SERIES EDITOR SCOTT KNASTER

Macintosh Inside Out

Programming THE LaserWriter®

DAVID A. HOLZGANG

Programming the LaserWriter®

Programming the LaserWriter®

David A. Holzgang

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

Library of Congress Cataloging-in-Publication Data

Holzgang, David A.
Programming the LaserWriter / David Holzgang.
p. cm. — (Macintosh inside out)
Includes bibliographical references and index.
ISBN 0-201-57068-8
1. LaserWriter (Printer) I. Title. II. Series.
TK7887.7.H65 1991
686.2'25445265—dc20 90-28918
CIP

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

Copyright © 1991 by David A. Holzgang

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor: Carole McClendon
Technical Reviewer: Scott Zimmerman
Cover Design: Ronn Campisi
Text Design: Copenhaver Cumpston
Set in 10.5 point Palatino by ST Associates, Inc.

ISBN 0-201-57068-8

1 2 3 4 5 6 7 8 9-MW-9594939291
First printing, April 1991

Contents

Foreword by Scott Knaster *xi*
Acknowledgments *xiii*

Introduction and Overview **xv**
Purpose of This Book xv
Thinking About Macintosh Printing xvi
Printing Manager and the LaserWriter xviii
PostScript and QuickDraw xix
Place of PostScript in Macintosh printing xx
LaserWriter Driver and the Laser Prep file xxi
Requirements xxi
Book Structure xxii
Expected methods of use xxiii
Resources available xxiii
Book Contents xxiv

1. LaserWriter Printing Process 1
Chapter Overview 1
LaserWriter History 2
LaserWriter Hardware 4
LaserWriter and LaserWriter Plus 4
LaserWriter II series 5
Personal LaserWriter series 5
Hardware overview 6

Software Components 8
QuickDraw 9
PostScript 10
Font Manager 10
Printing Manager 12
LaserWriter driver and Laser Prep 14
AppleTalk 14
Printing Process Overview 20
Chooser dialog 21
Printing Manager 22
Macintosh printing philosophy 26
LaserWriter Communications 28
Communication modes 28
Operating modes 30
Mode selection 32
Conclusion 33

2. PostScript Language Concepts 35
Chapter Overview 35
Language Characteristics 36
Important features 36
PostScript operators 41
PostScript objects 43
Device management 43
Page Structure 46
Output structure 47
Measurement and coordinates 49
Graphics state 55
Objects 56
Object notation 57
Object types 58
Writing PostScript 64
Stacks and Dictionaries 65
Stacks 65
Dictionaries 67
Fonts 69
Font measurements 70
PostScript fonts 71
Conclusion 73

3. Printing Manager Functions 75
Chapter Overview 75

Printing Environment 76
Printing Manager calls 76
QuickDraw translation 78
Example Code Structure 80
CSimpleLWDoc 81
CSimpleLWPane 87
First example 90
LaserWriter Printing 92
Revised example 93
PostScript code for example 100
Device-independent printing calls 102
PicComment usage 102
Additional PicComment examples 105
Printing Efficiency 116
Device independence 116
Optimizing printing functions 118
Resource management issues 120
Permanent and temporary downloading 122
Conclusion 123

4. PostScript Program Construction 125
Chapter Overview 125
Testing with PostScript 125
General example structure 126
Debugging and error handling 132
Creating Procedures 133
Procedure definitions 133
Procedures and dictionaries 139
Transfer of control 146
Performing Graphics Operations 158
Line operations 158
Graphics state 161
Closed paths 166
Shading and patterns 175
Clipping 183
PostScript and QuickDraw 192
Performing Text Operations 195
Text display 195
Text controls 200
Font operations 203

Foreword

If you've been using a Macintosh or LaserWriter for very long, you probably know that Apple often doesn't do things the same way that everybody else does. A good example of this was in 1985, when Apple shipped the first LaserWriter. While many other companies were taking advantage of cheap, new laser marking engines to produce very low-priced printers, Apple took a different approach. Apple build a laser printer, but it included a lot more: a powerful microprocessor, a lot of RAM, and a little-known language called PostScript that knew how to describe a page of text and graphics.

So, instead of building a cheap laser printer, Apple created a computer—in fact, one that was more powerful than any computer that Apple had shipped—and built a printer around it. Of course, this made the LaserWriter a lot more expensive than its apparent competition, and many folks predicted that Apple had really goofed this time. But the Macintosh and LaserWriter combination, along with nifty software like PageMaker, created desktop publishing, and, some say, kept Apple in business at the time.

There's no doubt that desktop publishing is still a vital part of what people do with their Macintosh computers, but another interesting, less-explored part of the LaserWriter story is that old line about the LaserWriter being the most powerful computer that Apple makes. Since 1985, Apple has adjusted its priorities, and it now makes Macintosh computers that are actually more powerful than some of its printers, but the latest LaserWriters are no slouches in the computing department, either. The LaserWriter IINTX, for example, features a 68020

microprocessor, not less than 2 Mb of RAM, and 1 Mb of ROM. You can add more RAM or a hard disk to get it to do more. That sounds an awful lot like a computer to me!

This book lets you take a look inside the computer that's at the heart of your LaserWriter just trying to get out. You'll learn about how the Macintosh communicates with the printer, and you'll find out more about what they're really saying to each other while you're waiting for your pages to come out. You'll also find out how to do all sorts of neat graphics tricks using the richness of the layers of software that the LaserWriter and the system software provide. And your tour guide, David Holzgang, is someone who's been down every wire and has seen every network packet go by.

Apple's LaserWriters provide an incredible set of software capabilities that are barely touched by most commercial applications. David shows you how you can really exploit that computer you didn't know you had—the one inside your printer.

Scott Knaster
Macintosh Inside Out Series Editor

Acknowledgments

Making any book requires the help and support of a variety of people; this is especially true of a book like this one, which deals with a process that is a little-understood part of the Macintosh system, combines several different pieces of hardware and software, and is highly complex in itself. I would, therefore, like to thank several important people who have helped me make this book better than it would otherwise be.

First of all, I must thank the generous and patient people at Addison-Wesley. Carole McClendon, who not only let me discuss this project with her, but even bought the lunch; Joanne Clapp Fullagar, my most patient muse and editor, who has now suffered my kicking against the goad through two books, to the benefit of both; and Rachel Guichard, who has always taken care of all my requests, even the most minor and temperamental, with charm and efficiency, and even makes me believe that she doesn't mind. I also appreciate very much the support and intelligent encouragement of Scott Knaster, my series editor, whose excellent work as an author sets a high standard indeed for the rest of us. Beth Burleigh, of Addison-Wesley's Production department, has done a remarkable job of taking the bits and pieces of the final edit and turning them into a coherent and collected book.

Of course, with any technical book, the support of the manufacturer is also important. Developer Technical Support has been very responsive to all my queries; both Scott "Zz" Zimmerman and Pete "Luke" Alexander have been very helpful. Moreover, "Zz" went above and beyond the call of duty by providing the technical review of the manuscript, thus saving me a number of embarrassing lapses. As always, any that remain are entirely my

fault, in spite of his (and other's) best efforts. In addition, I would like to thank Doedy Hunter and Ric Jones of the Apple Loan Program for their help in getting me a disk for testing purposes on a rush basis.

Last, but hardly least, I would like to thank my friend and associate, George Mastalir, who provided most of the initial code for the LaserWriter modules in this book and who has helped me in innumerable ways to better understand what was going on between the Macintosh and the LaserWriter.

Introduction and Overview

Purpose of This Book

This book is intended to give the Macintosh developer or other interested user information about the LaserWriter printer. Certainly anyone who works with a Macintosh on a regular basis has some experience with the LaserWriter; even users who don't have one themselves seem to have regular access to one at work, through a colleague or friend, or through a local service bureau. When you have used a LaserWriter, it seems that no lesser alternative is ever quite good enough again. Developers, therefore, always want to be sure that their software fully supports the LaserWriter. The Printing Manager calls in the Macintosh Toolbox provide good support for printers in general, but a LaserWriter can be used in many more ways than are generally made available to users by the standard Toolbox calls.

There is an important point that you should understand immediately about this book: It is primarily about the LaserWriter printers that include support for the PostScript language. This includes the original LaserWriter and most of the models identified as LaserWriters. There are, however, two important exceptions: the LaserWriter SC and the Personal LaserWriter SC. These two devices do not support PostScript output and are not attached to the host Macintosh by an AppleTalk connection. Instead, they use QuickDraw commands for printing and are connected by the SCSI port. If you have one of these printers, only a small part of the information presented here will be of use to you. Also, when the term "LaserWriter" is used in this book, it generally refers to

the LaserWriters that support PostScript and are connected by AppleTalk. If certain information applies to all the LaserWriter family of devices, including the non-PostScript devices, that is mentioned explicitly.

As an output device, the LaserWriter has qualities that are unique and valuable. It allows you to make both text and graphics that are of a higher quality than most output devices, and it is, in itself, a complex and versatile processor that can be a equal participant in making your output look good. One simple example is the LaserWriter's ability to place both text and graphics at any size and rotation on the page, but you will meet many more advanced features as you progress through this book. Giving your users the opportunity to take advantage of these special LaserWriter features makes your application stand out from most, and it gives you a market edge on the competition.

▶ Thinking About Macintosh Printing

Even experienced developers might be forgiven if they have not spent much time thinking about printing on the Macintosh. The Toolbox Printing Manager calls provide easy access to a wide variety of printing functions, and they also provide these functions without concern for the exact device that the user has chosen for printing. This allows both the developer and the user to ignore the specific device requirements for output and focus on more important jobs. This is, in fact, generally the correct method of handling printing on the Macintosh.

Using the Toolbox and the Printing Manager is completely general, and therefore new devices or printing architectures within the system cause no concern for the user or the developer. If this approach provides all the printing functionality that you require, it is absolutely the best way to provide printing in your application. This approach also ensures that you are in the mainstream of Macintosh development and helps you get technical support and code examples when you need them.

However, you may very well want to go beyond the basic printer support offered through the Printing Manager and take full advantage of the LaserWriter's advanced features. The LaserWriter is a printing tool of a much higher caliber than an ImageWriter, and you can, if you choose, allow your users full access to its many features. Also, you may want to take advantage of the intelligence built into the LaserWriter. In any case, such tasks can be successfully undertaken and accomplished by using the proper programming techniques.

There are, fundamentally, three levels of support from which you might choose. First, you can support the LaserWriter through the

Printing Manager, as part of general device support. Although you may not know it, the Printing Manager provides several useful and effective means for taking advantage of LaserWriter features. These means are structured so that the application using them does not even have to know that the printing device is a LaserWriter. When used correctly, the Printing Manager and the chosen printer driver change the application output to take advantage of the LaserWriter features if the device is a LaserWriter (or a compatible printer) and to ignore these commands if it is not.

The second method of working with the LaserWriter is still through the Printing Manager, but with the application being fully aware that the chosen printer is a LaserWriter. In this case, you must add code to test for the type of device and its name; then the rest is quite easy. You can make some useful changes in your application that make the users' life a little easier and their output a little more exciting.

The third and most direct approach is to communicate directly with the LaserWriter from your application, bypassing the Printing Manager altogether. This can be very useful, and it is the only way that you can get information directly back from the LaserWriter. Since the LaserWriter is a computer itself and a full participant on an AppleTalk network, you can get quite a bit of information when you know how to ask and how to listen for the answers. It's a lot like having a good conversation anyplace!

Like all powerful tools, however, the proper use of these advanced techniques requires working within a sensible framework of caution and with concern both for your user and your application. The first set of techniques is quite robust and is the method Apple itself recommends for providing LaserWriter support. These methods allow you to produce useful output on the LaserWriter yet they don't have negative consequences when the output device is something less intelligent and capable. These should survive no matter what changes occur in the system software. The second set is still fairly safe since many of the flagship Macintosh applications, such as page layout applications, use these techniques to create the necessary quality of output for their users. You can be reasonably sure that these techniques will not change without adequate advance notice and that, if they change, Apple will provide some replacement techniques that give the same or better functionality. The third set of techniques is the most risky and the most subject to change. However, even these techniques cannot be altered lightly; they are the foundation of Apple's own device management and control software. The major concern here for the outside developer is that the methods might change without any warning (except a broken

application, of course) because the primary users of these techniques are within Apple itself and are privy to forthcoming changes. In such an event, you might need to change your approach on short notice or get information from Apple about how to work around the problem. For a commercial application, this would be a serious concern.

Even if you never use any of these techniques except in the examples in this book, I think that you will be a much better developer for knowing how they work. When you are done here, you will have been through the bowels of both the Printing Manager and the LaserWriter, and you will certainly be in a much better position to choose your tools and create a robust and vigorous application that provides all the functions that you think the market can use.

▶ The Printing Manager and the LaserWriter

The general Macintosh printing process is quite complex compared to that of the less sophisticated microcomputers of earlier days when you simply sent a stream of characters to a device that was the equivalent of a typewriter. Today, much higher standards exist for both printed output and for user control and options, and so there are good reasons for each layer of the printing chain. With each additional layer comes new functionality and improved quality. After all, it was not so long ago when the idea of high-quality computer output was something that looked as if it had been typed; and the next best level was something called "near letter quality." The LaserWriter, on the other hand, produces output that is "near typeset quality," which is an altogether different level of performance.

The fundamental structure of the printing process on the Macintosh follows a series of steps that involve several different pieces of software and that, in addition, transform the output from a screen format to a printer format. This process generally begins when you choose Print... from the File menu. The application, which at that point has been displaying information on the Macintosh screen, now must transform this screen data (which may, of course, represent several actual physical screens) into printed output. It performs this transformation by calling Printing Manager functions from the Macintosh Toolbox. The Printing Manager checks the nature of the currently chosen printing device, which you set in the Chooser function, and also checks the current Page Setup parameters, which vary according to the print device that you have chosen. These parameters define such features as the page size and orientation of the desired output. Once this is set up, the Printing Manager returns to the application, which then redraws the infor-

mation that needs to be printed, just as if it were displaying that on another screen. The Printing Manager takes that information and passes it on to the printer driver, which converts the output to a format that the printer can understand. The printer driver also may actually send the information to the device, or it may send it to a file where the information is queued for later processing by a function such as the Print Monitor in MultiFinder.

Each printer driver has the name of the device that it supports, and therefore the LaserWriter driver is called LaserWriter. The LaserWriter driver, however, has to transform the data further before it can be processed by the LaserWriter device. The driver must convert the QuickDraw commands that it receives, which are the commands used to draw on the screen, into PostScript commands, which are what the LaserWriter device understands. It does this with the help of what may be thought of as a translation dictionary, called Laser Prep. Together, the LaserWriter driver and the Laser Prep file convert the QuickDraw commands that were generated by the application into PostScript commands that produce virtually the identical information that you saw on the screen onto your LaserWriter printer.

If any of this seems a bit confusing right now, don't be alarmed—this process is, essentially, the subject of the book, and you will learn about each of these steps in detail as you proceed. On the other hand, if any of that seems a bit short and superficial, the same comment applies—you will see the full range of complexity in the actual chapters where we work through this process.

▶ PostScript and QuickDraw

One purpose of this brief overview is to help you understand why you need to learn some PostScript in order to program effectively for the LaserWriter. The basic process from QuickDraw to PostScript uses the translation functions in the Laser Prep file to change QuickDraw commands into PostScript procedure calls. PostScript is the native language, so to speak, of the LaserWriter, and of a wide variety of other devices as well. It is both a graphics language and a programming language of great power and utility. You use PostScript, whether you know it or not, every time that you print a document to the LaserWriter, as described earlier. In addition, there are a variety of methods that your application can use to send PostScript commands directly to the LaserWriter itself. These techniques are what the most sophisticated applications use to create that great output that we have all admired, and they are the subject of this book.

Because of this translation process, you want to understand PostScript and its concepts to get full use of Macintosh printing. When you understand how the LaserWriter works internally and how its own programs are constructed and processed, you will be able to understand how you can take advantage of this processing to get the output that you (and your users) want onto the page. A developer who does not understand this process is likely to underutilize the LaserWriter as an output device, and, of perhaps more serious consequence, to mishandle the general printing process without realizing it. Certainly, in my own experience, I have often seen that the so-called slowness of Macintosh LaserWriter output is caused by applications that are doing things that no one who understands the LaserWriter would ever do. In this book, you will learn both what to do and not do, and—more importantly, I think—why to do or not to do it.

▶ Place of PostScript in Macintosh printing

PostScript is the glue that allows the Macintosh to perform the magic transformation that goes on when you print to a LaserWriter. Although it is, as it ought to be, mostly invisible to the end user, it is a subject well worth understanding because it is the complexities in this printing process that generally provide the opening for those problems that you, and your users, do encounter. Moreover, PostScript is a fascinating language in itself and will certainly repay the small amount of study needed to acquire the basic concepts and structure of the language.

PostScript is a page description language, which means that it is designed to produce units of output, which we normally call pages. It is, to a great degree, independent of the specific device that it is running on, so that pages produced for printing on the LaserWriter can be moved directly to a Linotronic imagesetter, for example, with virtually no change. This same independence allows users to create encapsulated PostScript files, which can be moved not only to other devices, but also embedded in other pages, which are created and manipulated by other applications. The reach of the PostScript language is really remarkable, and it has recently been extended, as PostScript Level 2, to add more comprehensive support for a variety of functions that should make the language even more prominent in the future. The very fact that PostScript can be so easily extended to support such functions as color correction and processing and non-Roman languages shows the robust flexibility that was inherent in the original design.

▶ LaserWriter Driver and the Laser Prep file

The translation process that takes place inside the PostScript output device is performed by a series of procedures that are stored in the System folder under the name of the Laser Prep file. This file provides the PostScript language routines that convert the QuickDraw commands that you use to draw text and graphics on the output screen into printed output. The Laser Prep file is managed by the LaserWriter driver, which downloads the file to the printer each time the printer is turned on or reset.

If you are a developer or user who has been working with the System 7.0 release of the Macintosh system software, you may be wondering if this discussion still is true because the old Laser Prep file has shrunk remarkably. Do not despair; in System 7.0 the Laser Prep file shrinks, but its functions have been retained and moved into the LaserWriter driver. All the techniques that you find here seem to work as well with System 7.0 as they did with earlier releases.

▶ Requirements

There are, naturally, certain requirements that a reader must fulfill to get the full value from this book. To begin with, you should be somewhat familiar with programming on the Macintosh. I do not explain ordinary things about the Toolbox calls, for example, and I assume that you are reasonably comfortable with them and with their structure both in the reference materials, such as *Inside Macintosh*, and in actual use in code. If you are not quite comfortable with this, I would recommend that you use one of the good basic tutorials on Macintosh programming, such as the *Macintosh Programming Primer* (Addison-Wesley, 1989) by Dave Mark and Cartwright Reed. Also work through the tutorial material in the Think C manual. As an aside, note that none of the references in the Bibliography is particularly basic.

You may wonder why I chose Think C 4.0 as the language for this book. I chose it, first of all, because I am much more comfortable in C than in Pascal and also because Think C 4.0 provides a set of object-oriented extensions that make this type of project much easier than it would otherwise have been. The great virtue of object-oriented programming, as most of you probably have heard already, is that it allows you to use and reuse code without a great familiarity with the internal structure and sequence of that code. Here, that ability allows you to get up and running with the full use of the Printing Manager functions without any real discussion of what those require or how they

are handled. You can simply and I can safely leave that part of the processing to the classes that are already included in Think C 4.0. In addition, Think C is both fast and efficient, thus allowing you to try various changes and options without too much frustration and overhead. Finally, Think C is, in my opinion, the best environment for this type of simple project development. It is easy to use, economical to purchase, and comes with a wide array of tools and classes to help you get started. For all these reasons, I chose it as the development environment here.

This doesn't mean that you can't use these techniques in other environments or other languages. Indeed you can; the only difficulty for you in those cases will be in translating the examples to that other environment or language, and in providing yourself with the equivalent functionality that we are assuming comes, without further discussion, from the classes in Think C.

▶ Book Structure

This book is basically structured as a series of lectures and tasks. In most cases, this "show and tell" approach would need some explanation because many books about computers do not have direct, useful code in them. Being part of a series, however, this book presents a different problem. The majority of the other titles in the *Macintosh Inside Out* series are very much directed toward practical, useful examples, which is the best way to approach many topics. This book differs in the opposite direction: It must have a large amount of reading material in addition to the hands-on examples. This approach is required by the nature of subject and of the PostScript language. If you are generally familiar with PostScript, you can probably skip the sections on language structure and use without too much problem; if not, you will need the information to understand the examples.

The intention in this book is to help the reader understand and use these tools and techniques. Therefore, it is important that all the necessary information for that understanding be included here, even when the information is also available in other sources. Indeed, one of the benefits of this book is that much of the information that was previously scattered over five or six sources is now concentrated in one place. Nevertheless, this means walking a fine road: on the one hand, to include all the information that is essential; on the other, not to repeat information that the reader already knows or can easily find out. I hope I have done this well; if anything, I have leaned toward including a little extra information, on the theory that, even if you already know it, you won't mind having it at your fingertips as you work through the material.

▶ Expected methods of use

I expect and recommend that most readers go rapidly once through a section of the material, looking at the code examples and the sample output and trying to follow the flow within the text. This allows you to see what the section is about and tell whether it is of interest, and also whether you are already familiar with the material. Then you can return for a detailed review of the code. In this way, you can be sure that you understand where you are going before you start coding and testing. Since there is a lot of information in a short compass in several chapters, this overview method will not cost you much time and will allow you to work through the exercises much more quickly.

