the tool for software designers

PDL/81™

Format Designers Reference Guide

(Version 2.0)

Caine, Farber & Gordon, Inc. Warren Point International Ltd.

Comments or questions relating to this manual or to the subject software are welcomed and should be addressed to:

In North America:	*In the Rest of the World:*
Caine, Farber & Gordon, Inc. 1010 East Union Street Pasadena, CA 91106 USA	Warren Point International Ltd. Babbage Road Stevenage, Herts SG1 2EQ England
Tel: (800) 424-3070 or (818) 449-3070	Tel: 0438 316311
Fax: (818) 440-1742	Fax: 0227 86521

Form Number: 9102-1

1 August 1988
1 December 1991

Contents

1. Introduction

This manual presents the lowest-level external description of the PDL/81 processor and its Format Definition Language (FDL). It is intended for the sophisticated and knowledgeable user who is faced with the task of developing an entirely new PDL/81 document style definition or of making extensive modifications to an existing definition. It is *not* intended for the normal PDL/81 user who is concerned with processing a program design or formatting a manual or report. Such users should refer to the manuals which describe the various document styles.

Minor modifications to existing document style definitions can usually be made without reference to this manual. An examination of the definition file to be changed, and some familiarity with macro and text processing systems, are generally all that are required for minor modifications.

1.1 Features and Capabilities of PDL/81

PDL/81 is a tool which integrates the capabilities commonly associated with a program design language processor and those of a text processing system.

This integration is accomplished by providing an extensive set of primitive formatting operations and a definitional language which allows a format designer to compose abstract constructs from these primitive operations. As an example, the document style for program designs might contain such concepts as “data segment” and “flow segment” while a style for manuals might contain such concepts as “chapter”, “enumerated list”, and “paragraph”.

The end user of PDL/81 uses these abstract concepts without any need to understand the underlying implementation or format design methods. Thus, writing and processing program designs is as simple with PDL/81 as with other PDL processors, but the local project manager has significantly more control over the layout and appearance of the resulting design document.

The primitive operations of the Format Design Language allow the format designer a very high degree of flexibility in creating document styles. Among the available capabilities are:

- Complete control over page layout including sheet dimensions and top, bottom, left, and right margins;

- Arbitrary running text at top and bottom of each page including security banners with document classification and sheet count;
- Definition of primary and secondary keywords for use in program designs;
- Definition of layout and characteristics of all program design segment types and the ability to create new types of segments;
- Continuation of all input lines – both commands and text;
- Tab expansion on source input;
- Case insensitivity for all dictionary searches;
- Ability to include input from alternate files;
- Automatic generation of table of contents and other such tables (e.g., table of figures, table of tables);
- Automatic generation of document indexes in various forms.

1.2 Document Styles and the PDL/81 Style Library

The document styles which are available at an installation reside in the PDL/81 style library. The form of the data base depends on the particular host operating system. The particular style to be used in a PDL/81 run is specified as an option when PDL/81 is invoked.

1.2.1 Distributed Document Styles

Several document styles are distributed along with the PDL/81 processor. These include:

design	the standard program design style
manual	a style for formatting manuals
letter	a style for formatting a letter
memo	a style for formatting a memorandum
text	a style for formatting general text

1.3 Related Publications

Other publications relating to the use of PDL/81 are:

- *PDL/81 Introduction and Invocation Guide* – a guide to invoking PDL/81 under various operating environments
- *PDL/81 Design Language Reference Guide* – a guide to using PDL/81 for producing software design documents
- *PDL/81 Ada Design Language Reference Guide* – a guide to using PDL/81 for Ada program design
- *PDL/81 Document Language Reference Guide* – a guide to using PDL/81 for producing manuals and reports
- *PDL/81 Installation Guide* – a guide to installing PDL/81 under the various supported operating systems.

1.4 Warning to the Reader

This is a *reference* manual; it is *not* a tutorial manual. It is assumed that the reader has a good working knowledge of macro processors and text processing systems.

This manual is not intended to stand alone. It should be read in conjunction with the listings of one or more actual document style definitions and with the corresponding document style manuals.

2. General Information

This chapter discusses various aspects of the PDL/81 processor which are of general interest to all persons who intend to define new design styles or to modify existing style definitions. It includes information on the overall operation of the processor, low-level input line syntax and scanning, and the syntax and semantics of expressions.

2.1 Overall Operation of PDL/81

When PDL/81 is invoked, it first performs various initialization operations. Then, it locates the format definition file using various invocation options. This file should contain a series of function definitions which define the particular formatting style. The processing of this file is known as the “definition phase”. At the conclusion of the definition phase, the function “$dev-xxx” is invoked, where “xxx” is the device type specified by an invocation option or “default” if the option is not given.

If the “$dev-xxx” function does not exist, a search is made for the file “xxx.d” in the style library directory. If it is found, it is input and executed as if it had been contained in the definition file.

If the “$dev-xxx” function is not defined, a “xxx.d” file does not exist, and “xxx” was specified with an invocation option, PDL/81 processing is terminated. However, this device proocessing may be supressed by defining the “$NoDev” string.

When the “$dev-xxx” function returns, the function “$start” is invoked. As long as the “$start” function is in control, the processor is said to be in the “process phase”. When the “$start” function returns, the “$end” function is invoked. When that returns, PDL/81 exits.

If the functions “$start” or “$end” are not defined, the attempts to invoke them are ignored.

All processing of the source input and production of the formatted output must be performed by the “$start” function. Usually, this is accomplished by invoking one or more of the primitive functions “pass1”, “pass2”, and “pass3”. Each of these is given the name of a source file which is read and processed. The “pass1” primitive is used during the first pass of processing a program design where it is necessary to collect definitions of design segments. The “pass2” primitive is used to produce the formatted output. The “pass3” primitive provides for special processing where only command lines are processed.

These *passes* are described in detail in Section 3.1.

2.1.1 Processing Cycle

The main cycle for each phase of PDL/81 consists of:

```
DO UNTIL end-of-file on source
    input and collect a line
    IF the line contained function calls
        expand the line
        IF the expanded result is not null
            process the line ..the result of the expansion
        ENDIF
    ELSE
        process the line
    ENDIF
ENDDO
```

The operation of “input and collect a line” is described in Section 2.3.

The operation of “process the line” depends upon the state of PDL/81 at the time the line is encountered. Note that the operation is not performed for a line which contained a function call if, after expanding the line, the result is empty.

If the operation is performed during the definition phase, it is considered to be an error. Thus, the expansion of a line of the definition file must be empty.

If the operation is performed under control of the “pass1” primitive, the line is processed to collect such information as data item definitions.

If the operation is performed under control of the “pass2” primitive, the line is formatted for output just as if it had been the argument to a call on the “fm” primitive (see Section 4.7).

2.2 Special Characters

The following characters have special significance to the PDL/81 processor. Their interpretation is described in Section 2.3 and Section 2.4.

%	the *command character* which, when appearing as the first character of a line, signals that the line is a *command line*.
#	the *break* character which, in combination with other special characters, signals the start of a function call or a reference to a parameter.
{	the *left bracket* which, together with “}”, is used for grouping.
}	the *right bracket* which, together with “{”, is used for grouping.
;	the *argument separator* which is used to separate arguments in a function call.
?	the *argument count character* which, in conjunction with the break character, is used to obtain the number of arguments with which a function is called.
\	the *escape character* which has various uses as described in Section 2.3.1.

With the exception of the escape character, these characters are only given special treatment when used in the contexts described in this manual.

Each of these special characters may be redefined by use of the “cc” primitive function as described in Section 3.9.

In addition, a *continue character* can be defined with the “cc” primitive which, when used at the end of an input line, allows the line to be continued in the manner of PDL/74 (see Section 2.3.2).

2.3 Input Scanning

Each input line is scanned and collected one character at a time. The only ASCII control codes allowed are “tab” and “newline”. Each tab is replaced on input by enough blanks to position the immediately following character to the next input tab stop column (1, 9, 17, ...).

When the sequence “#{” is encountered, it is replaced by the internal meta code CALL and the following sequence of characters up to the next “}” or “;” is considered to be the name of the function to be called.

Once a “#{” has been encountered, the following translations occur:

1. #n –> PARM*n* (0 <= n <= 25)
2. ##n –> PARM*n* ARG PARM*n+1* ARG PARM*n+2* ARG ... PARM*max*
3. #? –> NPARM
4. { –> LBR
5. } –> RBR if it matches a LBR or –> ECALL if it matches a CALL
6. ; –> ARG

Thus, the line

```
abc#{xy;zd;{aa#{bb;c}#3}}wxy
```

will yield

```
abc CALLxy ARG zd ARG LBR aa CALLbb ARG c ECALL PARM3 RBR
ECALL wxy
```

The case (upper, lower, mixed) of a function name is immaterial.

After the line is collected, it is “expanded” if it contained any CALL’s. When the line is expanded, it is scanned from left to right and characters are copied to an “output string” which is the result of the expansion. Characters between an LBR and the matching RBR are simply copied. Encountering a CALL (outside of LBR/RBR) causes arguments (separated by ARG’s) to be collected until an ECALL is encountered. The called entity is then invoked, with the output of the invocation being placed to the right of the scan arrow so that the output, itself, will be scanned in turn. The meaning of “invocation” depends upon the type of the called entity:

primitive	the output, if any, depends on the definition of the primitive as described in this manual.
string	the output is the string with any NPARM replaced by the number of arguments (always at least one – the name of the string counts as one argument) and any PARMn replaced by the nth argument (argument 0 is the name of the string being called).

	Strings are defined with the “ds” primitive (see Section 3.2).
number register	the output is the value of the number register. Number registers are defined with the “nr” primitive (see Section 3.3).

In all cases, excess arguments are ignored and a reference to a missing argument is the same as a reference to an explicitly null argument.

Primitives, strings, and number registers share the same name space. Collectively, they are sometimes referred to as *functions*.

A CALL of an undefined entity yields a null output.

Scanning is fully recursive so that a CALL within a CALL will result in invocation of the inner entity before invocation of the outer.

2.3.1 Escape Character Interpretation

The escape character (“\”) is used to remove the special significance of special characters or to assign special significance to some normal characters. During input scanning, the following translations occur:

1. \# –> #
2. \{ –> {
3. \} –> }
4. \; –> ;
5. * –> BULLET (this is a special character used to represent a “bullet” on the output device. Chapter 8 describes the method of defining the printer sequence to generate a bullet)
6. \ followed by a space –> unpaddable space (will not be expanded or contracted during line justification; does not act as a word delimiter)
7. \ followed by a newline are both deleted, causing continuation to the next line
8. \ followed by decimal digit –> decimal digit

2.3.2 Continuation of Input Lines

Input lines may be continued in two ways:

- The sequence “\<newline>” results in deletion of both characters, thus causing the following line to be considered part of the current line.
- The sequence “<continue-character><newline>” will be replaced by a single blank, thus causing the current and following lines to be a single line with their contents separated by a blank. This continuation mechanism is provided for compatibility with that of the PDL/74 processor.

Within a CALL, leading white space on continuation lines is ignored.

2.4 Command Lines

If the first character of a line is the command character (“%”), the line is considered to be a *command line*. When a command line is encountered, white space following the command character is skipped. If a newline is encountered, the command line is ignored. If an asterisk (“*”) is encountered, the line is considered a *comment line*, the rest of the line is skipped, and the whole line is ignored.

If anything else is encountered, it is assumed to start a command name which extends to the first blank or newline. The line is then transformed so that

```
%name text
```

becomes

```
#{cname;text}
```

where “c” is:

0	if encountered during the definition phase
1	if encountered during “pass1” processing
2	if encountered during “pass2” processing
3	if encountered during “pass3” processing

Thus, the command line

```
%Include test.p
```

would become

```
#{2Include;test.p}
```

if encountered during “pass2” processing.

Normally, all text on the command line following the command name is taken as a single argument to the derived function name and no special character interpretation (except for escape character processing) is performed. This can be varied, however, by use of certain built-in number registers:

- If the “.cmdarg” built-in number register is assigned the value of “1”, detection of arguments on command lines is enabled and leading spaces will be removed from each argument. If the register is assigned the value of “2”, detection of arguments on command lines is enabled but leading spaces will not be removed from arguments other than the first.
- If the “.cmdcall” built-in number register has a non-zero value, detection of the “#{” sequence is enabled during command line collection.

Normally, an undefined command name (undefined derived function name) will result in the command line being quietly ignored. This can be changed, however, by assigning a value to the “.cmderr” built-in number register. If the value is “1” and an undefined command name is detected during “pass1” processing, an error message will be issued. If the value is “2” and an undefined command name is detected during “pass2” processing, an error message will be issued.

2.5 Expressions and Number Registers

Many PDL/81 primitives take arguments which are numeric values. Values can generally be represented by *expressions* as described in this section.

An expression is composed of operators, operands, and parentheses. There is *no* operator precedence – except where modified by parentheses, the order of evaluation is *strictly left to right*.

Error messages are not issued for ill-formed expressions. If an error is detected, the value computed to that point is usually returned.

A null expression yields a value of zero.

If an expression is preceded by a "+" or a "-", it is said to be a *relative* expression; otherwise, it is said to be an *absolute* expression. In evaluating a relative expression, the leading "+" or "-" is stripped before processing the rest of the expression. Relative expressions are normally used to increment or decrement something, as described for certain primitives in this manual. If the context of the use of a relative expression does not imply something to increment or decrement, the value of the expression is taken relative to zero. In such a case, "-1" is the same as "(-1)", but "-5+1" is the same as "-(5+1)" since it is taken as "0 decremented by 5+1".

