Display PostScript System

Adobe Systems Incorporated

pswrap Reference Manual

15 April 1993

Adobe Systems Incorporated

Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/

Contents

List of Examples

pswrap Reference Manual

1 About This Manual

pswrap Reference Manual is a guide to the *pswrap* translator. It tells you how to use *pswrap* to create C-callable procedures that contain PostScript™ language code.

Section 2, “About pswrap,” introduces the *pswrap* translator.

Section 3, “Using pswrap,” tells you how to run *pswrap*, and documents the options in the *pswrap* command line.

Section 4, “Writing a Wrap,” tells you how to write wrap definitions for *pswrap*.

Section 5, “Declaring Input Arguments,” tells you how to declare input arguments.

Section 6, “Declaring Output Arguments,” tells you how to declare output arguments.

Section 7, “Syntax,” explains the syntax used in wrap definitions.

Appendix lists error messages from the *pswrap* translator.

2 About pswrap

The *pswrap* translator provides a natural way for an application developer or toolkit implementor to compose a package of C-callable procedures that send PostScript language code to the PostScript interpreter. These C-callable procedures are known as *wrapped procedures* or *wraps*. A *wrap* is a procedure that consists of a C declaration with a PostScript language body. A *wrap body* is the PostScript language program fragment in a wrap.

Here's how *pswrap* fits into the Display PostScript system:

- You write the PostScript language programs required by your application, using the *pswrap* syntax to define a C-callable procedure and specify input and output arguments.
- You run *pswrap* to translate these PostScript language programs into wrapped procedures.
- You compile and link these wraps with the application program.
- When a wrap is called by the application, it sends encoded PostScript language to the PostScript interpreter and receives the values returned by the interpreter.

A *pswrap* source file associates PostScript language code with declarations of C procedures; *pswrap* writes C source code for the declared procedures, in effect wrapping C code around the PostScript language code. Wrapped procedures can take both input and output arguments.

- Input arguments are values a wrap sends to the PostScript interpreter as PostScript objects.
- Output arguments are pointers to variables where the wrap stores values returned by the PostScript interpreter.

Wraps are the most efficient way for an application to communicate with the PostScript interpreter.

3 Using pswrap

The form of the *pswrap* command line (UNIX-specific) is:

```
pswrap [-apr] [-o outputCfile] [-h outputHfile] [-s maxstring]
  [inputFile]
```

where square brackets [ ] indicate optional items.

3.1 Command-Line Options

The *pswrap* command-line options are as follows:

inputFile A file that contains one or more wrap definitions. *pswrap* transforms the definitions in *inputFile* into C procedure definitions. If no input file is specified, the standard input (which can be redirected from a file or pipe) is used. The input file can include text other than procedure definitions. *pswrap* converts procedure definitions to C procedures and passes the other text through unchanged. Therefore, it is possible to intersperse C-language source code with wrap definitions in the input file.

Note: Although C code is allowed in a pswrap *input file, it is not allowed within a wrap body. In particular, no CPP macros (for example, #define) are allowed inside a wrap.*

–a Generates ANSI C procedure prototypes for procedure declarations in *outputCfile* and, optionally, *outputHfile*. The **–a** option allows compilers that recognize the ANSI C standard to do more complete typechecking of parameters. The **–a** option also causes *pswrap* to generate *const* declarations.

Note: ANSI C procedure prototype syntax is not recognized by most non-ANSI C compilers, including many compilers based on the Portable C Compiler. Use the **–a** *option only in conjunction with a compiler that conforms to the ANSI C Standard.*

–h *outputHFile* Generates a header file that contains *extern* declarations for non-static wraps. This file can be used in *#include* statements in modules that use wraps. If the **–a** option is specified, the declarations in the header file are ANSI C procedure prototypes. If the **–h** option is omitted, a header file is not produced.

-o *outputCFile* Specifies the file to which the generated wraps and passed-through text are written. If omitted, the standard output is used. If the **–a** option is also specified, the procedure declarations generated by *pswrap* are in ANSI C procedure prototype syntax.

–p Specifies that strings passed by wraps are padded so that each data object begins on a long-word (4-byte) boundary. This option allows wraps to run on architectures that restrict data alignment to 4-byte boundaries and improves performance on some other architectures.

–r Generates reentrant code for wraps shared by more than one process (as in shared libraries). Reentrant code can be called recursively or by more than one thread. Wraps generated without this option use local static variables, which can be overwritten by recursive calls or multiple threads. Since those variables need to be reused by reentrant wraps, the **–r** option causes local automatic variables to be used instead. The **–r** option causes *pswrap* to generate extra code, use it only when necessary.

