ULTRIX Worksystem Software

digital

Extensions for the Display PostScript® System

Order Number: AA-PAN9A-TE

POSTSCRIPT®
LANGUAGE

Extensions for the DISPLAY POSTSCRIPT® System

Order No. AA–PAN9A–TE

ADOBE SYSTEMS
INCORPORATED

POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System

October 25, 1989

This document replaces the previous version dated May 30, 1989.

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*.

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; $0 \leq index \leq 255$.
146	1	executable name from system name table indexed by *index*. The *index* parameter is contained in the character after the token type; $0 \leq index \leq 255$.
147	1	literal name from user name table indexed by *index*. The *index* parameter is contained in the character after the token type; $0 \leq index \leq 255$.
148	1	executable name from user name table indexed by *index*. The *index* parameter is contained in the character after the token type; $0 \leq index \leq 255$.
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 less and the overall length of the object sequence is 65535 characters or less, 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 greater than 255 top-level objects or the overall length of the object sequence is greater than 65535 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**. From **rectfill**'s point of view, this case is 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** notices that it has been given a string object, it interprets the *value* of the string, expecting to find the binary token encoding of a homogeneous number array. Indeed, 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 arbitrary 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 an arbitrary 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 occurred; 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 are:[6]

---

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 inter-context 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.

A program should not make any assumptions regarding the scheduling of contexts. 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 below.

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 creates an arbitrary object (in this case, a font dictionary). The second line associates the user object 17 with this dictionary. Subsequently,

```
17 execuserobject setfont
```

pushes the font dictionary on the operand stack, from which it is taken by **setfont**. **execuserobject** performs an implicit 'exec' of this object; however, since the object in this example is not executable, the result of the implicit 'exec' 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

[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.

attention among multiple, more-or-less independent imaging contexts in an unpredictable order. Switching entire graphics states by altering its 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 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 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

[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.

POSTSCRIPT number 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. 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

Associated with each opcode in the operator string are zero or more operands in the data string or data array. The order of the

[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.

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.)

Operators

There are four categories of user path operators:

- New path construction operators, intended for inclusion in

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

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 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 ac-

[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.

cumulated 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 accordingly. 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 per-

pendicular, 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 for the POSTSCRIPT 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 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 per-

[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.

form 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**.

[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.

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 because 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 Adobe POSTSCRIPT products 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 $i \leq x' < i+1$ and $j \leq y' < j+1$.

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 rasterization 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**

[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.

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.

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] 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 POSTSCRIPT 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

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

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 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',

[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.

'%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

[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.

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 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 *82*
obj int	**printobject**	–	write binary object to standard output file, using *int* as tag *99*
int	**setobjectformat**	–	set binary object format (0=disable, 1=IEEE high, 2=low, 3=native high, 4=low) *111*
file obj int	**writeobject**	–	write binary object to *file*, using *int* as tag *129*

Memory Management Operators

–	**currentshared**	bool	return current VM allocation mode *82*
any	**scheck**	bool	true if *any* is simple or in shared VM, false otherwise *105*
bool	**setshared**	–	set VM allocation mode (*false*=private, *true*=shared *113*
int	**setvmthreshold**	–	set the allocation threshold for garbage collection *115*
dict key	**undef**	–	remove *key* and its value from *dict* *120*
key	**undefinefont**	–	remove font definition *121*
int	**vmreclaim**	–	control garbage collector *127*
–	**vmstatus**	level used maximum	report VM status *128*

Multiple Execution Context Operators

–	**condition**	condition	create condition object *80*
–	**currentcontext**	context	return current context identifier *81*
context	**detach**	–	enable context to terminate immediately when done *85*
mark obj_1 .. obj_n proc	**fork**	context	create context executing *proc* with obj_1 .. obj_n as operands *90*
context	**join**	mark obj_1 .. obj_n	await context termination and return its results *96*
–	**lock**	lock	create lock object *97*
lock proc	**monitor**	–	execute *proc* while holding *lock* *97*
condition	**notify**	–	resume contexts waiting for *condition* *98*
–	**quit**	–	terminates the context *100*
lock condition	**wait**	–	release *lock*, wait for *condition*, reacquire *lock* *129*
–	**yield**	–	suspend current context momentarily *131*

User Object Operators

–	**UserObjects** array	return **UserObjects** array in **userdict** *123*
index any	**defineuserobject** –	associate *index* with *any* in **UserObjects** array *84*
index	**execuserobject** –	execute *index* element in **UserObjects** array *86*
index	**undefineuserobject** –	remove *index* element from **UserObjects** array *121*

Graphics State Object Operators

gstate	**currentgstate** gstate	read current graphics state into *gstate* *81*
–	**gstate** gstate	create graphics state object *91*
gstate	**setgstate** –	set graphics state from *gstate* *110*

