). The **name, version**, and **revision** fields should uniquely identify the ProcSet. The **name** may consist of a disk file name or it may use a PostScript language name under which the prologue is stored in the printer. In any case, these fields are used to identify the ProcSet to the document manager. See also the **%%?BeginProcSetQuery** comment convention, with which one may query the PostScript server for the prologue name and version fields.

Notice that this does not replace the **%%EndProlog** comment used in the original (version 1.0) document structuring conventions. It should still be used in the same manner, although now it is optional to provide the additional **%%BeginProcSet** and **%%EndProcSet** comments as well. A document manager may assume that the document prologue consists of everything from the beginning of the print file through the **%%EndProlog** comment, which may encompass several instances of the **%BeginProcSet/%%EndProcSet** comments.

%%BeginBinary: bytecount
%%EndBinary

These comments are designed to provide information about embedded bodies of binary data. When a PostScript language document file is being parsed, encountering raw data can complicate the parsing process immensely. These comments are designed to allow a document manager to effectively ignore anything encapsulated within these comments. In fact, the data may be anything, including hexadecimal representations of binary data or any other data which should be ignored by a document manager. Since a bytecount field is provided, the material may be skipped even if it contains embedded comment strings. If a PostScript language print job embeds binary data, it should be encapsulated within these **%%BeginBinary** and **%%EndBinary** comments so that parsing software can safely ignore it.

Note: In order to read data directly from the input stream in the PostScript language (using **currentfile**, for instance), it is necessary to invoke a procedure followed *immediately* by the data to be read. If the data is embedded in the **%%BeginBinary/%%EndBinary** construct, then those comments will effectively be *part of the data*, which is typically *not* what is desired. In order to avoid this problem, the procedure invocation should fall *inside* the comments (even though it is not binary), and the bytecount should reflect this, so that it can be skipped correctly. Make sure to allow for carriage returns, if any. In the example below (taken from the *PostScript Language Reference Manual*, there are 131072 bytes of binary data, but the call to the **image** operator is also encompassed within the **%%BeginBinary** and **%%EndBinary** comments. The resulting byte count includes 6 additional bytes, for the string "image" plus the newline character. *Note:* The entire **%%BeginBinary** comment line should be read before acting on the byte count.

```
/picstr 256 string def
25 140 translate
132 132 scale
256 256 8 [256 0 0 -256 0 256] { currentfile picstr readhexstring pop }
```

```
%%BeginBinary: 131078
image
4c47494b3187c237d237b137438374a
213769876c8976985a5c98767587575 % 131072 bytes binary
%%EndBinary
```

%%BeginPaperSize: sizename %%EndPaperSize

These comments delimit a PostScript language sequence that will invoke a particular paper size. This comment is provided specifically for environments where simplicity is needed; that is, where the full functionality and generality of the Printer Description files keyword replacement mechanism is not necessary. The **%%BeginFeature** and **%%Feature** comments will allow more specific control over invoking features as found in the Printer Description files. For the purposes of **%%BeginPaperSize** and **%%PaperSize** (see below under *Resource Requirements*), any valid method of securing that particular paper size is acceptable (whether by invoking a particular input bin, a paper tray, or by setting the page size). The appropriate keyword for the nominal paper size should be used as specified in the Printer Description file. Examples of these are: **"Letter, Legal, A3, A4, A5, B4, B5, Ledger, Statement, Tabloid.**" An example follows:

```
%%Page: 1 1
%%BeginPaperSize: Ledger
 statusdict begin ledgertray end
%%EndPaperSize
```

%%BeginFeature: featuretype [option] %%EndFeature

These comments are used to delimit a PostScript language code segment that will invoke a particular feature on a PostScript printer as specified in the Printer Description File format (described in another document). The **featuretype** will correspond to one of the keywords in the Printer Description file, and the **"featuretype option"** sequence should be exactly as found in the Printer Description file, in order to cooperate effectively with these conventions. The **%%BeginFeature** and **%%EndFeature** comments should delimit any PostScript language fragments that invoke a printer-specific feature on a printer. A document manager may choose to replace the enclosed PostScript language code with the proper sequence of instructions if the document is sent to a different printer than originally intended. This is, in a sense, the inverse of the **%%Feature** comment (see the *Resource Requirements* section of this document), which indicates that the specified printer feature needs to be invoked by the document manager at that position in the print file. The **%%Begin/EndFeature** comments are used when the printer feature is being explicitly invoked for a particular printer, and are intended to convey information about this decision to the document manager. This comment can be used instead of using the **%%BeginPaperSize** comment (or even in addition to, if nested properly). Three examples follow. The first example illustrates a PostScript language file in which an assumption is made about the presence of a file system. If a document manager were to send this document to a printer which did have a file system, it might actually replace the encapsulated "false" with the value of that feature (from the Printer Description file) for the chosen printer:

```
% example 1
%%BeginFeature: *FileSystem
 false
%%EndFeature
{ %ifelse
  (logfile) (w) file dup (message) writestring closefile }{ %else
  (message) print flush
} ifelse


% example 2
%%BeginFeature: *InputSlot Upper
 statusdict begin 1 setpapertray end
%%EndFeature


% example 3
%%BeginPaperSize: Ledger
%%BeginFeature: *PaperTray Ledger
 statusdict begin ledgertray end
%%EndFeature
%%EndPaperSize
```