2.5.1 Operands

An operand of an expression may be a *numeric constant*, a *character constant*, a *control character constant*, or a *number register*.

If an operand begins with a decimal digit, it is a numeric constant. The various types of numeric constants are:

- If the first two characters of the constant are "0x" or "0X", the constant is hexadecimal. The digits forming the constant should come from the set 0, 1, ..., 9, a, b, ..., f. The letters "a" through "f" may be given in upper or lower case.
- If the first character of the constant is "0" and the second character, if any, is neither "x" nor "X", the constant is octal. The digits forming the constant should come from the set 0, 1, ..., 7.
- Otherwise, the constant is decimal, and the digits forming the constant should come from the set 0, 1, ..., 9.

If an operand begins with the single-quote (') character, it is a character constant. The internal binary representation of the immediately following single character is taken as the value of the constant.

If an operand begins with the character '^', it is a control character constant. The value of the constant is the value of the rightmost five bits of the internal representation of the immediately following character. For example, "^h" has the value "8", which is the ASCII backspace control character.

If an operand begins with a letter or with one of the special characters ".", "$", or "_" it is the name of a number register. The remaining characters (if any) in the name come from the same set. No distinction is made between the upper and lower case forms of a letter. Number registers are the numeric valued variables of PDL/81 and are assigned values and are manipulated by the "nr" primitive (see Section 3.3).

2.5.2 Operators

The unary operators are:

+	unary plus (has no effect)
-	negation
!	not – result is zero if the argument is non-zero; otherwise the result is one

The binary operators are:

+	addition
-	subtraction
*	multiplication
/	integer division
%	remainder
&	and – result is one if both operands are non-zero; otherwise, it is zero
|	or – result is one if either operand is non-zero; otherwise it is zero
= or ==	equal to – one if left and right operands are numerically equal; otherwise, zero
!=	not equal to – one if left and right operands are not numerically equal; otherwise, zero
>	greater than – one if left operand is numerically greater than right operand; otherwise, zero
>=	greater than or equal to – one if left operand is numerically greater than or equal to right operand; otherwise, zero
<	less than – one if left operand is numerically less than right operand; otherwise, zero
<=	less than or equal to – one if left operand is numerically less than or equal to right operand; otherwise, zero

3. Definition and Control Primitives

This chapter discusses those Format Design Language primitives which are used for such purposes as defining strings and number registers, performing explicit input and output operations, and controlling the flow of execution. Chapter 4 discusses the primitives which are used for processing program design language input and for formatting output.

The description of each primitive gives the name of the primitive and the list of possible arguments. If an argument is described as being a *string*, a sequence of characters comprising the string, and *not* the name of a string, is meant. A sequence of characters is taken to be the name of a string *only* when explicitly so stated.

3.1 Execution Structure

Section 2.1 described the overall operation of the “process phase” which handles the actual processing of the source file. The two main drivers of this phase are the “pass1” and “pass2” primitives. There is also a “pass3” primitive driver that processes only command lines.

Each of these primitives takes one argument which is the name of the file to be processed. If the name is not given, is null, or is a single minus sign (“-”), the standard input file will be processed.

The name of the first (or only) source file specified at invocation of PDL/81 may be obtained by the “source” primitive:

```
#{source}
```

The “passsub{i}” primitives have the form

```
#{pass1;file}
```

```
#{pass2;file}
```

```
#{pass3;file}
```

and typical usage might be

```
#{pass1;#{source}}
```

During “pass1” processing, the following actions occur:

- The character “1” is prepended to each command name as described in Section 2.4;
- Each source line is scanned for explicit data item definitions as controlled by the “ddf” primitive (see Section 4.17) and for implicit segment definitions as defined by the “sdf” primitive (see Section 4.17).

During “pass2” processing, the following actions occur:

- The character “2” is prepended to each command name as described in Section 2.4;
- Each source line (except for those that contain function calls which result in a null line) is scanned for implicit data item definitions as controlled by the “ddf” primitive (see Section 4.17), for data item references as controlled by the “drf” primitive (see Section 4.17), and for segment references as controlled by the “srf” primitive (see Section 4.17). The line is then formatted for output by passing it to the “fm” primitive (see Section 4.7).

During “pass3” processing, the following actions occur:

- The character “3” is prepended to each command name as described in Section 2.4;
- Each command line is processed and all others are skipped.

Note that the term “pass3” is simply a name for this kind of processing. It is not intended to be indicative of processing order.

The primitive

```
#{$file}
```

returns the name of the *current* file being processed, whether as a result of definition processing, “pass1” processing, “pass2” processing, “pass3” processing, or “include” processing.

The primitive

```
#{scall;func}
```

calls the function *func* once for each source file given in the PDL/81 invocation line. At each call, *func* is given that source file name as its single argument. Thus, if PDL/81 were invoked under Unix as

```
pdl81 file1 file2 file3
```

or under VMS as

```
pdl81 file1,file2,file3
```

the call

```
#{scall;pass1}
```

would have the same effect as

```
#{pass1;file1}
#{pass1;file2}
#{pass1;file3}
```

If no source files were given when PDL/81 was invoked, “scall” will call *func* with no argument.

3.1.1 Establishing a Source Line Trap

The primitive

```
#{strap;function-name}
```

establishes a *source trap*. Once established, normal source lines and those which are not null after expanding any function calls will not be processed normally by PDL/81. Instead, the named function will be invoked with the line as its argument. Operation will continue in this way until the source trap is canceled by a call on “strap” without a function name.

3.2 String Definition Primitives

A string is a sequence of characters which is given a name and stored in the PDL/81 primary dictionary.

A string is defined by the “ds” (Define String) primitive

```
#{ds;name;contents}
```

where *name* is the sequence of characters by which the string will be referenced and *contents* is a sequence of characters to be placed in the string.

If a “ds” names an existing string, that string is replaced. It is an error for a “ds” to specify the name of a primitive or a number register.

The “as” (Append to String) primitive has the form

```
#{as;name;text}
```

which causes the characters of *text* to be appended to the contents of the string given by *name*. If the named string does not exist, the “as” primitive is treated as if it were “ds”. It is an error to apply “as” to a primitive or a number register.

3.3 Number Register and Evaluation Primitives

A number register is defined, or a new value is assigned to an existing number register, by

```
#{nr;name;value-expr}
```

Where *name* is the name of the number register (the syntax of number register names which are to be referred to in expressions is defined in Section 2.5) and *value* is an expression representing the value to be assigned to the number register. If *value* is a relative expression, it will be taken relative to the existing value of the register (or to zero, if the register does not yet exist). It is an error to apply the “nr” primitive to a string or a primitive.

An expression can be evaluated and the value returned as a string by:

```
#{ev;expression}
```

The value will be in the form of a decimal integer with a leading minus sign if the value of the expression is negative.

An expression can be evaluated and the value returned in one of several notational forms by:

```
#{nf;format;expression}
```

where *format* may be one of:

a	the value will be returned as 0, a, b, ..., z, aa, bb, ..., zz, aaa, ...
A	the value will be returned as 0, A, B, ..., Z, AA, BB, ..., ZZ, AAA, ...
i	the value will be returned as 0, i, ii, iii, iv, v, ...

I	the value will be returned as 0, I, II, III, IV, V, ...
1	the value will be returned as 0, 1, 2, ... (this is the same form as returned by the “ev” primitive)

If the value is negative, the result will be prefixed with a minus sign.

The primitive

```
#{eq;string1;string2}
```

compares the two strings and returns “1” if they are equal and “0” if they are not equal.

3.4 Duplication and Deletion

The primitive

```
#{del;name1;name2;...}
```

deletes the named item(s) which can be primitives, strings, or number registers. It is an error to attempt to delete an item if the first character of its name is a period (.). These are known as “permanent” items.

An item can be duplicated by:

```
#{dup;old;new}
```

where *old* is the name of a primitive, string, or number register. If *old* does not exist, it is taken to be the name of a null string. An item named *new* is created with all of the attributes of the item named *old*. It is an error to change the type (primitive, string, number register) of *new* if its name begins with a period (.).

3.5 Execution Control

The “if” primitive tests the value of an expression and returns one of two strings:

```
#{if;expression;string1;string2}
```

If the value of *expression* is non-zero, the result is *string1*; otherwise, the result is *string2*.

The “ifdef” primitive tests whether a given name corresponds to that of a currently defined item (primitive, string, number register) and returns one of two strings:

```
#{ifdef;name;string1;string2}
```

If *name* is defined, the result is *string1*; otherwise, the result is *string2*.

The “case” primitive performs a “one out of n” selection:

```
#{case;expression;string0;string1;string2;...}
```

The expression is evaluated and the string corresponding to the value is returned. A value less than zero is taken to be zero.

The primitive

```
#{loop;function-name;arg1;arg2;...;argn}
```

will cause the named function to be executed *n* times – once for each of the *arg* arguments and that argument will be passed as an argument to the function.

The primitive

```
#{lindex;key;string1;string2;...;stringn}
```

compares *key*, in a case insensitive way, with each string in turn. It returns 0 if there is no match; otherwise, it returns returns the number of the first string that is identical to the key.

The primitive

```
#{call;name;string1;string2;...}
```

is equivalent to

```
#{name;string1;string2;...}
```

except that *name* is treated as any other argument to a function and, thus, may contain function calls whose values are used to construct the actual name of function to be invoked.

The primitive

```
#{exit;status}
```

causes immediate exit from PDL/81. If the host operating system can accept an exit status value from a process, “status” is returned as that value. A value of zero is used if “status” is not provided.

3.6 String Processing Primitives

The primitive

```
#{substr;start-expr;length-expr;string}
```

returns a substring of *string* beginning at position *start* and extending for *length* characters. Characters in a string are numbered starting at one. If *length* is null, the substring extends to the end of *string*.

The length of a string, in characters, may be obtained by

```
#{length;string}
```

A string may be replicated by the “rep” primitive

```
#{rep;count-expr;string}
```

whose value is *string* replicated *count* times.

Special characters, such as device control codes, can be inserted into a string by:

```
#{sneak;expr1;expr2;...}
```

The expressions are evaluated and the resulting characters are concatenated to form the value of the primitive. Non-printing characters should be restricted to special purposes such as use with the “Out” primitive (Section 3.7.3) and in font control strings (Chapter 5). Note that 8-bit characters may be introduced into strings by this primitive.

3.7 Primitives for Input and Output

Most input and output in PDL/81 is performed implicitly as a result of the overall processing structure. However, provision is made for certain kinds of explicit input and output as described in this section.

3.7.1 Including Alternate Source Files

The primitive

```
#{include;file}
```

causes the current processing phase (definition phase, “pass1” processing, or “pass2” processing) to be invoked recursively to process the named file. When the end of file is reached, the primitive returns with a null value.

The primitive

```
#{lib;name}
```

operates in the same manner as the “include” primitive except that the file name is searched for in the data base library directory.

3.7.2 Input and Output to Auxiliary Files

Auxiliary files are identified by *file numbers* in the range zero through fourteen. In the descriptions below, *fn* will be used to refer to a file number.

An auxiliary file is opened and assigned a referencing file number by

```
#{open;fn-expr;file;special}
```

where *file* is the name of a file to be opened. The file is opened in *output* mode and the previous contents, if any, of the file are deleted. If the file does not exist, it is created. If the *file* argument is missing or null, the specified file number will be connected to the standard output file. If the *special* argument is present and non-null, the file will be opened in an operating system specific way which, for VMS, will supply the attributes

```
"rfm=var", "rat=cr", "mrs=255"
```

Information is written on an auxiliary file by

```
#{send;fn-expr;string;key-1;key-2;...;key-n}
```

If the specified file has not been opened with the “open” primitive, a temporary file will be created. The *string* argument will be written as the next line of the file. If the any *keys* are supplied, provision will be made so that when the file is input with the “rcv” primitive, the lines will be read in order sorted on the keys. Such a sorted file *must* be a temporary file and if any “send” for a file contains a key, every “send” for that file must contain a key.

If any *key*s are present, the file is said to be a *keyed* file and each record is composed of a series of fields which are, in order from left to right, the payload string, cardinal number of the record in the file represented as a string, and the key strings. For reference purposes, the fields are considered numbered starting at zero.

If *string* contains any internal meta codes (e.g., CALL, ARG, LBR) as described in Section 2.3, the auxiliary file should only be a temporary file since the internal form of some meta codes is specific to a given invocation of PDL/81. Escaped characters should be used to send lines containing function calls to non-temporary files.

An auxiliary file may be input and deleted by

```
#{rcv;fn-expr;control-1;control-2;...;control-n}
```

where the *control*s are strings which are used to control the sorting of file.

If there are no controls specified and if the file is not keyed, it is just received as written. If there are no controls but the file *is* keyed, it is first sorted on field 2 (the first key given with “send”) and then by field 1 (the sequence number).

If controls are given, the file is first sorted as specified by the controls. Each control consists of a field number followed immediately by one or more option codes. The option codes are chosen from the set:

u	Return only one of a set of identical (by sort criteria specified) records
b	Skip leading white space in this field before comparing.
f	Fold any upper-case characters in this field to lower case.
i	Ignore non-printing characters in this field.
n	Compare leading numeric strings on this field numerically.
s	Treat this field as a multi-part paragraph number and sort accordingly.