User Path Operators

x_1 y_1 x_2 y_2 r	**arct** –	append tangent arc *80*
ll_x ll_y ur_x ur_y	**setbbox** –	set bounding box for current path *107*
mark blimit	**setucacheparams** –	set user path cache parameters *114*
userpath	**uappend** –	interpret *userpath* and append to current path *118*
–	**ucache** –	declare that user path is to be cached *119*
–	**ucachestatus** mark bsize bmax rsize rmax blimit	return user path cache status and parameters *119*
userpath	**ueofill** –	fill using even-odd rule *119*
userpath	**ufill** –	interpret and fill *userpath* *120*
bool	**upath** userpath	create *userpath* for current path; include **ucache** if *bool* is true *122*
userpath	**ustroke** –	interpret and stroke *userpath* *124*
userpath matrix	**ustroke** –	interpret *userpath*, concatenate *matrix*, and stroke *124*
userpath	**ustrokepath** –	compute outline of stroked *userpath* *125*
userpath matrix	**ustrokepath** –	compute outline of stroked *userpath* *125*

Rectangle Operators

x y width height	**rectclip** –	clip with rectangular path *101*
numarray|numstring	**rectclip** –	clip with rectangular paths *101*
x y width height	**rectfill** –	fill rectangular path *102*
numarray|numstring	**rectfill** –	fill rectangular paths *102*
x y width height	**rectstroke** –	stroke rectangular path *103*

x y width height matrix	**rectstroke**	–	stroke rectangular path *103*
numarray|numstring	**rectstroke**	–	stroke rectangular paths *103*
numarray|numstring matrix	**rectstroke**	–	stroke rectangular paths *103*

Font Operators

font matrix	**makefont**	font'	produces new font *97*
key scale|matrix	**selectfont**	–	set font dictionary given name and transform *106*
text numarray|numstring	**xshow**	–	print characters of *text* using *x* widths in *numarray|numstring* *130*
text numarray|numstring	**xyshow**	–	print characters of *text* using *x* and *y* widths in *numarray|numstring* *131*
text numarray|numstring	**yshow**	–	print characters of *text* using *y* widths in *numarray|numstring* *132*

Halftone Definition Operators

–	**currenthalftone**	dict	return current halftone dictionary *82*
–	**currenthalftonephase**	x y	return current halftone phase *82*
–	**currentscreen**	frequency angle proc	return current halftone screen *82*
–	**currentscreen**	60 0 halftone	return current halftone dictionary (**sethalftone** was used) *82*
dict	**sethalftone**	–	set halftone dictionary *110*
x y	**sethalftonephase**	–	set halftone phase *111*
frequency angle proc	**setscreen**	–	set halftone screen *112*
num_1 num_2 halftone	**setscreen**	–	set halftone screen using halftone dictionary *112*

Scan Conversion Operators

–	**currentstrokeadjust**	bool	return current stroke adjust *83*
bool	**setstrokeadjust**	–	set stroke adjust (*false*=disable, *true*=enable) *114*

View Clip Operators

–	**eoviewclip**	–	view clip using even-odd rule *85*
–	**initviewclip**	–	reset view clip *92*
x y width height	**rectviewclip**	–	set rectangular view clipping path *104*
numarray|numstring	**rectviewclip**	–	set rectangular view clipping paths *104*
–	**viewclip**	–	set view clip from current path *126*
–	**viewclippath**	–	set current path from view clip *126*

Window System Support Operators

–	**deviceinfo** dict	return dictionary containing information about current device *85*
x y	**infill** bool	test whether point (*x*, *y*) would be painted by **fill** *92*
userpath	**infill** bool	test whether pixels in *userpath* would be painted by **fill** *92*
x y	**ineofill** bool	test whether point (*x*, *y*) would be painted by **eofill** *91*
userpath	**ineofill** bool	test whether pixels in *userpath* would be painted by **eofill** *91*
x y userpath	**inueofill** bool	test whether point (*x*, *y*) would be painted by **ueofill** *93*
$userpath_1$, $userpath_2$	**inueofill** bool	test whether pixels in $userpath_1$ would be painted by **ueofill** of $userpath_2$ *93*
x y userpath	**inufill** bool	test whether point (*x*, *y*) would be painted by **ufill** *94*
$userpath_1$ $userpath_2$	**inufill** bool	test whether pixels in $userpath_1$ would be painted by **ufill** of $userpath_2$ *94*
x y userpath	**inustroke** bool	test whether point (*x*, *y*) would be painted by **ustroke** *95*
x y userpath matrix	**inustroke** bool	test whether point (*x*, *y*) would be painted by **ustroke** *95*
$userpath_1$ $userpath_2$	**inustroke** bool	test whether pixels in $userpath_1$ would be painted by **ustroke** of $userpath_2$ *95*
$userpath_1$ $userpath_2$ matrix	**inustroke** bool	test whether pixels in $userpath_1$ would be painted by **ustroke** of $userpath_2$ *95*
–	**wtranslation** x y	return translation from window origin to device space origin *130*