You may wish, instead, to simply compile and run the examples in the section that you are reading as you proceed. Then, when you have finished your reading, return to the code explanations and rerun the programs in debug mode, following the execution at a reasonably high level so you don't spend too much time on the system functions or in the predefined classes. In this way you should get a good working knowledge of the code.

▶ Resources available

I have tried, in this book, to give you at least enough information so that you can see why each example works. However, this is a large and complex subject and, unless I want to kill a forest, it seems only reasonable to allow you to search out some of the more detailed information on your own. Also, this book could not possibly cover all of the material visited here in depth; the subject of PostScript alone has generated several books, most of them as long or longer than this one. Therefore, you should be prepared to explore alternate sources of information on many of the subjects covered in this book. Very often, when a specific subject is covered in another source, I have listed that source in the text. This is especially true where there is information in one or another of the Macintosh Technical Notes that is of special interest and relevance. The Technical Notes are very valuable to any developer and often provide the best, or even the only, source of much of the information that is important in programming for the LaserWriter. The list of works, given in the Bibliography, gives additional sources that can either help you use the information here or expand on it in various ways.

▶ Book Contents

This book consists of seven chapters, not including this introduction. A brief overview of each chapter is presented here to allow you to see how this book approaches the problems of getting the most out of your LaserWriter.

Chapter 1, "LaserWriter Printing Process," introduces you to the hardware and software components that enter into the process of making the LaserWriter print. In hardware, you review the various models of the LaserWriter family. In software, you look briefly at both QuickDraw and PostScript and then review the Macintosh software that is required for printing on the LaserWriter. The chapter contains a brief overview of how these components work together in the process and also discusses the various methods of communicating with the LaserWriter.

Chapter 2, "PostScript Language Concepts," gives you a quick introduction to the general concepts behind the PostScript language, making particular reference to differences between PostScript and QuickDraw. This chapter, which contains a large amount of material, is designed to help you get the most out of the PostScript code examples later in the book. The chapter begins with an overview of the PostScript language. Then it discusses the PostScript coordinate system in some detail. The next sections of the chapter discuss PostScript objects in general and then stacks and dictionaries in particular. The chapter ends with a brief look at PostScript fonts and font handling.

Chapter 3, "Printing Manager Functions," covers the standard Printing Manager calls and how they work to produce output on a LaserWriter. The chapter uses a series of simple examples to show you in some detail how the various components of the printing process work together to create LaserWriter output. The examples also show you how to take advantage of the LaserWriter, if present, by using QuickDraw PicComments to send PostScript commands to the LaserWriter. These examples provide some insights that lead to a discussion of printing efficiency on the LaserWriter, which ends the chapter.

Chapter 4, "PostScript Program Construction," uses the techniques that you learned in the previous examples to create a small application that you can use to send PostScript language programs to the LaserWriter. With this tool, the chapter teaches you some simple PostScript language programs that illustrate important concepts in the language. You learn how to use and work with PostScript graphics and text output. This allows you to study the structure of a standard PostScript program and to investigate encapsulated PostScript files.

Chapter 5, "Basic LaserWriter Programming," covers the first real code that integrates using the LaserWriter from your application. Here you have examples that allow you to identify that the LaserWriter is the actual output device and get its name from the network. With this and the PostScript code that you learned in Chapter 4, you can create some useful effects. The chapter also covers how to modify the job and style dialogs to allow the user to provide additional input, which you can then use to create special LaserWriter effects. The chapter also covers all the alternative methods of sending PostScript language commands to the LaserWriter, and discusses some of the advantages and drawbacks of each of these. It ends with a brief discussion of how to create and use your own PostScript header files for your application.

Chapter 6, "Font Handling," covers questions about font handling in both QuickDraw and PostScript. Because of differences in the two environments, it is very important that you have a clear appreciation of the similarities and differences that are inherent in these two approaches. The chapter includes discussions of System 7.0 outline fonts and font processing where that is relevant. The major issues in this chapter are how to identify fonts that are required by a document and how to get those fonts to the device for output. The chapter ends with a look at how to determine what fonts are present on the PostScript device and how spoolers and other document manager software deal with font requirements.

Chapter 7, "Advanced LaserWriter Operations," deals with direct communications with the LaserWriter, primarily over the AppleTalk network. Although the LaserWriter can communicate over both AppleTalk and serial ports, the AppleTalk connection is the most common. However, to do some tasks—especially to determine the current state of the LaserWriter and to carry on a two-way dialog with it—your application must be able to set up and use an AppleTalk connection that talks directly to the LaserWriter. This chapter discusses how to do that, using a series of concrete examples to show you what the requirements are for this process. The chapter then discusses some of the valuable information that you can derive by using direct communications, coupled with simple PostScript programs that query the device as required. The chapter ends with a brief look at the PostScript file system, with particular emphasis on how that is used for font storage.

1 ▶ LaserWriter Printing Process

▶ Chapter Overview

This chapter provides you with a general overview and review of printing from the Macintosh. It focuses specifically, but not exclusively, on printing to a LaserWriter device, which is the fundamental subject of this book. However, you must understand the entire printing process with reasonable clarity to program the LaserWriter. This first chapter helps point you in the right direction and provides important signposts for the remainder of the book.

All of the material in this chapter is covered in other sources—often in more depth and detail. The coverage here is a review, not a tutorial or comprehensive reference; for such coverage, you need other references. The point here is that you must be acquainted with the basic process and with some of its variants in order to move on in LaserWriter programming. Where appropriate and useful, the chapter does provide information about other reference works, both books and notes. The intention, then, is to provide the reader, in one place, a review and a checklist of important topics and information.

The chapter first reviews the LaserWriter family of devices. Although these are now quite familiar, they still are truly awesome (a much overworked word; here being used in its actual, dictionary definition) machines that provide a quality and flexibility of output unimagined and unimaginable twenty years ago.

Then the chapter looks over the software components that are essential for printing from the Macintosh to the LaserWriter. In this

process, you will look at the general requirements of Macintosh printing and then concentrate on the special software that enables you to print on the LaserWriter. You then take another look at the required software components in the order in which they are used in the printing process. Next you review the use of the common tools, the Chooser and the Printing Manager dialogs, that allow Macintosh users to get so much versatile output from these devices.

As you work through this book, and indeed as you work seriously with the LaserWriter, you need to see the actual output of the LaserWriter driver so you can review and debug it. This chapter tells you how to generate these important files and capture them on your disk. Once there, you can examine and modify the files using standard techniques. You will actually do some of this work in later chapters, so it's very important that you learn how to generate these files. Then the chapter discusses the potential benefits and drawbacks that are inherent in the techniques that you will learn here. This is important because it bears directly upon how your application serves (or abuses!) the Macintosh human interface guidelines and the Macintosh printing philosophy.

The chapter closes with a section on LaserWriter communications and modes of operation. Although this book pretty much assumes that you are working in a standard AppleTalk environment, you should remember that there is an alternative, in serial communications, that can provide some different and important features. This material is covered much more extensively in other books and documentation, which are referenced for you in the text and listed in the Bibliography.

▶ LaserWriter History

The Apple LaserWriter series of printers is one of the most interesting and innovative groups of peripheral devices to come out of the personal computing revolution. It is a small, relatively lightweight device that is capable of printing both text and graphics. It consists of a laser scanner and a xerographic printing mechanism, coupled to a device controller that includes a microprocessor and a sophisticated set of software that is especially dedicated to the printing process.

It is probably valuable to recall here a little of the history of these remarkable devices. The first LaserWriter was produced in 1985, and it began a notable change in the way in which documents are produced. It is not too much to say that this device, alone, began what is now called the "desktop publishing revolution."

There had certainly been laser output devices before. The first common variety of these came, most notably, from the Xerox Corporation, which had produced several office-sized products that could be used for printed output. However, these devices were sensitive, clumsy, difficult to use, and almost impossible to program. They were useful, but they required specialized software and careful handling to create even ordinary output.

The next generation of laser printers used the Canon laser-xerographic engines that had originally been developed as a small, lightweight copier that did not require much in the way of maintenance or support. The first of these was the Hewlett-Packard LaserJet. This printer was very well received and very popular. It allowed the user to produce high-quality pages of output with much more ease and flexibility than the previous printers, whether dot matrix or xerographic. It was small, fast, and relatively light. However, it still had many disadvantages. Most notably, it had a difficult time producing high-quality text and graphics; and providing and using additional type styles or faces was both expensive and limited. Although it was much improved over other, similar devices, it was certainly no threat to the publishing or typesetting industries.

The Apple LaserWriter, which was introduced shortly thereafter, moved to a new level of quality and flexibility. By incorporating the new page-description language, PostScript, it was able to produce elegant pages of output that seamlessly combined both text and graphics. The LaserWriter's type was of such high quality that it was rapidly termed "near-typeset" quality, as a kind of ironic parody of the previously used term, "near-letter quality," for the eye-straining dot matrix output that many users had come to abhor. Type could be set at any size or orientation that the user desired, and several type styles were available for the device. Moreover, its output could be transferred, with no additional processing, to a typesetting device and used to directly produce actual typeset output. Coupled with the fine graphics and intuitive interface of the Macintosh computer, this peripheral encouraged the development of software that provided page layout and typesetting controls directly at the computer screen. These in turn allowed both artists and businesspeople to produce quality publications with a flexibility and ease that had previously been unavailable at any price. This was the start of publishing from the desktop.

▶ LaserWriter Hardware

There are or have been seven models of the Apple LaserWriter family of devices. In chronological order, they are as follows:

- LaserWriter
- LaserWriter Plus
- LaserWriter IISC
- LaserWriter IINT
- LaserWriter IINTX
- Personal LaserWriter SC
- Personal LaserWriter NT

These printers, although all from the LaserWriter family, break naturally into three subgroups: the LaserWriter and LaserWriter Plus, the LaserWriter IIs, and the new Personal LaserWriters. These subgroups more or less correspond to the chronological appearance of the devices.

The full technical specifications of each of these devices are provided in the reference manual that accompanies each device, or you can find the detailed information in the *Apple LaserWriter Reference*, published by Addison-Wesley. Here you will just look briefly at the physical device specifications as a review and consolidation, so you don't have to pull out too many other manuals as you read along.

▶ LaserWriter and LaserWriter Plus

The original device is simply called the LaserWriter. This workhorse printer contains 0.5 megabytes of ROM (read-only memory) and 1.5 megabytes of DRAM (dynamic, random-access memory). The printer consists of a marking engine, the Canon LBP-CX laser-xerographic engine, with a maximum speed of 8 pages of output per minute; and the controller, the Motorola 68000 processor, operating at 12 MHz. The related LaserWriter Plus was introduced somewhat later. It contains 1 megabyte of ROM, which allows it to provide more fonts. Otherwise, it is the same as the original LaserWriter. The original LaserWriter could be upgraded to the Plus model in the field, without being sent back to the factory.

▶ LaserWriter II series

The next group of devices is the LaserWriter II series. The printing engine in all these devices is the same: the Canon LBP-SX engine, with a maximum speed of 8 pages a minute. This series begins with the LaserWriter IISC. This device is quite different from the LaserWriter printers that preceded it, in that it does not contain the PostScript language. Instead, it uses the internal resources of the Macintosh computer and the internal QuickDraw commands to print pages. Also, the IISC does not attach to the Macintosh in the same fashion as the other devices. All the other LaserWriters attach to the Macintosh using the AppleTalk communications link; the IISC however, attaches by the SCSI port, which makes it unavailable for network use. The suffix "SC" on the name indicates that this device only attaches on the SCSI port.

The LaserWriter IINT is similar to the LaserWriter Plus, which it replaces, in the fonts provided and in the processor. Like the Plus, it also uses a Motorola 68000 processor with 1 megabyte of ROM, but it has 2 megabytes of DRAM. A new device created for this series, the LaserWriter IINTX, has a Motorola 68020 microprocessor running at 16 MHz, and it has expandable ROM and DRAM configurations. ROM can be expanded from 1 megabyte to provide additional fonts, and DRAM can be expanded from 2 to 12 megabytes to provide additional internal storage. It can also support an external SCSI hard disk drive for additional storage. Both the IINT and the IINTX use the PostScript page description language, which means that any output produced for the original LaserWriters can be printed on these devices without any modification. As was true of the original LaserWriter series, any of these models could be upgraded to a higher model in the field.

▶ Personal LaserWriter series

The newest additions to this series are the Personal LaserWriters. These two devices are based on the Canon LBP-LX laser-xerographic printing engine, which produces a maximum of 4 pages a minute. The Personal LaserWriter SC is comparable to the IISC, which it replaces. For our purposes, the most important point about the Personal LaserWriter SC is that it, also, does not support the PostScript language. The Personal LaserWriter NT is a new addition to the LaserWriter family and does not replace the previous models. It has a Motorola 68000 processor running at 12 MHz, 1.25 megabytes of ROM, and 2 megabytes of DRAM.

Important ▸

In this book, the generic term "LaserWriter" will be used to mean all the members of the LaserWriter series of printers that support the PostScript language. In particular, much of what is discussed in this book does not apply to the LaserWriter IISC and the Personal LaserWriter SC. In discussions of information that does apply to these devices, the device will be named explicitly. Similarly, in discussions of information that does not apply equally to all the LaserWriter devices, whichever devices are included or excluded will be specified by the following names:

- original LaserWriter
- LaserWriter Plus, or LW Plus
- LaserWriter IINT, or IINT
- LaserWriter IINTX, or IINTX
- Personal LaserWriter NT, or Personal NT

▶ Hardware overview

Let's now concentrate on the LaserWriters that provide PostScript support. It is important for you to know about certain features of these devices. To begin with, they all support both LocalTalk connections (using the AppleTalk network protocol) and serial communications; however, none of them provides parallel communication support.

Note ▸

A major distinction that you should be aware of is that all the LaserWriter devices (that is, excluding the LaserWriter IISC and the Personal LaserWriter SC) use the LocalTalk connection to communicate with the host device. They can be shared among several users, and they provide device naming and status information over the network connection. The LaserWriter IISC and the Personal LaserWriter SC, on the other hand, as indicated by the "SC" suffix on their names, are connected to the Macintosh through the SCSI port. They cannot be shared with another user, and they are not accessed in the same way as the other LaserWriter devices. Programming for these two types of device is quite different.

The LaserWriter devices contain both ROM and DRAM. Each of these is used for different purposes. The ROM contains the following components:

- Adobe System's PostScript language interpreter
- printer diagnostic routines
- communication routines for handling AppleTalk and serial communications
- emulation routines for other devices
- character sets from certain internal fonts that have been preprocessed for quick access

The DRAM contains the following components:

- frame buffer for constructing the page image
- font cache buffer for storing the scanned font outlines
- communications data buffer
- working storage for the PostScript page description (or program, if you prefer)

Certain font families are built into each of the LaserWriter devices. The original LaserWriter contains the following fonts:

- Times
- Helvetica
- Courier
- Symbol

The LaserWriter Plus, IINT, and IINTX contain the following fonts:

- Times
- Helvetica
- Helvetica Narrow
- Courier
- Symbol
- Palatino
- ITC Avant Garde Gothic
- ITC Bookman

- ITC Zapf Chancery
- ITC Zapf Dingbats
- New Century Schoolbook

The Personal LaserWriter NT adds one more font to the preceding list of eleven:

- IBM PC Graphics Extended Character Set (ECS)

All of these devices also offer an emulation mode that allows the printer to behave like some other device. This is primarily useful for printing from applications that cannot produce PostScript output. All of the LaserWriters can emulate the Diablo 630 printer, which is a common dot matrix printer. The LaserWriter IINTX and the Personal LaserWriter NT can also emulate the Hewlett-Packard LaserJet Plus. There are, inevitably, some limitations on the emulation modes; see your printer documentation for full details.

▶ Software Components

Whereas the previous section gave you a brief review of the LaserWriter family of devices, now you will review, with equal brevity, the software components that are used in printing from the Macintosh. Table 1-1 summarizes these software components and their place in the Macintosh printing system. You can use this table as a handy reference during the following discussion.

Table 1-1. Summary of printing software components

Software Component	*Function*
QuickDraw	The standard graphics creation and presentation software used in the Macintosh system. These routines are built into the Macintosh Toolbox.
PostScript	The graphics creation and presentation language used in the LaserWriter and in other devices. This language is built into the LaserWriter.
Font Manager	Supporting software in the Macintosh Toolbox that QuickDraw and other Toolbox routines use to access and work with fonts.
Printing Manager	Toolbox routines that manage the transformation of screen data into printable output in a standard, transparent, and device-independent manner.

Table 1-1. Summary of printing software components (continued)

Software Component	*Function*
LaserWriter Driver	The device driver software that works with the Printing Manager to change screen information (in QuickDraw) into PostScript documents that the LaserWriter device can print.
Laser Prep	The PostScript language routines that the LaserWriter driver uses to translate QuickDraw commands into PostScript.
AppleTalk	The communication method that is used to transmit information, including commands, status requests, and data, between the LaserWriter printer and the Macintosh computer.

▶ QuickDraw

QuickDraw is the standard method of creating graphics in the Macintosh. It provides the Toolbox routines that perform all graphics operations, including text presentation. QuickDraw has several special attributes. Most notably, as its name implies, it is very fast and very efficient. It is designed to handle graphics on the screen as mathematical constructs, not simply as bitmaps. It provides clipping and off-screen buffering for both efficiency and speed. Both your application and the Macintosh system routines themselves use QuickDraw to make text and graphics appear on the screen.

You access QuickDraw by setting up an output region, called a *grafPort*. This is generally tied to a display area such as a window. Each grafPort has its own coordinate system and graphics settings that control the drawing in that region. QuickDraw makes it easy to switch rapidly from one grafPort to another. QuickDraw can manipulate graphics as arbitrary regions that are not limited in shape, but are limited in size to a maximum of 32K bytes. In addition, QuickDraw allows you to define collections of drawing commands to create graphics as units of output: These are polygons and pictures. Of these, the concept of pictures is the one that is most important in this book, since it allows you to group an arbitrary set of drawing commands as a unit, called a *picture*. Furthermore, you can include additional information with the drawing commands as *picture comments*, within the complete picture definition.

▶ PostScript

This naturally brings us to the issue of using PostScript as an output mechanism. PostScript is the page-description language that is included in the LaserWriter ROM. It is the most common page-description mechanism in current use and has become, without question, the *de facto* standard in the publishing industry. This provides important benefits for the user and programmer alike. For the user, it allows output that both matches the screen display to as great a degree as possible and can be produced on a wide variety of output devices without any changes, still taking advantage of the greatest possible resolution on that device if desired. For the programmer, it provides a device-independent method of specifying page output that allows multiple output devices, covering a wide range of resolutions and features, to be supported with a single set of device controls and a single device driver.

PostScript is also a complete programming language, and it can be used for a wide variety of tasks. However, it has been designed and optimized for production of text, graphics, and scanned or digitized images. Because PostScript produces page descriptions that are independent of the resolution of any specific device, its output can be used on typesetting and high-resolution printers for improved reproduction and publishing.

Since your application draws on the screen in QuickDraw, but the LaserWriter understands PostScript, the QuickDraw commands that were used to produce the screen must be converted into PostScript commands so that the correct output is generated when you want to print the images that you have created, and that the user sees, on the screen. This is done by a variety of software components within the Macintosh Toolbox; most notably, by the Printing Manager and the LaserWriter driver. You will learn about these two components and how your application can interact with them to control the LaserWriter in Chapter 3.

▶ Font Manager

QuickDraw requires certain additional supporting software within the Macintosh Toolbox. One piece of this software is the Font Manager, which takes care of all specific text information. QuickDraw calls on the Font Manager to control the font being used for text output, that is, to define character widths, style formats, and character spacing information such as kerning. Fonts are specified to the Font Manager by a font

number, and the sizes of the characters are given in *points*, which are approximately 1/72 inch—conveniently identical to the QuickDraw coordinates—and which are the common measurement for type specifications in the printing and publishing industries.

The important point to note here is that the fonts that the Font Manager provides and that QuickDraw uses on the screen are quite different from the fonts that the LaserWriter uses. The most important difference is that on the screen the characters are represented by bitmaps, whereas on the printer the characters are represented by mathematical outlines. This has two important consequences. First, the screen fonts are precisely accurate only at certain generated sizes, which must be exact integer numbers. The generated sizes are usually indicated on a font menu by an outline character for the font size, as shown in Figure 1-1.

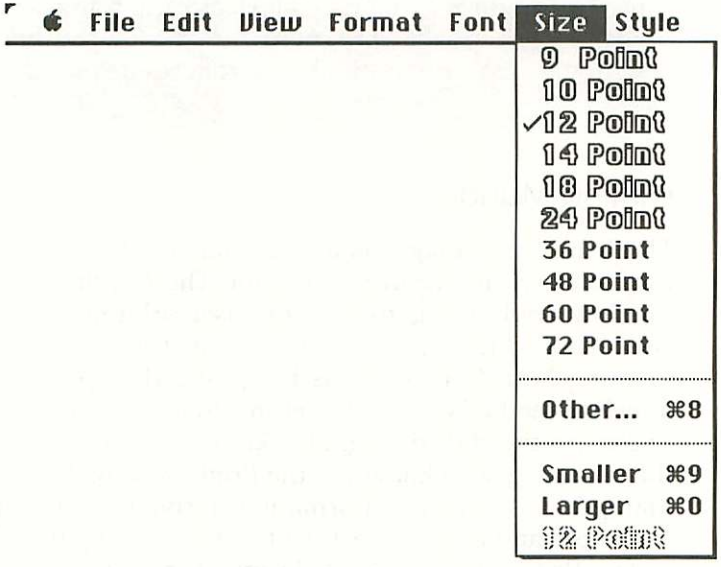

Figure 1-1. Font Size menu example

All other sizes of the given font are derived by scaling an appropriate bitmapped size to the desired size, as best as possible. In the LaserWriter, however, the fonts are accurate at all possible sizes and can be scaled to any desired size, including fractional sizes, without any loss of accuracy.