–s *maxstring* Sets the maximum allowable length of a PostScript string object or PostScript hexadecimal string object in the wrap body input. A syntax error is reported if a string is not terminated with) or > within *maxstring* characters. *maxstring* cannot be set lower than 80; the default is 200.

3.2 #line Directives

The C code that *pswrap* generates for wrapped procedures usually contains more lines than the input wrap body, so lines in the output file do not correspond to lines in the input file. This circumstance could make bugs that originate in the wrap body difficult to fix with a source-code debugger because the debugger displays C code from the output wrapped procedures, not PostScript language code from the input file.

pswrap solves the problem by using *#line* directives to record input file line numbers along with output file line numbers in the output file. When you use a C source code debugger, the directives refer the debugger to the correct line from the input file.

Note: Do not use the standard input and standard output streams as pipes to or from pswrap, because the resulting #line directives will be incomplete. pswrap expects both the input and output files to be named on the command line. If no input file is named, the references to input file line numbers will contain no filename; if the output file is not named, the name of the C source file produced by pswrap will be missing.

pswrap writes diagnostic output to the standard error if there are errors in the command line or in the input. If *pswrap* encounters errors during processing, it reports the error and exits with a nonzero termination status.

4 Writing a Wrap

Example 1 is a sample wrap definition. It declares the **PSWGrayCircle** procedure, which creates a solid gray circle with a radius of 5.0 centered at (10.0, 10.0).

Example 1 *Sample wrap definition*

Wrap definition:

```
defineps PSWGrayCircle( )
 newpath
 10.0 10.0 5.0 0.0 360.0 arc
 closepath
 0.5 setgray
 fill
endps
```

Procedure call:

```
PSWGrayCircle( );
```

PostScript language code equivalent:

```
newpath
10.0 10.0 5.0 0.0 360.0 arc
closepath
0.5 setgray
fill
```

4.1 The Wrap Definition

Following are the rules for defining a wrapped procedure. Each wrap definition consists of four parts:

- *defineps* begins the definition. It must appear at the beginning of a line without any preceding spaces or tabs.
- *Declaration of the C-callable procedure* is the name of the procedure followed by a list in parentheses of the arguments it takes. The arguments are optional. Parentheses are required even for a procedure without arguments. (Wraps do not return values; they are implicitly declared void.)
- *Wrap body* is a PostScript language program fragment, which is sent to the PostScript interpreter. It includes a series of PostScript operators and operands separated by spaces, tabs, and newline characters.
- *endps* ends the definition. Like *defineps*, *endps* must appear at the beginning of a line.

By default, wrap definitions introduce external (that is, global) names that can be used outside the file in which the definition appears. To introduce private (local) procedures, declare the wrapped procedure as static. For example, the **PSWGrayCircle** wrap in Example 1 can be made static by substituting the following statement for the first line:

```
defineps static PSWGrayCircle( )
```

Note: It is helpful for the application to give wraps names that identify them as such; for example, **PSWDrawBox**, **PSWShowTitle**, **PSWDrawSlider**, *and so on.*

4.2 Comments

C comments can appear anywhere outside a wrap definition. PostScript language comments can appear anywhere after the procedure is declared and before the definition ends. *pswrap* strips PostScript language comments from the wrap body. Comments cannot appear within PostScript string objects.

Example 2 *Comments in a wrap*

Wrap definition:

```
/* This is a C comment */
defineps PSWNoComment( )
 (/* This is not a comment */) show
 (% Nor is this.) length
 % This is a PS comment
endps
```

Wraps cannot be used to send PostScript language comments that contain structural information (%% and %!). Use another Client Library facility, such as **DPSWriteData**, to send comments.

4.3 The Wrap Body

pswrap accepts any valid PostScript language code as specified in the *PostScript Language Reference Manual, Second Edition*. If the PostScript language code in a wrap body includes any of the following symbols, the opening and closing marks must balance:

`{ }` Braces (to delimit a procedure)

`[ ]` Square brackets (to define an array)

`( )` Parentheses (to enclose a string)

`< >` Angle brackets (to mark a hexadecimal string)

Parentheses within a string body must balance or be quoted with \ according to standard PostScript language syntax.

Note: pswrap does not check a wrap definition for valid or sensible PostScript language code.

pswrap attempts to wrap whatever it encounters. Everything between the closing parenthesis of the procedure declaration and the end of the wrap definition is assumed to be an element of the PostScript language unless it is part of a comment or matches one of the wrap arguments.

Note: pswrap does not support the double slash (//) PostScript language syntax for immediately evaluated names. See the PostScript Language Reference Manual, Second Edition for more information about immediately evaluated names.