%%BeginExitServer: password
%%EndExitServer

These comments delimit the PostScript language sequence that will cause the rest of the file to be registered *outside* the normal server loop **save/restore** context. This convention is used to flag any code that will set up or execute the **exitserver** instruction, so that it can be caught and removed as necessary by a document manager. It may be used in conjunction with the **%%EOF** requirement convention to pinpoint where an end-of-file indication should be sent by the document manager. *Please see also the discussion under* **%!PS-Adobe-2.0** which can be found in the **Structure Comments** section of this document. PostScript language jobs which employ **exitserver** should be specially flagged by use of the **%!PS-Adobe-2.0 ExitServer** notation. An example of its appropriate employment follows:

```
%!PS-Adobe-2.0 ExitServer
%%CreationDate: Mon Feb 2 10:34:36 1987
%%EndComments
%%BeginExitServer: 000000
  serverdict begin 000000 exitserver
%%EndExitServer
% ...
%%EOF
```

%%BeginProcessColor: keyword
%%EndProcessColor

The **keyword** here is either **Cyan**, **Magenta**, **Yellow**, or **Black**. During color separation, the code between these comments should only be downloaded during the appropriate pass for that process color. Intelligent printing managers can save considerable time by omitting code within these bracketing comments on the other three separations. Extreme care must be taken by the document composition software to correctly control overprinting and knockouts if these comments are employed, since the code may or may not be executed.

%%BeginCustomColor: keyword
%%EndCustomColor

The **keyword** here is any PostScript language string except (Process Cyan), (Process Magenta), (Process Yellow), and (Process Black). During color separation, the code between these comments should only be downloaded during the appropriate pass for that custom color. Intelligent printing managers can save considerable time by omitting code within these bracketing comments on the other three separations. Extreme care must be taken by the document composition software to correctly control overprinting and knockouts if these comments are employed, since the code may or may not be executed.

%%CMYKCustomColor: cyan magenta yellow black *keyword*

This provides an *approximation* to the custom color specified by ***keyword***. The four components of **cyan**, **magenta**, **yellow**, and **black** must be specified as numbers from 0 to 1 representing the percentage of that process color. These numbers are exactly analogous to the arguments to the **setcmykcolor** PostScript language operator. The ***keyword*** follows the same custom color naming conventions for the **%%DocumentCustomColors** comment.

%%RGBCustomColor: red green blue *keyword*

This provides an *approximation* to the custom color specified by ***keyword***. The three components of **red**, **green**, and **blue** must be specified as numbers from 0 to 1 representing the percentage of that process color. These numbers are exactly analogous to the arguments to the **setrgbcolor** PostScript language operator. The ***keyword*** follows the same custom color naming conventions for the **%%DocumentCustomColors** comment.

%%Trailer*

This comment should occur once at the end of the document *script*. Any post-processing or "cleanup" should be contained in the *trailer* of the document, which is anything that follows this **%%Trailer** comment. Additionally, any of the document-level structure comments that were *deferred* by using the **(atend)** convention (permissible in **%%DocumentFonts, %%Pages, %%PageFonts, %%BoundingBox**, etc.) should be mentioned in the trailer of the document (*i.e.* after the **%%Trailer** comment).

Note: When entire documents are embedded in another document file, there may be more than one **%%Trailer** comment as a result. In order to avoid ambiguity, embedded documents should be delimited by the **%%BeginDocument** and **%%EndDocument** comments.

5.4 PAGE COMMENTS

The page-level comments will usually occur once for each page (with the exception of the **%%BeginObject** and **%%EndObject** comments), and provide information about that page's requirements or structure.

%%Page: label ordinal*

This comment is used to mark the beginning of the PostScript language code describing a particular page. It requires two arguments: a page label and a sequential page number. The label may be anything, but the ordinal page number should reflect the position of that page within the body of the PostScript language file (and should start with **1**, not **0**). This convention is widely used and very important. You *must* make sure that your pages do not rely on each other, but only on definitions made in the prologue of the document. A document manager should be able to *physically rearrange* the contents of the print file into a different order based on the **%%Page** comment (or the pages may be printed in parallel, if desired). Please do *not* include these comments if the document does not conform to these use constraints. See also the **%%Pages** comment.

%%PageFonts: fontname fontname ...*

Indicates the names of all fonts used on the current page. The notation **(atend)** is permissible here, but the list must be provided before the next instance of **%%Page** or **%%Trailer**. In other words, it must be supplied by the end of that page.

%%PageFiles: filename filename ...

Indicates the names of all files used on the current page. This should be used only if file inclusion is required of the document manager (i.e. there are subsequent instances of the **%%IncludeFile** directive on that particular page).