The second important difference between the screen fonts and the printer fonts is that the screen fonts are accessed by a number, but the fonts within the printer are accessed by name. This can be a source of some confusion and conflict between the two devices if the translation from the internal number to the external name is not done correctly. You will read about this in much more detail in Chapter 6, when we discuss font handling.

Note ▸

The preceding discussion is only true for versions of the system software before Version 7.0. With Version 7.0, Apple has begun using font outlines instead of bitmaps to generate the characters on the screen. This new mechanism, called TrueType, eliminates many of the complexities and confusions that were previously present in handling screen and printer fonts: For example, an application can now select fonts by name. This does raise some new issues, however, such as how to coordinate the new outline screen fonts with the PostScript outline fonts that reside on the printer.

▶ Printing Manager

The Printing Manager is at the heart of the conversion process that transfers screen output to a printer. The Printing Manager is designed to allow applications to offer the user printing services without much concern about the precise nature of the output device itself. The Printing Manager does this by giving the application two standard interface methods to use to get information about the printing process. These are the *style* dialog, also known as the *Page Setup...* dialog, and the *job* dialog, also known as the *Print...* dialog. By using these standard dialog boxes and the information derived from them, an application can create and control the exact form of printing that the user desires.

The Printing Manager is designed to bring printing services to an application in a manner that is independent of the specific output device the user selected. It does this by implementing a series of printing routines that provide functionally identical services to the application, regardless of the actual printing device that has been chosen. This is made possible by linking these service requests to actual printing code that is stored in a *printer resource file*. This file is stored in the System Folder on the user's disk, and it contains the device-specific code that is required to use the given output device. This *Printer driver* controls the communications between the Printing Manager and the actual device itself.

Important ▸

This book focuses on controls and printing techniques that are individual to a given device: the LaserWriter printer. Although some of these techniques are designed to be transparent to other devices if such devices are chosen as the output device, many of them rely on the LaserWriter's special qualities, in particular, that it understands and can process PostScript commands. Apple warns you repeatedly that use of such techniques can result in application code that may break when a new version of the printing software and new devices are available. This is a serious consideration, and you should not use these techniques lightly. On the other hand, many of the Macintosh flagship applications such as Aldus PageMaker, Adobe Illustrator, and Quark XPress also use these techniques, so you'll certainly not be alone. Therefore, I think, you can be reasonably certain that Apple is not likely to completely disable these techniques without some warning. Anyway, they're fun; and you'll sure know a lot more about Macintosh printing when you're done.

The user installs a specific output device by using the Chooser desk accessory to select one of the device drivers that have been loaded onto the system. When this is done, the new device driver is installed as the current printer resource file in the Printing Manager. This process happens asynchronously with any application processing (a fancy way of saying that it happens without anyone telling you!), and so your application cannot make any assumptions about the nature of the current device. Instead, you use the Print Default and PrValidate calls to determine information about the currently chosen printer.

You perform your printing in a special *printing grafPort*, which provides the same QuickDraw facilities that a standard grafPort does, with the addition of some features that make it suitable for printing. The Printing Manager creates a printing grafPort when you open a document for printing, and you draw into that grafPort using QuickDraw commands in the usual way, just as if you were drawing onto a screen grafPort. Obviously, one of the most efficient methods for converting the screen to a printed output is to use the QuickDraw picture construct to draw in both grafPorts; this, in fact, is the mechanism that you will generally use in this book.

▶ LaserWriter driver and Laser Prep

The printer driver for the LaserWriter is the LaserWriter file. This driver performs all the standard printing functions for the Printing Manager, as described earlier, including handling communications, transferring data, and handling responses from the LaserWriter. You normally access this driver by using the Printing Manager calls; however, in some instances, it is both useful and necessary to access the driver directly.

The driver provides the required translation from QuickDraw to PostScript. This is done by changing the QuickDraw information in the grafPort into PostScript commands. This process occurs in two steps. First, the QuickDraw commands are converted into the names of PostScript routines, or *procedures*, which produce the same effect on the LaserWriter that the QuickDraw commands produce on the screen. These PostScript procedures are stored in the Laser Prep file, which is stored in the System Folder and sent to the LaserWriter when you begin printing. The Laser Prep file contains all these translation procedures, called QuickDraw *bottlenecks*, which is certainly a suggestive, if not entirely flattering, epithet.

Note ▸

For System 7.0, the process is the same, but the Laser Prep file has shrunk within the System Folder. How is this possible? Easy. The former Laser Prep bottlenecks are now part of the LaserWriter driver software, so that there is only one file rather than two. Honestly, this doesn't make much difference to you unless you have been accessing the Laser Prep file directly, which is never necessary.

▶ AppleTalk

The LaserWriter driver communicates with the LaserWriter printer over an AppleTalk connection. As you probably already know, AppleTalk is the name of the communications network system used to connect a wide variety of computers and peripheral devices. The name "AppleTalk" refers to the network communications method and its associated architecture, collectively called a *protocol*. The AppleTalk protocol is implemented using specific hardware and on some form of data transmission medium. For most Macintosh computers and LaserWriter printers, this medium is LocalTalk. This book will generally use the term "AppleTalk" to mean both the network protocol and a

general communication medium that uses the AppleTalk protocol. The term "LocalTalk" will only be used if it is important to distinguish the specific transmission medium.

Note ▸

Since this is not a book about AppleTalk, you will just review here those components that are involved in communications with a LaserWriter. These features are important to anyone who wants to program specifically for the LaserWriter. Don't think that this is sufficient information to really understand or exploit AppleTalk. If you are going to do much programming that uses an AppleTalk connection, even if it's only for programming the LaserWriter, you should get and use the standard reference books on this subject: *AppleTalk Network System Overview* and *Inside AppleTalk*, both by Apple Computer and published by Addison-Wesley. The information presented here simply excerpts and recaps the more complete definitions that are presented in those volumes.

Fundamentally, there are two basic players in the printing process: the *workstation*, which is the device that produces the print job or request; and the *print server*, which is the device that processes the print job or request. For our purposes, we can assume that the workstation is a Macintosh computer, whereas the print server may be either a LaserWriter printer (or an output device that behaves like a LaserWriter, such as a typesetter) or another Macintosh computer that is being used as a network print server and spooler. You will read about these options in more detail in a moment; first, however, let's review the AppleTalk components of this system.

AppleTalk has a variety of layers that allow network configuration and control, along with data transmission. Figure 1-2 shows you a diagram of the AppleTalk layers that are involved in communications with the LaserWriter.

Let's briefly discuss each of the layers in Figure 1-2 in turn, from the top down. You have already met the PostScript language. In a very real sense, this is the communications method of the LaserWriter; it is how the LaserWriter identifies and processes information. As you will see in Chapter 2, PostScript is a good choice as a communications language since, among other virtues, it does not interfere with the communications process. The *Printer Access Protocol*, or PAP, is a specialized layer in the AppleTalk protocol that allows a workstation to communicate with a printer or print server. It provides connection setup, teardown, and

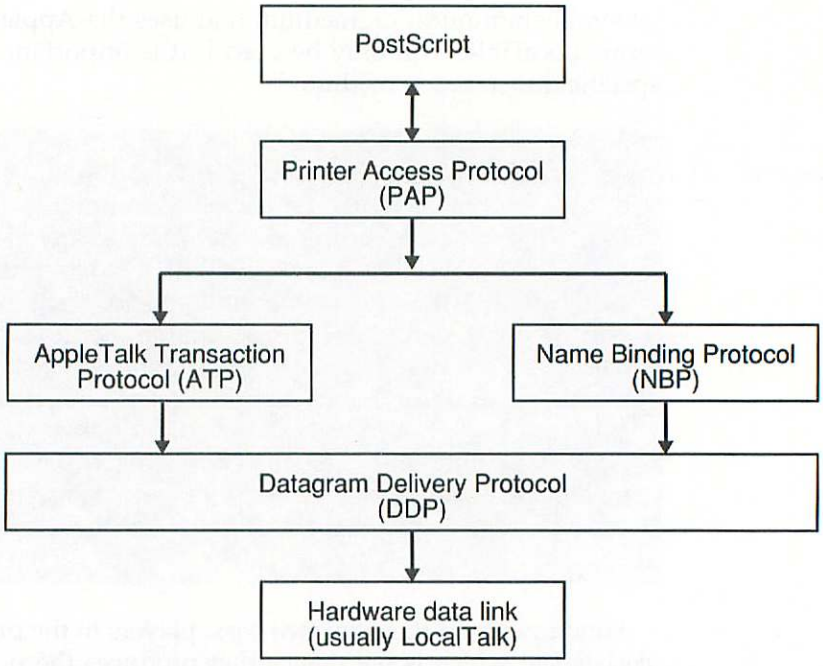

Figure 1-2. AppleTalk printer connection layers

maintenance operations along with the standard data transfer. PAP uses the functions provided by the NBP and ATP layers to find addresses and transmit data. From a programmer's viewpoint, PAP provides four important functions:

- Opening a connection
- Transferring data
- Closing a connection
- Providing print server status

PAP uses the NBP to determine the network address of a named print server.

Within this framework, you need to distinguish three methods of printing: foreground printing, background spooler printing, and remote spooler/server printing. Foreground printing applies when the workstation (the Macintosh) sends the print job directly to the printer, as shown in Figure 1-3. In this case, the LaserWriter becomes the print server.

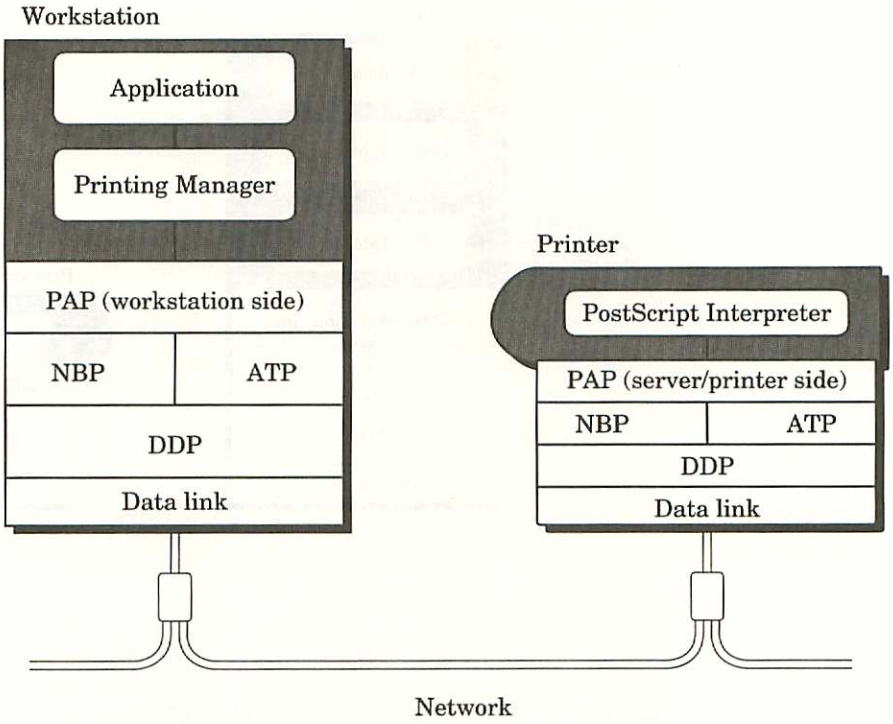

Figure 1-3. Foreground printing

In foreground printing, the application that produces the printed output is connected directly to the printer, and the workstation and the application keep processing the print job until the printer finishes receiving the entire job. Note, however, that this does not mean that the printer is finished processing the job; depending on the complexity of the job, it may be some appreciable time until the actual output appears on the printer.

Background spool printing applies when spooler software is loaded into the workstation, which is running system software that allows multiple tasks to process simultaneously. The MultiFinder application PrintMonitor is an example of such a background spooler. Figure 1-4 shows a schematic of this process.

In this case the application uses the Printing Manager to create the document, which is then passed to the LaserWriter driver. The LaserWriter driver then stores the print job on your disk in the Spool folder inside your System Folder. This frees up the application to return to its own processing without any more concern about the printing

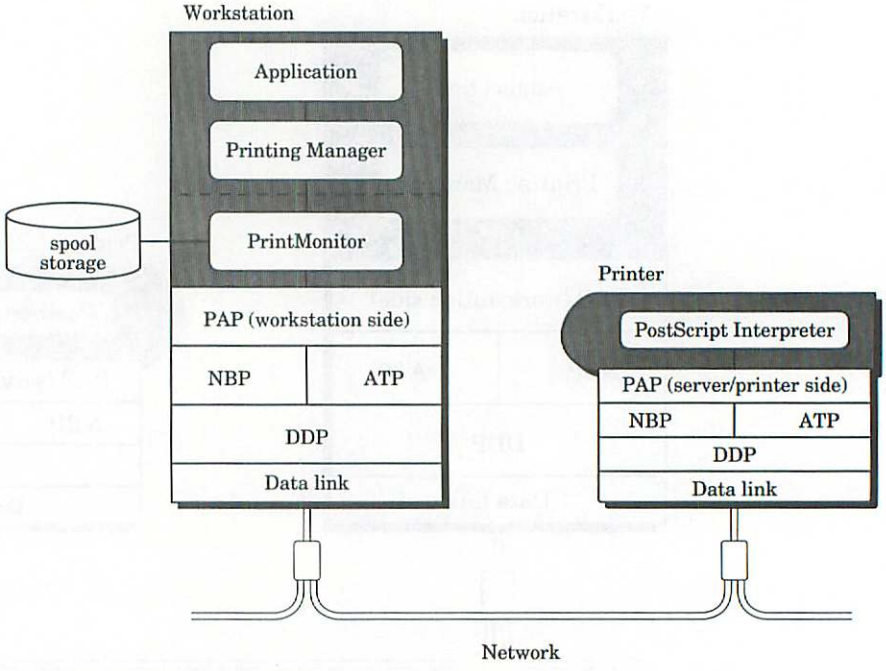

Figure 1-4. Background spooler printing

process. The PrintMonitor, which runs under MultiFinder as a background application, then takes over the communication with the LaserWriter, with the PrintMonitor providing the workstation side of the AppleTalk link and the LaserWriter providing the server side. Note that, although the application is free for more work, the workstation itself, which is running the PrintMonitor software, must remain connected to the network for the job to print completely.

Remote spool/server printing takes this process one step farther. A spooler/server is a dedicated workstation that provides common printer support (and possibly file support) to a group of devices. Figure 1-5 shows a schematic of this process.

Here the application prints normally, using AppleTalk protocols to direct the print job to the spooler/server in the same way it would if it were printing directly to the printer. The spooler/server takes the job, stores it, and then terminates the connection with the workstation that originated the job. This frees the workstation for other work and allows it to be shut off or otherwise disconnected from the network if desired.

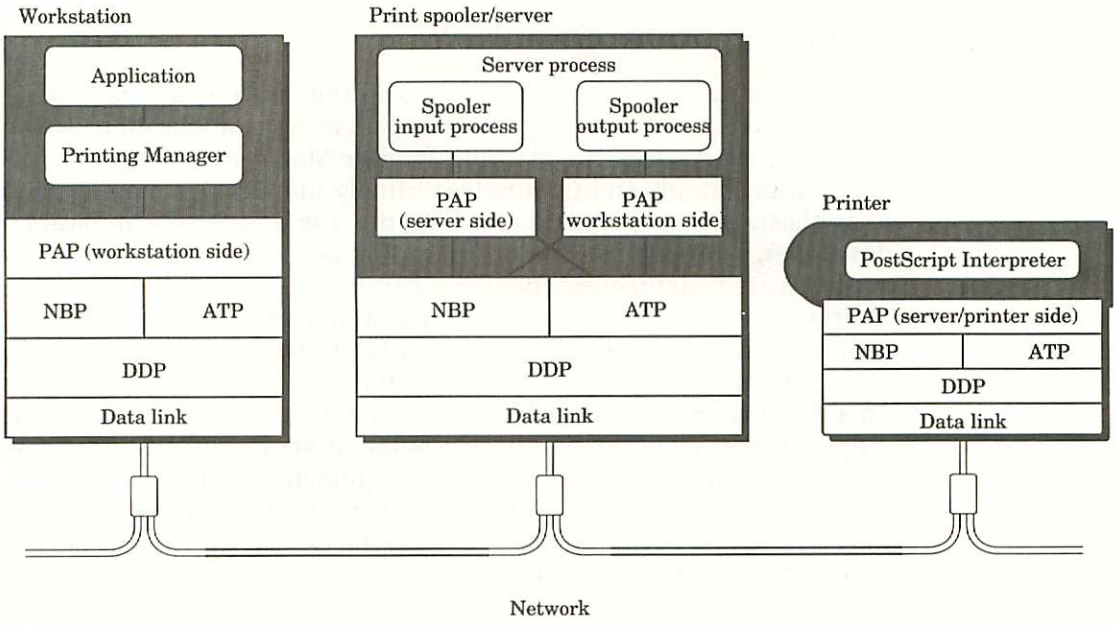

Figure 1-5. Remote spooler/server

The spooler/server takes charge of the print job and is responsible for getting it processed by an appropriate printer. Another important feature of the spooler/server is its ability to process several connections at the same time; the LaserWriter only allows a single connection, and therefore a single job, to be active at one time. The AppleShare print server is an example of spooler/server software.

Important ▸

Note that the spooler/server here plays both roles in the AppleTalk connection. It behaves to the LaserWriter as a workstation, but it appears to the application workstation as a server. This means that, from an AppleTalk viewpoint, the spooler/server looks and behaves just like a LaserWriter. For most printing tasks, this doesn't matter; however, if your application is managing the PAP connection directly, it can be a problem. Chapter 7 will teach you how to determine whether you are talking to a real LaserWriter or to a spooler/server.

▶ Printing Process Overview

Now that you have met all the players in this process, let's talk about how this works for an average user and an average application. As we all know, the process of printing from a Macintosh application is surprisingly simple and (naturally) virtually identical from application to application. This holds true for both the user and for the application developer, who can use the ordinary Toolbox calls to provide fairly sophisticated, device-independent printing. The standard printing interface, centered around the Printing Manager as described earlier, enables you to write an application without any further concern as to the exact nature or location of the output device, whether its location is on a network or directly connected, or whether the user is spooling the output or not. The user simply selects the desired output device using the Chooser and specifies the printing parameters. Then your application calls the Printing Manager, which supervises the print job, using the best method of printing according to the chosen device. The normal process follows the steps illustrated in Figure 1-6.

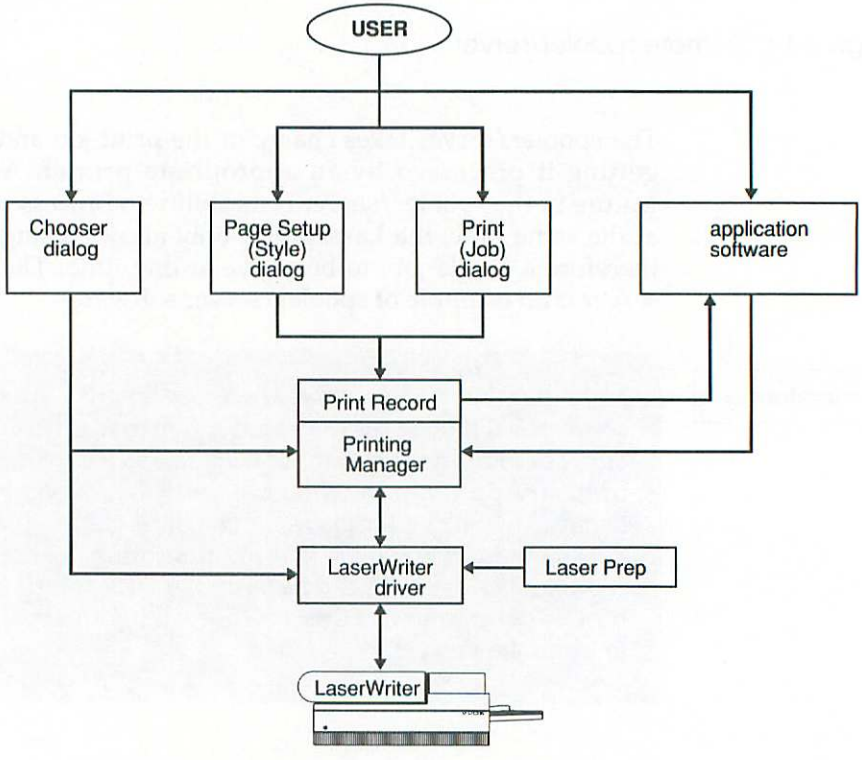

Figure 1-6. Normal printing process

▶ Chooser dialog

The normal printing process begins with the user selecting a printing device and a printing method using the Chooser desk accessory. The Chooser dialog box will look something like Figure 1-7.