4.3.1 Execution Considerations

A wrap body executes as if the entire body were enclosed in an extra set of braces and followed by the **exec** operator. In other words, the body is put into an executable array which is then executed. This form of execution places a few restrictions on wrap bodies:

- First, **restore** can be used in only two cases:

1. If the corresponding **save** is executed in the same wrap

   or

2. If **restore** is the last thing in the wrap and the wrap does not return any values

   In cases other than these two, the executable array on the operand stack causes an invalidrestore error.

- Second, literal composite objects within the wrap body are actually created before any code in the body is executed. If the wrap body changes virtual memory allocation mode, this change does not affect the literal composite objects. For example, if the current VM allocation mode is false, the wrap in Example 3 produces the output “true”, “false”.

Example 3 *Nested composite objects in a wrap*

Wrap definition:

```
defineps PSWtestshared( )
 true setshared
 3 string scheck ==
 (abc) scheck ==
endps
```

The string “abc” was actually created before any code was executed, so the change to VM allocation mode does not affect it. The same effect occurs with nested executable array objects (sequences within { } braces).

4.4 Arguments

Argument names in the procedure header are declared using C types. For instance, the following example declares two variables, *x* and *y*, of type *long int*.

```
defineps PSWMyFunc(long int x, y)
```

In addition, the following holds true for arguments:

- There can be an unlimited number of input and output arguments.
- Input arguments must be listed before output arguments in the wrap header.
- Precede the output arguments, if any, with a vertical bar *|*.
- Separate arguments of the same type with a comma.
- Separate arguments of different types with a semicolon.
- A semicolon is optional before a vertical bar or a right parenthesis; these two examples are equivalent:

```
defineps PSWNewFunc(float x, y; int a | int *i)
defineps PSWNewFunc(float x, y; int a;| int *i;)
```

4.5 Input Arguments

Input arguments describe values that the wrap converts to encoded PostScript objects at runtime. When an element within the wrap body matches an input argument, the value that was passed to the wrap replaces the element in the wrap body. Input arguments represent placeholders for values in the wrap body. They are not PostScript language variables (names). Think of them as macro definitions that are substituted at runtime.

For example, the **PSWGrayCircle** procedure can be made more useful by providing input arguments for the radius and center coordinates, as in Example 4.

Example 4 *Wrap with input arguments*

Wrap definition:

```
defineps PSWGrayCircle(float x, y, radius)
 newpath
 x y radius 0.0 360.0 arc
 closepath
```

```
  0.5 setgray
  fill
endps
```

Procedure call:

```
PSWGrayCircle(25.4,17.7, 40.0);
```

PostScript language code equivalent:

```
newpath
25.4 17.7 40.0 0.0 360.0 arc
closepath
0.5 setgray
fill
```

The value of input argument *x* replaces each occurrence of *x* in the wrap body. This version of **PSWGrayCircle** draws a circle of a specified size at a specified location.

4.6 Output Arguments

Output arguments describe values that PostScript operators return. For example, the PostScript operator **currentgray** returns the gray-level setting in the current graphics state. PostScript operators place their return values on the top of the operand stack. To return a value to the application, place the name of the output argument in the wrap body at a time when the desired value is on the top of the operand stack. In Example 5, the wrap gets the value returned by **currentgray**.

Example 5 *Wrap with output arguments*

Wrap definition:

```
defineps PSWGetGray(| float *level)
 currentgray level
endps
```

Procedure call:

```
float aLevel;
PSWGetGray(&aLevel);
```

PostScript language code equivalent:

```
currentgray
% Pop current gray level off operand stack
% and store in aLevel.
```

Note: See section 11, "Runtime Support for Wrapped Procedures," on page CL-54 of the Client Library Reference Manual for a discussion about how pswrap uses **printobject** *to return results.*

When an element within a wrap body matches an output argument in this way, *pswrap* replaces the output argument with code that returns the top object on the operand stack. For every output argument, the wrap performs the following operations:

1. Pops an object off the operand stack.

2. Sends it to the application.

3. Converts it to the correct C data type.

4. Stores it at the place designated by the output argument.

Each output argument must be declared as a pointer to the location where the procedure stores the returned value. To get a *long int* from a *pswrap*-generated procedure, declare the output argument as *long int* *, as in Example 6.

Example 6 *Output argument as long int*

Wrap definition:

```
defineps PSWCountExecStack(| long int *n)
 countexecstack n
endps
```

Procedure call:

```
long int aNumber;
PSWCountExecStack(&aNumber);
```

PostScript language code equivalent:

```
countexecstack
% Pop count of objects on exec stack
% and return in aNumber.
```

To receive information from the PostScript interpreter, use only the syntax for output arguments described here. Do not use operators that write to the standard output (such as **=**, **==**, **print**, or **pstack**). These operators send ASCII strings to the application that *pswrap*-generated procedures cannot handle.