%%PageBoundingBox: LLx LLy URx URy

This is a body comment, in that it will appear in the body of a document, analogous to the **%%PageFonts** comment. It specifies the bounding box that encloses all the marks painted on that particular page (not the whole document—see the **%%BoundingBox** convention). All four values must be integers; (LLx, LLy) and (URx , URy) are the coordinates of the lower left and upper right corners of the bounding box in the *default user coordinate system.* The specification **(atend)** is permitted, in which case the deferred comment must occur before the next **%%Page** or **%%Trailer** comment.

%%PageProcessColors: keyword keyword ...

This comment marks the use of process colors within the page. Process colors are defined to be *cyan*, *magenta*, *yellow*, and *black*. These four colors are indicated in this comment by the keywords **Cyan**, **Magenta**, **Yellow**, and **Black**. See also **%%DocumentProcessColors**. The **(atend)** conventions is allowed.

%%PageCustomColors: name name ...

This indicates the use of custom colors within a document. These colors are arbitrarily named by an application, and their CMYK or RGB approximations are provided through the **%%CMYKCustomColor** or **%%RGBCustomColor** comments within the body of the document. The **(atend)** conventions is allowed.

%%BeginPageSetup
%%EndPageSetup

These directives are analogous to the **%%BeginSetup** comments, except that they will fall within the body of a document, right after a **%%Page** comment. Use them to delimit areas where manual feed is set, margins are established, landscape mode is chosen, particular paper colors are invoked, etc.

%%BeginObject: [name] [code]
%%EndObject

These comments are used to delimit individual graphic elements of a page. In a context where it is desirable to be able to recognize individual page elements, this comment will provide a mechanism to label and recognize them at the PostScript language level. This can be especially useful for a document printing system that may have the capability of only printing selected objects in a document (or on a page). For instance, the **code** field of this comment may be used to represent *proofing levels* for a document *(or color separations).* Then the printing manager may be requested to "only print those objects with proofing levels less than, say, 4. This can save printing time when proofing various different elements of a document. It can also be useful in systems which allow PostScript language segments to be parsed in and reedited, to allow convenient grouping and categorization of graphic page elements. In a document production system which is highly object-oriented, this comment is strongly recommended.

%%PageTrailer

This comment marks the end of a page. Any page comments that may have been deferred by the **(atend)** convention should follow the **%%PageTrailer** comment.

6. RESOURCE REQUIREMENTS

These comments may appear anywhere in a document, and indicate that the named resource (whether it is a font, a disk file, or other) should be included in the document at the point where the comment is encountered. *These should not appear in the body of a document without a corresponding comment in the header of the document indicating that the files are required by the document as a whole.*

%%IncludeFont: fontname

Indicates that the specified font must be included at this point in the document by the document manager. The **fontname** specified should be the correct PostScript language name for the font (without the leading slash). Due to the presence of multiple **save/restore** contexts, a font server may have to supply a specific font more than once within a single document, and should do so whenever this comment is encountered.

%%IncludeProcSet: name version revision

This is a special case of the more general **%%IncludeFile** directive. It requires that a PostScript language ProcSet with the given name, version, and revision be inserted into the print file at the current position. If a version numbering scheme is not used, these fields should still be filled with a "dummy" value (such as **""** or **0**). See the **%%BeginProcSet** comment for more information.

%%IncludeFile: filename

Provides a directive that the specified file must be inserted by the document manager at the current position in the document.

%%ExecuteFile: filename

This is much like the **%%IncludeFile** directive except that it specifies that the included file is an *executable* PostScript language file, rather than perhaps a ProcSet or prologue file. This means that in all probability it will contain at least one instance of **showpage**, and means that it can (and should) be wrapped with a **save** and **restore**. This convention should usually be used to include whole PostScript language documents within other documents, and should be used whenever the enclosed file is entirely self-contained (and especially when it can be nested in a **save/restore** context without difficulty). In particular, illustrations and files which have no effect other than to make marks on a page are perfectly suited for the **%%ExecuteFile** convention.

%%ChangeFont: fontname

This comment is being *phased out* as of version 2.1 of this specification. It is not very useful, and is fraught with problems. Please don't use it or feel that you have to support it.

6.1 PAPER COMMENTS

Note: The following comments which request particular paper forms, colors, or sizes *may appear either in the document setup area or on a particular page.* If these comments fall within the document setup section of the document file, then they may be construed to be in effect for the entire print job. If they are found within the page-level comments for a page, then they should *only be in effect for that page.*

%%PaperForm: formname

This requests a particular form for printing. For example, a page may be printed on company letterhead, on an order form, or other particular form specified within a document handling system. There are no clearly defined values for **formname**; it may vary from one site to another.

%%PaperColor: colorname

The syntax is like **%%DocumentPaperColors**. This comment will appear in the body of a document description after an instance of **%%Page**, and will only remain in effect for that particular page.

%%PaperWeight: integer

The syntax is like **%%DocumentPaperWeights**. This comment will appear in the body of a document description after an instance of **%%Page**, and will only remain in effect for that particular page.