For example,

```
#{rcv;1;2;1n}
```

is equivalent to

```
#{rcv;1}
```

if the file is keyed.

An open, non-temporary, auxiliary file may be closed by

```
#{close;fn-expr}
```

3.7.3 I/O to Standard Output and Standard Error Files

The primitive

```
#{ps;string;code}
```

will display *string* on the standard error file. A newline will be appended to the string if *code* (which is otherwise ignored) is null. The “ps” (Print String) primitive is intended for displaying messages and for debugging. See Section 3.8 for primitives which are intended for issuing error messages.

The primitive

```
#{out;string}
```

immediately writes *string* on the standard output file. This is intended for outputting such things as device control codes and initialization sequences.

3.7.4 Miscellaneous Input and Output Primitives

The primitive

```
#{access;file;lib}
```

returns the character “1” if *file* exists and is readable; otherwise, it returns null. If the *lib* argument is present and non-null, the file is searched for in the style library directory.

The primitive

```
#{base;string}
```

considers *string* to be a file name in the syntax of the host operating system and returns the base portion of the name. Under Unix, for example, this will be the name with any extension stripped. Thus,

```
#{base;abc/def.hi}
```

yields “abc/def”.

The primitive

```
#{backup}
```

will cause the source line last read in the normal process cycle to be reread as the next source line.

3.8 Error Reporting Primitives

Error messages may be issued by

```
#{error;message-string}
```

which reports the error and continues processing, or by

```
#{quit;message-string}
```

which reports the error and terminates PDL/81 processing.

If applicable, the message will be prefixed with the name of the current source file and the line number within that file of the line which caused the error.

3.9 Special Character Redefinition

The special characters described in Section 2.2 may be redefined by

```
#{cc;code-expr;char}
```

where *code* specifies the special function to be redefined and *char* is the new character to assume that function. If *char* is null, that special function is deleted. The possible values of *code* are:

0	command character (%)
1	break character (#)
2	left bracket ({)
3	right bracket (})
4	escape character (\)
5	continue character (no default)
6	argument separator character (;)
7	argument count character (?)

For example, the command

```
#{cc;5;/}
```

will define the continue character to be a “/”.

3.10 Debugging Primitives

The primitive

```
#{tn}
```

turns on trace mode and the primitive

```
#{tf}
```

turns off trace mode. When trace mode is on, the name and arguments of each function are displayed just prior to calling the function. If the pause mode invocation option was specified, the processor will pause after printing the trace information and will wait for a newline from the standard input before proceeding.

The primitive

```
#{pns;name}
```

will display the name, type, and contents (except for primitives) of the item named *name*.

3.11 Processor Information Primitives

Various information about the identification of the PDL/81 processor and about its execution may be obtained by the primitives described in this section.

The primitive

```
#{lpn}
```

returns the Licensed Program Number of the PDL/81 processor.

The primitive

```
#{ver}
```

returns the version number of the PDL/81 processor.

The primitive

```
#{ver2}
```

returns a secondary version number for the PDL/81 processor.

The primitive

```
#{memuse}
```

returns an integer showing the amount of dynamic memory currently in use by the PDL/81 processor. It may be displayed for statistical purposes.

The primitive

```
#{systype}
```

returns a string which indicates the type of the host operating system. Among these types are:

unix	most Unix systems
vms	the VMS operating system
dos	the MSDOS or PCDOS operating systems

3.12 Saving the Dictionary

The current PDL/81 dictionary may be saved by the primitive

```
#{dump;file-name}
```

which causes the dictionary to be dumped in encoded form to the named file. The file is prefixed with a *magic number* so that special processing will take place when the file is later input to the PDL/81 processor.

Dumping saves most of the dictionary items. However, only the following permanent number registers are dumped: .stree, .cwidth, .dclass, .cmdcall, .cmdarg, .cmderr, .noff, .notab, .boxfont, .mcfont, .nobs, and .po.

Note that this primitive is intended only to support preprocessed style files and is not a general mechanism.

4. Formatting and Design Primitives

This chapter discusses those Format Design Language primitives which are used for such purposes as specifying modes of formatting, defining page dimensions and margins, defining segments and data items, and preparing various indexes.

The description of each primitive gives the name of the primitive and the list of possible arguments. If an argument is described as being a *string*, a sequence of characters comprising the string, and *not* the name of a string, is meant. A sequence of characters is taken to be the name of a string *only* when explicitly so stated.

A number of primitives described in this chapter use an argument described as a “setting” to set the state of some internal control variable. If such an argument is null, or if its first character is the digit “0”, or if its first character is the letter “n” (upper or lower case), or if its first two characters are the letters “of” (upper or lower case), the internal control variable is placed in its off state. Otherwise, it is set to the first character of the argument. The meanings of the various settings are discussed with each such argument.

4.1 Page Size Specification

The overall dimensions of the output page are defined by

```
#{psize;width-expr;depth-expr}
```

where *width* specifies the width of the page in characters, and *depth* specifies the depth of the page in lines. Prior to the first use of “psize”, the page width will be 120 characters and the page depth will be 66 lines. The maximum supported page width is 256 characters and the maximum supported page depth is 330 lines. If *width* is a relative expression, it is taken relative to the current page width. If *depth* is a relative expression, it is taken relative to the current page depth.

If the page depth is set to zero by “psize”, the output will be considered as a single page of indefinite depth. The page footer trap (Section 4.2) will never be sprung and the “bp” primitive (Section 4.3) will be ignored.

The primitive

```
#{$width}
```

returns the current page width and the primitive

```
#{$depth}
```

returns the current page depth.

4.2 Heading and Footing Traps

The primitive

```
#{head;name}
```

specifies that *name* is the name of a function to be invoked by PDL/81 just before processing the first line of a page. This so-called “header trap” can be used to format and print page headers.

The primitive

```
#{foot;name;loc-expr}
```

specifies that *name* is the name of a function to be invoked by PDL/81 just before processing the line following the line whose number is given by *loc*. If *loc* is a relative expression, it is taken relative to the page depth specified by the “psize” primitive. The maximum value of the trap location is the current page depth. When the trap function returns, PDL/81 positions the output to the top of the next page. The current position of the footing trap (i.e., the line number given in the last “foot” primitive) is returned by

```
#{$foot}
```

The footing trap will never be sprung if the page depth is set to zero with the “psize” primitive (Section 4.1).

4.2.1 Forcing Heading and Footing Traps

If output is positioned at the top of a page but the heading trap has not been sprung, the primitive

```
#{fht}
```

will spring the heading trap.

If the line with the number given in the “foot” primitive has been printed but the footing trap has not been sprung, the primitive

```
#{fft}
```

will spring the trap.

4.3 Vertical Spacing

Vertical spacing is performed by

```
#{sp;lines-expr}
```

where *lines* specifies the number of lines to space. If the argument is null, it is taken as one. If a footing trap is encountered while spacing, the space request is terminated and the trap is sprung.

A new page is started by

```
#{bp}
```

which is equivalent to a “sp” request with a line count sufficient to spring the footing trap. If the page depth is set to zero with the “psize” primitive (Section 4.1), the “bp” primitive will be ignored.

Both the “sp” and the “bp” primitives force a line break (see Section 4.6). Spacing can be inhibited by

```
#{spcok;setting}
```

where *setting* is

off	vertical spacing requests (“sp”, “bp”) will be ignored until either spacing is resumed with the “spcok” primitive or until the next line has been printed
other	vertical spacing is resumed

The current vertical position on the output page is returned by

```
#{$line}
```

The first line on a page is line number one. However, “$line” will return a value of zero if output is positioned at the top of a page but the heading trap has not been sprung.

4.4 Horizontal Margins and Indenting

The left and right margins define the horizontal boundaries within which output formatted by the “fm” primitive (see Section 4.7) is placed.

The current left margin is set by

```
#{lm;margin-expr}
```

where *margin* specifies the margin with a value of one referring to the leftmost column on the page. If *margin* is a relative expression, it is taken relative to the current left margin. If *margin* is null, the left margin is restored to its previous position. The initial left margin is set at position one. The position of the current left margin is returned by

```
#{$lm}
```

The current right margin is set by

```
#{rm;margin-expr}
```

where *margin* specifies the position of the desired right margin. It may not exceed the page width. If *margin* is a relative expression, it is taken relative to the current right margin. If *margin* is null, the right margin is restored to its previous setting. The position of the current right margin is returned by

```
#{$rm}
```

The primitive

```
#{in;indent-expr}
```

specifies that each formatted output line is to be indented the number of positions specified by *indent*. If *indent* is a relative expression, it is taken relative to the current indentation setting. The minimum indentation value is zero, which implies no indentation. If *indent* is null, the indentation is restored to its previous value. The current indentation is returned by

```
#{$in}
```

The primitive

```
#{ti;indent-expr}
```

sets indentation for the *next* formatted line. Thus, it provides a temporary indentation. If *indent* is a relative expression, it is taken relative to the current indentation as set by the “in” primitive. Use of “ti” in any formatting mode other than *filled* (see Section 4.6) is undefined.

The “lm”, “rm”, “in”, and “ti” primitives force a line break (see Section 4.6).

4.5 Design Oriented Special Characters

PDL/81 supports both *data characters* and *comment strings* for use in design segments.

A data character is used to flag a word as being a *data item*. The definition and referencing of data items are controlled by the “ddf” primitive (see Section 4.17) and the “drf” primitive (see Section 4.17), respectively. Data characters may be defined by

```
#{dc;char1;char2;...}
```

where each *char* is a string whose first character will become a data character. A null argument causes deletion of all previously defined data characters.

Normally, data item names are composed of alphanumeric characters and data characters (if any are defined). Any other characters are considered to be break characters which delimit the names. Special characters, known as *data special characters*, can be included in data item names by declaring them with

```
#{dsc;char1;char2;...}
```

where each *char* is a string whose first character will become a data special character. A null argument will cause deletion of all previously defined data special characters.

A comment string is used to indicate the point in a line where scanning is to stop for segment definition and segment referencing. These actions are controlled by the “name” (see Section 4.9), “sdf” (see Section 4.17), and “srf” (see Section 4.17) primitives. Comment strings may be defined by

```
#{cm;string1;string2;...}
```

where each *string* is to become a comment string. Comment strings may be at most two characters long and no two comment strings may have the same first character. A null argument will cause deletion of all previously defined comment strings.

4.6 Modes of Automatic Formatting

The current mode of automatic formatting is selected by

```
#{fmt;setting}
```

where *setting* may be one of:

off	*nofilled* mode is selected.
f	*filled* mode is selected
s	*structured* mode is selected

Output to be formatted in one of these modes is specified by use of the “fm” primitive (see Section 4.7) either by an actual call on this primitive or by an implicit call as generated during “pass2” processing.

In *nofilled* mode, each explicit or implicit call on “fm” is considered to represent a line. If the line cannot fit within the current output boundaries (from left margin plus indent to right margin), the line will be split following a word and the continued portion will be further indented by an amount known as the *tab width*. The tab width is normally four characters but may be redefined by

```
#{tabw;width-expr}
```

If *width* is a relative expression, it is taken relative to the current tab width. The current tab width is returned by

```
#{$tabw}
```

In *filled* mode, each explicit or implicit call of “fm” is considered to represent an input line, but words are collected, regardless of input line boundaries, and are put into the output line until a word does not fit. The output line is then printed and a new output line is started. This action is known as “breaking” the line and may be forced by

```
#{br}
```

A break is also forced by the primitives “bp”, “ce”, “env”, “envs”, “fft”, “fht”, “fmt”, “in”, “lm”, “rm”, “sp”, “ti”.

In *structured* mode, each explicit or implicit call on “fm” is considered to be a design statement. If the line begins with a keyword (see Section 4.11), the definition of the keyword will control the indenting of the statement. If the statement cannot fit between the output boundaries, it will be continued with continuation lines being indented twice the current tab width.

4.6.1 Line Justification

If *justified* mode is selected, output lines will be justified. Justification can be turned on and off by

```
#{just;setting}
```

where *setting* can be

off	turn off justification
other	turn on justification.

Justification is initially off.

A line is justified by expanding inter-word spaces so that the right end of the line is at the current right margin of the page. For most output devices, inter-word spaces are expanded by integral multiples of the character width. However, some devices allow a finer horizontal resolution so that PDL/81 may distribute the extra inter-word spacing more equally during justification. This type of justification is accomplished by setting the “.cwidth” number register to the width of a character in device basic horizontal resolution units.

For example, the Diablo 1620 uses 1/60th of one inch as its horizontal resolution in graphics mode. Thus, at 12 characters per inch, “.cwidth” should be set to 5 to request equalized justification. When PDL/81 needs to output less than a full space character, it first outputs the contents of the special string “*gon”. It then outputs the required number of space characters. Finally, it outputs the contents of the special string “*goff”. For example, the following definitions can be used for a Diablo 1620:

```
#{nr;.cwidth;5}
#{ds;*gon;#{sneak;033;’3}}
#{ds;*goff;#{sneak;033;’4}}
```

4.6.2 Spaces and Blank Lines

Leading spaces on input lines can be ignored or kept by

```
#{lsp;setting}
```

where *setting* may be

r	leading spaces are removed
k	leading spaces are kept

Initially, leading spaces are kept.

Imbedded spaces on input lines can be kept or compacted by

```
#{isp;setting}
```

where *setting* may be

c	imbedded spaces are compacted (i.e., a sequence of more than one space is replaced by one space)
k	imbedded spaces are kept

Initially, imbedded spaces are kept.