For an operator that returns results, the operator description shows the order in which results are placed on the operand stack, reading from left to right. When you specify a result value in a wrap body, the result is taken from the top of the operand stack. Therefore, the order in which wrap results are stated must be the reverse of their order in the operator description.

For instance, the PostScript operator description for **currentpoint** *returns two values, x and y:*

– currentpoint *x y*

Because the y value is left on the top of the stack, the corresponding wrap definition must be written

```
defineps PSWcurrentpoint (| float *x, *y)
 currentpoint y x   % Note: y before x.
endps
```

Note: Putting output parameters in the wrong order is one of the most common errors made with the Display PostScript system.

5 Declaring Input Arguments

This section defines the data types allowed as input arguments in a wrap. Note that *pswrap* accepts only *pswrap* data types as parameters. Although some *pswrap* data types correspond to C data types, they really are not the same. Also, not all defined C data types have corresponding *pswrap* data types (*long long* and *signed char*, for example).

In the following list, square brackets indicate optional elements.

- *DPSContext*. If the wrap specifies a context, it must appear as the first input argument. (*DPSContext* is a handle to the context record.)
- One of the following *pswrap* data types (note the *boolean* and *userobject*, data types, which are exclusive to *pswrap*):

```
boolean             userobject

int                 unsigned [int]

short [int]         unsigned short [int]

long [int]          unsigned long [int]

float               double
```

- An array of a *pswrap* data type.
- A character string (*char** or *unsigned char**).
- A character array (*char [ ]* or *unsigned char [ ]*). (The square brackets are part of C syntax.)

A string (*char**) passed as input can't be more than 65,535 characters. An array can't contain more than 65,535 elements.

5.1 Sending Boolean Values

If an input argument is declared as *boolean*, the wrap expects to be passed a variable of type *int*. If the variable has a value of zero, it is translated to a PostScript Boolean object with the value *false*. Otherwise, it is translated to a PostScript Boolean object with the value *true*.

5.2 Sending User Object Values

Input parameters declared as type *userobject* should be passed as type *long int*. The value of a *userobject* argument is an index into the **UserObjects** array.

When *pswrap* encounters an argument of type *userobject*, it generates PostScript language code to obtain the object associated with the index, as in Example 7.

Example 7 *Wrap with a userobject argument*

Wrap definition:

```
defineps PSWAccessUserObject(userobject x)
 x
endps
```

Procedure call:

```
long int aUserObject;
 ...
/* assume aUserObject = 6 */
PSWAccessUserObject(aUserObject);
```

PostScript language code equivalent:

```
6 execuserobject
```

If the object is executable, it executes; if it's not, it is pushed on the operand stack.

If you want to pass the index of a user object without having it translated by *pswrap* as described in Example 7, declare the argument to be of type *long int* rather than type *userobject*. Example 8 is a wrap that defines a user object.

Example 8 *Wrap that defines a user object*

Wrap definition:

```
defineps PSWDefUserObject(long int d)
 d 10 dict defineuserobject
endps
```

Procedure call:

```
long int anIndex;
 ...
/* assume anlndex = 12 */
PSWDefUserObject(anlndex);
```

PostScript language code equivalent:

```
12 10 dict defineuserobject
```

5.3 Sending Numbers

An input argument declared as one of the *int* types is converted to a 32-bit PostScript integer object before it is sent to the interpreter. A *float* or *double* input argument is converted to a 32-bit PostScript real object. These conversions follow the C conversion rules. The *int*, *long* and *short* types correspond to the data sizes in the native C environment. On some architectures, a long integer or double float is 64 bits, but the usable range of values is still 32 bits.

See *The C Programming Language, Second Edition*, B.W. Kernighan and D.M. Ritchie (Englewood Cliffs, N.J., Prentice-Hall, 1988) or *C: A Reference Manual, Second Edition*, Harbison and G. L. Steele, Jr. (Englewood Cliffs, N.J., Prentice-Hall, 1987).

Note: Since the PostScript language doesn't support unsigned integers, unsigned integer input arguments are converted to signed integers in the body of the wrap.

5.4 Sending Characters

An input argument composed of characters is treated as a PostScript name object or string object. The argument can be declared as a character string or a character array.

pswrap expects arguments that are passed to it as character strings (*char** or *unsigned char**) to be null terminated (*\0*). Character arrays are not null terminated. The number of elements in the array must be specified as an integer constant or an input argument of type *int*. In either case, the integer value must be positive.

5.4.1 Text Arguments

A text argument is an input argument declared as a character string or character array and converted to a single PostScript name object or string object.