%%PaperSize: sizename

Specifies a particular page size (see **%%BeginPaperSize**) to be set. The **sizename** should be a simple keyword as found in the Printer Description file, and which basically consists of the nominal paper size expressed in lowercase letters (for example: **ledger, legal, a4**). This will only remain in effect for that particular page.

%%Feature: featuretype [option]

This comment specifies the need for a particular printer feature as specified in the Printer Description file format. Its use specifies a *requirement* that must be fulfilled by a document manager before printing (see also the discussion under **%%BeginFeature**). The document file may make the assumption that the **%%Feature** line in the file will be replaced by the appropriate PostScript language fragment from the appropriate Printer Description file, and the execution of the file may be contextually dependent upon this replacement. This offers a very powerful way of making a document behave differently on different printers in a device-independent manner. An example follows. This example will request that the appropriate page dimensions for B4 paper be inserted into the document, and will then use these for a call to the **setpageparams** operator, if it is present. The ***ImageableArea** keyword in the Printer Description file always produces four numbers representing the bounding box of the imageable region of a particular paper size. Two of these numbers can be used as arguments to **setpageparams**, as in the example. The entire invocation is contained within **%%BeginPaperSize** / **%%EndPaperSize** comments, and may be replaced by entirely different code for a different printer.

```
%%BeginPaperSize: A4
mark 595 842
%%Feature: *ImageableArea A4
 statusdict begin
 /setpageparams where {
   pop 0 0 setpageparams pop pop
 } if
 end
 cleartomark
%%EndPaperSize
```

%%EOF

This comment is used to request that an end-of-file indication be sent to the PostScript device. It may be used, for instance, with the **%%BeginExitServer** comment *(see above)*.

7. QUERY CONVENTIONS

A *query* is defined to be any PostScript language segment that will generate some returned information back to the host across the communications channel that is expected *before* a document can be formatted or created. This might result from the execution of any of the **=**, **==**, **print** or **pstack** operators, for instance. In particular, this definition covers information which is expected back from the PostScript printer for decision-making purposes. This might include the generation of font lists, inquiries as to the availability of resources, printer features, or the like.

All of the query conventions consist of a *BEGIN* and *END* construct, with the keywords reflecting the type of query. For all of them, the **%%?EndQuery** comment should include a field for a *default* value, which should be returned by document managers if they cannot understand or do not support query comments. The value of the default (see below for examples) is entirely application dependent, and can be used by an application to determine specific information about the spooling environment (if any), and to take appropriate default action.

Any print file which embeds PostScript language queries within the job should adhere very carefully to these query conventions, to allow the document to be spooled properly.

7.1 SPOOLER QUERY RESPONSIBILITIES

A spooler which expects to be able to interpret and correctly spool documents conforming to version 2.0 of the Adobe Document Structuring Conventions must, at a minimum, perform certain tasks in response to these query conventions. In general, the spooler must recognize the queries, remove them from the print stream, and send some reply back to the host. If a spooler cannot interpret the query, it is expected to return the value provided as the argument to the **%%?EndQuery** comment (see below). A query can be minimally recognized by the sequence **"%%?Begin"** followed by any number of characters (up to 256 maximum per line, by convention) through the end-of-line indication (the "%" is decimal ASCII 37, and the "?" is decimal ASCII 63). The end of the query will be delimited (minimally) by the sequence **"%%?End"** followed by some keywords and optionally by a colon **":"** (decimal ASCII 58) and the default response to the query (any text through end-of-line). A spooler should make attempts to recognize the full query keyword (like **%%?BeginProcSetQuery**) if it can, but it is obligated at least to respond to any validly formed query.

7.2 QUERY COMMENTS

%!PS-Adobe-2.0 Query

A PostScript language query must be sent as a separate job to the printer in order to be fully spoolable. This means that an *end-of-file* indication should be sent immediately after the query job. A job which is a query job should always begin with this **%!PS-Adobe-2.0 Query** convention, which further qualifies the file as being a special case of a version 2.0 conforming PostScript language file. A query job will contain only query comments, and need not contain any of the other standard structuring conventions. A document manager should be prepared to extract query information from any print file which begins with this comment convention. A document manager should fully parse a query job file until the EOF indication is reached.

Note: It is permissible to include more than one query in a single print job, but not to include queries within the body of a regular print job. It cannot be guaranteed that a print job with embedded queries will be handled properly by a document manager.

%%?BeginQuery: identifier
%%?EndQuery: default

These comments are very general-purpose, and may serve any function that is not adequately covered by the rest of this specification. The reader will notice that the query keywords in this section are very specific, since in order to understand and intelligently respond to a query, a document manager *must* semantically understand the query. Therefore specific keywords like **%%?BeginPrinterQuery** were used. When this generic **%%?BeginQuery** comment is encountered, a spooler may be forced to return the default value. The comment is included primarily for large installations which need to implement specific additional queries that are not covered here, and which will likely implement both the document composition software and the spooling software themselves.