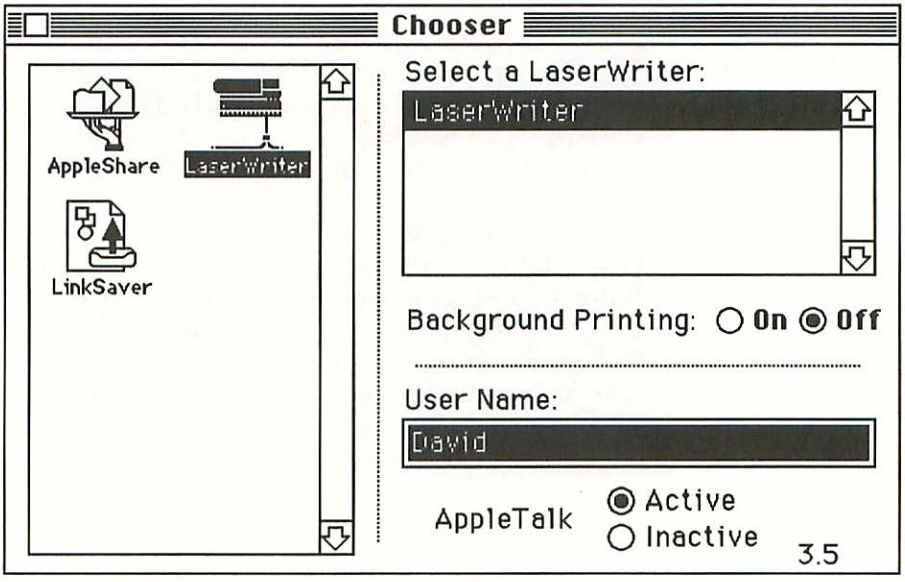

Figure 1-7. Chooser dialog box example

Note ▸

The following discussion presents several examples of the standard dialog boxes that the standard printing process on the Macintosh provides: the Chooser, the Page Setup..., and the Print... dialog boxes. The screens shown here are from a specific device and network configuration and represent System Version 6.0.4 and LaserWriter Version 6.0.1. Other system configurations, especially older or newer versions of the system and printer software, will have dialog boxes that differ from those shown here in content, format, or both.

This Chooser dialog box is quite straightforward and therefore needs minimal discussion. As you can see, it allows you to choose a LaserWriter or other output device. There are two points for you to note. First, you can control your AppleTalk connection at this level. If

you disconnect AppleTalk here, no printing takes place even if everything else is set correctly. Second, if you are running under MultiFinder, you can control whether or not the printing process is performed in the foreground or the background, as discussed earlier. This becomes important later in the book as you consider and test different types of interaction with the printer.

By the Way ▸

The Chooser dialog shows you all the available names of whatever printer you selected in the left panel on the dialog box. These are the AppleTalk names of the devices on the network that are of the given type. When you select a name, it is stored in the LaserWriter driver file. Similarly, the name of the workstation is entered into the User Name section of the Chooser dialog box as editable text. Generally, this is the name of the person who uses the workstation (mine, as you see in Figure 1-7, is "David"); however, it could be the location name or any other name that identifies this workstation. The Chooser stores this information in the System file. You will learn how to access and use both of these items of information in your application.

▶ Printing Manager

Once a device and the associated information is provided in the Chooser, the user can proceed to printing. This is generally done from within the application code, by selecting menu choices from the File menu. Two sets of information are necessary for printing. The first information group is provided by the Page Setup... dialog box, which is also called the *style* dialog box. This dialog box is generated by the Printing Manager when you call PrStlDialog from your application. Actually, this dialog box comes in two pieces. The first section is shown in Figure 1-8.

The choices shown in Figure 1-8 are the default page setup options for the LaserWriter. All of the information shown here is available to your application when you are ready to print, if you need it. For the average user, the most commonly used options here are the paper size and the page orientation.

In addition to the Page Setup dialog box, there is another dialog box that appears when you click on the Options button. This other dialog box is shown in Figure 1-9.

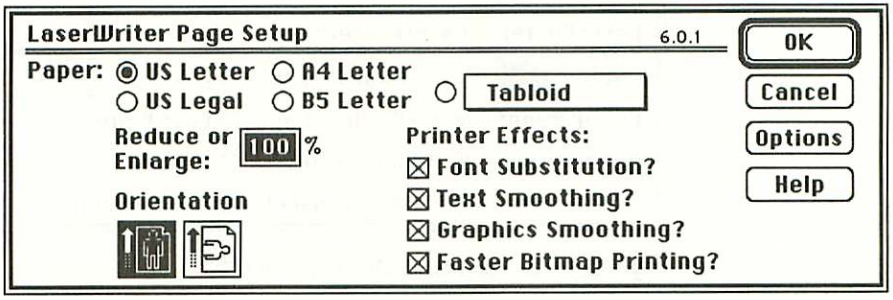

Figure 1-8. Page Setup basic dialog

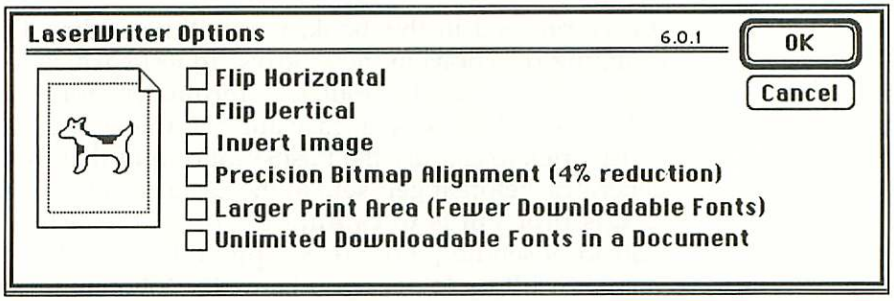

Figure 1-9. Options dialog from Page Setup

The Options dialog consists of a series of check boxes that allow the user to select a variety of options for their printing. You will see what these options do and exactly how they work in more detail as we proceed through the book.

Now the user must provide information for the second information group. After the user has made selections from the Page Setup dialog boxes (or has used the default selections that are provided automatically), he or she can print the document by choosing Print... from the File menu. This brings up a dialog box, which is also called the *job* dialog box, as shown in Figure 1-10. This dialog box is generated by the Printing Manager when you call PrJobDialog from your application.

The job dialog box collects information about this specific print job (as opposed to the style dialog, which collects general information about print jobs). The information provided in the style dialog is stored permanently in the Print record, while the information that is provided in the job dialog is stored in the printer resource file. This information is used during printing by the application and the driver, and the temporary data from the job dialog is then reset for the next printing request.

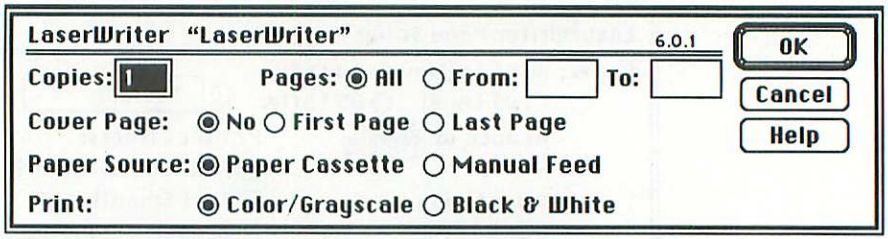

Figure 1-10. Print selection dialog box

Intercepting Printing Manager output

As we proceed in this book, you will be adding PostScript code and changing selections in these boxes. To test such code, it is essential that you be able to see the results. Unfortunately, for System software prior to Version 7.0, there is no easy and positive way to do this. Let's look at a little trick to capture the PostScript output that the LaserWriter driver generates, before it gets sent to the LaserWriter printer. The easy answer to this problem is to capture the output as a text file on your disk instead of sending it out over AppleTalk to the LaserWriter. This is not very difficult to do once you know the following trick. First, make sure that Background Printing is set to Off in the Chooser dialog box—this method doesn't work if Background Printing is enabled. Then prepare to print in the usual way, by performing the Page Setup that you want and then selecting Print. At this point you should have a dialog box something like the one shown in Figure 1-10.

Hold down both the Command key and the F key (don't use the Shift key—you need a lowercase letter *f*) as you click on the OK button. I usually hold these keys down before I actually click the button. This may cause the computer to complain audibly, but it won't affect the result. You should see a dialog box that looks like Figure 1-11 if all goes well.

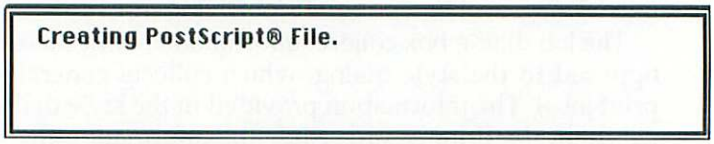

Figure 1-11. Creating PostScript dialog box

The dialog box tells you that, instead of sending the file to the printer, you have saved the document as a series of PostScript commands somewhere on your disk. The name of the file is PostScript*n*, where *n* is 0 to begin with. If there is already a PostScript0 in the folder, the number is incremented by 1, and goes up to 9. After 9, the names start over at 0 again, so you will write over the first file if you save more than 10 files without renaming them.

Note ▸

I use the words "somewhere on your disk" advisedly. There does not seem to be a simple rule for where the file gets put. I generally begin looking in the folder where the application itself is located—that's the application that was running when you gave the Print command. Even if the document that you were printing is located in some other folder, there is a good chance that the PostScript output is located with the application itself, and not with the document (unless, of course, they are both in the same folder). If the PostScript file isn't there, try the System Folder. If all else fails, use the Find File desk accessory to search out all files with the name PostScript.

This method saves just the document itself, without saving the Laser Prep file that goes with it. If you want to see the Laser Prep output as well, use the same process except press the K key instead of the F. This generates a longer file that includes all the Laser Prep bottleneck definitions.

The resulting file is a straight text file, and it can usually be opened and read with any word processing program or text editor that will read straight text. For example, the Think C editor can open the file. To do so, of course, you must be in a project; but, once you are, you can open these files at will. Apple Computer's TeachText editor can also open the file as long as the file is not larger than TeachText's 32K maximum. Unfortunately the most current version of the Laser Prep file is about 41K, so if you try to save a document plus the Laser Prep file, you won't be able to open it with TeachText.

Note ▸

This technique for capturing files only applies to System software up to Version 7.0. With Version 7.0, there is a new radio button selection in the job dialog box that allows you to specify output to a PostScript File instead of to the Printer. This is more obvious, more elegant, and more friendly than the method just outlined. Of course, every silver lining has a cloud—how will you show everybody that you're a power user now?

▶ Macintosh printing philosophy

At several points in this book, you will see warnings about how the use of the techniques described here make your application vulnerable to changes in how the Macintosh system software handles printing requests. This is not intended to scare you, nor is it intended to prevent or discourage you from using these methods. As mentioned in an earlier note, many of these techniques are used by major commercial applications, applications that represent the best and most distinctive work that is available on the Macintosh. It is, it seems to me, quite reasonable to assume that Apple isn't going to lightly or quickly make changes to the system software that break all these applications; and, if such changes do occur, there will be alternative methods provided to do the same things.

It is equally important to understand and appreciate the basic concepts behind the Macintosh printing philosophy. The Macintosh is, by design and continuous work, the computer for users who want to get work done, not for those who just want to learn about computers. For this reason, the Macintosh has always tried to make users' lives a little easier, so that they can do things without giving too much thought to the mechanics of the task. In the area of printing, users should be able, generally, to select Print from the file menu of any application and get what is on the screen printed onto whatever device is available without having to worry about having installed some special software or whatever. Thus, all applications come to look and act alike with regard to printing (and most other things, as well!), so that the new user doesn't have to relearn all the commands and quirks of an application in order to use it.

For the application developer, this means following the Macintosh printing process described previously. If you use the Printing Manager calls that are defined in *Inside Macintosh*, your application is entirely

device independent, and your application behaves, with regard to printing, exactly like every other Macintosh application. When you begin to apply the techniques in this book to your application, you are in a position to violate these comforting rituals that the user has come to both know and rely on. Don't do so needlessly. However clever you may be, both your sales manager and your users will thank you for adhering strictly to the spirit of the Macintosh human interface, even when you are doing the work yourself rather than letting the Printing Manager have all the fun.

Working with the standard Printing Manager calls has very definite and positive results. It allows your application to use a wide variety of printing devices, and it gives the user a common interface for all devices and across most applications, to everyone's benefit. By using a standard interface, you ensure that your product will be accepted on the market, that it will not become obsolete at the next release of the system software, and that your users will most likely be able to take advantage of new peripheral devices without any changes to the application. All are good reasons for sticking to the standard.

Unfortunately all these advantages come at a price, albeit a small one for most applications: loss of flexibility. Generally this is not a problem. However, when you need to modify or bypass parts of the printing process to make your application do what you want, these techniques are available. You can change or add to the standard style and job dialogs. For example, you can provide your own PostScript code to do things that the Printing Manager and QuickDraw cannot do, or you can manage the communications with the LaserWriter yourself to perform status queries or identify specific conditions. The important point is to make these fancy features as transparent to the user as possible.

The best way to ensure this transparency is to apply the techniques in this book in a structured way. In other words, you should use the technique that is closest to the standard Printing Manager calls and that still does the job that you need done. Thus, for example, if you can do everything through the standard calls, do it that way. If you must provide specialized code for the LaserWriter, use the device-independent calls first (we will discuss these in Chapter 3), and then use the device-specific methods. Manage your own communications only when that is required by your application; otherwise, let the Printing Manager do it for you. In other words, do as little work as possible. Be lazy! Believe it or not, your code will be better and more useful because of it.

Important ▸ There are very few occasions when you need to bypass the Printing Manager. When you do, you should do so in a structured way that preserves the look and feel of the standard Macintosh interface. Be aware, as you do so, that the resulting application depends on specific device features and software components. For this reason, such applications may not work with future Apple printing products or system software.

▶ LaserWriter Communications

The LaserWriter printer supports two types of communications and three operating modes. Although these are conceptually distinct, they should be discussed together since the modes influence the communication choices and vice versa.

▶ Communication modes

There are two specific communications modes for the LaserWriter family of printers: network communications using the AppleTalk protocol, and serial communications. Each of these types of communications is well described in the appropriate documentation, with the practical issues of ordinary setup and communications choices covered in the reference manual that comes with the individual device and with more technical detail and additional specific information in the *LaserWriter Reference*, published by Addison-Wesley. Particularly if you intend to use the serial communications to the LaserWriter, you should review these reference materials in some detail.

Two main points should be noted here about these communications choices. First, they are controlled by external switches on the device itself: by a rotary switch on the original LaserWriter and LaserWriter Plus, and by DIP switches on the LaserWriter II series. Second, they are mutually exclusive; that is, the LaserWriter can only accept information from one or the other of the communications channels. Generally, you must reset the device in order to switch from one channel to the other.

AppleTalk communications is the most straightforward of these modes. As is generally true of AppleTalk devices, you simply connect an AppleTalk connector kit to the device and it appears on the network, ready to go to work. Serial communications is a bit more complex. To begin with, you must make the correct selections for baud rate and the other transmission variables at both ends of the communications link. If these are not set correctly, communications will be garbled. You also

must have a cable with the correct connections or use a null modem to set the communications; without this, you will never see any response. In addition, you must determine what type of communications control your computer hardware and software use. All of these issues are covered in detail in the reference materials mentioned earlier.

By the Way ▸

The communications control method, or *protocol*, defines how two connected devices determine when to start and stop sending data. This is necessary since it is certainly possible for the receiving device (in this case, the LaserWriter printer controller) to get more data than it can process. The receiving device must have some way to stop the transmission process at the sending end so that no data is lost while it is working and cannot receive data.

Basically two mechanisms can be used for this process: hardware control or software control. Hardware control uses specific wires and pins on the interface plug to determine when transmission is acceptable. In this method, usually called DTR (for Data Terminal Ready) control, the receiving device uses a pin on the communications line as a sort of stoplight, to control the sending device transmissions. When the light is on, the transmissions can proceed; when the light is off, the transmissions must wait. Software control accomplishes the same task by setting aside special characters or flags that the receiving device can send back. When the sender sees one of these characters, it knows whether to stop or start sending, as specified by the character. Obviously this requires that both sender and receiver agree on what characters to use and what those characters mean.

For the early versions of the original LaserWriter, this wasn't a problem since the LaserWriter itself only supported the XON/XOFF protocol. This is one of the most common software protocols, but it is somewhat limited in flexibility because it requires that the software at the transmission end support this protocol. Later models of the original LaserWriter allowed you to select one of two specific communications protocols by setting parameters within the PostScript interpreter. These two were the XON/XOFF software protocol and the DTR hardware protocol. This was a big improvement because now you could use either hardware or software as a control. These same two protocols are also supported on the LaserWriter IINT. The IINTX provides more versatility because it has more switch settings; it supports XON/XOFF, DTR, and Etx/Ack protocols.

▶ Operating modes

The LaserWriter, like almost all PostScript devices, provides three distinct modes of operation: batch, emulation, and interactive. These modes define the way in which the PostScript interpreter handles and processes input. Each of these is useful and important, but they behave in quite distinct fashions. Let us, first, examine each of these modes in turn to see how they function and why you use them.

Batch mode

Batch mode is the standard operating mode of the device. In this mode, data is received as a continuous stream over a transmission path, usually using one of the communications channels described earlier. Data is processed until one of two things occurs: an end-of-transmission character is reached or there is a timeout on the communications channel. The end-of-transmission character is an ASCII character 0x04, which is generated by, and usually referred to as, Control-D. On standard keyboards, the Command (⌘) key, also called the Apple key, performs the function of the Control key in most applications. On the extended keyboards, there is an actual Control key. The amount of time that the device waits before assuming that the transmission has failed is determined by settings within the PostScript interpreter. In later chapters, you will see how to determine and set these values.

Errors that occur during the processing of the PostScript file are handled in a standard default manner. The nature of the error is reported back to the workstation over the same communication channel that the job was received on, using a standard error message format, and the remainder of the job, to the end-of-transmission character, is flushed. Later chapters discuss the format of the standard error messages, how they are handled by your application, and how you can alter the standard processing, if you want.

Emulation mode

Emulation mode is a special variant of batch mode. As in batch mode, data is received as a continuous stream over one of the communications channels of the device. This data is not treated as PostScript, however. Instead, the interpreter emulates some other output device and processes the data in the same manner that the emulated device would. Two common emulation modes are used in the LaserWriter family of

printers: Diablo 630 and Hewlett-Packard LaserJet Plus. Both modes are well documented in the reference materials for the LaserWriter family and thus are not discussed any further here.

Note ▸

A major drawback and difficulty in using the emulation modes of the LaserWriter is that there is no method to determine when a job has completed printing. This happens because the emulation modes require that all possible characters be transmitted to the printer; this means that the Control-D termination character is not recognized as a control character in emulation mode. The LaserWriter simply waits until timeout and then assumes that the job is complete. At that point, it resets itself to the initial job state. In this mode, it is important that you understand the printing and job cycles; your documentation will tell you how these work.

Interactive mode

The last mode available is interactive mode. In this case, the interpreter sets itself to receive data over a bidirectional channel, and essentially it carries on a "conversation" with the programmer. Data is received line by line over one of the communications channels and is processed immediately. Any errors are reported back to the host device as they are discovered, and the interpreter waits for another line of code. Data is processed until a specific **quit** operator is processed, which terminates the bidirectional connection and the interactive mode. Obviously, there is no timeout.

This mode is particularly useful for developing or debugging PostScript code because it allows immediate feedback and correction of errors. There are, however, some serious drawbacks. Most notably, you tie up the LaserWriter device, and no one else can access or use it while it is in interactive mode. Also, running in interactive mode over AppleTalk requires a program interface; otherwise, you must connect to the LaserWriter over a serial port and use terminal emulation software for processing. This process, using the serial connection, is well described in the *LaserWriter Reference* and is also presented in the reference manuals for the devices. For example, there is a tutorial describing an interactive session in the *LaserWriter IINT/IINTX Owner's Guide*; this describes a session using MS-DOS, but the basic information is the same for a Macintosh.

For anyone doing serious PostScript programming, the best method is to get Lasertalk, which provides interactive access to the LaserWriter over AppleTalk and also provides a complete programming environment, including online reference materials, structured access to the printer, full editing and debugging facilities, and much more. The Lasertalk program is distributed as a commercial product by Adobe Systems, Inc., the creators of the PostScript language.

This discussion assumes that you are connected to your LaserWriter by AppleTalk and are using the Printing Manager facilities. Thus it is possible for you to create arbitrary PostScript code and download it to the LaserWriter for execution quite easily. (You will learn just how to do this in Chapter 3.) This eliminates the need for using any special software or changing the cable connections to the LaserWriter, and it will fit in nicely with the network environment, so that any of your co-workers can share the printer with you just as they would in the ordinary course of a day.

▶ Mode selection

Basically, you can change the communications parameters of the LaserWriter in two ways: by using software or hardware selection. Since these two methods interact to some extent, you generally must make a hardware choice, which you can then modify by using the software modes. Let's look at some of these options.

On the hardware side, communications options are controlled by a switch or switches on the LaserWriter. Setting these switches sets the communications to a specific state. Typical choices are AppleTalk communications, 1200-baud serial communications, 9600-baud serial communications, and the emulation mode. Each specific LaserWriter has its own method of setting the appropriate switches, which is documented in the reference manual for that device. Since this is a programming book, switches will not be discussed any further here. The important point for you to understand is that the physical setting of the switch or switches controls to some degree how the printer responds to communications.

This raises the other half of the equation, the software controls for these communications channels. Basically, by using the software settings, you can interrogate the interpreter regarding all the current communications parameters and you can reset most of them. For now, you just need to know that there are software controls for the communications mechanism and that you can report the current settings and change these parameters using a PostScript program.

▶ Conclusion

This ends your short review of the Macintosh printing process. You have covered quite a bit of material in a short space, touching only lightly on many of the interesting and important features of Macintosh printing, always with a view toward the LaserWriter as an output device. The apology for such a quick and, in many cases, superficial review is twofold. First, certainly some of this material is familiar to you, perhaps even daily bread-and-butter. Second, any material that is not completely familiar is well documented and accessible in the reference materials that are discussed here and that are listed in the Bibliography. This review is intended to recall some of the most important points so that you will be ready for the more detailed material in the following chapters, and to make it unnecessary for you to go rooting through all the references to refresh yourself on the general process and techniques of printing. With this outline fresh in your mind, you can proceed to a more in-depth discussion of using the LaserWriter. The next chapter introduces the PostScript language, which is the most important and distinguishing feature of LaserWriter programming.

