ULTRIX Worksystem Software

digital

Display PostScript® pswrap
Reference Manual

Order Number: AA-PAJTA-TE

DISPLAY POSTSCRIPT
S Y S T E M

pswrap Reference Manual

Order No. AA–PAJTA–TE

ADOBE SYSTEMS
I N C O R P O R A T E D

pswrap Reference Manual

Writer: Amy Davidson

October 25, 1989

This manual replaces the previous version dated October 6, 1988.

Contents

1 ABOUT THIS MANUAL

This manual is the programmer's reference manual for the *pswrap* translator. It tells you how to use *pswrap* to create C-callable procedures that contain POSTSCRIPT® language code.

Section 2 introduces the *pswrap* translator.

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

Section 4 tells you how to write wrap definitions for *pswrap.*

Section 6 tells you how to declare input arguments.

Section 5 tells you how to declare output arguments.

Appendix A lists error messages from the *pswrap* translator.

Appendix B describes the syntax used in wrap definitions.

This manual does not provide information on the POSTSCRIPT language, the DISPLAY POSTSCRIPT® system, or the Client Library (the programming interface to the DISPLAY POSTSCRIPT system). For more information regarding these topics, see the following manuals:

- *POSTSCRIPT Language Reference Manual*
- *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System*
- *POSTSCRIPT Language Color Extensions*
- *Client Library Reference Manual*

2 ABOUT *PSWRAP*

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

Here's how *pswrap* fits into the DISPLAY POSTSCRIPT system:

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

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

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

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

3 USING *PSWRAP*

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

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

where square brackets [ ] indicate optional items.

3.1 COMMAND-LINE OPTIONS

The *pswrap* command-line options are described below.

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

Note: Although C code is allowed in a *pswrap* input file, it is *not* allowed within a wrap body. In particular, C '#define' macros cannot be used inside a wrap.

-a	Generates ANSI C procedure prototypes for procedure declarations in *outputCfile* and, optionally, *outputHfile*. (See the **-h** option.) The **-a** option allows compilers that recognize the ANSI C Standard to do more complete typechecking of parameters. To save space, the **-a** option also causes *pswrap* to generate 'const' declarations.

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

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

-o *outputCFile*	Specifies the file to which the generated wraps and passed-through text are written. If omitted, the standard output is used. If the **-a** option is also specified, the procedure declarations generated by *pswrap* are in ANSI C procedure prototype syntax.
-r	Generates reentrant code for wraps that are shared by more than one process (as in shared libraries). Since the **-r** options causes *pswrap* to generate extra code, use it only when necessary.
-s *maxstring*	Sets the maximum allowable length of a POSTSCRIPT string object or POSTSCRIPT hex string object in the wrap body input. A syntax error will be reported if a string is not terminated with ')' or '>' within *maxstring* characters. *maxstring* cannot be set lower than 80. The default is 200.

3.2 '#LINE' DIRECTIVES

Since the C source code generated for wrapped procedures usually contains more lines than the input wrap body does, *pswrap* inserts '#line' directives into the output wrap. These directives record input line numbers in the output wrap source file so that a source-code debugger can display them correctly. Since a debugger displays C source code, not the POSTSCRIPT language code in the wrap body, *pswrap* inserts **#line** directives for both the *inputFile* and the *outputCfile*.

Note: Unless both the input and output files are named on the command line, the '#line' directives will be incomplete; in the latter case, they will lack the name of the C source file *pswrap* produces. Use of the standard input and standard output streams is discouraged for this reason.

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

4 WRITING A WRAP

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

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

The rules for defining a wrapped procedure are given in the next section.

4.1 THE WRAP DEFINITION

Each wrap definition consists of four parts:

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

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

```
defineps static grayCircle()
```

Note: It is helpful for the application to use a naming convention for wraps that identifies them as such; for example, *PWdrawbox*, *PWshowtitle*, *PWdrawslider*, and so on.

4.2 COMMENTS

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

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

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

4.3 THE WRAP BODY

pswrap accepts any valid POSTSCRIPT language code as specified in the *POSTSCRIPT Language Reference Manual*, *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System*, and *POSTSCRIPT Language Color Extensions*. If the POSTSCRIPT language code in a wrap body includes any of the following symbols, the opening and closing marks must balance.

'{ }'	Braces (to delimit a procedure)
'[ ]'	Square brackets (to define an array)

'( )'	Parentheses (to enclose a string)
'< >'	Angle brackets (to mark a hexdecimal string)

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

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

---

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

Note: *pswrap* does not support the // POSTSCRIPT language syntax for immediately evaluated names. See the *POSTSCRIPT Language Reference Manual* for more information about immediately evaluated names.

4.4 ARGUMENTS

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

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

There can be any number of input and output arguments. Input arguments must be listed before output arguments in the wrap header. Precede the output arguments, if any, with a vertical bar '|'. Separate arguments of the same type with a comma. Separate arguments of differing types with a semicolon. A semicolon is optional before a vertical bar or a right parenthesis; for example:

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

4.5 INPUT ARGUMENTS

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

For example, the *grayCircle* procedure defined on page 5 can be made more useful by providing input arguments for the radius and center coordinates:

```
defineps grayCircle(float x,y, radius)
      newpath
      x y radius 0.0 360.0 arc
      closepath
      0.5 setgray
endps
```

The value of input argument 'x' replaces every occurence of 'x' in the wrap body. This version of *grayCircle* draws a circle of any size at any location.