%%?BeginPrinterQuery
%%?EndPrinterQuery: default

This comment delimits PostScript language code that will return information describing the printer's product name, version and revision numbers. The standard response consists of the printer's product name, version, and revision strings, each of which should be followed by a newline character, which should match the information in the printer's Printer Description file. This comment may be used also to identify the presence of a spooler, if necessary. In the following example the *default* response as represented in the **%%?EndPrinterQuery** line is the word **spooler**, which would be returned by spooling software that did not have a specific printer type attached to it:

```
%%?BeginPrinterQuery statusdict begin revision == version == productname == flush
end %%?EndPrinterQuery: spooler
```

%%?BeginVMStatus
%%?EndVMStatus: default

This comment delimits PostScript language code to return the state of the PostScript interpreter's virtual memory. The standard response consists of a line containing the results of the PostScript language **vmstatus** operator.

```
%%?BeginVMStatus vmstatus = = = flush
%%?EndVMStatus: unknown
```

%%?BeginFeatureQuery featuretype option
%%?EndFeatureQuery: default

This query provides information describing the state of some specified printer-specific feature, as defined by the printer's Printer Description file. The **featuretype** field identifies the keyword as found in the Printer Description file specification. The standard response will vary with the feature and is defined by the printer's Printer Description file. In general, the *keyword* associated with the feeture should be returned. In the example that follows, the Printer Description File keywords **True** or **False** are returned:

```
%%?BeginFeatureQuery: *InputSlot manualfeed
 statusdict /manualfeed known {
   statusdict /manualfeed get { (True) }{ (False) } ifelse
 }{
```

```
    (None)
} ifelse = flush
%%?EndFeatureQuery: Unknown
```

%%?BeginFileQuery: filename
%%?EndFileQuery: default

The PostScript language code between these comments causes the printer to respond with information describing the availability of the specified file. This presumes the existence of a file system that is available to the PostScript interpreter, which is not available on all implementations. The standard response consists of a line containing the file name, a colon, and either **Yes** or **No**, indicating whether or not the file is present. Look at the example below for the font query, which works the same way.

%%?BeginFontQuery: fontname fontname ...
%%?EndFontQuery: default

This provides a PostScript language query that should be combined with a particular list of font names being sought. It looks for any number of names on the stack, and will print a list of values depending on whether or not the font is known to the PostScript interpreter. The font names should be provided on the operand stack by the Document Manager, This is done by simply emitting the names, with leading slash "/" characters, before emitting the query itself.

To keep the Document Manager from having to keep track of the precise order in which the values are returned, and to guard against errors from dropped information, the syntax of the returned value will be **/FontName:Yes** or **/FontName:No**, where each font in the list is returned in this manner. The slashes delimit the individually returned font names, although newlines should be expected (and ignored) between them: A final **'*'** character will follow the returned values.

```
%%?BeginFontQuery: Times-Roman Optima CircleFont Adobe-Garamond
 mark
 /Times-Roman
 /Optima
/CircleFont
/Adobe-Garamond
{ %loop
 counttomark 0 gt { %ifelse
  dup (/) print (12345678901234567890123456789012345678901234567890) cvs print
  FontDirectory exch known { (:Yes) } { (:No) } ifelse =
 }{ %else
  pop exit
 } ifelse
} bind loop (*) = flush
%%?EndFontQuery: Unknown
%%EOF
```

```
/Times-Roman:Yes
/Optima:Yes
/CircleFont:No
/Adobe-Garamond:No
*
```

%%?BeginFontListQuery
%%?EndFontListQuery: default

Provides a PostScript language sequence to return a list of all available fonts. It should consult the FontDirectory dictionary as well as any mass storage devices available to the device. The list need be in no particular order, but each name should be returned separated by a slash "/" character. This is normally the way the PostScript == operator will return a font name. All white space characters should be ignored. The end of the font list should be indicated by a trailing **"*"** sign on a line by itself (decimal ASCII 42). Here is a look at two valid returns from the query:

```
/Optima/Optima-Bold/Optima-Oblique/Optima-BoldOblique/Courier/Symbol
*

/Courier
/Symbol
/Times-Roman
*
```

Note:
*In previous versions of this document, it was recommended to use **flush** to separate names into packets. This turns out to result in major performance degradation, and is hereby and subsequently disrecommended.*

%%?BeginProcSetQuery: name version.revision
%%?EndProcSetQuery: default

These comments delimit a ProcSet query. The combination of the **name, version** and **revision** fields should uniquely identify the ProcSet. The standard response to this query will consist of a line containing any of the values **"0, 1, 2"** where a value of **0** means that the ProcSet file is *missing*, a value of **1** means that the ProcSet is *present and OK*, and a value of **2** indicates that the ProcSet is *present but is an incompatible version.*

```
%%?BeginProcSetQuery: adobe_distill 1.1
/adobe_distill_dict where {
 begin VERSION (1.) anchorsearch {(1)}{(2)} ifelse clear
 end
}{
 (0)
} ifelse print flush
 %%?EndProcSetQuery: unknown
```