File System Operators

string	**deletefile** –	delete named file *84*
pattern proc scratch	**filenameforall** –	execute *proc* for each file name matching *pattern* *88*
file	**fileposition** int	return current position in *file* *89*
$string_1$ $string_2$	**renamefile** –	rename file $string_1$ to $string_2$ *104*
file int	**setfileposition** –	set *file* to specified position *110*
string	**status** pages bytes referenced created true *or* false	return information about named file *116*

Miscellaneous Operators

–	**cleardictstack**	–	pop all nonpermanent dictionaries off dictionary stack *80*
–	**currentcacheparams**	mark size lower upper	return current characteristics of font cache *81*
index name	**defineusername**	–	define encoded name index *83*
–	**realtime** int		return real time in milliseconds *100*
mark size lower upper	**setcacheparams**	–	change characteristics of font cache *109*
–	**usertime** int		return context execution time in milliseconds *123*

Errors

invalidcontext	improper use of context operation *95*
invalidid	invalid identifier for window-system-specific operator *96*

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

returns the current halftone dictionary in the graphics state. If the current halftone was defined by **setscreen** instead of by **sethalftone**, **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 the **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 a **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:
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 **curveto**s 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:
rangecheck, 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**.

Positioning beyond the existing end-of-file will lengthen the file if it is open for writing and the file's access permits; the storage appended to the file has unspecified contents. Otherwise, an **ioerror** occurs. 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

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.

ERRORS:
limitcheck, rangecheck, stackunderflow, typecheck

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:
rangecheck, 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 enclosing user path is to be retained 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, stackoverflow, 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 POSTSCRIPT 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 past 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. Additionally, 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 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.

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 smaller 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*.

setpacking bool **setpacking** –

sets the array packing mode to the specified boolean value. This determines the type of executable arrays subsequently created by the POSTSCRIPT 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
124 readhexstring
125 readline
126 readstring
127 rectclip
128 rectfill
129 rectstroke
130 rectviewclip
131 repeat
132 restore
133 rlineto
134 rmoveto
135 roll
136 rotate
137 round
138 save
139 scale
140 scalefont
141 search
142 selectfont
143 setbbox
144 setcachedevice
145 setcachedevice2
146 setcharwidth
147 setcmykcolor
148 setdash
149 setfont
150 setgray
151 setgstate
152 sethsbcolor
153 setlinecap
154 setlinejoin
155 setlinewidth
156 setmatrix
157 setrgbcolor
158 setshared
159 shareddict
160 show
161 showpage
162 stop
163 stopped
164 store
165 string
166 stringwidth
167 stroke
168 strokepath
169 sub
170 systemdict
171 token
172 transform
173 translate
174 truncate
175 type
176 uappend
177 ucache
178 ueofill
179 ufill
180 undef
181 upath
182 userdict
183 ustroke
184 viewclip
185 viewclippath
186 where
187 widthshow
188 write
189 writehexstring
190 writeobject
191 writestring
192 wtranslation
193 xor
194 xshow
195 xyshow
196 yshow
197 FontDirectory
198 SharedFontDirectory
199 Courier
200 Courier-Bold
201 Courier-BoldOblique
202 Courier-Oblique
203 Helvetica
204 Helvetica-Bold
205 Helvetica-BoldOblique
206 Helvetica-Oblique
207 Symbol
208 Times-Bold
209 Times-BoldItalic
210 Times-Italic
211 Times-Roman
212 execuserobject
256 =
257 ==
258 ISOLatin1Encoding
259 StandardEncoding
260 [
261]
262 atan
263 banddevice
264 bytesavailable
265 cachestatus
266 closefile
267 colorimage
268 condition
269 copypage
270 cos
271 countdictstack
272 countexecstack
273 cshow
274 currentblackgeneration
275 currentcacheparams
276 currentcolorscreen
277 currentcolortransfer
278 currentcontext
279 currentflat
280 currenthalftone
281 currenthalftonephase
282 currentmiterlimit
283 currentobjectformat
284 currentpacking
285 currentscreen
286 currentstrokeadjust
287 currenttransfer
288 currentundercolorremoval
289 defaultmatrix
290 definefont
291 deletefile
292 detach
293 deviceinfo
294 dictstack
295 echo
296 erasepage
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

How to Order Additional Documentation

Technical Support

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

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local DIGITAL subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———————	Local DIGITAL subsidiary or approved distributor
Internal[1]	———————	SDC Order Processing - WMO/E15 *or* Software Distribution Center Digital Equipment Corporation Westminster, Massachusetts 01473

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