The action of PDL/81 on encountering a blank (or empty) input line can be specified by

```
#{bll;setting}
```

where *setting* may be

r	blank lines are removed
k	blank lines are kept. Each blank line will act as a call on the “sp” primitive to space one line and will force a line break.
a	in *filled* mode, blank lines trigger the “automatic paragraph” mechanism which will cause the user function “$pp” to be invoked; in *nofilled* mode, blank lines will be treated as described for the “k” setting, above.

4.7 Output Formatting

The primary way of formatting output is with the primitive

```
#{fm;string}
```

where *string* is considered to be the text of a line to be formatted and printed in the manner controlled by the current formatting modes (see Section 4.6). This primitive is invoked implicitly during “pass2” processing as described in Section 2.1 and Section 3.1.

A line may be centered between the current left and right margins by

```
#{ce;text}
```

A three-part line (as in a page heading or footing) may be output by

```
#{ttl;left-string;center-string;right-string}
```

which will cause *left* to be printed flush with the left margin, *right* to be printed flush with the right margin, and *center* to be centered between the left and right margins.

The primitive

```
#{stuff;loc-expr;text}
```

places the characters of *text* into the output line starting at the position given by *loc*. If *loc* is a relative expression, it is taken relative to the current output position. The specified position may be to the left of the left margin or to the right of the right margin.

The primitive

```
#{leader;leader-string;prefix-string;suffix-string}
```

defines a *leader* string, *prefix* string, and *suffix* string in the current environment. If a leader string is defined and a line break occurs, the space between the end of the line and the right margin is filled by placing the prefix string at the left, the suffix string at the right, and as many replications of the leader string as are needed to fill the remaining space.

4.7.1 Establishing a Format Trap

The primitive

```
#{fmtrap;function-name}
```

establishes an *fm trap*. Such a trap will be sprung, and the given function invoked, just before processing the next “fm” primitive. When the trap is sprung, it is deleted so that the named function may, if desired, use the “fm” primitive. For example, the definitions

```
#{ds;xxx;#{fm;(TS)}}
```

and

```
#{fmtrap;xxx}
```

would cause “(TS)” to be displayed just in front of the output from the next “fm”.

4.8 Extra Line Spacing

The primitive

```
#{els;n-expr}
```

will cause *n* blank lines to be inserted following each non-blank line printed. If the footer trap is sprung while printing the blank lines, any remaining blank lines will be absorbed.

4.9 String Processing Primitives

The primitive

```
#{width;string}
```

returns the number of characters which *string* would occupy if printed. This calculation takes into account the setting of the “imbedded spaces” mode as set by the “isp” primitive.

The primitive

```
#{name;string}
```

returns the contents of *string* up to, but not including, the first comment string. Leading and trailing spaces will be removed, and imbedded spaces will be compacted.

The primitive

```
#{sqz;string}
```

returns its argument with each sequence of more than one consecutive blank replaced by a single blank.

The new primitive

```
#{nargs;string}
```

returns a count of the number of *arguments* encountered in the string. The string is assumed to be a *call* to a flow segment and may have an argument list enclosed in parentheses. If a list is found, the arguments are assumed to be strings separated by zero-level (with respect to other parentheses and to single and double quote marks) commas.

4.10 Box Drawing

An automatic box is started by

```
#{box;lm-expr;rm-expr;h-char;lv-char;rv-char;c-char}
```

where the arguments are

lm	specifies the left margin of the box
rm	specifies the right margin of the box. If this is a relative expression, it is taken relative to the left margin of the box.
h	the character to draw horizontal lines
lv	the character to draw the left vertical edge of the box
rv	the character to draw the right vertical edge of the box
c	the character to form the corners of the box

The abbreviated form

```
#{box;lm-expr;rm-expr;x-char;rv-char}
```

will use *x* as the *h*, *lv*, and *c* characters, and the abbreviated form

```
#{box;lm-expr;rm-expr;y-char}
```

will use *y* as all of the characters.

An expanded form, which provides for specifying all portions of the box, is also available:

```
#{box;lm-expr;rm-expr;h-char;lv-char;rv-char;
        ul-char;ur-char;ll-char;lr-char}
```

where the additional arguments are

ul	character to draw upper-left corner
ur	character to draw upper-right corner
ll	character to draw lower-left corner
lr	character to draw lower-right corner

The “box” primitive will draw the top of the box and arrange that any subsequently printed lines will be enclosed in the appropriate vertical edge characters.

All box drawing characters will be printed in the font specified by the “.boxfont” number register.

A box is terminated by

```
#{ebox}
```

which will draw the bottom of the box.

Boxes may not be nested.

4.11 Secondary Dictionary Operations

The so-called *secondary dictionary* is the repository for such items as keywords, secondary keywords, flow segment names, and data item names. An entry is made in the secondary dictionary by

```
#{dx;type-expr;name-string;code-expr;
         val1-expr;val2-expr;val3-expr;val4-expr}
```

where *type* specifies the type of entry to be made, *name* is the name of the entry, and *code* and the *valsubi* depend upon the type of entry. The maximum value of “code” is 255. If the entry was previously defined, the call on “dx” overrides this and the primitive returns a “1”; otherwise, the definition is made and the primitive returns a “0”.

There is a separate name space for each of the entry types. The possible types are:

1 the entry is a *keyword*. The “val1” argument specifies the indent to apply prior to printing the line; the “val2” argument specifies the indent to apply after printing the line. The indents are interpreted as integral multiples (positive or negative) of the current tab width (see Section 4.6). The “val3” argument controls flow figure checking (Section 4.12). The low-order eight bits of the “val4” argument represent the cyclomatic complexity of the keyword; the high-order 8 bits are flags (0x01 means do not process what follows for references). Keywords have significance only in *structured* mode formatting (see Section 4.6). Keywords are printed in the current keyword font (see Section 5.4).

2 the entry is a *secondary keyword*. The low-order eight bits of the “val3” argument represent the cyclomatic complexity of the keyword; During *structured* mode formatting, secondary keywords are recognized as such if they immediately follow a keyword or another secondary keyword. Secondary keywords are printed in the current keyword font (see Section 5.4).

3 the entry is a *flow segment name*. The “code” argument may be any desired value and is usually used to distinguish various types of flow segments. The code value is made available during index processing (see Section 4.27). The “val1” argument should be the page number of the definition and the “val2” argument should be the line number of the definition (if applicable).

4 the entry is a *data item name*. The “code” argument may be any desired value and is usually used to distinguish various types of data items. The code value is made available during index processing. The “val1” argument should be the page number of the definition and the “val2” argument should be the line number of the definition (if applicable).

5 Defines an executable pass 1 keyword (Section 4.13). The “val1” argument is the name of the function to be executed.

6 Defines an executable pass 2 keyword (Section 4.13). The “val1” argument is the name of the function to be executed.

If *xsw* 4 (Section 4.21) is set during pass one, the complexity measure for each keyword scanned in the source is added to the value in the “.ixcmplx” number register. If *xsw* 8 is set during pass two, the complexity measure for each keyword scanned in the source is added to the value in the “.ixcmplx” number register.

During automatic cross-reference collection (see Section 4.17, in “pass2” processing, the current flow segment must be known to PDL/81. This is accomplished by

```
#{sx;name-string}
```

where *name* is the name of the current flow segment which must have been defined with the “dx” primitive.

The primitive

```
#{fx;type-expr;name-string}
```

may be used to search the secondary dictionary for a named item of a particlar type and to retrieve information about that entry. If the named entry is *not* found with the specified type, a value of “0” is returned. If it *is* found, the value “1” is returned and, in addition, the following special number registers are set:

.ixdcode	The “code” value specified when the item was defined with the “dx” primitive.
.ixdpage	The “val1” value specified when the item was defined with the “dx” primitive.
.ixdline	The “val2” value specified when the item was defined with the “dx” primitive.
.ixdval3	The “val3” value specified when the item was defined with the “dx” primitive.
.ixdval4	The “val4” value specified when the item was defined with the “dx” primitive.

4.12 Flow Figure Checking

The *val3-expr* as set by the “dx” primitive is used to contain a *class/code* pair in the form *class*256+code*.

The classes are used to distinguish between flow figures and are positive numbers. The codes are used to distinguish among types of keywords within a given flow figure and are in the range 0–5, inclusive. The codes are used to access the state table

	0	**1**	**2**	**3**	**4**	**5**
0	0	1	4	5	5	5
1	0	1	2	3	3	3
2	-	-	-	-	-	-
3	0	1	2	6	3	3
4	0	1	2	6	3	3
5	0	1	2	6	6	6

where the column headings correspond to the *code* of an incoming keyword, the row headings correspond to the *code* at the top of the *keyword stack*, and the table entries are *actions* to be performed.

The *codes* may be thought of as corresponding to

0 As an incoming keyword, this is one that doesn’t make any change in the structure (such as *RETURN* or *UNDO*). Since these are never pushed on the stack, a *code* of 0 on the stack means the stack is empty.

1 A figure-opening keyword (such as *DO* or *IF*).

2 A figure-ending keyword (such as *ENDDO* or *ENDIF*).

3 An intermediate keyword that may only occur once following a figure-opening keyword.

4 An intermediate keyword that may occur any number of times after a *code* 2 or 3 keyword. *ELSEIF* is an example.

5 An intermediate keyword that may occur once after a *code* 2, 3, or 4 keyword and may not be followed by a *code* 3 or 4 keyword. *ELSE* is an example.

A separate *class* should be assigned to each flow figure. For example, the *IF...ENDIF* figure might be *class* 1 and the *DO...ENDDO* figure might be *class* 2.

The *actions* are

0 Do nothing.

1 Push the *class* and *code* of the incoming keyword onto the stack.

2 If the incoming *class* matches that on the top of the stack, pop the stack. Otherwise, invoke the function “$kwerr” with an argument of 3.

3 If the incoming *class* matches that on the top of the stack, pop the stack and push the *class* and *code* of the incoming keyword onto the stack. Otherwise, invoke the function “$kwerr” with an argument of 3.

4 Error: invoke the function “$kwerr” with an argument of 0.

5 Error: invoke the function “$kwerr” with an argument of 1.

6 Error: invoke the function “$kwerr” with an argument of 2.

Flow figure processing is enabled in pass one if *xsw* 16 is set and in pass two if *xsw* 32 is set. Processing may be done explicitly by the primitive

```
#{kwcheck;class-expr;code-expr}
```

In the special case where *class* is negative, *code* is interpreted as

0 Reset the keyword stack.

1 Return the depth of the keyword stack.

4.13 Executable Keywords

Keywords are defined to be “executable” by defining type 5 or type 6 secondary dictionary entries with the “dx” primitive (Section 4.11). Such keywords will not, however, be “executed” unless such execution has been enabled for the current environment (Section 4.21).

When keyword execution is enabled for a pass and a line begins with an executable keyword appropriate to that pass (as defined by the “dx” primitive), the associated function (as specified by the “val1” argument to the “dx” function) will be invoked as

```
#{func;kw-string;rest-string}
```

where “kw” is the keyword that caused the invocation and “rest” is the remainder of the input line. In addition, the “.ixdcode” number register will be set to the “code” value that was given when the keyword was defined.

If a line results in keyword execution, no other processing (including printing during pass two) is performed on the line. All desired processing must be performed by the invoked function.

4.14 Specifying Keyword Alignment

The primitive

```
#{kwlm;loc-expr}
```

specifies the leftmost position in which a keyword may be automatically placed. If the processor attempts to place a keyword to the left of “loc”, the user-defined function “$inderr” will be invoked.

4.15 Case Control

The primitive

```
#{kwcase;setting}
```

specifies the case in which keywords and secondary keywords are to be printed during *structured* mode formatting. The value of *setting* may be:

off	do not change the case of the keyword
u	print the keyword in upper case
l	print the keyword in lower case

The primitive

```
#{cap;text}
```

returns *text* with each lower-case alphabetic character promoted to upper-case.

The primitive

```
#{icap;string}
```

returns *string* with the first character promoted to upper-case if it is a lower-case letter.

4.16 Flow Figure Enhancement

The primitive

```
#{kwvc;char;font-expr}
```

specifies that the *char* character in the given font will be used to connect the beginning and end of each flow figure. *Font* defaults to zero (the base font). If *char* is null or blank, the mechanism is turned off.

For keywords automatically processed in structured mode, the mechanism requires that the appropriate *code*, as used by the keyword checking mechanism (Section 4.12), be defined with the "dx" primitive.

The enhancement line may be manually specified with the primitive

```
#{kwv;col-expr}
```

This pushes the *col* value onto a stack and the current active flow figure enhancement character will be printed in that column (if the column would otherwise be blank). A *col* of zero will pop the stack and a negative *col* will reset the stack to its empty state.

4.17 Design Attribute Controls

This section describes the primitives which control the processing of various design attributes. This processing includes data item definition and referencing, segment definition and referencing, and treatment of labels. The primitives described in this chapter have effect only during *structured* mode formatting.

The primitive

```
#{ddf;setting}
```

controls the definition of data items. Data items consist of letters, digits, data characters, and data special characters as described in Section 4.5. The value of *setting* may be:

off	do not perform data item definition
f	define the first word of each source line as a data item (active during “pass1” processing only)
d	define each word which contains a data character as a data item; if the first character of the word is a data character, however, do not consider the leading data character as part of the name of the data item (active during “pass1” processing only)
i	during “pass2” processing, define each word which contains a data character to be a data item if it has not previously been defined as a data item

If the source line begins with a keyword, the keyword and any secondary keywords are skipped before applying the above data item definition requests.