The PostScript language interpreter does not process the characters of text arguments. It assumes that any escape sequences (*\n*, *\t*, and so on) have been processed before the wrap is called.

To make *pswrap* treat a text argument as a PostScript literal name object, precede it with a slash, as in the **PSWReadyFont** wrap definition in Example 9. (Only names and text arguments are preceded by a slash.)

Example 9 *Using a text argument as a literal name*

Wrap definition:

```
defineps PSWReadyFont(char *fontname; int size)
  /fontname size selectfont
endps
```

Procedure call:

```
PSWReadyFont("Sonata", 6);
```

PostScript language code equivalent:

```
/Sonata 6 selectfont
```

To make *pswrap* treat a text argument as a PostScript string object, enclose it within parentheses. The **PSWPutString** wrap definition in Example 10, shows a text argument, *str*.

Example 10 *Using a text argument as a string*

Wrap definition:

```
defineps PSWPutString(char *str; float x, y)
 x y moveto
 (str) show
endps
```

Procedure call:

```
PSWPutString("Hello World", 72.0, 72.0);
```

PostScript language code equivalent:

```
72.0 72.0 moveto
(Hello World) show
```

Note: Text arguments are recognized within parentheses only if they appear alone, without any surrounding white space or additional elements. In the following wrap definition, only the first string is replaced with the value of the text argument. The second and third strings are sent unchanged to the interpreter.

```
defineps PSWThreeStrings(char *str)
 (str) ( str ) (a str)
endps
```

If a text argument is not marked by either a slash or parentheses, *pswrap* treats it as an executable PostScript name object. In Example 11, *paintOp* is treated as executable.

Example 11 *Using a text argument as an executable name*

Wrap definition:

```
defineps PSWDrawPath(char *paintOp)
 0 setgray
 paintOp
endps
```

Procedure call:

```
PSWDrawPath("stroke");
```

PostScript language code equivalent:

```
0 setgray
stroke
```

5.5 Sending Arrays of Numbers or Booleans

Each element in the wrap body that names an input array argument represents a PostScript literal array object that has the same element values. In Example 12, the current transformation matrix is set using an array of six floating-point values.

Example 12 *Wrap with an array argument*

Wrap definition:

```
defineps PSWSetMyMatrix (float mtx[6])
 mtx setmatrix
endps
```

Procedure call:

```
static float anArray[ ] = {1.0, 0.0, 0.0, -1.0, 0.0, 0.0};
PSWSetMyMatrix(anArray);
```

PostScript language code equivalent:

```
[1.0 0.0 0.0 -1.0 0.0 0.0] setmatrix
```

The **PSWDefineA** wrap in Example 13 sends an array of variable length to the PostScript interpreter.

Example 13 *Wrap with a variable-length array argument*

Wrap definition:

```
defineps PSWDefineA (int data[x]; int x)
 /A data def
endps
```

Procedure call:

```
static int d1[ ] ={1, 2, 3};
static int d2[ ] = {4, 5};
 ...
PSWDefineA(d1, 3);
PSWDefineA(d2, 2);
```

PostScript language code equivalent:

```
/A [1 2 3] def
/A [4 5] def
```

5.6 Sending a Series of Numeric or Boolean Values

Occasionally, it is useful to group several numeric or Boolean values into a C array, and pass the array to a wrap that will send the individual elements of the array to the PostScript interpreter, as in Example 14.

Example 14 *Using array elements within a wrap*

Wrap definition:

```
defineps PSWGrayCircle(float nums[3], gray)
 newpath
 \nums[0] \nums[1] \nums[2] 0.0 360.0 arc
 closepath
 gray setgray
 fill
endps
```

Procedure call:

```
static float xyRadius = {40.0, 200.0, 55.0};
PSWGrayCircle(xyRadius, .75);
```

PostScript language code equivalent:

```
newpath
40.0 200.0 55.0 0.0 360.0 arc
closepath
.75 setgray
fill
```

In Example 14,
nums[i]
identifies an element of an input array in the wrap body, where *nums* is the name of an input boolean array or numeric array argument, and i is a nonnegative integer literal. No white space is allowed between the backslash (\) and the right bracket (]).

5.6.1 Specifying the Size of an Input Array

As the previous examples illustrate, you can specify the size of an input array in two ways:

- Give an integer constant size when you define the procedure, as in the **PSWGrayCircle** wrap definition

- Give an input argument that evaluates to an integer at runtime, as in the **PSWDefineA** wrap definition

In either case, the size of the array must be a positive integer with a value not greater than 65,535.

