EXHIBIT B

PostScript®
LANGUAGE REFERENCE
third edition

Adobe Systems Incorporated

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California • New York • Don Mills, Ontario
Harlow, England • Amsterdam • Bonn • Sydney • Singapore • Tokyo
Madrid • San Juan • Paris • Seoul • Milan • Mexico City • Taipei

Library of Congress Cataloging-in-Publication Data
PostScript language reference manual / Adobe Systems Incorporated. — 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-37922-8
1. PostScript (Computer program language) I. Adobe Systems.
QA76.73.P67 P67 1999
005.13'3—dc21 98-55489
CIP

ISBN 0-201-37922-8
1 2 3 4 5 6 7 8 9 CRS 03 02 01 00 99
First printing February 1999

Contents

Preface xiii

Chapter 1: Introduction 1

1.1 About This Book 3
1.2 Evolution of the PostScript Language 5
1.3 LanguageLevel 3 Overview 6
1.4 Related Publications 7
1.5 Copyrights and Trademarks 9

Chapter 2: Basic Ideas 11

2.1 Raster Output Devices 11
2.2 Scan Conversion 12
2.3 Page Description Languages 13
2.4 Using the PostScript Language 15

Chapter 3: Language 23

3.1 Interpreter 24
3.2 Syntax 25
3.3 Data Types and Objects 34
3.4 Stacks 45
3.5 Execution 46
3.6 Overview of Basic Operators 51
3.7 Memory Management 56
3.8 File Input and Output 73
3.9 Named Resources 87
3.10 Functions 106
3.11 Errors 114
3.12 Early Name Binding 117
3.13 Filtered Files Details 123
3.14 Binary Encoding Details 156

Chapter 4: Graphics 175

4.1 Imaging Model 176
4.2 Graphics State 178
4.3 Coordinate Systems and Transformations 182

It begins with a brief overview of the PostScript interpreter. The following sections detail the syntax, data types, execution semantics, memory organization, and general-purpose operators of the PostScript language (excluding those that deal with graphics and fonts). The final sections cover file input and output, named resources, function dictionaries, errors, how the interpreter evaluates name objects, and details on filtered files and binary encoding.

3.1 Interpreter

The PostScript interpreter executes the PostScript language according to the rules in this chapter. These rules determine the order in which operations are carried out and how the pieces of a PostScript program fit together to produce the desired results.

The interpreter manipulates entities called PostScript *objects*. Some objects are data, such as numbers, boolean values, strings, and arrays. Other objects are elements of programs to be executed, such as names, operators, and procedures. However, there is not a distinction between data and programs; any PostScript object may be treated as data or be executed as part of a program.

The interpreter operates by executing a sequence of objects. The effect of executing a particular object depends on that object's *type*, *attributes*, and *value*. For example, executing a number object causes the interpreter to push a copy of that object on the operand stack (to be described shortly). Executing a name object causes the interpreter to look up the name in a dictionary, fetch the associated value, and execute it. Executing an operator object causes the interpreter to perform a built-in action, such as adding two numbers or painting characters in raster memory.

The objects to be executed by the interpreter come from two principal sources:

- A character stream may be scanned according to the syntax rules of the PostScript language, producing a sequence of new objects. As each object is scanned, it is immediately executed. The character stream may come from an external source, such as a file or a communication channel, or it may come from a string object previously stored in the PostScript interpreter's memory.
- Objects previously stored in an array in memory may be executed in sequence. Such an array is known as a *procedure*.

The interpreter can switch back and forth between executing a procedure and scanning a character stream. For example, if the interpreter encounters a name in a character stream, it executes that name by looking it up in a dictionary and retrieving the associated value. If that value is a procedure object, the interpreter suspends scanning the character stream and begins executing the objects in the procedure. When it reaches the end of the procedure, it resumes scanning the character stream where it left off. The interpreter maintains an *execution stack* for remembering all of its suspended execution contexts.

3.2 Syntax

As the interpreter scans the text of a PostScript program, it creates various types of PostScript objects, such as numbers, strings, and procedures. This section discusses only the *syntactic* representation of such objects. Their internal representation and behavior are covered in Section 3.3, "Data Types and Objects."

There are three encodings for the PostScript language: *ASCII*, *binary token*, and *binary object sequence*. The ASCII encoding is preferred for expository purposes (such as this book), for archiving documents, and for transmission via communications facilities, because it is easy to read and does not rely on any special characters that might be reserved for communications use. The two binary encodings are usable in controlled environments to improve the efficiency of representation or execution; they are intended exclusively for machine generation. Detailed information on the binary encodings is provided in Section 3.14, "Binary Encoding Details."

3.2.1 Scanner

The PostScript language differs from most other programming languages in that it does not have any syntactic entity for a "program," nor is it necessary for an entire "program" to exist in one place at one time. There is no notion of "reading in" a program before executing it. Instead, the PostScript interpreter *consumes* a program by reading and executing one syntactic entity at a time. From the interpreter's point of view, the program has no permanent existence. Execution of the program may have side effects in the interpreter's memory or elsewhere. These side effects may include the creation of procedure objects in memory that are intended to be invoked later in the program; their execution is *deferred*.