When a word is to be defined as a data item, PDL/81 will invoke the user function

```
#{$ddi;name-string;code}
```

where *name* is the name of the data item and *code* is “0” if the data item resulted from a “ddf” setting of “f”, “d”, or “l” and is “1” if the data item resulted from a “ddf” setting of “i”. It is the responsibility of the “$ddi” function to perform the actual definition by use of the “dx” primitive (see Section 4.11).

If a line begins with a comment string, data item definition is controlled by

```
#{cdata;setting}
```

where a *setting* of “off” specifies no definition processing for such a line and anything else specifies that definition processing is to be performed according to the current “ddf” setting.

Collection of references to data items occurs during “pass2” processing and is controlled by

```
#{drf;setting}
```

where *setting* is “off” if collection is not desired. Any other setting will result in data item reference collection. References are only collected if *structured* formatting mode is specified. An initial keyword and any following secondary keywords will be skipped before collecting references. The page and line numbers for the reference are taken from the values of the “.page” and “.stanr” built-in number registers, respectively (see Section 9.4). Regardless of the “drf” setting, data item references will not be collected if the value (initially zero) of the “.xrsw” built-in number register is zero.

The primitive

```
#{sdf;setting}
```

controls automatic definition of segment names. If *setting* is “off” automatic definition is not performed; otherwise, each line is considered to be the definition of a segment. The name of the segment begins following any initial keyword and secondary keywords and extends to the end of the line or to the first comment string. Leading and trailing blanks are deleted and each sequence of imbedded blanks is replaced by a single blank. If the line begins with a comment string, definition is controlled by the current “cdata” setting.

PDL/81 requests the definition by invoking the user function

```
#{$dseg;name-string;line-string}
```

where *name* is the name of the segment and *line* is the source line that caused the invocation. It is the responsibility of the “$dseg” function to perform the actual definition by use of the “dx” primitive (see Section 4.11).

During “pass2” processing, collection of references to segments is controlled by the primitive

```
#{srf;loc-expr;width-expr}
```

If *loc* is zero, collection is not performed. Otherwise, each line (following any initial keyword and secondary keywords) is scanned for a possible reference to a segment. If a reference is found, an entry is made in the cross-reference data base for use in generating a segment index (see Section 4.27). In addition, the page number of the definition of the segment is placed right adjusted in a field of *width* characters starting at the position given by *loc*. The page and line number of the reference are taken from the values of the “.page” and “.stanr” built-in number registers, respectively (see Section 9.4). Regardless of the “srf” setting, references to segments will not be collected if the value (initially zero) of the “.xrsw” built-in number register is zero.

The primitive

```
#{lbl;char;indent-expr}
```

specifies that if the first word of a source line ends with the single character *char*, the word is considered to be a *label*. Label detection is disabled if *char* is null. A line beginning with a label is printed starting at the position given by *indent*. If *indent* is null, the line is printed flush against the current left margin; otherwise, *indent* is taken as the indentation (which may be negative) of the line. The text, if any, following the label on the line should generally be commentary.

4.18 Specifying Marginal Information

The primitive

```
#{snr;loc-expr;width-expr}
```

will cause the value of the “.stanr” built-in number register (see Section 9.4) to be printed right adjusted in a field of *width* characters starting at the position given by *loc*. This will be printed on each output line, but not on overflow lines produced in *nofilled* or *structured* modes. If the value of *loc* is zero, these numbers will not be printed.

The primitive

```
#{lnr;loc-expr;width-expr}
```

will cause the input source line number to be printed right adjusted in a field of *width* characters starting at the position given by *loc*. If the value of *loc* is zero, these numbers will not be printed.

The primitive

```
#{mc;char;loc-expr}
```

will cause the character *char* to be printed at the position given by *loc* in each output line which contains some text. If *loc* is null, it is taken to be the current right margin plus two. If *char* is null, the mechanism is disabled. The character will be printed in the font given by the “.mcfont” number register.

4.19 Explicit Reference Processing

Normally, reference processing is automatically performed during pass two. It may be explicitly performed by the primitive

```
#{rf;string}
```

which causes segment and data item reference scans to be made on “string”.

4.19.1 Establishing a Reference Trap

The primitive

```
#{rtrap;function-name}
```

establishes a *reference trap*. Once established, the named function will be invoked whenever a reference is being collected by the "rf" primitive or under control of the "Srf" processing mode (Section 4.17). Once established, the trap will be in effect until cancelled by a call on "rtrap" without a function name.

When the trap function is invoked, it will be given two arguments:

1. The name of the segment being referenced; and
2. The entire source line containing the reference.

In addition, the built-in number register ".ixdpage" is set to the page containing the definition of the segment and ".ixdline" is set to the line number of the definition.

4.20 Environments

PDL/81 contains twenty *environments* which are used to contain many of the formatting parameters. The user can switch between these environments during processing. Thus, for example, one environment can have the proper settings for flowing and justified text, another can have settings for processing flow segments, etc. The environments are numbered zero through nineteen and each initially contains the various default settings described in this manual.

At the start of the processing phase (just before "$start" is called) the current environment is set to "one". Processing can be switched to a new environment by

```
#{envs;en-expr}
```

where *en* is the number of the desired environment.

The current environment can be pushed down and a new environment can be established by

```
#{env;en-expr}
```

where *en* is the number of the desired new environment. If *en* is null, the environment stack is popped and the previous environment is restored.

If the current environment is zero and a line is encountered which contains text (after any possible function expansion), the user function

```
#{$ev0txt;0}
```

is invoked. An entirely empty (or blank) line in environment zero will invoke

```
#{$ev0txt;1}
```

This function might, for example, handle design text outside of a segment by issuing an error message, creating a dummy segment, and issuing the "backup" primitive (see Section 3.7.4) to reprocess the line.

The ".envnr" built-in number register always contains the current environment number.

The "env" and "envs" primitives force a line break (see Section 4.6).

The various parameters stored in an environment are:

- blank line mode ("bll" primitive) [default = keep]
- box mode and characters ("box" primitive) [default = no box]
- margin character and position ("mc" primitive) [default = no margin character]
- keyword case setting ("kwcase" primitive) [default = off]
- keyword font setting ("kwfont" primitive) [default = base font]
- data item definition mode ("ddf" primitive) [default = off]
- data item reference mode ("drf" primitive) [default = off]
- formatting mode ("fmt" primitive) [default = off]
- tab width ("tabw" primitive) [default = 4]
- indent value and previous indent value ("in" primitive) [default = 0]
- temporary indent value ("ti" primitive) [default = 0]
- imbedded spaces mode ("isp" primitive) [default = keep]
- justification mode ("just" primitive) [default = off]
- cdata mode ("cdata" primitive) [default = off]
- label character and position ("lbl" primitive) [default = no label character]
- left margin and previous left margin ("lm" primitive) [default = 1]
- right margin and previous right margin ("rm" primitive) [default = page width]
- line number mode, position, and width ("lnr" primitive) [default = off]
- leading spaces mode ("lsp" primitive) [default = keep]
- segment definition mode ("sdf" primitive) [default = off]
- statement number mode, position, and width ("snr" primitive) [default = off]
- space OK mode ("spcok" primitive) [default = yes]
- segment reference mode, position, and width ("srf" primitive) [default = off]
- extra line spacing ("els" primitive) [default = 0]
- keyword left margin warning mode and position ("kwlm" primitive) [default = off]

- the *x switch* values (Section 4.21) executable keyword activation mode (“kwx” primitive) [default = off]

4.21 Switch Manipulation Primitives

There is an environment value, known as the *x switch* (or *xsw*) which can hold up to 16 switches. These are manipulated by the primitives

```
#{xswn;expr}
```

which turns *on* each switch that corresponds to a one-bit in *expr*,

```
#{xswf;expr}
```

which turns *off* each switch that corresponds to a one-bit in *expr*, and

```
#{xswv}
```

which returns the current *xsw* value. The value of *xsw* is saved in the PDL/81 environment.

The currently defined switches are:

0x0001	enable executable keyword detection in pass one
0x0002	enable executable keyword detection in pass two
0x0004	enable complexity processing in pass one
0x0008	enable complexity processing in pass two
0x0010	enable flow-figure checking in pass one
0x0020	enable flow-figure checking in pass two
0x0040	enable Ada label handling

4.22 Saving and Restoring the State of the Formatter

The primitive

```
#{fmtsave}
```

will stack the current state of the formatter, except for the current environment and its contents, but including the current partially constructed output line. The primitive

```
#{fmtrestore}
```

will restore the state of the formatter, except for the current environment, to the state just before the last “fmtsave”.

4.23 Text Diversions

A *diversion* is a named place in which to temporarily place formatted output instead of printing it. Diversions are normally used to measure the vertical extent of some text so that such things as whether or not it will fit in the remaining space on the current page can be determined.

The primitive

```
#{di;name}
```

opens the diversion named *name* and causes all formatted output to be placed into it. If the diversion already has text in it, the new output is appended. If output is being diverted when the “di” primitive is encountered, the previous diversion is first closed. If *name* is null, the current diversion is closed.

When a diversion is closed, a count of the number of lines in the diversion is placed in the “.dline” built-in number register.

While output is being diverted, all normal formatting is performed. However, since the vertical position on the page is not changed, heading and footing traps will not be sprung.

The contents of a diversion is printed by the primitive

```
#{rdi;name}
```

where *name* is the name of the diversion. The contents of the diversion will be read and printed – no formatting is done since that was accomplished while the diversion was being made. The vertical page position is tracked during printing of a diversion so that heading and footing traps will be handled normally. If there is an open diversion during “rdi” processing, the contents of the diversion being read will be diverted again instead of being printed.

4.24 Final Output Diversion

The final, completely formatted, and paginated output from PDL/81 is sent to a file known as the “final output file”. Normally, this is the UNIX “standard output”. The current final output file may be redesignated by the primitive

```
#{dfo;fn-expr}
```

where *fn* is an auxiliary file number as described in Section 3.7.2. If the *fn* argument is absent, “standard output” will again become the final output file.

The contents of an auxiliary file may be copied to the current final output file by the primitive

```
#{cfo;fn-expr;text}
```

Normally, no formatting or other processing is done during this operation – it is a strict copy. However, if the *text* argument is not null, it is taken to be an *escape sequence* which may trigger special processing while the diverted final output is being processed.

Each line of the diverted output is examined to see if it begins with the specified escape sequence. If it does, the characters on the line following the escape sequence, up to the first non-alphanumeric, non-punctuation character, are taken to be the name of a function. Any white space is then skipped and the rest of the line, if any, is taken to be an argument to the function which is then invoked.

The function may perform any desired processing. In practice, it will normally output exactly one line which will replace the triggering line in the final output.

4.25 Definition and Referencing of Tags

A *tag* is a named entity which may be used for marking a point in formatted output for purposes of referring to that point in the text. The tag mechanism supports both forward and backward referencing.

A tag is defined or redefined by

```
#{dt;name;string1;string2;string3;string4}
```

where *name* is the name of the tag and the *string*s are arbitrary strings. The operation of “dt” is:

1. If *name* is not the name of a previously defined tag, it is defined as a tag, the four strings are associated with the definition, and the primitive returns a null value.
2. If it *has* been previously defined, and if the definition occurred in the same processing phase (“pass1”, “pass2”), “dt” returns “1” and associates the four strings with the definition.
3. If it has been previously defined in a *different* processing phase, the primitive returns a null value and, if the strings are not the same as those saved with the previous definition, the value of the “.tagerr” built-in number register (initially zero) is incremented by one. In either case the strings are asso-

ciated with the definition.

The strings associated with a tag may be retrieved by

```
#{tv;name;expr}
```

where *name* is the name of the tag and *expr* should be zero through three to retrieve strings one through four, respectively. The selected string is returned as the value of the primitive. If the tag is undefined, it will be defined and the two strings will be the name of the tag promoted to upper case.

All of the defined tags are retrieved by

```
#{tags;func-name}
```

where *func* is the name of a function to be executed for each tag. The tags will be presented in sorted order and each call on *func* will have the form

```
#{func;name;code;string1;string2;string3;string4}
```

where *name* is the name of the tag, *code* is “1” if the tag was not defined by the “dt” primitive or “0” if it was so defined, and the *string*s are the four associated strings.

WARNING

Once the “tags” primitive has been invoked, further use of the “dx”, “sx”, “di”, “rdi”, “dt”, and “tv” primitives will cause unpredictable results.

4.26 Printing Design Trees

Printing of design trees is initiated by

```
#{trees;root-func;node-func;end-func}
```

where *root* is the name of a function to be called for each tree root; *node* is the name of a function to be called for each node of a tree; and *end* is the name of a function to be called after all trees have been processed. The “trees” primitive returns the number of trees which were processed (i.e., the number of roots found). There will always be at least one root unless there were no flow segments.

WARNING

> Once the “trees” primitive has been invoked, further use of the “dx”, “sx”, “di”, “rdi”, “dt”, and “tv” primitives will cause unpredictable results.

The information necessary to produce the trees will not be collected unless the value (initially zero) of the .xrsw built-in number register is non-zero during “pass2” processing.

The processing of each tree begins with a call on the *root* function specified in the “trees” primitive. It is called as

```
#{root-func;name}
```

where *name* is the name of the segment which corresponds to the root of the tree.