EXAMPLE 1: QUERY COMMENTS

This is an example of a *query job.* It is a separate PostScript language job that is sent before a print file is generated, its purpose being to obtain information about the state of the printer that can be used to generate the print file. Notice the **Query** keyword on the first line of the file, and the use of the query comments.

```
%!PS-Adobe-2.1 Query
%%Title: Query job to find out if some fonts exist
%%?BeginFontQuery: Palatino-Roman Palatino-Bold
mark
/Palatino-Roman
/Palatino-Bold
{
counttomark 0 gt {
dup (/) print (12345678901234567890123456789012345678901234567890) cvs print
FontDirectory exch known { (:Yes) } { (:No) } ifelse =
}{
pop exit
} ifelse
} bind loop (*) = flush
%%?EndFontQuery: Unknown
%%EOF
% send the appropriate EOF indication, depending on protocol
```

EXAMPLE 2: STRUCTURE COMMENTS

This is an example of an actual print file generated by a document manager. It is a representative document with a short prologue and script, a special paper size, and a downloaded font. It does not employ any of the **Requirement Conventions**, and will function properly with no intervention from a document manager or spooler. Since it uses **setpageparams**, it will only run on a PostScript language device that supports this operator unless a spooler changes the PostScript language code for that paper size.

```
%!PS-Adobe-2.1
%%Title: Example print file
%%Creator: /usr/local/bin/emacs
%%CreationDate: Mon Dec 1 09:24:58 1986
%%For: Software Developers
%%BoundingBox: 0 0 612 792
%%Pages: 1
%%DocumentFonts: Palatino-Roman Special
%%DocumentSuppliedFonts: Special
%%DocumentProcSets: /usr/lib/ps/draw.prologue 1.00 0
%%DocumentPaperSizes: ledger
%%EndComments
%%BeginProcSet: "/usr/lib/ps/draw.prologue" "(1.00)" ""
%!
% draw.prologue  Mon Dec 1 09:56:09 1986
/VERSION (1.00) def
/S /save load def
/RS { restore save } bind def
/R /restore load def
/g /setgray load def
/m /moveto load def
/s /show load def
/l /lineto load def
/c /closepath load def
/f { findfont exch scalefont setfont } bind def
```

```
%%EndProcSet
%%EndProlog

%%BeginSetup
%%BeginFeature: *PageSize Ledger
statusdict begin
792 1224 0 0 setpageparams
end
%%EndFeature
%%EndSetup
%%BeginFont: Special
%!
% downloaded font file for the "Special" font
70 dict begin
/FontName /Special def
/etc (more definitions) def
currentdict end
dup /FontName get exch definefont pop
%%EndFont

%%Page: one 1
%%PageFonts: Palatino-Roman Special
%%PaperColor: buff
S
100 100 m
147.5 318.0 l
301.7 77.0 l
0.9 g
c fill
RS
110 110 m
24 /Palatino-Roman f
(Example) s
40 500 m
24 /Special f
(Special Stuff) s
R
showpage
%%Trailer
```

EXAMPLE 3: RESOURCE REQUIREMENTS

This example shows use of the Requirement Conventions. This file is *dependent* on a document manager or spooler to supply it with additional resources. This kind of file is common in large, distributed networks where print spooling is taken for granted and resource management may be centralized.

```
%!PS-Adobe-2.1
%%Title: Example print file
%%Creator: /usr/local/bin/emacs
%%CreationDate: Mon Dec 1 09:24:58 1986
%%For: Software Developers
%%BoundingBox: 0 0 612 792
%%Pages: 1
```

```
%%DocumentFonts: Palatino-Roman Palatino-Bold Sonata
%%DocumentNeededFonts: Sonata Palatino-Roman
%%+ Palatino-Bold
%%DocumentNeededProcSets: "draw.prologue" (1.00) 0
%%DocumentPaperSizes: ledger
%%DocumentPaperColors: buff
%%Requirements: duplex punch3
%%EndComments

%%IncludeProcSet: "draw.prologue" (1.00) 0
%%EndProlog

%%BeginSetup
%%Feature: *PageSize Ledger
%%EndSetup

%%Page: one 1
%%PageFonts: Palatino-Roman
%%PaperColor: buff
S
100 100 m
147.5 318.0 l
301.7 77.0 l
0.9 g
c fill
RS
110 110 m
%%IncludeFont: Palatino-Roman
 24 /Palatino-Roman f
(Example) s
R
showpage
%%Page: two 2
%%PageFonts: Palatino-Bold Sonata
%%PaperColor: white
S
%%IncludeFont: Palatino-Bold
 100 100 m
24 /Palatino-Bold f
(Print something in bold) s
RS
%%IncludeFont: Sonata
 24 /Sonata f
gsave (=======) show grestore
(&Q q|372) show
R
showpage
%%Trailer
```

INDEX

% Comments

N

S

T

CHARACTER BITMAP DISTRIBUTION FORMAT
Specification
Version 2.1

January 16, 1989
PostScript® Developer Support Group

Adobe Systems Incorporated
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400

PN LPS5005

POSTSCRIPT®

CHARACTER BITMAP DISTRIBUTION FORMAT Specification Version 2.1

January 16, 1989
PostScript® Developer Support Group
(415) 961-4111

1. INTRODUCTION

This document describes Adobe Systems's character bitmap distribution format. The format is intended to be easily understood by both humans and computers. The format described in this document is subject to change without prior notification.