5.6.2 Sending Encoded Number Strings

A number sequence in the PostScript language can be represented either as an ordinary PostScript array object whose elements are to be used successively or as an encoded number string. Encoded number strings are described in section 3.12.5, “Encoded Number Strings,” of the *PostScript Language Reference Manual, Second Edition*.

The encoded number string format efficiently passes sequences of numbers, such as coordinates, to PostScript operators that take arrays of operands (**xyshow** and **rectfill**, among others). In this form, the arrays take up less space in PostScript VM. In addition, the operator that consumes them executes faster because the data in an encoded number string, unlike a PostScript array object, does not have to be scanned by the PostScript scanner.

To simplify passing encoded number strings in a wrap, *pswrap* syntax provides the *numstring* data type, which lets you pass PostScript operands as numeric elements in a normal C array. The *pswrap* translator generates code that produces the encoded number string corresponding to this C array.

Note: numstring is used only for input. It is invalid as an output parameter in a wrap definition.

The syntax of the *numstring* declaration is as follows, where braces enclose optional parts (the square brackets enclosing the array size are actual brackets):

```
{modifier} numstring variablename[arraysize] {: scale};
```

The modifier can be *int*, *long*, *short*, or *float*, and describes the numbers passed in as a parameter. For example, if a system defines long to be 64-bit integers, the array passed as a parameter should be 64-bit integers. If no modifier is specified, the default is *int*.

Scale applies only to fixed-point types and specifies the number of fractional bits in the number. If it isn’t specified, *scale* defaults to zero.

Arraysize and *scale* can be specified as either constants or variables. Any variable that is used must be declared immediately after the *numstring* parameter and must be an integer type.

Example 15 *Examples of numstrings in wrap definitions*

Wrap definitions:

```
defineps PSWNums1(numstring a[5];)
% Array of 5 elements of default format
% Native integer size, zero scale.

defineps PSWNums2(float numstring a[6];)
% Floating point, constant array size.

defineps PSWNums3(float numstring a[n]; int n;)
% Floating point, variable array size.

define PSWNums4 (int numstring a{6}:8)
% Native integer size, constant array size and scale.

defineps PSWNums5(int numstring a[n]:6; int n;)
% Native integer size, variable array size, constant scale.

defineps PSWNums6(long numstring a[n]:s; int n, s;)
% Long integer size, variable array size and scale.
```

Note: Number string parameters with int, long, or short modifiers are packed into 16- or 32-bit PostScript language number strings. If an integer type is 16 bits or shorter, it converts into a 16-bit number string, otherwise it converts into a 32-bit number string. If an integer type is longer than 32 bits, values will be truncated to 32 bits.

PSWXShowChars, as shown in Example 16, is a wrap that uses the *numstring* data type to pass an array of user-defined widths to the **xshow** operator.

Example 16 *Wrap with a numstring argument*

Wrap definition:

```
defineps PSWXShowChars(char str[4];
                       long numstring widths[4]:0)
  /Times-Roman 30 selectfont
  100 100 moveto
  str widths xshow
endps
```

Procedure call:

```
char str[4] = "test";
long widths[4] = {7, 10, 9, 7};
  ...
PSWXShowChars(str, widths);
```

PostScript language code equivalent:

```
/Times-Roman 30 selectfont
/str (test) def
/widths <9580040007000000A0000000900000007000000> def
     % encoded number string, hex format,
     % preceded by 4-byte generated header
100 100 moveto
str widths xshow
```

5.7 Specifying the Context

Every wrap communicates with a PostScript execution context. The current context is normally used as the default. The Client Library provides operations for setting and getting the current context for each application. To override the default, declare the first argument as type *DPSContext* and pass the appropriate context as the first parameter whenever the application calls the wrap. Example 17 shows a wrap definition that explicitly declares a context.

Note: Do not refer to the name of the context in the wrap body.

Example 17 *Wrap that declares a context*

Wrap definition:

```
defineps PSWGetGray(DPSContext c | float *level)
 currentgray level
endps
```

Procedure call:

```
DPSContext myContext;
float aLevel;
   ...
PSWGetGray(myContext, &aLevel);
```

PostScript language code equivalent:

```
currentgray
% Pop current gray level off operand stack
% and store in aLevel
```

6 Declaring Output Arguments

To receive information from the PostScript interpreter, the output arguments of a wrap must refer to locations where the information can be stored. One of the following can be declared as an output argument:

- A pointer to one of the *pswrap* data types listed previously except *userobject*
- An array of one of these types
- A character string *(char*** or *unsigned char**)*
- A character array *(char [ ]* or *unsigned char [ ])*

If an output argument is declared as a pointer or character string, the procedure writes the returned value at the pointed-to location.

For an output argument declared as a pointer, previous return values are overwritten if the output argument is encountered more than once in executing the wrap body.