When it is time to print a tree node, the *node* function specified in the “trees” primitive is called as

```
#{node-func;code;name}
```

where *name* is the name of the segment corresponding to the node and *code* is one of

0	this is a normal tree node
1	this node corresponds to a branch of the tree that has previously been printed. This code is used only if the value (initially zero) of the “.stree” built-in number register is non-zero.
2	this node is the start of a branch that has previously been printed and which is on the same path from the root – this indicates a recursive reference

When the *node* function is called, the following built-in number registers will have the values shown:

.trlnr	contains the tree line number for this node. Tree line numbers start at one for each tree and are useful for making backward references for code 1 and code 2 nodes.
.trlevel	contains the nesting level of the node. The root of the tree is at level zero.
.trpage	contains the page number on which the segment for this node was defined.
.trshort	contains, for codes 1 and 2, the tree line number of the line where this branch was previously printed.

If any roots were detected, the *end* function specified in the “trees” primitive is called as

```
#{end}
```

at the conclusion of tree processing.

4.27 Data Item and Segment Indexes

The printing of a data item or flow segment index is initiated by

```
#{index;type-expr;start;entry;ref;line;end}
```

where *type* should have the value 3 for a flow segment index or 4 for a data item index and the other arguments are names of functions to be called at appropriate points during index processing. After producing the index, the “index” primitive returns a count of the number of entries in the index.

WARNING

Once the “index” primitive has been invoked, further use of the “dx”, “sx”, “di”, “rdi”, “dt”, and “tv” primitives will cause unpredictable results.

Upon encountering the first index entry, the *start* function specified by the “index” primitive is called as

```
#{start}
```

For each index entry (i.e., segment name or data item name), the *entry* function is called as

```
#{entry;name}
```

where *name* is the name of the segment or data item. At each call on the *entry* function, the following built-in number registers will have the indicated values:

.ixdpage	contains the page number on which the entry was defined
.ixdline	contains the line number (statement number) on which the entry was defined; by convention, a zero value means that a line number is not applicable to this entry

.ixdcode	contains the code value assigned to this segment or data item name with the “dx” primitive (see Section 4.11)

For each referencing segment within an entry, the *ref* function is called as

```
#{ref;name}
```

where *name* is the name of the referencing segment. The contents of the “.ixrpage” built-in number register will contain the page number of the reference and the contents of the “.ixrline” built-in number register will be zero.

For each referencing line within a referencing segment within an entry, the *line* function will be called as

```
#{line}
```

The contents of the “.ixrpage” built-in number register will contain the page number of the reference and the contents of the “.ixrline” built-in number register will contain the line number (statement number) of the reference.

At the conclusion of index processing, the *end* function will be called as

```
#{end}
```

if there were any entries in the index.

5. Font Definition and Use

PDL/81 provides a base font and up to fourteen additional, user defined fonts.

NOTE

The font mechanism allows special device actions to be associated with the printing of each character. However, these actions, no matter how complex, *must* result in exactly one net character width of positive horizontal motion and exactly zero net vertical motion per character output.

5.1 Using Fonts

The primitive

```
#{uf;font-expr;text}
```

returns *text* with each character represented in the font given by the absolute value of *font*. If *font* is zero, the base (i.e., normal) font is used. If *font* is positive, only non-blank characters will be printed in the given font. If *font* is negative, blanks will be converted to unpaddable spaces and they, along with the non-blank characters, will be printed in the given font.

The absolute value of *font* should range from 0 to 14, inclusive.

Uses of the *uf* function may be nested. However, each use represents a switch from the current font to the new one.

5.2 Defining Fonts

Each font may specify actions to be performed when the font is entered, for each character printed in the font, and when the font is left.

5.2.1 Per-Character Font Action

Each of the definable fonts has an associated *font control string* which specifies how a character is to be printed in that font. The control strings are normal strings with the names *f1, *f2, ..., *f14 and may be defined by

```
#{ds;*fn;control-string}
```

where *n* is 1, 2, ..., 14.

A control string acts as a template for printing of one character in the associated font. Each character in the string is sent to the final output file, except that the character "#" will be replaced by the character to be printed in this font. The sequence "\#" will result in the output of a literal "#".

As an example, defining

```
#{ds;*f1;_#{sneak;008}#}
```

will cause each character printed in font 1 to be printed as an underscore. followed by a backspace, followed by the character. Thus, the function

```
#{uf;1;some text}
```

will result in "some text" being underscored on output.

The following may be used to define a "us" primitive to provide underscoring of non-blank text and a "uc" primitive to provide underscoring of all text:

```
#{ds;us;{#{uf;1;#1}}}
#{ds;uc;{#{uf;-1;#1}}}
```

As another example, consider defining font 2 to produce "bold face" output. One way to do this on a normal printer would be to overstrike each character twice as in

```
#{ds;*f2;#{sneak;'#;^h;'#;^h;'#}}
```

The notation "^h" results in the low-order five bits of the character "h", thus supplying an ASCII backspace (see Section 2.5.1). The function

```
#{ds;bf;{#{uf;2;#1}}}
```

could then be defined so that

```
#{bf;some text}
```

would print "some text" in bold face.

Some output devices allow complicated device motions to be performed for each character. On these, bold face printing might be done by printing the character more than once with a slight offset between each instance. If this is done, remember that the net horizontal motion must be exactly one character width and that there must be no net vertical motion. Also, note that a sequence which gives desirable results on one device may result in horrible looking output on another device. Results depend heavily on the choice of type face, the ribbon, the paper, and the “state of tune” of the particular device.

As a final example, consider a requirement for printing “some text” bold face and underscored. The form

```
#{us;#{bf;some text}}
```

will not work, since it would enter font 1 and then immediately switch to font 2, thus resulting in bold face output without underscores. The solution is to define a new font to produce the desired result:

```
#{ds;*f3;#{sneak;'_;^h}#{*f2}}
```

The function

```
#{ds;bfu;{#{uf;3;#1}}}
```

can then be defined for printing bold face, underscored text.

5.2.2 Actions on Beginning and Ending a Font

In addition to the per-character control strings discussed in Section 5.2.1, the strings *f1b, *f2b, ..., *f14b act as “begin font” control strings and the strings *f1e, *f2e, ..., *f14e act as “end font” control strings. When a sequence of one or more characters is to be printed in a given font, the contents of the corresponding “begin font” control string is output, each of the characters is output under control of the corresponding per-character control string, and the contents of the corresponding “end font” control string is output.

As an example, some video display terminals have an “underline” mode which is entered with the escape sequence

```
^[&dA
```

and terminated with the escape sequence

```
^[&d@
```

where “^[” is the ASCII escape character (octal value 033). To define font 1 (the usual underscore font) to use this mode, the definitions

```
#{ds;*f1b;#{sneak;^[}&dA}
#{ds;*f1;#}
#{ds;*f1e;#{sneak;^[}&d@}
```

could be used. Then,

```
#{uf;1;some text}
```

would cause “some text” to be displayed on the terminal using the terminal’s underline mode.

5.3 Initial Font Definitions

When PDL/81 is started, font 1 is initialized as shown above so that each character in this font prints as “underscore, backspace, character”. The remaining fonts all have null control strings.

Printing of characters in an undefined font or in one defined to have a null control string is equivalent to printing in the base font.

5.4 Selecting Keyword Fonts

The primitive

```
#{kwfont;n}
```

specifies that keywords are to be printed in font *n*.

5.5 Miscellaneous Font Control Primitives

The primitive

```
#{defont;text}
```

returns *text* with all font change information removed.

The primitive

```
#{ups;text}
```

returns *text* with each blank replaced by an unpaddable space.

5.6 Explicit Keyword Enhancement

Keyword enhancement (i.e., display is special case and font) is normally an automatic function of pass two processing. It may be performed explicitly by the primitive

```
#{enhance;string}
```

which returns “string” in the currently prevailing keyword case and font.

6. Regular Expression Processing

PDL/81 supports a means of scanning and processing regular expressions similar to those which are handled by many of the utilities found in the Unix operating system. Primitives are provided to determine if a string matches a regular expression and to extract specified substrings of a matched string.

6.1 Scanning for Regular Expressions

The primitive

```
#{rxs;re-string;source-string}
```

determines if the “source” string, or any substring of it, matches the regular expression given by “re”. It returns “1” if it matches and “0” if it does not match. If the “source” argument is absent, the “source” string provided in the most recent previous call on “rxs” is scanned again.

A regular expression is made up of the following characters and character sequences:

1. A period (.) matches any character.
2. The apostrophe (′) is the “escape character”. If followed by any character other than a digit (0-9), it matches that character.
3. The form [s], where “s” is non-empty, matches any character in the string “s”. The form [~s], where “s” is non-empty, matches any character not in “s”. The character $ as the last character in either “s” matches the end of the source string. In either of these “character class” forms, a substring of the form “a-b” matches any character in the inclusive ASCII character range “a” through “b”.
4. Any other character except “(” and “*” matches itself.
5. Any of the above followed by “*” matches zero or more matches of the regular expression.

6. A regular expression of the form “(x)” matches whatever “x” matches. Such an expression is known as a *group*.
7. The form “′n”, where “n” is a digit (1-9), matches a copy of the string that group “n” matched.
8. A regular expression “xy” matches the longest “x” such that a match for “y” is still possible.
9. ^ as the first character of a regular expression matches the beginning of the source string.
10. $ as the last character of a regular expression matches the end of the source string.

In the absence of a leading “^” or trailing “$”, the longest, leftmost substring of the source string that matches the regular expression will be chosen.

6.2 Extracting Matched Substrings

The primitive

```
#{rxg;group-expr}
```

will return the specified portion of the last string scanned with the “rxs” primitive. The possible “group” values are

0	the entire matched substring
1-9	the substring that matched the specified *group* (1-9)

6.3 Input Scanning With Regular Expressions

The primitive

```
#{skip;regex}
```

will cause input lines to be skipped until a line is encountered which matches the regular expression. Normal processing will then be resumed with the matched line being rescanned in the normal mode.

The primitive

```
#{cpf;regex;fn-expr}
```

will cause input lines to be copied to the output file specified by the “fn” expression until a line which matches the regular expression is encountered.

7. Escape Functions

At various points during processing, PDL/81 will call certain user-defined functions. Collectively, they are known as *escape functions*. These functions are described at appropriate points in previous chapters. If a particular escape function has not been defined, its invocation is ignored.

This chapter summarizes the escape functions and their usage:

$dev-xxx	called at the end of the definition phase. *xxx* is the device name specified at invocation or is “default” if a device is not specified.
$ddi	invoked when a data item definition is encountered. The general form is `#{$ddi;name;code}` where *code* is “1” if this is an implicit data item definition and is “0” otherwise.
$dseg	invoked when an automatic segment definition is encountered. The general form is `#{$dseg;name;line}}`
$start	invoked at the beginning of the processing phase. In fact, whatever this function does *is* the processing phase.
$end	invoked upon return from the “$start” function.
$ev0txt	invoked if a line is not empty, after any requested function expansion, and the current environment is number zero.
$pp	invoked for each blank line if the blank line mode has been set to “a” by the “bll” primitive.
$inderr	invoked if the “kwlm” primitive has set a keyword left margin in the current environment and indentation to the left of that value has just been requested.

8. Special Strings

There are a few named strings which have special interpretations during PDL/81 processing. They have been described in previous chapters and are summarized here:

*bu
: contains the output sequence to print the bullet character. For example, the bullet will be printed as a plus sign superimposed on a “o” if the definition

```
#{ds;*bu;o#{sneak;010}+}
```

is given. In order to maintain proper output alignment, the contents of the “*bu” string must occupy only a single character position when actually printed.

*f1b, *f2b, ..., *f14b
: begin-font control strings (Chapter 5).

*f1, *f2, ..., *f14
: per-character font control strings (Chapter 5).

*f1e, *f2e, ..., *f14e
: end-font control strings (Chapter 5).

*gon
: contains the sequence to put the output device into “graphics” or “fine resolution” mode if needed during line justification (see Section 4.6.1). For example, the definition

```
#{ds;*gon;#{sneak;033;’3}}
```

can be used with a Diablo 1620.

*goff
: contains the sequence to turn off “graphics” mode. For example, the definition

```
#{ds;*goff;#{sneak;033;’4}}
```

can be used with a Diablo 1620.

*eol
: contains the sequence of characters to be output at the end of every printed line. If this string is not defined, a newline is postpended to each printed line.

*bol	contains the sequence of characters to be output at the beginning of each printed line. If the string is not defined, nothing special is output.

9. Built-In Number Registers

Most of the various built-in number registers have been described in previous chapters. This chapter describes those not described elsewhere and summarizes the others.