1.1 TAPE FORMAT

These character bitmaps are typically distributed on magnetic tape. Each tape is 1600 BPI, nine track, unlabeled, and contains two or more files. Each file is followed by an EOF mark. The last file on the tape is followed by two EOF marks. Physical records contain 512 bytes. The last physical record in a file (preceding an EOF mark) may contain fewer than 512 bytes.

Each file is encoded in the printable characters (octal 40 through 176) of USASCII plus carriage return and linefeed. Each file consists of a sequence of variable-length lines. Each line is terminated by a carriage-return (octal 015) and line-feed (octal 012). The first file on the tape is the Adobe Systems Copyright notice. Following files are font files. The format of font files is described in the following sections.

Note:
Font tapes may also be obtained in UNIX **tar** *format. Be sure to specify tar format if desired. No other tape formats are currently supported by Adobe Systems.*

1.2 FILE FORMAT

Character bitmap information is distributed in an USASCII encoded, human readable form. The information about a particular family and face at one size and orientation is contained in one file. The file begins with information pertaining to the face as a whole, followed by the information and bitmaps for the individual characters.

A font bitmap description file has the following general form, where each item is contained on a separate line of text in the file. Items on a line are separated by spaces.

GLOBAL HEADER INFORMATION

- The word STARTFONT followed by a version number indicating the exact file format used (for example, 2.1)
- One or more lines beginning with the word COMMENT. These lines may be ignored by any program reading the file.
- The word FONT followed by the **family name** and the **face name** separated by a hyphen. This should exactly match the PostScript outline font name.

- The word SIZE followed by the **point size** of the characters, the **x resolution**, and the **y resolution** of the device for which these characters were intended. All are represented as integers.

- The word FONTBOUNDINGBOX followed by the **width in x**, **height in y**, and the **x and y displacement** of the lower left corner from the **origin**. (See the examples in section 1.3). These are all integers.

- Optionally the word STARTPROPERTIES followed by the number of properties (**p**) that follow. This is a recent addition to the format. Within the properties list, there may be **p** lines consisting of a word for the **property name** followed by either an integer or string surrounded by ASCII double quotes (ASCII octal 042). Internal quote characters are indicated (or "quoted") by using two in a row. The property section, if it exists, is terminated by ENDPROPERTIES

THE INDIVIDUAL CHARACTER INFORMATION

The character section is introduced by the word CHARS followed by the number of character segments (**c**) that follow. This is an integer value. Error checking is recommended at the end of the file, to make sure that **c** characters were actually read and processed. Each of the **c** characters is then represented by the following:

- The word STARTCHAR followed by up to 14 bytes (no blanks) containing the **name** of the glyph. This should correspond to its name in the PostScript outline font's encoding vector.

- The word ENCODING followed by a positive integer representing the Adobe Standard Encoding value. If the character is *not* a member of the Adobe Standard Encoding, ENCODING is followed by -1 and optionally by another integer specifying the glyph index.

  The word SWIDTH followed by the **scalable width** in x and y of character. Scalable widths are in units of 1/1000th of the size of the character, and correspond to the widths found in AFM files (for outline fonts). If the size of the character is *p* points, the width information must be scaled by *p*/1000 to get the width of the character in printer's points. This width information should be considered as a vector indication the position of the next character's origin relative to the origin of this character. To convert the scalable width to the width in device pixels, multiply SWIDTH times *p*/1000 times *r*/72 where *r* is the device resolution in pixels per inch. The result is a real number giving the ideal print width in device pixels. The actual device width must of course be an integral number of device pixels and is given in the next entry.

- The word DWIDTH followed by the width in x and y of the character in device units (pixels). Like the SWIDTH, this width information is a vector indicating the position of the next character's origin relative to the origin of this character.

- The word BBX followed by the **width** in x (*BBw*), **height** in y (*BBh*) and x and y displacement (*BBxoff, BByoff*) of the lower left corner of the bitmap from the **origin** of the character.

- The word BITMAP. This introduces the hexadecimal data for the character bitmap.

- From the BBX value for *h*, find *h* lines of **hex-encoded bitmap**, padded on the right with zero's to the nearest byte (i.e., multiple of 8). Hex data can be turned into binary by

taking two bytes at a time, each of which represents 4 bits of the 8-bit value. For example, the byte 01101101 is two hex digits: 6 (0110 in hex) and D (1101 in hex).

- The word ENDCHAR.

- The entire file is terminated with the word ENDFONT. If this is encountered before **c** characters have been read, it is an error condition.