2 ▶ PostScript Language Concepts

▶ Chapter Overview

As you saw in Chapter 1, the PostScript language is an important component of the LaserWriter. The Macintosh and the LaserWriter began the desktop publishing revolution, and they paved the way for the proliferation of software and hardware that is now available to create and manipulate documents from the desktop. This was not simply a function of the high quality of LaserWriter output. The major difference between the LaserWriter and other similar output devices is its capacity to understand the PostScript language. LaserWriter output has a number of outstanding features that are of substantial benefit to its users and that are directly derived from the PostScript language.

To program the LaserWriter, therefore, you must learn something about the PostScript language. This chapter and the following one are intended to give you a solid overview of the PostScript language. You can't learn PostScript programming in such a short space, of course, but you can get enough information to be able to read simple PostScript programs and follow the limited PostScript code that is necessary to the tasks that are covered in this book, for example, determining device status, listing fonts available, identifying dictionary versions, and so on.

This chapter introduces you to the few major topics that form the basic concepts of the PostScript language. It begins with an overview of the vocabulary that you need to discuss the PostScript language, explains the language's important features, and shows you how those features have made the LaserWriter such an important link in desktop

publishing. This section also discusses how a PostScript device such as a LaserWriter goes about creating its output. The section on device management is very important for understanding where the LaserWriter fits into the spectrum of PostScript output devices and also for trouble-shooting many typical types of problems that may occur when you are writing code specifically for the LaserWriter.

After this overview, the chapter covers the structure of PostScript output and how a PostScript page is put together. This is of particular importance to the Macintosh programmer who wishes to program the LaserWriter because the assumptions, coordinate structure, and imaging model of PostScript are different from those of QuickDraw in subtle but important ways. To help you understand PostScript programs, the next topic is PostScript objects and how they are described and used. This leads to a discussion of the fundamental PostScript operating structures: stacks and dictionaries. The final topic in the chapter is a brief discussion of PostScript fonts and their use in programs. Altogether, these topics should give you a good basic understanding of the PostScript language and how it fits into the LaserWriter printing process.

▶ Language Characteristics

PostScript has some important characteristics that have made it the *de facto* standard for desktop and advanced publishing tasks. As you know, the basic printing engine in the LaserWriter is generally no different from that in other laser output devices; it is the additional features and functionality that PostScript provides that distinguish the LaserWriter from its competition.

▶ Important features

The PostScript language has certain characteristics that make it both distinctive and superior for high-quality output that combines text and graphics. PostScript is

- page oriented
- interpreted
- dynamic
- encoded
- device independent
- graphically powerful

When you appreciate these features, you will understand both why PostScript is important to the user and how you can use it enhance your output.

Page oriented

PostScript is often called a "page description" language. This means that it is designed to create units of output, which we ordinarily think of as "pages." In other words, PostScript is oriented toward making pages. This is quite different from the traditional computer output. Originally computers produced listings of data, programs, or internal information. In any case, these outputs were generally conceived and produced as continuous lists, and any division into page units was completely arbitrary. Consider how different this is from the approach of a graphic artist or a designer. When such people conceive of a page of output such as an advertisement or a magazine layout, they think of it as a unit. They lay out the page so that all the elements flow together harmoniously with the correct visual impact on the reader or viewer. This is not possible with traditional computer output because the output is created and printed as (at best) lines of information rather than as pages. PostScript, on the other hand, is designed to create units of output and thus allows you to make design decisions based on page layouts.

Important ▸

Although these units are normally described as "pages," the actual units need not be physical pages. For this purpose, pages are only a convenient shorthand for any output unit; this may be as much as a large sheet of output or as little as a single graphic element. The output may be on a printer such as a LaserWriter, but it may equally well be onto a continuous roll of paper or even film. It can also be a display screen or some other output device. For both the programmer and the user, the important point is that a page represents a simple and easily handled output unit that is independent of any specific device or output medium. From this point on I will continue to use "page," but you will understand that I mean the unit of output, not just a physical page of any particular dimensions or medium.

A PostScript page may contain any type of output, either text or graphics. This is another major break with what was, until the LaserWriter, the traditional manner of creating output. Until recently,

most computer printers, and even many computer displays, could not produce text and graphics on the same page. Since pages that have the most information and the greatest impact generally contain both text and graphics, this was a great drawback. PostScript, however, allows you to combine them both without any limitations and with no loss of quality.

The PostScript language can produce both text and graphics on a single page because it does not distinguish text from any other type of graphic. In a very real sense, the shapes of letters are just another form of complex graphic. PostScript has special features that allow you to use text as you would any other graphic element, while still providing special processing to enable letter shapes to be displayed quickly and with great clarity. PostScript provides for all types of general graphics as well. Using standard PostScript commands, you can create lines and shapes of all types. These can be either stroked or filled, as required. PostScript also provides for the creation and rendering of general images created out of bitmaps, which may be either scanned in or generated synthetically.

Interpreted

PostScript is an interpreted language, that is, PostScript language elements are understood and acted on by another program, the *interpreter*. The interpreter takes these elements and creates the desired output on the given device. In a LaserWriter, the interpreter is inside the printer, as was discussed in Chapter 1. In general, the interpreter may reside in a separate device, as it does in the LaserWriter, or it may share the resources and processing facilities of the host computer. In some devices, the interpreter is a separate unit that can be connected to the basic output device. A variation of this principle allows you to upgrade the LaserWriter IISC to an NT or NTX by providing a new processing board.

An interpreted language provides several benefits, with one drawback. The benefits are that the interpreter is flexible and sensitive to the context of the current operation. It can provide immediate feedback on the state of the device and the output; it can also provide sophisticated error handling and reporting. All these features allow an interpreted language to take full advantage of the output device, however complex or unusual it may be.

The drawback, however, is that the interpreter represents an additional layer of software that needs to be run in order to produce the output. With traditional output methods, when the application software has completed its processing, the job is done. All that remains is to send

the output down a communications channel to the designated device, which then can follow the relatively simple commands to produce the output. These commands are, of course, device-specific and do not provide the marvelous output quality or features that are available through PostScript; however, they also don't require any additional interpretation when they reach the output device.

Moving the interpreter and its associated processing to the device itself, as is done in the LaserWriter, minimizes or eliminates some of these overhead issues. The interpreter does not consume any processing resources in your Macintosh or other computer, nor do the output file and any intermediate results have to be accommodated with file space on your storage media. This also makes it possible to control and speed up the processing by adding more resources to the interpreter itself; the change from the LaserWriter to the LaserWriter NT is a good example. Further increases in output speed may be available in the future, as the interpreter is given faster internal processors or is adapted to special-purpose processing chips. For this reason, the benefits of having and using an interpreted language are great relative to the small overhead involved.

Finally, for the programmer, an interpreted language has one more area of concern. Interpreted languages, by their nature, generally do not have a strong structural component, and PostScript is quite typical in that it provides almost no structure that is internally imposed by the language itself. Instead, the PostScript programmer must define and adhere to standards that come from a personal discipline. The PostScript language has certain conventions and coding standards that are encouraged and even essential if the resulting output is to be included in other pages or used by other applications. Nevertheless, you must realize that PostScript, of itself, does not enforce any programming standards. This is only a drawback, however, if the programmer does not provide the necessary discipline to establish and follow clear coding practices.

Dynamic

A particular benefit of an interpreted language is that it is dynamic. For example, the PostScript language provides feedback on the current state of the device—which fonts are available and how the output page is oriented. It also allows graphics to be modified as they are output based on internal data or other information that may not have been available at the time the graphic was originally created. This type of dynamic structure means that you can define new operations from the basic set

of graphic operations and from the control and program operations that are available through the interpreter. This process makes PostScript page descriptions much more flexible and significantly more powerful than the traditional escape codes or other page coding methods.

PostScript output is converted before being sent to the interpreter, which means that all the elements of the language are converted into units before processing. This conversion is called *encoding*. A variety of methods can be used to encode any computer language, and PostScript supports several of them, depending on the environment and the requirements of the output device. The standard, and most common, method of encoding PostScript is by using ASCII text strings; this is the only encoding method that the LaserWriter supports. These text strings require only the printable subset of the ASCII codes (ASCII codes 32 through 127).

This type of encoding has a number of advantages. It is transparent to networks and still allows a full range of values to be transmitted and operated upon. The files can be easily created and read by standard editing software, and the resulting files are readable by both computers and humans. The only disadvantage to ASCII encoding is that ASCII files are quite bulky in comparison to other encodings of the same data, and thus they require more space to store and more time to transmit. Where this is a consideration, some PostScript devices support two forms of binary encoding: binary object encoding and binary token encoding.

Device independent

PostScript output is, by design, independent of the specific device that was used to create it. This independence is one of PostScript's great advantages. It allows you to create both text and graphic output, which can be proofed on one device, such as a LaserWriter, and then produced in the final form on a high-resolution device, such as an imagesetter. Device independence means that the user doesn't have to be concerned about the type of output device or the resolution of that device; PostScript output takes the maximum advantage of the device features without any significant user intervention. You get the best output that the device is capable of creating, without any special requirements. This allows the user to take a file directly from the LaserWriter to an imagesetter and create output that maintains the graphic components of the proof output but uses the higher resolution of the new output device.

PostScript output also provides portability between platforms. Because PostScript files are generally encoded as ASCII text, they can

be easily and quickly moved from one computer or operating system to another, and they will produce identical output in the new environment. The interpreted nature of PostScript also allows devices to simulate features that are not present on the specific device, or it allows some substitution to take place. Finally, although PostScript is designed to be device independent, it does provide device-specific operations and controls if required or desired. The only drawback to using these, of course, is that your output may become tied to the specific device.

Graphically powerful

The PostScript language is graphically powerful. PostScript is a complete programming language that also provides powerful graphics primitive operators. Over 30 percent of PostScript operators deal directly with graphics operations; and most of these operations are intuitively similar to the natural actions of an artist drawing a picture—selecting a pen, positioning it, inking it, and then stroking an image onto the output surface. This is a marked difference from general-purpose languages such as C or Pascal, where graphics operations must be created as procedures within the language. In this sense, PostScript is something like Pascal plus QuickDraw, in that it has within it many graphics operations that produce the same results that would be handled on the screen by QuickDraw calls. These graphics operators function naturally and intuitively. The operators provide many basic graphics tools as primitives, and they allow you to combine text and graphics as a single unit.

▶ PostScript operators

PostScript is a language of operators. Standard PostScript provides more than 270 primitive operators, which cover both ordinary programming tasks and graphics operations. Before you begin reading or modifying PostScript code, you need to acquaint yourself with a basic subset of these operators and become familiar with the structure of PostScript programs—or, as they are normally called, page descriptions.

This extensive set of operators, which covers both ordinary programming tasks and graphic operations, is one of the strengths of the PostScript language. The library of PostScript operations can be conceptually divided into five major groups.

Stack and mathematical operators

This group of operators provides stack handling, array and dictionary processing, and all mathematical operations for the language.

Graphics operators

This group of operators consists of all the operators that produce or manipulate graphics, except those that relate to font definition and handling. This includes operators to handle path construction and painting and also to handle PostScript's internal graphics memory.

Font operators

This group includes all operators that define and control fonts. It also contains operators that provide text processing. These are distinguished from the other graphics operators because they have facilities and rules that allow specialized processing to speed up and improve character handling.

Program control operators

This group includes operators to handle relation tests of various types and to alter and control the sequence of execution within the PostScript program. It also includes operators that support file processing. Error handling is provided by operators in this group as well.

Device control and status operators

This group contains all the operators that set or report the current state of the output device being used; this includes controls for selection of paper, for example, and for setting and reporting job statistics and other job information. The exact nature of these operators and their functions naturally depend on the features that are available on a specific device.

Together, these operators provide all the controls and features of a complete programming language, together with the graphics operations that you have already reviewed. This represents a rich set of possible operations; and each of these has a specific and useful place in the overall PostScript lexicon. However, as is so often true in natural languages, 90 percent of your work can generally be accomplished with 15 percent of these operators. Thus you don't need to learn as large a number of operators as you might expect in order to program the LaserWriter effectively.

▶ PostScript objects

The PostScript interpreter acts on a series of entities called *objects*. This is a convenient name to describe all the different things that the PostScript interpreter can act on and understand. Sometimes these PostScript objects are referred to as *tokens*, and the process of turning a stream of characters into PostScript objects is called *tokenizing*. The two terms "object" and "token" mean the same thing, and you may find either one used in various reference materials. Here we will exclusively use the word "object" for all these types of PostScript entities.

For our purposes, these objects can be grouped into three intuitive categories:

Literals	As you would anticipate from their name, these are numbers, characters, strings, arrays, and so forth.
Names	Labels for other objects—entries into various dictionaries, as you will discover in the next chapter.
Procedures	Groups of other objects, which can be treated as units and stored in PostScript memory. Like procedures in C or Pascal, these procedures are a series of program commands that are meant to be executed in a precise sequence.

Since PostScript programs generate graphics as objects, the elements of the graphics are stored and manipulated as mathematical descriptions of the desired shape. This process is applied to both text and graphics.

▶ Device management

The PostScript language is designed to work on a broad, but specific, class of output devices called *raster-output* devices. Raster-output devices work by making pictures—both text and graphics—out of a series of dots. In general, these dots may be fixed or they may vary in size; they may vary in intensity and color; and, although they are commonly called "dots," they may be of any convenient shape: circular, oval, or square.

As you might expect, the LaserWriter is one such device. In fact, all laser printers, as well as other devices from dot matrix printers to display screens, use a similar process to create images. Television, which was the first raster-output device, uses electron beams to turn dots of phosphor on and off to make an image. Any method of creating

dots works equally well; on the LaserWriter, the dots are created by a laser beam and toner. On some devices, the dots themselves can be adjusted to create shading; on other devices, for example, the LaserWriter, the dots are only present or absent and must be grouped to create the effects of shading.

Using dots to form images has, not surprisingly, both advantages and disadvantages. The advantage is that the dots can be composed into a representation of any image, no matter how complex. The disadvantage is that, since fundamentally a raster-output device has no continuous lines, all objects must be ragged and uneven at the finest level. Fortunately the human eye compensates for this to some degree—when the dots are small enough and close enough together, the image appears smooth and solid.

Device resolution

Each dot on a raster-output device represents one picture element, or *pixel*. Devices are often classified by the maximum number of pixels in a unit of area. This classification is called *device resolution*. The more pixels that there are in a unit of area, the closer together the pixels are and the better the image produced by that device. Devices with a high resolution, therefore, provide a finer and higher quality output than lower resolution devices.

Resolution is measured by the number of pixels per unit area. Resolution may be uniform in both the horizontal and vertical directions; in such cases, the resolution is usually given as *dots per inch*, or *dpi*. The LaserWriter has a resolution of 300 dpi, whereas the Macintosh screen display has a resolution of 72 dpi. Some devices, however, have different resolutions in the horizontal and vertical directions; in such cases, the resolution in each direction must be specified separately, for example, 240 by 400 dpi.

Devices are generally divided into groups according to their resolution. This is not a precise division, but simply a matter of convenience when designing and referring to the devices. Low-resolution devices range between 50 and 150 dpi; screen displays and dot matrix printers usually fall into this category. Medium-resolution devices range between 240 and 600 dpi; laser printers and ink-jet printers are in this category. Particularly at the upper end of this range, it becomes quite difficult for the eye to distinguish output quality unaided. High-resolution devices such as film output devices or imagesetters typically range between 1000 and 2800 dpi. This is the area of typeset quality output, and this is generally the highest quality that most applications

require. For some specialty requirements, however, there are devices that provide even higher resolutions. PostScript page descriptions are capable of being transferred from the lowest to the highest resolutions without further processing; this is one of the features, mentioned earlier, that makes PostScript so valuable.

Scan conversion

Even on devices with a very high resolution, the issue remains of how to mark the boundary of an object. The process of defining which pixels should be turned on and which pixels should be turned off is both important and complex; this process is called *scan conversion*. The PostScript interpreter performs the scan conversion for all PostScript objects by converting the mathematical descriptions of the object into a set of pixels on the output page. This is probably the task that consumes most of the processing resources in the interpreter.

Scan conversion for graphic objects essentially consists of determining which pixels lie within the shape and which lie outside of it. Since it would most unusual if the edge of an object lay exactly along the line of pixels, the scan conversion process must somehow decide which pixels must be activated and, for devices where the pixels can be varied, what values the pixels must have, in order to best render the desired object. This process is exactly the same, no matter what the nature of the object; the same process is used for circles, rectangles, text characters, or even more complex objects. In addition, some technique must be used to adjust the pixel setting when fractional pixels are required; this process is called *tuning*. Letters are both the most common graphics on a typical page and the most familiar to our eye; for this reason, tuning is especially important for letter shapes.

Processing loop

The PostScript interpreter manages jobs and the output device by executing a continual loop, called the *server loop*. The server loop performs several tasks:

1. Checks the communication channel or channels for incoming jobs. Generally, the choice of communication channel is made by a switch or panel control on the device, but it may also be made under software control.
2. Initializes the state of the device before each job.

3. Provides an execution context for each job.
4. Cleans up after each job. This process discards all objects created by the job, erases any marks left on the device output, and recovers resources, particularly memory resources, used by the job.
5. Processes any exceptions or faults that may occur during processing.

Some processes want or need to leave information behind them for later use. Since the server loop removes all job-related information, it is necessary to leave, or *exit*, the server loop to do this. When a process exits the server loop, it leaves any procedures and values that it creates behind after the job terminates. These objects consume virtual memory until the output device is powered off or reset. This allows a job to provide further information for subsequent jobs. This is the process that is used to place the Laser Prep file into the LaserWriter. However, this process should be used with caution. Exiting the server loop also allows a program to modify permanent device parameters, and it even allows modification of the interpreter itself.

▶ Page Structure

Each PostScript program begins work on a simple, two-dimensional space that represents a standard output unit, or page. Images are built by placing "marks" in selected areas of the page. These marks are made visible by applying "paint" to them, and they may represent characters, filled or outlined shapes, or halftone representations of images. The marks may be colored or black and white, and they may be shaded to any degree. The actual output, of course, depends on the specific characteristics of the device used. However, one of the features of the PostScript interpreter is that it converts images into representations that can be used on the given output device; thus, for example, figures that are placed on the output page in color will be rendered in shades of gray on a black and white output device. This is part of the device independence that is inherent in the PostScript language.

These marks are always opaque, so that the last element painted onto the output page overlays and obscures any marks underneath it. Any element on the page may be rotated, cropped, or otherwise transformed as it is placed onto the page. Then, when the page has reached the desired state, it can be rendered onto a physical output medium.

In concept, this all follows a natural model of a graphic artist working with a pen or brush on a piece of paper. Even though the "page" is entirely electronic and the "marks" are binary numbers, the underlying

model remains the same. In PostScript, the object to be rendered is defined by a mathematical path. This path is then "painted" by the appropriate operators, turning on a set of pixels that render the path visible on the output page. This is the most notable difference between PostScript and the natural model of the working artist. Even this has its analogy, as artists often use a light pencil to trace a path before inking or painting it. The important point here is that these operations form a natural set and are both easy to visualize and use.

Operators that apply paint onto the output page are in four basic categories, represented here by the most fundamental PostScript operator in each category. Each of these operators takes some information and renders the results onto the output page.

Operator	*Function*
fill	Fills an area on the page
stroke	Marks lines on the page
image	Renders bitmaps onto the page
show	Paints character shapes onto the page

▶ Output structure

Both the paths that are created and the marks that render the paths visible are applied onto an ideal output space called the *current page*. When PostScript begins, the current page is blank, that is, free from all marks or paths. Paint is applied to the current page, by the painting operations described earlier, along the *current path*. The path controls where on the page the paint is placed. The path may consist of straight or curved line segments or individual points. The path may be made up of multiple subpaths. Each subpath is independent of any other subpaths and each may consist of a single point, a segment, or many segments joined together. The last point added to the current path is called the *current point*.

The current path is built on the page by PostScript path construction operators. Typical path construction operators are as follows.

Operator	*Function*
newpath	Starts a new path and erases any old path elements.
moveto	Begins a new subpath by moving to a given point and makes that the current point.
lineto	Adds a straight-line segment to the current path, extending from the current point to a given point. The end of the segment becomes the new current point.

Remember that these operators and all path construction operators do not make any marks on the page. They only create the path that is later marked by one or another of the painting operators. This means that drawing in PostScript is a two-step process: First, a path is constructed, and then it is painted. Here is a simple example, using standard PostScript coordinates:

```
72 72 moveto
432 504 lineto
```

This creates a path from the point (72, 72) to the point (432, 504), as illustrated by the dashed gray line in Figure 2-1.

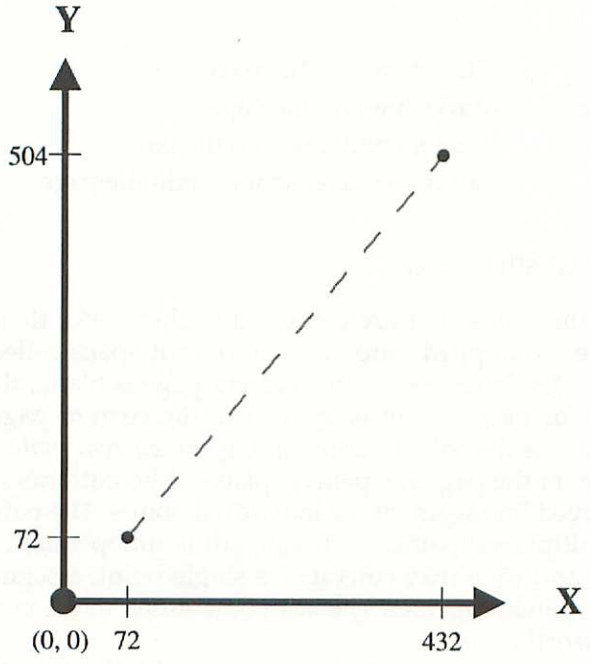

Figure 2-1. Path construction

To make this line visible, issue a painting operator:

```
stroke
```

Figure 2-2 shows the result.