9.1 Command Control Registers

These number registers control various aspects of command recognition and processing:

.cmdarg	if set to “1”, detection of arguments on command lines is enabled and leading spaces are removed from each argument; if set to “2”, the treatment is the same, except that leading spaces are not removed from arguments other than the first
.cmdcall	if set non-zero, detection of the “#{” sequence is enabled during command line collection
.cmderr	if set to “1”, an undefined command name detected during “pass1” processing will result in an error message; similarly, if set to “2” during “pass2” processing

9.2 Date and Time Registers

The values of the number registers described in this section are set from the current date and time of day when PDL/81 is invoked. If the host operating system is unable to supply a particular value, the register will be set to a value of zero. The date number registers are:

.month	the current month, with January being “one”
.mday	the day of the month, starting at “one”
.wday	the day of the week, with Sunday being “one”
.yday	the day of the year, with the first day of January being “one”
.year	the year minus 1900

The time of day number registers are:

.hour	the hour (0 - 23)
.min	the minute (0 - 59)
.sec	the second (0 - 59)

9.3 Output Control Registers

These number registers control various aspects of the formatted output:

.cwidth	set to the width of an output character in output device basic units. Used to control fine-resolution justification (see Section 4.6.1). Initially has a value of “one” which means normal justification.
.noff	set to non-zero to prevent PDL/81 from generating form feed characters in the output. Initially set to zero which means that PDL/81 is allowed to generate form feed characters in the output.
.notab	set to non-zero to prevent PDL/81 from generating horizontal tab characters in the output. Initially set to zero which means that PDL/81 is allowed to generate tab characters in the output.
.nobs	set to non-zero to prevent PDL/81 from generating backspace characters in the output. Initially set to zero which means that PDL/81 is allowed to generate backspace characters in the output.
.po	set to the number of character positions by which each output line is to be indented, thus shifting the output page a constant amount to the right. Initially has a value of zero.

9.4 Cross Reference Control Number Registers

These number registers control various aspects of cross-reference collection and processing:

.page	should contain the current page number if the selective printing option is used or if any of the cross referencing modes (“drf”, “srf”) are enabled. The manipulation of this number register is entirely up to the user – PDL/81 never modifies its value (initially zero).
.stanr	incremented by one just before performing “pass1” or “pass2” processing on an input line. It normally is considered to hold the current segment statement number and normal practice would be to set its value to zero at the start of each segment.
.xrsw	set to non-zero to allow collection of reference information under control of the “drf” and “srf” primitives. The initial value of zero inhibits such collection.

9.5 Tree Registers

These number registers control tree collection and provide information about a tree which is being displayed:

.stree	set non-zero to print short trees; zero to print long trees

.trlevel	holds level of current node
.trlnr	holds line number of current node
.trpage	holds page number of current node definition
.trshort	holds line number of line where node was printed previously

9.6 Indexing Registers

These number registers provide information about an item being indexed:

.ixdcode	holds code for current index entry
.ixdpage	holds page number of definition of current entry
.ixdline	holds line number of definition of current entry
.ixrpage	holds page number of current reference
.ixrline	holds line number of current reference

9.7 Tag Related Registers

This number register provides information about tags:

.tagerr	holds the count of the number of tags whose second definitions are different from their first definition

9.8 Font Control Registers

These number registers control the font for printing certain information:

.boxfont	font to print box drawing characters
.mcfont	font to print marginal character

9.9 Informational Registers

These number registers provide general-purpose information:

.envnr	number of current environment
.ixcmplx	complexity value for current keyword
.ixdval3	value from “val3” for current item
.ixdval4	value from “val4” for current item
.ndefl	count of the number of lines in definition files
.nsrcl	count of the number of lines in source files
.ndict	count of the number of primary and secondary dictionary entries made
.nivb	count of the number of internal string storage units allocated
.nfivb	count of the number of internal string storage units which were once allocated but are now free

10. Examples of Use

This chapter presents a number of examples which illustrate how certain formatting objectives can be reached by use of the Format Definition Language. There are usually a number of ways to reach a desired objective. Generally, only one of the many possibilities will be illustrated.

The examples presented here are mostly just outlines of the definitions that would be necessary to fully handle the desired objective. The best source of complete examples is any of the definition files distributed with PDL/81.

10.1 Defining the Processing Structure

On completion of the definition phase, PDL/81 invokes the "$start" function which, presumably, has been defined during that phase. It is the purpose of the "$start" function to cause processing of the source document.

For two-pass processing, such as for a design document, the function should be at least

```
#{ds;$start;{\
    #{if;#{eq;#{source}};{\
        #{quit;no source file given}\
    }}\
    #{pass1;#{source}}\
    #{pass2;#{source}}\
}}
```

where the following should be noted:

- The first call (on "if") causes processing to terminate immediately if there was no source file specified.
- The body of the "$start" function and of the true-branch of the "if" are enclosed in brackets so that the imbedded function calls will occur during execution.

For one-pass processing, such as a manual or report, a simpler definition such as

```
#{ds;$start;{\
    #{pass2}\
}}
```

may be used.

10.2 Obtaining Printable Dates

First, obtain the name of the current month by

```
#{ds;mname;#{case;.month;;January;February;\
    March;April;May;June;July;August;\
    September;October;November;December\
}}
```

The three-character name of the month can then be obtained by

```
#{ds;mname3;#{substr;1;3;#{mname}}}
```

From these, the string “date” can be defined to obtain the date in the form 20 September 1986 and the string “date3” can be defined to obtain the date in the form 20 Sep 86. The definitions for these strings are

```
#{ds;date;#{.mday} #{mname} 19#{.year}}
#{ds;date3;#{.mday} #{mname3} #{.year}}
```

10.3 Establishing Page Headings and Footings

The definitions

```
#{head;$$head}

#{ds;$$head;{\
    #{env;10}\
    #{nr;.page;+1}\
    #{sp;2}\
    #{ttl;DRAFT;;#{date3}}\
    #{sp;3}\
    #{env}\
    #{spcok;no}\
}}
```

will cause each page to begin with two blank lines, followed by a header consisting of the word “DRAFT” left adjusted and the date (assuming the definitions of Section 10.2) right adjusted. The header is followed by three blank lines. Note the following in this example:

- The body of the “$$head” definition is enclosed in left and right brackets. If this were not done, the function calls would be executed at the time of the definition and not when the “$$head” function is invoked.
- The header is processed in an environment of its own. Presumably, the left and right margins in this environment have been established where desired for headers. Thus, the processing state which existed at the time of the trap will not interfere with the desired format of the header.

- The page number (.page number register) is incremented.
- The “spcok” mode is set to inhibit spacing on the new page until some text is printed.

In a similar manner

```
#{foot;$$foot;-6}

#{ds;$$foot;{\
    #{env;10}\
    #{sp;2}\
    #{ttl;;- #{.page} -}\
    #{env}\
}}
```

will place a footing trap six lines from the bottom of the page. The footing will consist of two blank lines followed by a line with the page number centered and surrounded by dashes.

10.4 Formatting a Flow Segment

Suppose it is desired to process flow segments in the same general style as that used in PDL/74. First, it is necessary to choose an environment, say number 5, which will be used for flow segments and to initialize it:

```
#{envs;5}          (switch to environment 5)
#{lm;11}           (left margin at column 11)
#{rm;103}          (right margin at column 103)
#{snr;7,3}         (statement numbers in
                         column 7, 3 wide)
#{bll;remove}      (remove blank lines)
#{lsp;remove}      (remove leading spaces)
#{isp;compact}     (compact imbedded spaces)
#{fmt;s}           (use structured formatting mode)
#{srf;2;3}         (collect segment references,
                         putting page ref
                         in column 2, 3 wide)
#{drf;on}          (collect data item definitions)
#{ddf;implicit}    (make implicit data
                         item definitions)
#{kwcase;off}      (don’t change case of keywords)
#{kwuscr;on}       (underscore keywords)
#{lbl;:}           (: is label character, print
                         at left margin)
```

Then, “%s” command needs to be defined. For the first pass, this could be something like

```
#{ds;1s;{\
    #{if;#{dx;3;#{name;#1};;.page;0};{\
        #{error;duplicate segment name}\
    }}\
    #{nr;.stanr;0}\
    #{nr;.page;+1}\
    #{envs;5}\
}}
```

and for the second pass, something like

```
#{ds;2s;{\
    #{bp}\
    #{envs;5}\
    #{fm;#1}\
    #{sp}\
    #{sx;#{name;#1}}\
    #{stuff;1;REF}\
    #{br}\
    #{stuff;1;PAGE}\
    #{box;6;#{$width}-1;*}\
    #{nr;.stanr;0}\
}}
```

On completion of the segment, it will be necessary to issue

```
#{ebox}
```

Note that the definitions shown here are only an outline of the needed operations. Such things as error detection, table of contents contribution, and interactions with other segment types have not been handled.

10.5 Table of Contents Handling

Almost any style of table of contents can be produced automatically with PDL/81. The general scheme is:

1. When an item is to be placed into the table of contents, use the “send” primitive to write a call on a function into a temporary file.
2. When the table of contents is to be printed, use the “rcv” primitive to read the file, thus executing the function calls which were previously written out.

As an example, consider producing a simple single-level table of contents. To put something in the table of contents, call

```
#{tocsend;text;page-nr}
```

where “tocsend” is defined as

```
#{ds;tocsend;{\
    #{send;1;{#{tocline;#1;#2}}}\
}}
```

As processing continues, temporary file number one will contain lines of the form

```
#{tocline;text;page-nr}
```

When it is desired to print the table of contents, execute

```
#{rcv;1}
```

which will cause each of the calls on “tocline” to be executed.

The “tocline” function might be something like

```
#{ds;tocline;{\
    #{rstuff;#{$width};#2}\
    #{fm;#1}\
}}
```

This should, of course, be done in an environment with attributes appropriate to the table of contents. For example, one would normally want the right margin to be somewhere to the left of the page boundary so as to assure there is space for the page number.

The “rstuff” function, used above, just prints its second argument right adjusted in a field ending at the column given by its first argument. It might be defined as

```
#{ds;rstuff;{\
    #{stuff;(#1)-#{width;#2}+1;#2}\
}}
```

10.5.1 Printing an In-Line Table of Contents

The method of printing a table of contents as outlined above will result in the table being printed at the end of the document. By use of the “dfo” and “cfo” primitives, and at the expense of additional processing time, the table can be printed in its accustomed place at the front of the document. The general method is:

1. Process any introductory material, such as a title page, with the final output file being left at its default assignment of the “standard output”.
2. Before starting to process the body of the document, divert the final output to an auxiliary file, e.g.:

   ```
   #{dfo;3}}
   ```

3. While processing the body, collect table of contents entries onto another auxiliary file as outlined above.
4. When finished with the body, return the final output file assignment to “standard output” by:

   ```
   #{dfo}
   ```

   and print the table of contents as outlined above. At this point, the standard output will contain the introductory material, followed by the table of contents.

5. Finally, copy the previously diverted body to the standard output by:

```
    #{cfo;3}}
```

10.6 Index Handling

Full support for automatically collected data item and flow segment indexes is supplied by the “index” primitive (see Section 4.27). This section describes an alternate mechanism for creating word and phrase indexes such as the one found at the end of this manual.

Items for such an index are collected by defining a specific command, say “%ix” which enters its argument into the index. The item is entered by writing a function call onto a keyed temporary file with the “send” primitive. The index is formatted by calling the “rcv” primitive to sort, input, and execute these calls.

The “%ix” command might be defined as

```
#{ds;2ix;{#{ix;#1;#{.page}}}}
```

where the “ix” function is defined as

```
#{ds;ix;{\
    #{send;2;{#{ixent;#1;#2}};#1}\
}}
```

so that executing

```
%ix test phrase
```

on page 25 would be the same as executing

```
#{send;2;{#{ixent;test phrase;25}};test phrase}
```

When the index is to be printed, executing

```
#{rcv;2}
```

will cause the file to first be sorted on the phrases and then input, thus executing each call on “ixent” in turn.

The “ixent” function can be defined to perform any desired formatting for the index. For example, it can detect changes in its first argument so as to start new entries in the index, capitalize the first word of each entry, and detect a change in the first character of each entry so as to skip a blank line on the output for each new letter of the alphabet.

10.7 Tag Handling

The tag mechanism (see Section 4.25) is used to allow referencing various places in a document without requiring the author to know what referencing numbers (e.g., section or page numbers) will be assigned by PDL/81. An example of such a reference appears in the preceding sentence.

So that both forward and backward references can be handled without requiring two passes for every formatting run, an auxiliary file can be associated with a source file and the auxiliary file can contain the definitions of the tags. This file can be read in at the start of a run and its up-to-date contents can be written back out at the end of the run. If the definitions of a tag are changed during the run, a message can be displayed to the effect that the document must be reprocessed if it is desired to get correct references to tags (correctness is often not necessary while a document is in draft form).

As an example, assume that we want to implement tags which behave as:

- The auxiliary file will have the same name as the source file with any extension replaced by an extension of “a”. For example, the source file “doc.p” would have an auxiliary file named “doc.a”.
- A tag is to be placed into the text by the command

  ```
  %tag name
  ```

  where *name* is the name which will be used to refer to the point of the tag. We desire that the number of the page on which the tag appears be associated with the tag.
- A tag is to be referenced by

  ```
  #{ref;name}
  ```

  where *name* is the name of the tag. This function is to return “page n” where *n* is the page number associated with the tag.

At the start of a run, the auxiliary file must be input if it exists. This can be done by the function

```
#{ds;getaux;{\
    #{if;#{access;#{auxname}};{\
        #{include;#{auxname}}\
    }}\
}}
```

where *auxname* has been defined as

```
#{ds;auxname;#{base;#{source}}.a}
```

The “%tag” command can be implemented as

```
#{ds;2tag;{\
    #{if;#{dt;#1;page #{.page}};{\
        #{error;duplicate tag: #1}\
    }}\
}}
```

The “ref” function can be implemented as

```
#{ds;ref;{#{tv;#1;0}}}
```

which will return the string associated with tag by the “%tag” command. Thus,

```
#{ref;testing}
```

will return “page 25” if *testing* had been previously defined (on page 25) and will return TESTING if the tag had not been previously defined. Of course, *previous* definition includes the case of being defined at a following point in the document as long as the definition had been placed in the auxiliary file by a preceding run of PDL/81.