For an output argument declared as a character string *(char *)*, the value is stored only the first time it is encountered.

For an output argument declared as an array of one of the *pswrap* data types or as a character array, the wrap fills the slots in the array.

For example, the wrap in Example 18 returns 2 in *nump*, "abc" in *charp*, the array {3,4} in *numarray*, and the string "ghijkl" in *chararray*.

Example 18 *Returning output values more than once*

Wrap definition:

```
defineps PSWreturn( | int *nump, char *charp,
                int numarray[2], char chararray[6])
  1 nump
  2 nump
  3 numarray
  4 numarray
  (abc) charp
  (def) charp
  (ghi) chararray
  (jkl) chararray
endps
```

Note: Whenever an array output argument is encountered in the wrap body, the values on the PostScript operand stack are placed in the array in the order in which they would be popped off the stack. When the array bounds have been exceeded, no further storing of output in the array is done. No error is reported if elements are returned to an array that is full.

You can specify output arguments in the defineps statement in any order that is convenient. The order of the output arguments has no effect on the execution of the PostScript language code in the wrap body.

pswrap does not check whether the wrap definition provides return values for all output arguments, nor does it perform type checking for declared output arguments.

6.1 Receiving Numbers

PostScript integer objects and real objects are 32 bits long. When returned, these values are assigned to the variable provided by the output argument. On a system where the size of an *int* or *float* is 32 bits, pass a pointer to an *int* as the output argument for a PostScript integer object; pass a pointer to a *float* as the output argument for a PostScript real object:

```
defineps PSWMyWrap ( | float *f; int *i)
```

A PostScript integer object or real object can be returned as a *float* or *double*. Other type mismatches cause a **typecheck** error (for example, attempting to return a PostScript real object as an *int*).

6.2 Receiving Boolean Values

A procedure can declare a pointer to a *boolean* as an output argument.

Example 19 *Wrap with a boolean output argument*

Wrap definition:

```
defineps PSWKnown(char *Dict, *x | boolean *ans)
  Dict /x known ans
endps
```

Procedure call:

```
int found;
 ...
PSWKnown("statusdict", "duplex", &found);
```

PostScript language code equivalent:

```
statusdict /duplex known found
```

This wrap expects to be passed the address of a variable of type *int* as its output argument. If the PostScript interpreter returns the value *true*, the wrap places a value of 1 (one) in the variable referenced by the output argument. If the interpreter returns the value *false*, the wrap places a value of 0 (zero) in the variable.

6.3 Receiving a Series of Output Values

To receive a series of output values as an array, declare an array output argument; then write a wrap body in the PostScript language to compute and return its elements, one or more elements at a time. Example 20 declares a wrap that returns the 256 font widths for a given font name at a given font size.

Example 20 *Returning a series of output values*

Wrap definition:

```
defineps PSWGetWidths(char *fn; int size | float wide[256])
 /fn size selectfont
 0 1 255 {
  (X) dup 0 4 -1 roll put
  stringwidth pop wide
 } for
endps
```

Procedure call:

```
float widths[256];
PSWGetWidths("Serifa", 12, widths);
```

PostScript language code equivalent:

```
/Serifa 12 selectfont
0 1 255 {
 (X) dup 0 4 -1 roll put
 stringwidth pop
 % Pop width for this character and insert width
 % into widths array at current element;
 % point to next element.
} for
```

In Example 20, the loop counter is used to assign successive ASCII values to the scratch string (“X”). The **stringwidth** operator then places both the width and height of the string on the PostScript operand stack. (Here it operates on a string one character long.)

The **pop** operator removes the height from the stack, leaving the width at the top. The occurrence of the output argument *wide* in this position triggers the width to be popped from the stack, returned to the application, and inserted into the output array at the current element. The next element then becomes the current element.

The **for** loop (the procedure enclosed in braces followed by **for**) repeats these operations for each character in the font, beginning with the first, 0, and ending with 255th element of the font array.

6.3.1 Receiving a Series of Array Elements

A PostScript array object can contain a series of elements to be stored in an output array. The output array is filled in, one element at a time, until it's full. Therefore, the **PSWTest** wrap defined below returns {1, 2, 3, 4, 5, 6}:

```
defineps PSWTest(| int Array[6])
  [1 2 3] Array
  [4 5 6] Array
endps
```

The **PSWTestMore** wrap defined below returns {1, 2, 3, 4}:

```
defineps PSWTestMore(| int Array[4])
  [1 2 3] Array
  [4 5 6] Array
endps
```

6.3.2 Specifying the Size of an Output Array

The size of an output array is specified in the same manner as the size of an input array. Use a constant in the wrap definition or an input argument that evaluates to an integer at runtime. If more elements are returned than fit in the output array, the additional elements are discarded.