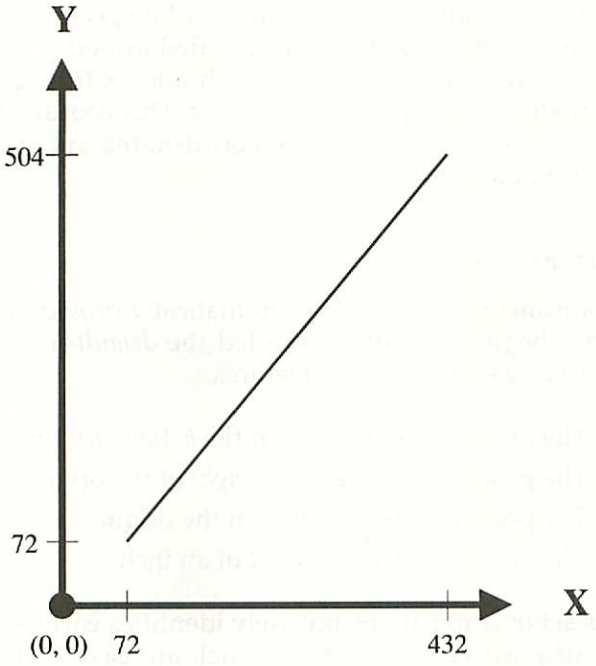

Figure 2-2. Path painted

Notice that this is different from QuickDraw, where a LineTo, for example, both moves the pen and draws the line.

The painting operators, such as **fill** and **stroke**, "use up" the current path as they mark it. After you have painted a path, the current path is empty and the current point is undefined. If you wish to retain the path, you must take special steps to preserve it before you paint it.

▶ Measurement and coordinates

All PostScript objects—text, graphics, and images—are controlled by a rectilinear coordinate system that is independent of the device and its resolution. The paths and points that make up PostScript graphics are defined in this coordinate system. This coordinate system uses a pair of independent coordinate axes, at right angles to one another. By convention, the horizontal axis is called the *x-axis* and the vertical axis is called the *y-axis*. In this way, any point on the page can be determined by two numbers, *(x, y)*, which give its position from the *origin*, the point

(0, 0). In the same way, any path can be specified as a mathematical equation, and any point can be specified as being inside or outside the path according to some rule, which allows filling the interior of an object with a given gray value or color. This coordinate system is called the *user space*. All user space coordinates are represented as real (floating-point) numbers.

Default user space

The coordinate system that is automatically provided when a PostScript program begins execution is called the *default user space*. The default user space has the following features.

- The origin, point (0, 0), is at the *bottom, left corner* of the page.
- The positive *x*-axis is to the *right* of the origin.
- The positive *y*-axis is *up* from the origin.
- The unit of measure is 1/72 of an inch.

This set of conventions precisely identifies each point on the page by a pair of positive coordinates, which are easy and natural to use for both printing and graphics composition.

The choice of 1/72 of an inch may seem to mimic the resolution of the Macintosh screen. However, it is also almost exactly a printer's point (which is fractionally larger than 1/72 inch), and thus it makes the coordinates on the PostScript page correspond to point measures in the ordinary printing process. This is a great convenience for laying out pages. Figure 2-3 illustrates the resulting coordinate system.

Note that this entire structure is quite different than QuickDraw coordinates. In particular, you must get used to the fact that the positive *y* coordinates are positive up the page rather than down, as they are in QuickDraw. The QuickDraw orientation, running from top to bottom, is in some ways more natural because we read a page, and usually print it, from top to bottom. The PostScript coordinate system, however, corresponds to the more traditional mathematical practice and makes many mathematical tools readily available for working on paths and objects. In any case, as you will see in the next section, PostScript provides coordinate transformation tools that allow you to readjust these coordinates if you find it necessary. With a little experience, you will find this system as easy and natural as QuickDraw.

There are some other differences that you should note between QuickDraw and PostScript coordinates. First, QuickDraw coordinates

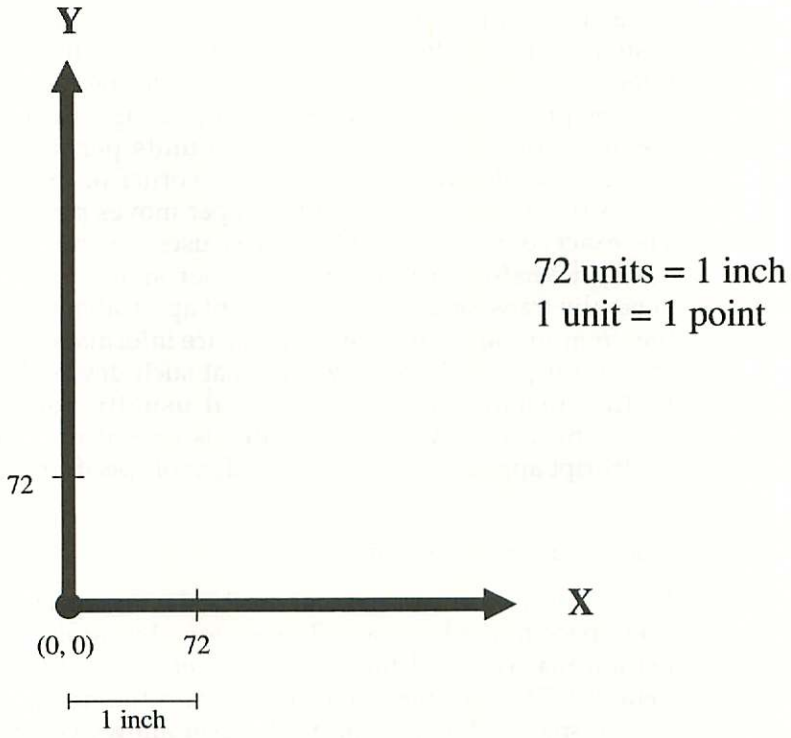

Figure 2-3. Default user space

are always integer values, whereas PostScript coordinates are real numbers. Therefore, the position (72.25, 100.84), which has no meaning in QuickDraw, is a perfectly correct value for PostScript. This difference exists because QuickDraw coordinates are tied to the specific output device. Remember that all QuickDraw coordinates are pixel positions, with each point representing the upper, left corner of a device pixel. In such a system, fractional coordinates are impossible because you cannot have a fraction of a pixel. PostScript coordinates, on the other hand, are independent of the device. In PostScript, the default user space has no direct connection to the specific output device or the output device pixels. Fractional coordinates for points, therefore, are quite reasonable and possible; the interpreter decides, by scan conversion, which pixel corresponds to the coordinate point. The default user space represents ideal coordinates and measures that always retain a fixed relationship to the current page, no matter what the device resolution or orientation.

Each individual output device has its own measure and coordinate system. Typically, an output device such as the LaserWriter uses an internal coordinate system that is tied into the resolution of the device and the process that produces the output page. For the LaserWriter, this means a coordinate measure of 300 units per inch and a coordinate system with its origin at the top, left corner of the output page—since that is the direction in which the paper moves past the printing station. The exact coordinates that the device uses are known as the *device space*.

The transformation from the user space to the device space is generally transparent to the PostScript application. However, if required, the program can access the device space information and coordinates for special purposes. Note, however, that such device-dependent coding is neither usually necessary nor is it usually good coding practice. Freedom from device coordinates is one of the major reasons why PostScript applications are independent of specific devices.

Coordinate transformations

PostScript uses a very general mechanism to specify the changes from user space to device space. This is done by means of a linear transformation matrix called the *coordinate transformation matrix*, or CTM. The default CTM specifies the changes from the default user space to the device space. This general mechanism allows you to make any set of linear changes to the default coordinates. By using various PostScript operators, you can make a wide variety of changes to the default coordinates; for example, you could change the coordinate structure to match the QuickDraw coordinates if you wanted.

PostScript's standard operators effect the following common transformations:

- translation
- rotation
- scaling

PostScript also has operators for general transformation, but these are somewhat more difficult to use.

Translation moves the origin of the user space to a given point. The orientation of the axes and the unit of measure remain unchanged. For example, the code sequence

```
72 72 translate
```

moves the new point (0, 0) to the old point (72, 72). This transformation is shown graphically in Figure 2-4.

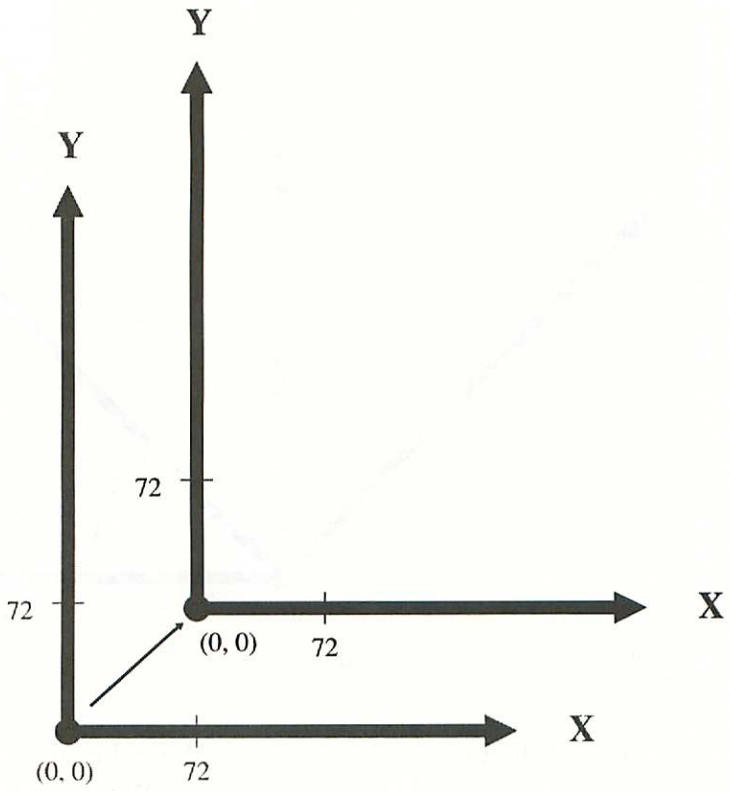

Figure 2-4. Example of translation

Rotation changes the orientation of the axes by rotating the coordinate system counterclockwise around the current origin by a given amount. The position of the origin and the unit of measure remain unchanged. For example, the code sequence

```
45 rotate
```

positions the new *x*-axis at a 45-degree angle to the horizontal. This transformation is shown graphically in Figure 2-5.

Scaling changes the default units in the user space to new values without changing either the origin or the orientation of the axes. The scale factors can be set independently for the *x*- and *y*-axes. For example, the code sequence

```
2 3 scale
```

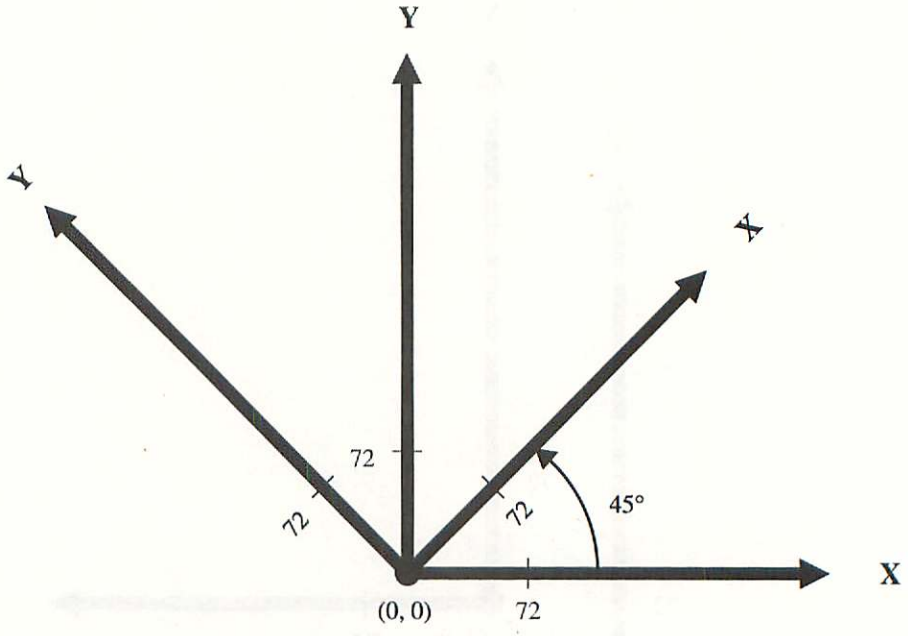

Figure 2-5. Example of rotation

makes all the units twice as large in the x direction and three times as large in the y direction. This transformation is shown graphically in Figure 2-6.

Note that all these transformations are cumulative; that is, each one transforms the axes from the state they were left in by the preceding transformation. Thus, for example, a rotation through 45 degrees followed by another rotation by 45 degrees leaves the x-axis pointing vertically, at 90 degrees to its original position. As another example, performing

```
2 3 scale
```

twice magnifies the x coordinates four times and the y coordinates nine times, and so on.

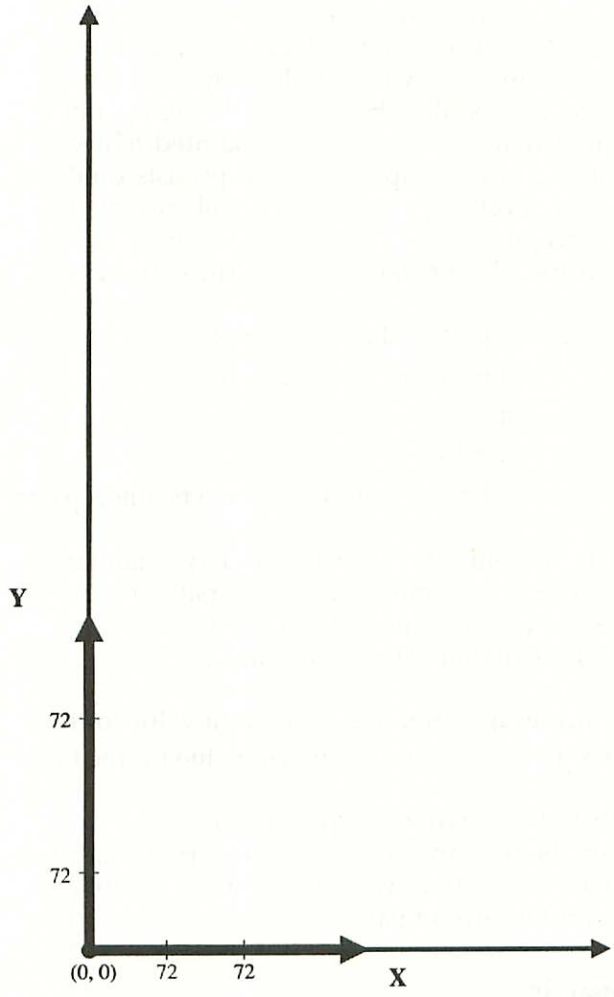

Figure 2-6. Example of scaling

▶ Graphics state

The PostScript graphics state is a collection of parameters that affect what graphics will look like when they are painted. In many ways, it contains data similar to that contained in a QuickDraw grafPort. The complete collection of the current settings of these parameters is referred to as the *current graphics state*. PostScript programs are able to reference these

parameters both individually and collectively, as the current graphics state. PostScript programs establish parameters such as the current color and the current line width in the graphics state. These parameters affect the execution of all subsequent painting operations; they do not affect anything that has already been painted. Once a parameter has been established in the graphics state, it persists until it is explicitly changed by another PostScript operator, or until the end of the job.

The graphics state contains parameters that control all of the following factors, which make up part of the graphics state:

- current path and current point
- current transformation matrix
- current font
- current color
- current line formation parameters: linecap and linejoin

Many graphic state parameters have standard default values; however, some, for example the current path and current point, do not. The PostScript language provides the following specific operators to set and report these parameters; for example:

currentlinecap	Returns the current value for the linecap parameter.
setlinecap	Sets the current value for the linecap parameter.

Most of the individual parameters are set and reported in a similar fashion. Setting and reporting the current path is a notable exception; there are a variety of ways to construct a path and several ways to test or report the current path.

▶ Objects

A PostScript object is any syntactic entity that the interpreter can recognize. Objects are generally created by scanning a stream of data, which is usually a file. A typical stream of data comes from a file being received over an AppleTalk connection or from some other transmission channel such as a serial connection. A string may also be used as a data stream. As you read earlier, the data stream is analyzed into tokens, which represent individual PostScript objects.

All data and procedures in a PostScript program consist of objects. The interpreter does not distinguish data from programs; all are simply objects to be processed. However, the same object may be treated in different ways at different points in the processing; procedures are the most notable example. You will examine procedure processing later in this chapter.

▶ Object notation

Ten special characters (delimiters) serve to distinguish one object from another.

Delimiter	*Function*
{ and }	Begin and end a procedure.
/	Begins a name literal.
%	Begins a comment.
(and)	Begin and end a string.
< and >	Begin and end a hexadecimal string.
[and]	Begin and end an array.

As you can see, several of these characters occur in pairs, to begin and end important forms of PostScript objects. As in C and Pascal, failure to use matching forms of delimiters may cause severe processing problems in your applications.

Three *white space* characters serve to separate ordinary PostScript objects.

- space
- newline (return or linefeed)
- tab

Generally, all white space characters are treated in the same manner; that is, the interpreter processes them all identically. The PostScript interpreter uses white space characters to delimit objects. Any number of consecutive white space characters is treated in the same fashion as a single white space character. The only instance where this is not true is when white space characters occur within a string or a comment; in those cases, multiple white space characters are treated individually.

▶ Object types

The PostScript language has 13 distinct types of objects in two major groups, as shown in Table 2-1.

Table 2-1. PostScript object types

Simple			*Composite*
Numeric	*Nominal*	*Special*	
integer	name	mark	string
real	operator	null	array
Boolean	file	save	dictionary
			fontID

Simple objects have a standard structure that is invisible to a PostScript program. Most PostScript objects are simple objects. Composite objects, on the other hand, have an internal data structure that is both visible and accessible to a PostScript application program. Components of composite objects can be accessed by using a pointer to their internal structure. Composite objects also consume virtual memory.

Composite objects are strings, arrays, and dictionaries. Each of these has an internal structure that can be accessed by an application. Dictionaries are accessed by the use of dictionary operations, as described in the next section. The individual elements of arrays and strings can be referenced by using an *index* value. All indices in PostScript are integers and begin with 0 as the first index value to the last available entry in the object.

Numeric objects are simple objects and may conveniently be divided into the following three types.

- Integer numbers (such as 1 or 12)
- Real numbers (such as 1.2 or 123.44)
- Boolean values (*true* and *false*)

Note that Boolean values in PostScript are not actual numbers, but instead they are independent objects. Unlike C, for example, numbers can never be used directly as Boolean values.

Nominal objects are objects that are similar to, or function like, names. They can be grouped as follows.

- Names
- Operators
- Files

Names are quite straightforward in both creation and use. You will learn about names and their use in more detail in the next section of this chapter under the discussion of dictionary operations. Operators are built-in PostScript processes. A file is a stream of characters that can be read or written. The PostScript file object is tied by the interpreter to the underlying stream of characters that it represents. Several standard files are available in PostScript; most notably, the %stdin, %stdout, and %stderr files.

PostScript also has the following group of special objects that serve the requirements of the interpreter.

Object	*Function*
mark	Marks a position on the operand stack.
null	Is a placeholder for other objects.
save	Is the result of a **save** operator.
fontID	Is a special object used in font processing.

Strings

PostScript strings are any characters grouped between matching parentheses, (and). You may place any characters within a string. The only special characters that are recognized within a string are the left and right parentheses, (and), and the backslash character, \. Examples of valid strings are as follows.

```
(this is a string)
(so is 123#@$%*add{} this)
(white space characters  may be  freely used..!!!!)
```

Strings in PostScript are exactly the length of the characters that you see. Unlike strings in C, for example, there is no termination character in a PostScript string; and, unlike strings in Pascal, there is no length byte at the beginning of the string. Since PostScript is an interpreted language, it has no need of special coding for strings. Since PostScript strings are composite objects, you can access individual elements in a string by using an index. The first character in the string is at index value 0, and valid indices run to the last character in the string. For

example, the string (abcdefg) has seven characters. You can access the character *a* by an index of 0, and the character *g* by the index value of 6. An index of 3 retrieves the character *d*, and so on. Any attempt to use an index value greater than the length of the string raises an error.

The backslash character, \, is an escape mechanism to signal that special processing is required within a string. The character or characters that immediately follow the \ determine exactly what processing occurs. This is quite similar to the use of the \ in C strings. The special processing that takes place is determined as follows.

\n	Newline
\r	Carriage return
\t	Horizontal tab
\b	Backspace
\f	Formfeed
\\	Backslash
\(	Single left parenthesis
\)	Single right parenthesis
ddd	Octal character code, where *ddd* is the octal value of the character that is to be placed in the string.
newline	No character; that is, a backslash followed by a newline character inside a string means that the newline is ignored and not included in the string.

Any other combination of characters with the backslash is ignored.