4.6 OUTPUT ARGUMENTS

Output arguments describe values that POSTSCRIPT operators return. For example, the standard POSTSCRIPT operator **currentgray** returns the gray-level setting in the current graphics state. POSTSCRIPT operators return values by placing them on the top of the operand stack. To return the value to the application, place the name of the output argument in the wrap body at a point in which the desired value is on the top of the operand stack. For example, the following wrap gets the value returned by **currentgray**:

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

When an element within a wrap body matches an output argument in this way, *pswrap* substitutes code that gets the returned value from the stack. For every output argument, the wrap will perform the following operations:

- Pop an object off the operand stack.
- Send it across the connection to the application.
- Convert it to the correct C data type.
- Store it at the place designated by the output argument..

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

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

To call the wrap from a C program, pass it a pointer to a 'long int':

```
long int val;
countexecstack(&val);
```

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

Sections 5 and 6 discuss the details of input and output arguments, respectively.

5 DECLARING INPUT ARGUMENTS

This section defines the types allowed for input arguments. If the wrap specifies a context argument, it must appear as the first input argument and it must be of type 'DPSContext'. ('DPSContext' is a handle to the context record; see the *Client Library Reference Manual* for more information.)

All other input arguments are declared as one of the data types in the following list. Square brackets indicate optional elements.

- One of the following *pswrap data types* (equivalent to C data types except for 'boolean' and 'userobject', which are special to *pswrap*):

'boolean'	'userobject'
'int'	'unsigned [int]'
'short [int]'	'unsigned short [int]'
'long [int]'	'unsigned long [int]'
'float'	'double'

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

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'; see *POSTSCRIPT Language Extensions for the DISPLAY POSTSCRIPT System* for a description of user objects.

When *pswrap* encounters an argument of type **userobject**, it will generate POSTSCRIPT language code to get the value of the user object. For example, *pswrap* expands

```
defineps access_userobject(userobject x)
    x
endps
```

to

```
    x execuserobject
```

The 'userobject' argument in a wrap is expanded to the object that was used to define that user object.

If you want to use the value of a 'userobject' argument without having it expanded by *pswrap* as described above, declare the argument to be of type 'long int'. Here is an example of a wrap that defines a user object:

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

5.3 SENDING NUMBERS

An input argument declared as one of the 'int' types is converted to a 32-bit POSTSCRIPT integer object before it is sent to the interpreter. A 'float' or 'double' input argument is converted to a 32-bit POSTSCRIPT real object. These conversions follow the usual C conversion rules.[1]

Note: Since the POSTSCRIPT language does not support unsigned integers, unsigned integer input arguments are cast to signed integers in the body of the wrap.

5.4 SENDING CHARACTERS

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

pswrap expects arguments that are passed to it as character strings ('char *' or 'unsigned char *') to be null terminated ('\0'). Character arrays are not null terminated, but the number of elements in the array must be specified as a positive integer. This is done either by an integer constant or by an input argument of type 'int'. See Section 5.5 for an example of this rule.

5.4.1 Text Arguments

An input argument declared as a character string or character

[1]See *The C Programming Language*, R. W. Kernighan and D. M. Ritchie (Englewood Cliffs, NJ: Prentice-Hall, 1978) or *C: A Reference Manual*, S. P. Harbison and G. L. Steele, Jr. (Englewood Cliffs, NJ: Prentice-Hall, 1984).

array is converted to a single POSTSCRIPT name object or string object. Such an argument is referred to as a *text argument.*

The POSTSCRIPT interpreter does not process the characters of text arguments. It assumes that any character escape processing has been done 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 *readyFont* wrap definition below. (Only names and text arguments can be preceded by a slash.)

```
defineps readyFont( char *fontname; int size )
    /fontname findfont
    size scalefont
    setfont
endps
```

To make *pswrap* treat a text argument as a POSTSCRIPT string object, enclose it within parentheses. The *putString* wrap definition below shows a text argument, '(str)':

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

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

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

If a text argument is not marked by either a slash or parentheses, *pswrap* treats it as an executable POSTSCRIPT name object. In the following example, 'mydict' is treated as executable:

```
defineps doProcedure(char *mydict )
    mydict /procedure get exec
endps
```

5.5 SENDING ARRAYS OF NUMBERS OR BOOLEANS

Each element in the wrap body that names an input array argument represents a literal POSTSCRIPT literal array object that has the same element values. In the *grayRect* wrap definition below, the wrap paints a gray rectangle with a given gray value at the location specified by the element values of its 'rectNums' parameter:

```
defineps grayRect (float gray, rectNums [4])
    gsave
    gray setgray
    rectNums fillrect
    grestore
endps
```

The *defineA* wrap below sends an array of any length to the POSTSCRIPT interpreter:

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

5.6 GROUPING NUMERIC OR BOOLEAN VALUES

Occasionally, it is useful to group several numeric or boolean values into a C array, pass the array to a wrap, and then send the individual elements of the array to the POSTSCRIPT interpreter, as in the following example.