1.3 METRIC INFORMATION

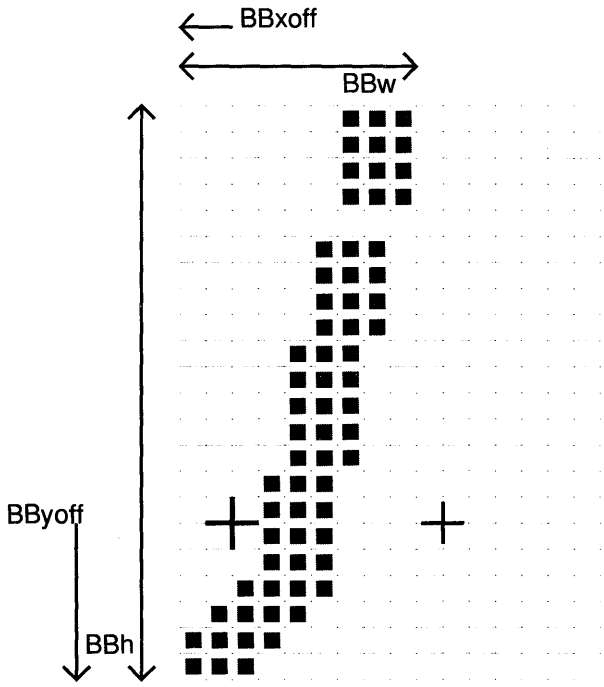
The font metrics include both the scalable width (really the width of the corresponding printer font character) and the character width of the screen font glyph, expressed in pixels. The scalable width is more accurate, and can be used by applications for keeping track of roundoff error and compensating in placement.

The following figures best illustrate the bitmap format and character metric information:

```
STARTCHAR j
ENCODING 106
SWIDTH 355 0
DWIDTH 8 0
BBX 9 22 -2 -6
```

The Bounding Box is expressed differently than other PostScript language files; the first two are the *width* and *height*, the second two are the offsets in *x* and *y*. This can bee seen in the illustration at right.

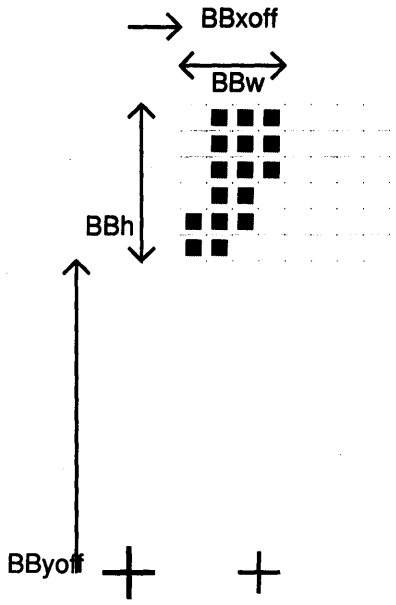
The character width from the origin (between + indicators) is 8 pixels, which has nothing to do with the actual bits, but is how far the current point moves after rendering the character.



The bounding box of the bitmap character can be used to predict how much data to read in the BITMAP section; the first two numbers give the width and height of the bitmap, and correspond exactly to how much data is supplied. The offset then allows positioning without repeating lots of white bits (look at the following **quoteright** character, which doesn't have very many bits, but is located far above the baseline. That is what the offset fields are for):

```
STARTCHAR quoteright
ENCODING 39
SWIDTH 223 0
DWIDTH 5 0
BBX 4 5 2 12
```

Here the actual bitmap is much smaller, and the offset (2 in *x*, 12 in *y*) positions the glyph with respect to its origin. These bitmaps are actually both from an italic font; notice that the character width of the quoteright leaves the origin still to the *left* of the actual bits after the character is drawn. Since all the characters are slanted, the next one will not interfere.



The bitmap itself is started by the BITMAP keyword and finished with the ENDCHAR keyword. It is best to "predict" the amount of data needed (using the BBX information) and use the ENDCHAR as an error-checking method: if you have consumed what you think is the appropriate amount of data, the very next thing in the file should be ENDCHAR. If not, either your parser is in error or the file is not complete (or is incorrect).

The bitmap itself is represented as hexadecimal digits, where each row corresponds to one row of the character bitmap. The bits are padded out to the nearest byte boundary with 0's, and the BBX bounding box information should be carefully consulted to determine how to extract the data.

The following is an abbreviated example of a bitmap file containing the specification of two characters (the j and quoteright from the previous examples):

```
STARTFONT 2.1
COMMENT This is a sample font in 2.1 format.
FONT Helvetica-BoldOblique
SIZE 8 200 200
FONTBOUNDINGBOX 9 24 -2 -6
STARTPROPERTIES 2
MinSpace 4
Copyright "Copyright (c) 1987 Adobe Systems, Inc."
ENDPROPERTIES
CHARS 2
STARTCHAR j
ENCODING 106
SWIDTH 355 0
DWIDTH 8 0
BBX 9 22 -2 -6
BITMAP
0380
0380
0380
0380
0000
0700
0700
0700
0700
0E00
0E00
0E00
0E00
0E00
1C00
1C00
1C00
1C00
2C00
7800
F000
E000
ENDCHAR
STARTCHAR quoteright
ENCODING 39
SWIDTH 223 0
DWIDTH 5 0
BBX 4 5 2 12
BITMAP
70
70
60
E0
C0
ENDCHAR
ENDFONT
```

International Business Machines
Corporation
11400 Burnet Road
Austin, Texas 78758–3493

Printed in the
United States of America

SC23-2211-00