The octal character code escape is the most frequently used escape sequence. This escape mechanism allows you to place any arbitrary character value into a string. This allows you to use simple digits to generate characters that are outside of the standard ASCII printable subset that is used in PostScript. Note that the character generated is simply an octal value, from 0 through 377. It is the application's or programmer's responsibility to ensure that the character generated matches the desired character code in the current font. (You will review font encoding and processing later in this chapter.) Examples of the octal escape sequence, using the standard PostScript encoding, are as follows.

(\242)	Cents sign
(\267)	Bullet
(\263)	Double dagger

Note that the interpreter translates the three-digit octal code into a single character. The strings in the preceding list consist of a single character, and only the index value of 0 is valid for these strings. It is not possible to access the decimal characters that were used to make up the character in the string, and any attempt to index a character greater than 0 would be an error.

Since the parentheses characters mark the beginning and end of a string, including these characters in a string requires some special processing. Parentheses may be included in a string in two ways: by using matching parentheses and by using the escape mechanism. The two methods are illustrated in the following examples.

```
(this is a (valid) string with matching parentheses)
(this is also a \( valid string with a single parenthesis)
```

PostScript also provides for a special form of *hexadecimal string*. In this string format, the delimiters are angle brackets, < and >. When these brackets are used, the only valid characters that can occur between the delimiters are the digits 0 through 9, the characters *a* through *f*, and white space characters. The characters *a* through *f* may be either upper- or lowercase. The presence of any other characters within the delimiters causes an error. In these strings, each character represents one hexadecimal (base 16) digit. White space characters, which may be freely used for readability, are ignored. Since all ASCII characters are represented by decimal values from 0 through 255, it takes two hexadecimal digits to represent one character. For example,

`<61 73 64>`	is identical to	`(asd)`
`<31 39 34>`	is identical to	`(194)`
`<32 A2>`	is identical to	`(2\242)` or 2¢

Comments

Like all programming languages, PostScript has a method for including comments in the program text. Comments begin with the special character % and end at the next *newline* character. Any character, including the %, may be used in a comment. The PostScript interpreter will neither recognize nor process any special characters that are included in a comment. No white space character except the *newline* character terminates a comment; therefore, spaces, tabs, and so on may be freely used in a comment. Note that this is a difference from ordinary PostScript processing, where all white space characters are treated in the same fashion.

When the PostScript interpreter encounters a % character in ordinary processing, it assumes that the remainder of the line (until the next *newline* character) is a comment. Therefore, comments always end a line. Also, since a comment ends with a *newline* character, no comment can be longer than a single line. If you want or need multiple comment lines, each line must begin with the % character.

The only restriction on comment processing is within a string. If the % character occurs within a string, the interpreter does not process that as a comment, but as part of the string. By the same token, once the interpreter has recognized a comment, any occurrence of the string delimiters—or of any other special characters—is ignored. A good example of this occurs in the file names. Standard PostScript file names such as %stdin, which you met earlier, begin with a %. File names are always used as strings, so that the complete PostScript reference for the file is (%stdin). Starting the names with the comment character—which can never be used in a normal PostScript name, since the name definition would end at the % character—ensures that file names can only be used within a string and cannot conflict with other internal PostScript names. Device names, such as %disc%, which are used in certain file references, have the same structure for the same reasons. Examples of comments are as follows.

```
/abc %this is all comment (this is not a string {1 2
%add}
/abc      %this is effectively identical to the line
%above
%this is a valid comment - the entire line is a comment
12%(asdf) {1 2 add} this is just the integer '12'
```

Arrays and procedures

Arrays are any valid PostScript objects grouped between matched brackets, [and]. Arrays can also begin with the *mark* object. In fact, the left bracket, [, and the *mark* object, which is generated by the PostScript **mark** operator, are exactly equivalent objects in all PostScript processing. Examples of valid arrays are as follows.

```
[12 34 (abcd) 56 78]
[/name name .12 34]
[ 1 2 3 [10 20 30] 4 5]
```

As you see from the preceding examples, arrays can contain other arrays as objects. Although PostScript does not directly support

multidimensional arrays, you can use this feature to create a type of multidimensional array, if required.

As was true of strings, arrays are composite objects and individual elements of the array can be accessed by using an index value. Once again, indices run from 0 through one less than the number of elements in the array. Any attempt to index beyond the array results in an error.

Data to be included in an array is processed by the interpreter as it is received; this can be a source of some confusion. Consider the array

```
[12 15 add]
```

This array consists of only one element, the number 27. The interpreter begins the array when the left bracket, [, is encountered. Then it processes the numbers 12 and 15. Next it processes the **add** operator. This causes the two previous numbers to be added together, and only the result, 27, remains. Then the array is formed by the closing bracket,]. In this way, the array that results contains only one element.

Procedures are operators and other objects grouped together between matched braces, { and }. Procedures are a special form of array, called *executable arrays*. They have all the attributes of an array, including that individual elements within the procedure may be accessed as in an array. Examples of valid procedures are as follows.

`{12 15 add}`	Adds the numbers 12 and 15.
`{Xpos 12 add}`	Adds 12 to the Xpos variable (which must have been previously defined).
`{add 5 div}`	Adds two numbers that are supplied when the procedure is executed and divides the result by 5.

As the first example indicates, the primary difference between procedures and arrays lies in the method of processing. For an array, as you saw earlier, the addition takes place immediately and the result is stored in the array. For a procedure, the execution processing is somewhat more complex. If the procedure is encountered directly, that is, if it comes to the interpreter over the data stream, it is not processed immediately, but rather is stored for future processing. On the other hand, if the procedure is encountered indirectly, that is, if it is invoked from within an application, it is processed immediately. This allows you to define and store procedures for future execution in an intuitive fashion.

▶ Writing PostScript

PostScript uses a special notation because of its special operation and features. Operators are executed by the interpreter as soon as they are received. The data for an operator, called an *operand*, must precede the operator in PostScript. Thus PostScript is written

```
1 2 add
```

instead of in any of the following formats:

```
1 plus 2
1 + 2
add 1 to 2
```

Thus, in a sense, PostScript is written "backwards." This form of presentation is called *post-fix* notation.

Since the order of presentation for the interpreter is in a post-fix notation, naturally PostScript programs are written in the same fashion. Thus PostScript examples will look like the following simple PostScript fragments.

```
20 60 add    =>  80

20 60 sub    =>  -40

2 4 mul      =>  8
```

In a similar fashion, all PostScript reference materials present operator listings and other PostScript code references in a standard format that mimics the post-fix notation. Since this is not a book about PostScript programming, it will not provide any extensive discussion of PostScript operators; but here are two examples of typical PostScript operators in the standard reference format. All PostScript reference materials use this format.

num1 num2 **add** sum
adds the operands *num1* and *num2* and returns the result *sum*.

any_n ... any_0 int **index** any_n ... any_0 any_{int}
counts down *int* items on the operand stack and returns that item, any_{int}, to the top of the stack.

▶ Stacks and Dictionaries

The PostScript interpreter operates through a series of stacks and dictionaries. The PostScript interpreter uses these techniques to manage and access data that it must use for internal operations. As you will learn in this section, the use of the various PostScript stacks and the use of various dictionaries, both those created by the PostScript interpreter and those created by a programmer or application, are features of almost every PostScript operation. Understanding these two mechanisms is essential to understanding the nature of the PostScript language.

▶ Stacks

Certainly one of the most notable characteristics of the PostScript language is that it works through a series of stacks. Stacks are data structures that are created and controlled by the interpreter. All PostScript stacks follow certain specific rules that control data going to and from the stacks. These rules define how you can add data to a stack and how you can remove data from the stack. The PostScript interpreter maintains and uses four distinct stacks:

- execution stack
- dictionary stack
- graphics state stack
- operand stack

All of these stacks are independent of one another; each can be accessed and used without direct reference to the others. All these stacks are not equally accessible nor are they all equally important to the PostScript programmer. However, they all share a common structure and have common rules for their use.

Because all PostScript stacks have a common structure, all PostScript stacks work on similar principles. They are push-down, pop-up stacks; they are also sometimes called last-in, first-out (LIFO) stacks, which means the same thing. Personally, I find the push-pop analogy easier to visualize, and stack control is such an important part of PostScript programming that clear visualization is a very useful skill. For the remainder of this section, therefore, I will refer to these operations as though you were placing the last item on the stack on the top of the stack and all previous items are positioned below that item in the same order as they were put on the stack.

Whatever image or analogy you choose, you should know several rules of stack operations to help this process. First, only the top item on the stack is accessible. Nothing below it can be reached without removing or replacing the top item. Second, motion up and down the stack is automatic. That is, when you remove the top item, the item immediately below it in the stack becomes the new top item and is available for access automatically; at the same time, all items further down on the stack move up one position. Since every position in the stack is indexed, you can determine how many items are in the stack, with the top item on the stack being item number 0, and so on. Finally, use of the stacks in PostScript is not usually entirely automatic; it requires some attention and management from the PostScript program. For this reason, the PostScript language provides operators that allow you to control and manage the information on each of the stacks.

The *execution stack* holds PostScript procedures as they are being executed. This stack is the single exception to the rule that the PostScript program must manage the stack; the execution stack is under the direct control of and completely managed by the PostScript interpreter. It can be queried, but not changed, by direct program action. The execution stack contains executable objects, normally procedures and files, that are in the process of being executed by the interpreter. Whenever the interpreter interrupts execution of one object in order to execute some other object, the suspended operation is placed onto the execution stack as an object. Then, when the interpreter finishes executing each object, it retrieves the previous object from the execution stack and continues processing. In particular, the execution stack contains the server loop processing at all times.

The *dictionary stack* and the *graphics state stack* are special stacks that the PostScript program controls by using specific operators. The dictionary stack contains only PostScript dictionaries; its use and operation are covered in the next section where you review the creation and use of PostScript dictionaries. The graphics state stack contains the complete elements of the graphics state. Each entry in this stack consists of a complete copy of the graphics state as it existed in the interpreter at a given point in time. The PostScript program manages and uses this stack to help control the PostScript graphics environment.

Operation of the graphics state stack works in this fashion. A copy of the current graphics state is added to the stack by the **gsave** operator. This copy contains all the elements of the current graphics state. This is very similar to saving the current grafPort in a data structure. The difference is that the PostScript graphics state is saved on the stack, and the top element of the stack (and only the top element) can be restored

by the use of the **grestore** operator. This operator removes the top element of the graphics state stack, which represents the most recent version of the graphics state that was saved by the PostScript application, and makes that the current graphics state, replacing all previous graphics state parameters. The next state on the stack now becomes available for access. If no other state is on the stack, the original, or default, state is restored by the next **grestore**.

The *operand stack* is the most commonly used stack in PostScript, and it is also the stack that requires the most management by the PostScript application. This is the stack that is used to pass parameters and data to and from PostScript operators and procedures. Like all the PostScript stacks, this is a push-down, pop-up stack. This stack is so common in PostScript programming that it is generally referred to as "the stack"; any references to the other stacks are made explicitly by name. The operand stack is implicitly referenced by most PostScript operators; that is, they use the stack without making any direct reference to it. Operators that require data remove these objects (called *operands*) from the stack; and they return any data that they create (called *results*) to the stack. The operand stack can also be directly controlled by an application to reorder operands on the stack, to add new operands to the stack, or to remove operands from the stack.

Each of these stacks has a maximum number of objects that it can hold; this maximum number depends on the implementation of the PostScript interpreter that is being used and on the nature of the device itself. Any attempt to add an item to a stack that is at its maximum capacity or to remove an item from a stack that is empty (or at its minimum capacity) results in an error. Some general limits can be given for the LaserWriter. In the standard LaserWriter, these maximums are 500 for the operand stack, 20 for the dictionary stack, 31 for the graphics state stack, and 250 for the execution stack. Remember also that the PostScript interpreter uses these stacks itself, so that the full limit may not always be available to the application program.

▶ Dictionaries

The PostScript language makes extensive use of dictionaries to implement its processing. Like stacks, dictionaries form an important and distinctive feature of the PostScript language. Neither of these concepts is entirely unfamiliar; however, stacks are more common in computer programming than dictionaries. Dictionaries are familiar from the everyday world where language dictionaries are used in the process of reading and writing. An ordinary language dictionary associates a word

with a definition of that word. In PostScript, dictionaries are tables that associate two items: a *key*, which is usually the name of a procedure or variable; and a *value*, which may be any PostScript object. As you might guess, you use dictionaries to define procedures or variables in PostScript so that you can reuse them repeatedly during processing.

Although almost any PostScript object may be used as the key entry in a dictionary, the keys used for dictionary references are usually names of PostScript objects, and many PostScript implementations are designed to work optimally when names are used as keys. A *name* in PostScript can consist of any number of characters, and it can use alphabetic or numeric characters as well as most punctuation characters. In fact, the only characters that cannot be used in a valid PostScript name are the white space characters and the 10 special characters described earlier. Of these, only two are particularly significant: the slash (or / character) and the space. Since a space character terminates a name, it cannot be used within a valid name. The slash character is used to define a name literal.

PostScript makes a distinction between ordinary, or executable, names and name literals. A *name literal* is any valid name preceded by a /. All name literals are preceded by this character. The difference between ordinary names and name literals is in how the interpreter processes them. When the interpreter encounters an ordinary name, it looks that name up in the dictionary stack. On the other hand, if it encounters a name literal, it pushes that value onto the operand stack and does not look it up.

There are three main mechanisms for storage and retrieval of key-value pairs in a dictionary. First, there are special operators that will access any specific dictionary, which must be supplied to the operator as an operand. Second, there are operators that access the current dictionary implicitly. Third, the interpreter accesses the dictionary stack to identify ordinary names. When the interpreter encounters an ordinary name, it assumes that the name represents a key for one of the dictionaries on the dictionary stack. It then proceeds to access the dictionary stack in a consistent fashion, by looking for the name first in the current dictionary, that is, the dictionary on the top of the dictionary stack. If the name is not found there, the interpreter continues to search for the name down through all the dictionaries on the dictionary stack until it either finds the key or exhausts the stack. If it finds the key, it stops the search and retrieves the value associated with the key. If it exhausts the stack, it raises the **undefined** error.

The dictionary stack is the mechanism PostScript uses to hold all the currently available dictionaries. These are the dictionaries that contain

the keys for those objects in memory that are directly available to the PostScript program, without any explicit dictionary reference. Dictionaries can be created by a PostScript application and can be added to or removed from the dictionary stack. The dictionary at the top of the dictionary stack is called the *current dictionary*.

The dictionary stack always contains at least two dictionaries: *userdict* and *systemdict*. These two dictionaries are built into the interpreter and cannot be removed from the dictionary stack. The *systemdict* is the bottom dictionary on the stack; it contains the definitions for all the PostScript operators and for some other essential PostScript procedures that the interpreter uses. The *userdict* is directly above the *systemdict* in the standard PostScript implementation, and it contains all the code that is intended to be modified by programs, as well as containing certain support procedures and other variables. Application programs and other users may place their own information into *userdict*, but very often applications will create their own dictionaries for storing their procedures and data. These two dictionaries are always the two bottommost entries on the dictionary stack, and neither can be removed from it. Any attempt to remove these dictionaries will raise the **dictstackunderflow** error.

▶ Fonts

A *font* is a complete collection of characters and punctuation in a specific design, style, and size. A font, therefore, provides the specific form and size to each letter or other character as it is presented on the screen or on the printer. Most fonts are grouped by the design of the typeface; each typeface is known as a *font family*. Thus the names that you see on your Macintosh applications Font menu are names of font families such as Helvetica, Times Roman, and so on. Fonts are grouped into families so that you will have a complete and complementary range of styles and sizes for display or printing. Particularly in the Macintosh, it is common to use the word "font" to refer to what we have defined here as a font family. This is usually innocuous, and the meaning is generally quite clear from the context. In this book, the word "font" is used for font families as long as there is no possibility of confusion.

Font families are further divided by the style of the characters. Typical styles are bold, italic, narrow, oblique, and so on. On the Macintosh, styles also encompass such features as underlined, outlined, and shadowed text. The two most common styles are *italic*, in which the characters slope to the right to give an effect similar to handwritten text, and **bold**, where the characters are heavier and darker than usual for

added emphasis. In some cases, certain styles may be used to substitute for other style features; for example, the oblique style is substituted for the italic in some font families.

Fonts are an important aspect of visual communication. Fonts are used to make a document more legible. Fonts also impart a style and tone to any document, and the selection of the fonts used in the document has a marked effect on its appearance and impact. The visual effect of documents is a means of calling attention to the document's content. Many fonts have been designed to create a specific visual effect or to be used in a particular environment. Finally, fonts are also used to enhance the quality of the finished output. High-quality fonts contribute to a typeset look that provides eye appeal and impact. Emphasis can be created in a document by a judicious mixture of type faces; however, too many faces can cause the page to look garish and ugly. This can be avoided by using styles and sizes from a single font family. This variation of style and size is often used within a document to distinguish certain types of information; for example, bold letters may be used to identify important instructions on a form.

▶ Font measurements

The characters that make up a font can be distinguished by size as well as design and style. Font sizes are measured in a printer's unit called *points*, which is fractionally larger than 1/72 inch. Since the default units in both PostScript and QuickDraw are 1/72 inch, this measure is used as an approximation of the size in points in most applications. Font sizes measure the font height, which is the distance required to separate two successive lines of text. Thus the font size measures the distance required to keep the bottom of a 'y' on one line from touching the top of an 'h' on the line below it. Naturally, every character in a font has both height and width. The width of characters in a font is called *pitch*, which is related to the font height and the design of the font. All characters in a font may have the same width, in which case it is called a *monospaced font*. For most fonts, however, each character has an individual width, so that an 'I', for example, takes less room than a 'W'; these are called *proportional fonts* since the width of a character is in proportion to its actual width on the page.

Font height and font width are connected by the *font metrics*, which are a series of measurements for each character in a font. When you are printing text, the characters are generally set along a straight line, called the *baseline*. The part of a lowercase character, such as a 'y' or a 'p', that extends below the baseline is called the *descender*; whereas the part of a

character, such as an 'h' or a 't' that extends above the body of a letter is called the *ascender*, or riser. This allows a more precise definition of font height. The point size of a font is the distance between the baselines of two successive lines of text, such that the ascenders of the lower line do not touch the descenders of the upper line. For this reason, you can approximate the point size of a font by measuring the distance from the top of an ascender to the bottom of a descender on a line of text. For example, keeping in mind that points are 1/72 of an inch, 36-point type will measure about 1/2 inch from the top of a 'k' to the bottom of a 'y'.

▶ PostScript fonts

There are a variety of ways to produce letter shapes on a page. In raster-output devices, these come down to two basic types of fonts: bitmapped and outline. The simplest way to create a font for a raster-output device is to design each character as a series of dots, precisely set for the desired point size of the font and the resolution of the device. Such fonts are called *bitmapped fonts*. These fonts allow some flexibility of presentation and use, but they have some serious limitations. For example, they cannot be rotated and scaled to arbitrary angles and sizes. Furthermore, since bitmapped fonts by their very design are limited to a single device resolution, you must have one font for each device type that you use.

PostScript fonts are stored as mathematical outlines. This provides maximum versatility in both presentation and in device choice. In this case, the description of a letter is stored as a graphic outline and the character is converted into the pixel representation for the device at the time the character is required. Each character, then, is a separate PostScript program that draws the character as required. This technique allows characters to be scaled to any point size or to be printed at any angle.

The major obstacle to using outline fonts is the time that is required to convert the outlines into pixel representations of the characters. In PostScript, the characters that are generated for each font are cached to optimize printing speed while still retaining the flexibility of font outlines. The PostScript interpreter caches character bitmaps so that they only have to be rasterized once. In this way, the process of rendering a page of text data can be made quite fast and efficient. Characters in a font are rasterized regardless of size or orientation, although some very large characters may not be cached to save space. The PostScript language provides operators to control the caching process.

PostScript font organization

A PostScript font is implemented as a dictionary that contains a variety of information regarding the font. In particular, the PostScript font dictionary contains the routines that define the shapes for each character in a given typeface and style. This makes a PostScript font different from either of the earlier definitions of the word "font"; it is not a complete font family since it does not include different styles of characters in it, but it does provide information for creating all point sizes of the font. From here on, we will use the word "font" to mean a PostScript font.

A font specifies the shape of a character at a given size, with small adjustments for different sizes. The standard size for a font is 1 point in the default user coordinates, which is much too small to use. The ordinary font machinery is used to scale a font up to the desired point size. PostScript can support any point size, including fractional point sizes, if required.

PostScript font dictionaries contain pointers to several other sub-dictionaries. A PostScript font dictionary that is properly formed contains certain key-value pairs. Some of these must be present in any PostScript font, whereas others are optional and, in some cases, user defined. A font dictionary is created by, and remains accessible to, ordinary PostScript processing. In addition to the normal dictionary operations, there are special font operators that access the font dictionary entries.

PostScript font metrics

Characters in a PostScript font are defined in their own independent coordinate system, which is called the *character coordinate system*. Each character has its own origin—the point (0, 0)—which is called the *reference point* for the character. This point ties the character coordinate system into the user coordinates since this point corresponds to the current point in the user coordinates when the character is printed. The line connecting the reference points for successive characters is the baseline of the text, as you read earlier.

Every character in a PostScript font has individual character metric information, as shown in Figure 2-7.

The *character width* is the distance from the origin of one character to the position where the next character would be placed when writing successive characters. This value is a relative displacement from the current character reference point and is given by an x and a y value. In general, the x value will be some positive number and the y value will be zero for most roman fonts. However, fonts in other, non-roman writing systems may have negative x values or nonzero y values.