At the conclusion of the run, the “dumpaux” function should be called. It can be defined as

```
#{ds;dumpaux;{\
    #{open;9;#{auxname}}\
    #{tags;dumptag}\
    #{close;9}\
    #{if;.tagerr;{\
        #{error;One or more tags are wrong.}\
    }}\
}}
```

The “dumptag” function, which is called automatically for each tag, is just

```
#{ds;dumptag;{\
    #{if;#2;{\
        #{error;undefined tag: #1}\
    };{\
        #{send;9;\#\{dt\;#1\;#3\}}\
    }}\
}}
```

Note the use of the escaped special characters in the call on the “send” primitive in the “dumptag” function. This is necessary since the file is a non-temporary file and the internal encoding of the special characters cannot be saved from run to run of PDL/81 (see Section 3.7.2).

A. Error Messages

This Appendix lists all of the error messages which may be issued by PDL/81. Note that error messages may also be issued by the “error” and “quit” primitives. Error messages are displayed on the standard error file. If applicable, the message is prefixed with the name of the current input file and the current line number within the file.

A.1 Non-Terminal Error Messages

The error messages described in this section do not cause termination of PDL/81 processing:

- CFO: REQUESTED FILE NOT AVAILABLE – the file specified in a call on the “cfo” primitive is not open.
- DEL: CAN’T DELETE PERMANENT ITEM – an item named in a call to the “del” primitive has a period as the first character of the name.
- DS: NAME IS NOT THAT OF A STRING – the item being defined in a call on the “ds” primitive exists but is not the name of a string.
- DUP: CAN’T CHANGE TYPE OF PERMANENT ITEM – the item named in the second argument of a call on the “dup” primitive exists, has a name beginning with a period, and has a different type than that of the item named in the first argument.
- INVALID CHARACTER IN LINE – an input line contains an ASCII control character other than “tab” or “newline”.
- NR: NAME IS NOT A NR – the item being defined in a call on the “nr” primitive exists but is not a number register.
- TEXT IN DEFINITION FILE – after any indicated expansion, a line in a definition file was not empty.
- UNBALANCED BRACKETS – the number of unescaped left brackets is not equal to the number of unescaped right brackets in the line.
- UNKNOWN COMMAND – a command name is not defined and the “.cmderr” number register is set to detect and report on this.

A.2 Terminal Error Messages

The error messages described in this section cause immediate termination of PDL/81 processing:

- BCOPY: INVALID META CHARACTER – this is an internal processing error and should not occur. If it does occur and is repeatable, report it as a possible processor bug.
- CAN'T OPEN OUTPUT FILE <file name> – the "open" primitive is unable to open the named file for output.
- CAN'T OPEN TEMP FILE <file name> – the "send" primitive is unable to open the named temporary file. The name is an internal name but is displayed as a possible aid in diagnosing the problem.
- CANNOT ALLOCATE DYNAMIC MEMORY FOR A BUFFER – memory was needed for an input/output buffer, but insufficient memory was available.
- CFO: CAN'T COPY DIVERTED OUTPUT: <file> – the file cannot be copied by the "cfo" primitive because it would be copied to itself since the current final output file has been set to be the named file.
- CFO: CAN'T OPEN TEMP FILE <file> – the "cfo" primitive is unable to open the named temporary file. The name is an internal name but is displayed to help in diagnosing the problem.
- DYNAMIC MEMORY OVERFLOW (n) – all available dynamic memory is allocated and more is needed. The character "n" indicates the particular point in the processor at which overflow was detected and is of interest only to PDL/81 processor maintenance personnel.
- ERRORS IN DEFINITION FILE – issued at the end of the definition phase if any errors have been detected up to that time.
- EXIT FUNCTION INVOKED – BUT NOTHING SHOULD INVOKE IT!!! – something has invoked the so-called UNIX "cleanup exit" function, but nothing in PDL/81 is supposed to invoke it. This message indicates system trouble of some kind.
- FOPEN: FILE DESCRIPTOR OUT OF RANGE: <file> – when the internal PDL/81 "fopen" routine invoked the UNIX "open" routine, a file descriptor was returned which was too large for PDL/81 to handle. This is only likely to happen if PDL/81 is invoked with a very large number of files already open in the invoking environment.
- FOPEN: IMPOSSIBLE MODE: <file name> – something invoked the internal PDL/81 "fopen" function with a mode request not supported by PDL/81. This message indicates some kind of system trouble.
- INPUT ROLL OVERFLOW – the current state of FDL execution has resulted in too many function expansions which have not yet been rescanned by the interpreter.
- INPUT STATE STACK UNDERFLOW – this is an internal processing error and should not occur. If it does occur and is repeatable, report it as a possible processor error.
- INVALID FILE NAME: <file name> – the file name in a call on the "open" primitive is syntactically invalid.

- INVALID META CHARACTER – this is an internal processing error and should not occur. If it does occur and is repeatable, report it as a possible processor error.
- I/O ERROR ON CLOSE: <code> – a permanent I/O error has occurred while closing a file. The UNIX error number is displayed as “code”.
- I/O ERROR ON READ: <code> – a permanent I/O error has occurred while reading from a file. The UNIX error number is displayed as “code”.
- I/O ERROR ON WRITE: <code> – a permanent I/O error has occurred while writing on a file. The UNIX error number is displayed as “code”.
- LINE IS TOO COMPLEX TO PROCESS – the current input line contained too many lexically nested function invocations.
- MKTEMP: CANNOT GENERATE UNIQUE FILE NAME: <file name> – names of PDL/81 temporary files are generated by the internal PDL/81 “mktemp” function. This function can generate up to 26 unique names for each invocation of PDL/81. Since names will be reused when possible, and since PDL/81 deletes temporary files after they are closed, this message usually means that a large number of temporaries were left around following a system crash. Examine the directory given in the message and delete the abandoned temporaries.
- OPENING FILE WOULD DESTROY SOURCE FILE: <file name> – opening the named file with the “open” primitive would cause the current source file to be erased.
- PARM ROLL OVERFLOW – function execution is too deeply nested.
- PSORT: CAN’T FIND SORT!! – the sorter used by the “rcv” primitive to process keyed temporary files cannot find the sort program. Under Unix, this program is either /bin/sort or /usr/bin/sort. This message should not occur.
- PSORT: CAN’T OPEN INPUT FILE <file name> – the named input file cannot be opened during sort processing of the “rcv” primitive. The name will be that of an internal file and this message normally indicates some kind of system problem.
- PSORT: CAN’T OPEN OUTPUT FILE <file name> – the named output file cannot be opened during sort processing of the “rcv” primitive. The name will be that of an internal file and this message normally indicates some kind of system problem.
- PSORT: UNABLE TO FORK SORT – a UNIX fork could not be created to hold the sorter used by the “rcv” primitive. This generally means that the system is so heavily loaded that no UNIX process slots are available.
- TARGET OF -v OPTION ALREADY DEFINED: <name> – the named item was specified as the target of a string assignment invocation option, but the item was already defined.
- TOO MANY NESTED ENVIRONMENTS – the “env” primitive has been used to push the environment too many times without any intervening pops.
- UNABLE TO OPEN <file name> – the named file cannot be opened for input.
- UNKNOWN DEVICE TYPE: <name> – the named device was specified with an invocation option but no such device type exists.

- UNKNOWN INVOCATION OPTION: <option> - the given invocation option is not one recognized by PDL/81.
- UNKNOWN NAME: <function name> – an attempt was made to call the named function but it did not exist and the *quit on undefined function* option was specified during invocation.

B. List of Primitives

$depth	obtain current page depth
$file	obtain name of current input file
$foot	obtain location of footing trap
$in	obtain current indent
$line	obtain current line number on output page
$lm	obtain current left margin
$rm	obtain current right margin
$tabw	obtain current tab width
$width	obtain current page width
access	test if a file is accessible
as	append to string
backup	back up current input
base	obtain base portion of a file name
bll	set blank line treatment mode
box	start drawing a box
bp	begin a new page
br	force a line break
call	call a function with a composed name
cap	capitalize a string
case	select one of n strings
cc	redefine control characters
cdata	set reference detection mode in comment statements
ce	center text

cfo	copy final output
close	close a file
cm	define comment strings
cpf	copy file until regular expression match
dc	define comment characters
ddf	set data item definition mode
defont	remove font information from a string
del	delete one or more functions
dfo	divert final output
di	open or close a diversion
drf	set data item reference mode
ds	define a string
dsc	define data special characters
dt	define a tag
dump	output the encoded dictionary
dup	duplicate a function
dx	define keywords, secondary keywords, data items, and segments
ebox	end drawing of a box
els	set extra line spacing
enhance	enhance a string using current keyword font
env	push current environment and set a new one
envs	set a new current environment
eq	test two strings for equality
error	issue a non-terminal error message
ev	evaluate an expression
exit	exit from PDL/81 with a status code
fft	force footing trap
fht	force heading trap
fm	format some text
fmt	set formatting mode
fmtrap	set a format trap
fmtrestore	restore state of formatter
fmtsave	save state of formatter
foot	define page footing trap
fx	find information on secondary dictionary entry

head	define page heading trap
hpos	set a given horizontal position
hsp	horizontal space
icap	capitalize initial character of a string
if	select one of two strings
ifdef	test if a name is defined
in	set indent
include	input from named file
index	initiate design index processing
isp	set imbedded space treatment mode
just	set line justification mode
kwcase	set case for printing keywords
kwcheck	check flow figure
kwfont	set font in which to print keywords
kwlm	establish keyword left margin warning position
kwv	control flow figure enhancement
kwvc	define flow figure enhancement character
kwx	allow executable keywords
lbl	define the label character and label printing position
leader	fill with leaders
length	obtain the length of a string
lib	input from a library file
lindex	determine set membership
lm	set left margin
ln	set location to print line numbers
loop	loop over arguments
lpn	obtain Licensed Program Number of PDL/81
lsp	set mode for treatment of leading spaces
mc	define margin character and position
memuse	obtain memory use information
name	extract segment name from a string
nargs	count arguments in a string
nf	format a number
nr	define a number register
open	open a named output file

out	output unprocessed text
pass1	perform “pass1” processing
pass2	perform “pass2” processing
pass3	perform “pass3” processing
ps	print a string (on standard error)
psize	set page size
quit	issue a fatal error message
rcv	receive contents of a file written with “send”
rdi	receive a diversion
rep	replicate a string
rf	do reference processing on a string
rm	set right margin
rtrap	establish a reference trap
rxg	obtain value of a regular expression group
rxs	scan and match a regular expression
scall	call a function for each source file
sdf	set segment definition mode
send	send text to an intermediate file
skip	skip until regular expression match
sneak	put arbitrary characters into a string
snr	set location to print statement numbers
source	obtain name of source file
sp	perform vertical spacing
spcok	inhibit or allow vertical spacing
sqz	squeeze out blanks
srf	set segment reference collection mode
strap	set a source trap
stuff	put text at given horizontal position on output line
substr	obtain a substring of a string
sx	set current segment for reference collection
systype	obtain type of operating system
tabw	set current tab width
tags	initiate tag retrieval scan
tf	turn on tracing mode
ti	set temporary indent

tn	turn off tracing mode
trees	initiate segment reference tree processing
ttl	print a three-part title
tv	obtain value of a tag
uf	use a font for printing some text
ups	convert blanks to unpaddable spaces
ver	obtain primary version number of PDL/81 processor
ver2	obtain secondary version number of PDL/81 processor
width	obtain width of a string
xswf	turn off switches
xswn	turn on switches
xswv	get switch values

C. List of Number Registers

.boxfont	font to print box drawing characters
.cmdarg	switch to allow argument separators on command lines
.cmdcall	switch to allow calls on command lines
.cmderr	switch to report undefined command names as errors
.cwidth	width of a character in output device basic units
.dline	number of lines in last closed diversion
.envnr	current environment number
.hour	current hour of day (0 - 23)
.ixcmplx	complexity value for a keyword
.ixdcode	code of an index entry
.ixdline	definition line of an index entry
.ixdpage	definition page of an index entry
.ixdval3	“val3” value for a dictionary entry
.ixdval4	“val4” value for a dictionary entry
.ixrline	reference line for an index reference
.ixrpage	reference page for an index reference
.mcfont	font to print marginal character
.mday	day of month (1 - 31)
.min	current minute (0 - 59)
.month	month (1 - 12)
.ndefl	number of lines in definition files
.ndict	number of dictionary entries
.nfivb	number of allocated but now free string storage units (ivb’s)

.nivb	number of allocated string storage units (ivb's)
.nobs	switch to inhibit issuing of backspace characters
.noff	switch to inhibit issuing of form feed characters
.notab	switch to inhibit issuing of tab characters
.nsrcl	number of lines in source files
.page	current page number
.po	page offset
.sec	current second (0 - 59)
.stanr	current segment statement number
.stree	switch to enable "short tree" format
.tagerr	count of number of mismatched tags
.trlevel	level of current tree node
.trlnr	line number of current tree node
.trpage	page number for definition of current tree node
.trshort	line number of previous display of current tree node
.wday	day of week (1 - 7, Sunday = 1)
.xrsw	switch to enable collection of reference information
.yday	day of year (1 - 366)
.year	year minus 1900

Index