6.4 Receiving Characters

To receive characters from the PostScript interpreter, declare the output argument as either a character string or as a character array.

If the argument is declared as a character string, the wrap copies the returned string to the location indicated. Provide enough space for the maximum number of characters that might be returned, including the null character (\0) that terminates the string. Only the first string encountered will be returned. For example, in the following **PSWStrings** procedure, the string "123" is returned:

```
defineps PSWStrings(| char *str)
  (123) str
  (456) str
endps
```

Character arrays, on the other hand, are treated just like arrays of numbers. In the **PSWStrings2** procedure, the value returned for *str* will be "123456".

```
defineps PSWStrings2(| char str[6])
 (123) str
 (456) str
endps
```

Note: The string is not null terminated. If the argument is declared as a character array (for example, char s[num]), the procedure copies up to num characters of the returned string into the array. Additional characters are discarded.

6.5 Communication and Synchronization

The PostScript interpreter can run as a separate process from the application; it can also run on a separate machine. When the application and interpreter processes are separated, the application programmer must take communication into account. This section alerts you to communication and synchronization issues.

A wrap that has no output arguments returns as soon as the wrap body is transferred to the client-server communications channel. In this case, the communications channel is not necessarily flushed. Since the wrap body is not executed by the PostScript interpreter until the communications channel is flushed, errors arising from the execution of the wrap body can be reported long after the wrap returns.

In the case of a wrap that returns a value, the entire wrap body is transferred to the client-server communications channel, which is then flushed. The client-side code awaits the return of output values followed by a special termination value. Only then does the wrap return.

7 Syntax

Square brackets, [ ], mean that the enclosed form is optional. Curly brackets, { }, mean that the enclosed form is repeated, possibly zero times. A vertical bar, |, separates choices in a list.

```
Unit =
 ArbitraryText {Definition ArbitraryText}

Definition =
 NLdefineps ["static"] Ident "(" [Args] ["|" Args]")"
 Body
 NLendps

Body =
 {Token}

Token =
 Number | PSIdent | SlashPSIdent
 | "(" StringLiteral ")"
 | "<" StringLiteral ">"
 | "{" Body "}"
 | "[" Body "]"
 | Input Element

Args =
 ArgList {";" ArgList} [";"]

ArgList =
  Type ItemList

Type =
 "DPSContext" | "boolean" | "float" | "double"
 | ["unsigned"] "char"
 | ["unsigned"] ["short" | "long"] "int"
 | ["int" | "long" | "short" | "float"] "numstring"

ItemList =
 Item {"," Item}

Item =
 "*" Ident | Ident ["["Subscript"]"]
 | Ident "["Subscript"]" [Scale]

Subscript =
 Integer | Ident

Scale =
 ":" Integer | ":" Ident
```

7.1 Syntactic Restrictions

- *DPSContext* must be the first input argument if it appears.
- A simple char argument *(char Ident)* is never allowed; it must be * or [ ].
- A simple *Ident* item is not allowed in an output item list; it must be * or [ ].

7.2 Clarifications

- *NLdefineps* matches the terminal *defineps* at the beginning of a new line.
- *NLendps* matches the terminal *endps* at the beginning of a new line.
- *Ident* follows the rules for C names; *PSIdent* follows the rules for PostScript language names.
- *SlashPSIdent* is a PostScript language name preceded by a slash.
- *StringLiteral* tokens follow the PostScript language conventions for string literals.
- *Number* tokens follow the PostScript language conventions for numbers.
- Integer subscripts follow the C conventions for integer constants.
- *Input Element* is \n[*i*] where *n* is the name of an input array argument, *i* is a nonnegative integer literal, and no white space is allowed between \ and].

Appendix A

Error Messages from the pswrap Translator

The following is a list of error messages the pswrap translator can generate:

```
input parameter used as a subscript is not an integer

output parameter used as a subscript

char input parameters must be starred or subscripted

hex string too long

invalid characters

invalid characters in definition

invalid characters in hex string

invalid radix number

output arguments must be starred or subscripted

out of storage, try splitting the input file

-s 80 is the minimum

can’t allocate char string, try a smaller -s value

can’t open file for input

can’t open file for output

error in parsing

string too long

usage:pswrap[-s maxstring][-ar][-h headerfile][-o outfile]
  [infile]

endps without matching defineps

errors in parsing
```

```
errors were encountered

size of wrap exceeds 64K

parameter reused

output parameter used as a subscript

non-char input parameter

not an input parameter

not a scalar type

wrong type

parameter index expression empty

parameter index expression error

end of input file/missing endps
```