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

In the example just above, '\nums[*i*]' identifies an element of an input array in the wrap body, where 'nums' is the name of an input boolean array or numeric array argument, 'i' is a non-negative integer literal, and no whitespace is allowed between '\' and ']'.

5.6.1 Specifying the Size of an Input Array

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

- Give an integer constant as the size when you define the procedure, as in the *grayRect* wrap definition.
- Give a name that evaluates to an integer at run time as the size, as in the *defineA* wrap definition.

In either case, the size of the array must be a positive integer.

5.7 SPECIFYING THE CONTEXT

Every wrap communicates with a POSTSCRIPT execution context. The current context is normally used as the default. The Client Library provides operations for setting and getting the current context for each application. To override the default, declare the first argument as type 'DPSContext' and pass the appropriate context as the first parameter whenever the application calls the wrap. Here is an example of a wrap definition that explicitly declares a context:

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

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

6 DECLARING OUTPUT ARGUMENTS

To receive information back from the POSTSCRIPT interpreter, the output arguments of a wrap must refer to locations where the information can be stored. An output argument can be declared as one of the following:

- A pointer to one of the basic data types listed in Section 5 , except for 'userobject'.
- An array of one of these types.
- A character string ('char *' or 'unsigned char *').
- A character array ('char [ ]' or 'unsigned char [ ]').

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

For an output argument declared as a pointer, previous return values are overwritten if the output argument is encountered more than once in executing the wrap body. For an output argument declared as a character string ('char *'), the value is stored only the first time it is encountered.

If an output argument is declared as an array of one of the *pswrap* data types (see page 10 for a list) or as a character array, the wrap fills the slots in the array (see Section 6.3).

Note: Whenever an array output argument is encountered in the wrap body, the values on the POSTSCRIPT operand stack are placed in the array in the order in which they would be popped off the stack. When no empty array elements remain, no further storing of output in the array is done.

Output values can be returned in any order.

pswrap does not check whether the wrap definition provides return values for all output arguments, nor does it check whether the wrap body is compatible with the 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. Typically, 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 myWrap ( | 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:

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

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

6.3 RECEIVING A SERIES OF OUPUT VALUES

To receive a series of output values in a specified region of memory, declare an output argument to be an array; then write a wrap body in the POSTSCRIPT language to compute and return its elements, either one at a time or in chunks. The example below declares a wrap that returns the 256 font widths for a given font name at a given font size:

```
defineps getWidths(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
```

In the above example, the loop counter is used to assign successive ASCII values to the scratch string '(X)'. The **stringwidth** operator then places both the width and height of the string on the POSTSCRIPT operand stack. (Here it operates on a string just one character long.) The **pop** operator removes the height from the stack, leaving the width at the top. The occurrence of the output argument 'wide' in this position triggers the width to be popped from the stack, returned to the application, and inserted into the output array as the next element.

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

6.3.1 Returning a Series of Array Elements

A POSTSCRIPT array object can contain a series of elements to be stored in an output array. The elements are treated as if they had been returned one at a time. The array is written to until all of the elements have been filled. Therefore the *test* wrap defined below will return '{1, 2, 3, 4, 5, 6}':

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

The *testmore* wrap defined below will return '{1, 2, 3}':

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

6.3.2 Size of an Output Aray

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

6.4 RECEIVING CHARACTERS

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

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

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

Character arrays, on the other hand, are treated just like arrays of numbers. In the *strings2* procedure, the value returned for 'str' will be '123456':

```
defineps strings2(| char *str[6])
        (123) str
        (456) str
endps
```

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

6.5 COMMUNICATION AND SYNCHRONIZATION

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

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

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

See the *Client Library Reference Manual* for information concerning synchronization, run-time errors, and error handling.

A ERROR MESSAGES FROM THE *PSWRAP* TRANSLATOR

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

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

output parameter used as a subscript

char input parameters must be starred or subscripted

hex string too long

invalid characters

invalid characters in definition

invalid characters in hex string

invalid radix number

output arguments must be starred or subscripted

out of storage, try splitting the input file

-s 80 is the minimum

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

can't open file for input

can't open file for output

error in parsing

string too long

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

endps without matching defineps

errors in parsing
```

errors were encountered

size of wrap exceeds 64K

parameter reused

output parameter used as a subscript

non-char input parameter

not an input parameter

not a scalar type

wrong type

parameter index expression empty

parameter index expression error

end of input file/missing endps

B SYNTAX

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

```
Unit =
    ArbitraryText {Definition ArbitraryText}

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

Body =
    {Token}

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

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

ArgList =
     Type ItemList

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

ItemList =
      Item {"," Item}

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

Subscript =
      Integer | Ident
```

B.1 SEMANTIC RESTRICTIONS

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

B.2 CLARIFICATIONS

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

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).