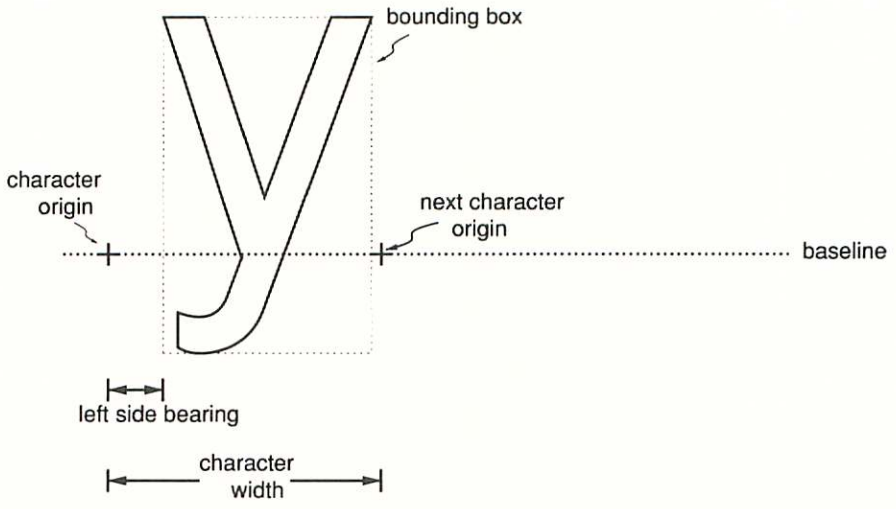

Figure 2-7. Character metric information

The *bounding box* of the character is a rectangular box that contains all of the marks made by the character. The sides of the box are parallel to the character coordinate axes, and the box is defined by giving the coordinates of the lower left and upper right corners of the box in character coordinates. The *left side bearing* is the distance from the character origin to the left side of the bounding box, again measured in character coordinates. Note that the left side bearing may be a negative number if the character extends to the left of the reference point. PostScript fonts are designed so that the left side bearing of any character may be adjusted independently of any other character metrics.

▶ Conclusion

To effectively program the LaserWriter, you have to know something about the PostScript language, which forms the essential link between the LaserWriter and the Macintosh software. Even if you are not going to program especially for the LaserWriter, understanding the PostScript language can help you use the general PostScript printing routines that are built into the Printing Manager through the LaserWriter printer driver and the Laser Prep file. This chapter has given you a basic, although condensed, introduction to the PostScript language and how it works in the LaserWriter.

3 ▶ Printing Manager Functions

▶ Chapter Overview

This chapter discusses the standard process of printing in the Macintosh environment. As stated in the Introduction, this book assumes that its readers are reasonably familiar with Macintosh programming, including the standard Toolbox calls and their use. Obviously the Printing Manager falls into that category. Nevertheless, as you will see, it is important that you have a short review of the basic Printing Manager functions and calls. The first section of this chapter, therefore, provides this review, along with a brief review of how QuickDraw and the Printing Manager interact to create printed output.

This is a how-to book, and it would be a pretty poor example of one if you didn't have some real, hands-on code with which to work. Unfortunately, when dealing with something as complex as Macintosh printing and the LaserWriter, a certain amount of groundwork is essential. However, since you are now ready to start programming, the chapter presents a code example at this point. This example uses the Think C object library to build a simple application that draws and prints. The first example simply uses QuickDraw with no special consideration for the LaserWriter. This allows you to see how the example application is put together and how it runs in the ordinary printing environment.

Next, you proceed to the first exercise that is tailored to using the LaserWriter as output. This exercise demonstrates the use of the QuickDraw PicComment to pass PostScript code to the LaserWriter. At this point, the code is quite simple, but it allows you to test and work with the process of specializing your output for the LaserWriter in a

consistent and compatible way. This section shows you three different types of PicComments that can be used, ending with a version that you will adapt in Chapter 4 to allow you to create and test PostScript code.

The last section of the chapter discusses how to print efficiently. Now that you are starting to customize your code for the LaserWriter, you should know some of the points, both positive and negative, that are inextricably linked to this type of programming. Some of these points are specific to the type of coding shown here. However, some of them are more general and will help you create applications that use printing resources more effectively even if you are not creating specialized code for the LaserWriter. We also discuss, briefly, the important issues surrounding how to manage the resources on your PostScript printer; in particular, how you can control virtual memory use in the device. This is an important point—one that recurs in future chapters. One important aspect of this process is the issue of permanent and temporary downloading. We discuss this both in relation to the Laser Prep dictionary and to other PostScript files, such as fonts, as well.

▶ Printing Environment

This section discusses the standard types of Printing Manager interactions with an application. Since you are expected to be reasonably familiar with this process, the review is (mercifully) brief. It is important, however, for the familiar two reasons. First, I need to make sure that all the readers understand the basic elements. The printing process involves a lot of complexity, but only some of it is relevant to the issues that will be discussed in this book. This review helps by identifying what elements are going to be discussed. Second, this review functions as a checklist. Readers who are quite familiar with these elements and the printing process can simply nod their heads as each item passes by; readers who may not fully recognize or remember an item can review it or get out the appropriate reference materials for a refresher.

▶ Printing Manager calls

Some basic issues always arise when you start to print on the Macintosh. In the section that follows, we are going to look briefly at how you use the Printing Manager. All Printing Manager calls follow the same basic set of steps.

1. Call PrOpen to prepare the Printing Manager for use.
2. Call PrValidate to ensure that the current printing record is compatible with the currently chosen printer.

3. Do whatever Printing Manager chores you wanted to do.
4. Call PrClose to release the resources used by the Printing Manager.

As this discussion proceeds, recall that this sequence of steps must take place for each task that you want to accomplish. Following this sequence ensures that you will always have the correct printer information regarding the current printer. Technical Note #161 discusses this issue in some depth; if you are not clear on this process, you may want to review it. Since you will be using the class CPrinter, which already handles this issue, no further discussion is needed here.

The printing process normally begins when the user selects Print... from the File menu. At this point, the application should call PrJobDialog to set the job dialog parameters, as discussed in Chapter 1. Although an application does not have to use the standard print dialogs, such action would open up a large can of worms because the application must set a number of fields for printing to take place correctly; such interference is not recommended. (See Technical Note #122 for more information.) The job dialog sets certain information in the TPrJob subrecord, which tells the application the page range, number of copies, and so on. This record also determines whether draft or spool printing will take place; for the LaserWriter, the printing method is *always* draft.

Note ▸

This may seem a bit surprising to the uninitiated; after all, what does *draft* mean on a LaserWriter? What it means is that these names for the two methods of printing were extremely poorly chosen. These names date from the period when the ImageWriter was the only supported printing device; and they make sense only in that context. You see, on an ImageWriter, you can either print directly onto the device, which is the *draft* mode, so called because you must print left to right and down the page. This prevents any fancy features such as bold face type, underline characters, and so on. Hence the name *draft*. Alternatively, you could spool a page to disk (or memory) and get additional features; this is called the *spool* mode. However, for the LaserWriter, none of these distinctions makes any sense. You are not limited by the motion of the printer. In fact, you aren't driving the printer directly at all; you're sending AppleTalk packets to it. It's just as bright as the host (your Macintosh) and is perfectly capable of handling its own features without any interference from the bleachers, thank you! Hence, on a LaserWriter, you use *draft* mode to get high-quality output; and you never set the *spool* mode at all.

One of the major occurences when you print in draft mode is that your data and the printing code from the driver must reside in the workstation memory at the same time. Generally you need at least 25K or more for the printer driver, AppleTalk communications routines, and other required code whenever you print to the LaserWriter. You should plan this into your memory requirements when you are looking at how your application uses memory.

Once the printing record and its job subrecord are set up correctly, the printing process begins. This basically involves the following steps.

1. Call PrOpenDoc to establish and initialize the printing grafPort.
2. Call PrOpenPage to start a new page to be printed.
3. Perform the QuickDraw routines to draw the page into the printing grafPort.
4. Call PrClosePage to end the page.
5. If there are more pages to print, loop back to step 2.
6. Call PrCloseDoc to close the printing grafPort.

As with other Toolbox calls, you need to provide error handling for the Printing Manager calls. However, because the Printing Manager and the printer driver take care of some of these, you should be careful about exactly when and how you test for and handle errors. (Review Technical Note #118 regarding this process.) Correct processing here is especially important when printing to the LaserWriter (or any other AppleTalk printer) because incorrect handling may, in fact, cause unnecessary errors in the processing.

▶ QuickDraw translation

Once the printing process is done, the device driver takes over. For the LaserWriter, this means translating the QuickDraw commands that you have used to draw in the printing grafPort into PostScript commands. This is done, effectively, by using two different pieces of code: the LaserWriter driver and the Laser Prep dictionary. First, the LaserWriter driver sets up a standard preamble that prepares the document for PostScript printing. Specifically, this resets the default PostScript coordinates to QuickDraw coordinates. It also sets the user name and the job name inside the printer. (You read about these differences in coordinate systems in Chapter 2.) This preamble also sets the physical size of the output page and does some other general tasks. Then the driver converts the specific QuickDraw commands that you have used

into calls to predefined PostScript routines that are intended to mimic the effect of those commands. Finally, the LaserWriter driver inserts a series of cleanup routines that make sure the processing is terminated correctly. These routines are defined in the Laser Prep dictionary, called **md**, which is designed to hold all these conversion routines and all the startup and cleanup routines that are required for printing on the LaserWriter (or any other PostScript device, for that matter).

Warning ▸

Note that the following text is not intended to be an actual description of the timing or sequence of the events within the driver. The communications and translation processes can and do, to some extent, overlap in actual processing. It is this overlap, in fact, that makes it important for you not to hold up the communications process with an error alert. Conceptually, however, the conversion process happens first, followed by the communications. It's just easier for simple humans, like us, to think of these as happening sequentially.

Once the conversion takes place, the document is ready to be sent to the printer. If you are using foreground printing, the driver opens the AppleTalk connection and begins a dialog with the LaserWriter; if you are using background or spool/server printing, the same dialog is initiated by the server instead of the LaserWriter driver. In any case, the first task is to be sure that the Laser Prep file has been downloaded and to determine what fonts are available on the device. If Laser Prep has not been loaded, it is loaded now. A check is also made to ensure that, if a Laser Prep version is already loaded, it is the same as the version being used by the LaserWriter driver. If you think about the process here, you should see immediately that any variation in versions could be a disaster since the LaserWriter driver has converted the QuickDraw calls into PostScript according to a specific set of procedures. Any change in the Laser Prep file means that those procedure calls may be invalid.

As the document is prepared, a list of the fonts used in the document is also made and checked against the currently available fonts in the output device. If any fonts are missing from the printer, the driver looks for additional PostScript fonts in the System folder and downloads those that are available. Any fonts not found are replaced by bitmapped versions from the screen. Don't worry too much about font handling here; this whole process is quite complex, and Chapter 6 is entirely devoted to these issues.

Once prepared, the document is sent to the LaserWriter for processing. The device stores the job name so that it can respond to requests for status by showing both the status of the device and the job that is in process. The LaserWriter driver includes the user name as part of the default job name, so that a status request can show both the user and the job. Errors inside the LaserWriter (PostScript errors) are communicated back over the AppleTalk channel in a standard format. You will learn about these error messages in more detail in Chapter 7, when you practice two-way communications with the LaserWriter.

▶ Example Code Structure

At this point, it's time that you began looking at some actual code. As stated in the Introduction, this book uses the Think C 4.0 compiler and the object-oriented extensions that come with it as a foundation for the example code. This allows you to move rather rapidly into actual code and provides a solid base for the extensions that you want to explore.

The example code presented here is derived from the Starter and Pedestal sample applications that are provided by Think C in the TCL Demos folder on your distribution disks. Both of these applications have the same basic structure and code segments, with some minor differences in how the classes and objects within them are handled. Since much of the code that you need is routine and all of it is derived from these two libraries, a detailed analysis of the code is not necessary. Instead this section concentrates on the specific changes that have been made to these basic classes to make them conform to your requirements.

The example is called SimpleLW instead of Pedestal or Starter. This makes the project file SimpleLW.π, and so on. Our example project follows the structure of the starter files and contains the following modules, presented here in descending order of hierarchy.

- SimpleLW.c
- CSimpleLWApp.h
- CSimpleLWApp.c
- CSimpleLWDoc.h
- CSimpleLWDoc.c
- CSimpleLWPane.h
- CSimpleLWPane.c

To set this up, simply follow the instructions in the Think C manual for creating a new project, located in the manual's object-oriented section. Look in Chapter 16 on using the Think C Class Library for detailed instructions. I expect here that you are already familiar with the class library, at least in general terms, and can use the class definitions comfortably. If not, you will probably need to work a little slower through this material, or you can work through one or two examples from the Think C manual.

If you review the code in Appendix A of this book in detail, you will see that there are a few changes from the basic applications described here. Specifically, I have rearranged some of the #include files from the code to the headers and I have added some basic code, such as an About... box at the Apple menu. All of this is just simple, personal preference. If you wish to do the same, go ahead; on the other hand, none of it is really required so you may ignore these small accommodations if you prefer others or none at all. The examples work just as well without them.

Note ▸

For those readers who have the accompanying disk, this is, of course, not an issue. You can simply use the project and code files that are already on the disk. However, you will have to both compile the project and run RMaker to create the necessary resources before you can run the exercises.

In addition, there must be a standard resource file, SimpleLW.π.rsrc, that contains, at this point, only the same resources that the Pedestal.π.rsrc file contains. You can make it by simply copying the Pedestal.π.rsrc file and renaming it. Later, you will add some more resources and customize some of what's already here.

The two basic application modules, SimpleLW.c and CSimpleLWApp.c, and the associated header file, CSimpleLWApp.h, are virtually identical to those in the Pedestal application. You may want to review these, in Appendix A of this book, at your leisure. Let's look at the other two code classes that form the basis for this application.

▶ CSimpleLWDoc

This class, and its associated header file, are quite close to the samples given in the Think C Pedestal application; however, there are some significant differences. Let's look first at the CSimpleLWDoc.h header file in Listing 3-1.

Listing 3-1. CSimpleLWDoc.h header file

```
/**********************************************************************
 * CSimpleLWDoc.h
 *
 *  Document class header for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
 **********************************************************************/

#define _H_CSimpleLWDoc     /* Include this file only once */

#include <Global.h>
#include <Commands.h>
#include <CApplication.h>
#include <CBartender.h>
#include <CDataFile.h>
#include <CDesktop.h>
#include <CDecorator.h>
#include <CDesktop.h>
#include <CDocument.h>
#include <CError.h>
#include <CPanorama.h>
#include <CScrollPane.h>
#include <TBUtilities.h>

#define  BASE_RES_ID    400

#define  BASE_WINDOW    BASE_RES_ID   /* Resource ID for WIND template */
#define  BASE_PANE      BASE_RES_ID   /* Resource ID for ScPn template */

struct CSimpleLWDoc : CDocument {

                /** Construction/Destruction **/
    void    ISimpleLWDoc(CBureaucrat *aSupervisor, Boolean printable);
    void    Dispose();

    void    DoCommand(long theCommand);
```

Listing 3-1. CSimpleLWDoc.h header file (continued)

```
    void    Activate(void);
    void    Deactivate(void);

    void    NewFile(void);
    void    OpenFile(SFReply *macSFReply);
    void    BuildWindow(Handle theData);

            /** Filing **/
    Boolean DoSave(void);
    Boolean DoSaveAs(SFReply *macSFReply);
    void    DoRevert(void);
};
```

The major addition here is the use of the #define for the resource ids. Also note that the resource id numbers are now 400. This technique for setting resource numbers allows you to choose any value that you want.

At this time, you should also enter ResEdit and change the number of the 'ScPn' and 'WIND' resources to 400 to match the BASE_RES_ID that you have used here (or whatever number you have entered). This is done quite easily. Launch ResEdit and open the SimpleLW.π.rsrc file that you made earlier by copying Pedestal.π.rsrc. Open the 'ScPn' resource and use Get Info to change the ID number of Scroll Pane from 1 to 400. Close the 'ScPn' resource. Next, open the 'WIND' resource and use Get Info on the Pedestal resource. Change the Name of the resource to SimpleLW and the ID number to 400. I have also added a 'vers' resource, but that is optional. If you want to add one, do so at this point. Then close and save the resource file.

Next you must modify the CSimpleLWDoc.c file. As in all the classes, some small cosmetic changes must be made to the file. These include changing the names of the include files and the references to other classes, and so on. All of these are quite straightforward; look at Appendix A if you have any questions. The **OpenFile** method is taken from the TinyEdit sample application so that it provides some error checking and basic functions that are otherwise lacking in the other sample projects. The major change in this class comes in the **BuildWindow** method, shown in Listing 3-2.

Listing 3-2. The **BuildWindow** method

```
/**********************************************************************
 * BuildWindow
 *
 *  This is the auxiliary window-building method that the
 *  NewFile() and OpenFile() methods use to create a window.
 *
 *  In this implementation, the argument is the data to display.
 *
 ***/

void CSimpleLWDoc::BuildWindow (Handle theData)

{
    CScrollPane     *theScrollPane;
    CSimpleLWPane   *theMainPane;
    Rect            sizeRect;

        /**
         ** First create the window and initialize
         ** it. The first argument is the resource ID
         ** of the window. The second argument specifies
         ** whether the window is a floating window.
         ** The third argument is the window's enclosure; it
         ** should always be gDesktop. The last argument is
         ** the window's supervisor in the Chain of Command;
         ** it should always be the Document object.
         **
         **/

    itsWindow = new(CWindow);
    itsWindow->IWindow(BASE_WINDOW, FALSE, gDesktop, this);

        /**
         ** After you create the window, you can use the
         ** SetSizeRect() message to set the window's maximum
         ** and minimum size. Be sure to set the max & min
         ** BEFORE you send a PlaceNewWindow() message to the
         ** decorator.
         **
```

Listing 3-2. The **BuildWindow** method (continued)

```
        ** The default minimum is 100 by 100 pixels. The
        ** default maximum is the bounds of GrayRgn() (The
        ** entire display area on all screens.)
        **
        **/
    SetRect(&sizeRect, 100, 100, 600, 500);
    itsWindow->SetSizeRect(&sizeRect);

    theScrollPane = new(CScrollPane);

        /**
         ** You can initialize a scroll pane two ways:
         **     1. You can specify all the values
         **        right in your code, like this.
    theScrollPane->IScrollPane(itsWindow, this, 10, 10, 0, 0,
                               sizELASTIC, sizELASTIC,
                               TRUE, TRUE, TRUE);
         **     2. You can create a ScPn resource and
         **        initialize the pane from the information
         **        in the resource as we do here:
         **
         **/

    theScrollPane->IViewRes('ScPn', BASE_PANE, itsWindow, this);

        /**
         ** The FitToEnclFrame() method makes the
         ** scroll pane be as large as its enclosure.
         ** In this case, the enclosure is the window,
         ** so the scroll pane will take up the entire
         ** window.
         **
         **/

    theScrollPane->FitToEnclFrame(TRUE, TRUE);

        /**
         ** itsMainPane is the document's focus
         ** of attention. Some of the standard
         ** classes (particularly CPrinter) rely
         ** on itsMainPane pointing to the main
         ** pane of your window.
         **
```

Listing 3-2. The **BuildWindow** method (continued)

```
        ** itsGopher specifies which object
        ** should become the gopher. By default
        ** the document becomes the gopher. It's
        ** likely that your main pane handles commands
        ** so you'll almost always want to set itsGopher
        ** to point to the same object as itsMainPane.
        **
        ** Note that the main pane is the
        ** panorama in the scroll pane and not
        ** the scroll pane itself.
        **
        **/

    itsMainPane = new(CSimpleLWPane);
    itsGopher = itsMainPane;

        /** The FitToEnclosure() method makes the pane
         ** fit inside the enclosure. The inside (or
         ** interior) of a scroll pane is defined as
         ** the area inside the scroll bars.
         **/

    ((CSimpleLWPane*)itsMainPane)->ISimpleLWPane(theScrollPane, this);

        /**
         ** Send the scroll pane an InstallPanorama()
         ** to associate our pane with the scroll pane.
         **
         **/

    theScrollPane->InstallPanorama((CPanorama*)itsMainPane);

    if (theData)
        /* do something here if this window contains data */;


        /**
         ** Let Decorator place window on screen
         ** ...mostly useful for multiple windows
         **
         **/

    gDecorator->PlaceNewWindow(itsWindow);
}
```

This method is not difficult to follow. You begin by sending the **new** message to the Window class to generate a new window. Then you initialize this window using the 'WIND' resource that you created in the previous step. This is given the name itsWindow. Next you set the display rectangle for the window. (You might have noticed that the window definition in the resource file does not have any set size.) Then you create a new ScrollPane in the window and initialize it with the name theScrollPane. In this code as shown, you are using the 'ScPn' resource that you created earlier; the comment, however, shows you how you might do this with inline code instead of using a resource, if that's more convenient. The ScrollPane is then made to fit the window that encloses it.

The next step brings us to the beginning of our private methods. You now create a new instance of SimpleLWPane. This requires the methods of CSimpleLWPane.c, which will be discussed in the next section. Notice here that this follows the standard processes of setting up a new instance of a class. This new instance is made the Gopher so that it will respond to commands directly. Then it is initialized with its enclosure being the ScrollPane that you created before. Finally, the ScrollPane is tied to the Panorama that you want to show and gDecorator is called to place and display the window. If you want to handle data in the window, that is done just before you place it on the screen; in this version, no data handling is provided, just a comment to show where it would be placed. All this is quite standard code.

▶ CSimpleLWPane

We next turn to the real heart of the example program: the class CSimpleLWPane. The code used in this class, although similar in spirit to that in the example applications, has some marked differences. As before, the analysis begins with the header file, CSimpleLWPane.h, shown in Listing 3-3.

First, notice that this *class* is a subclass of CPicture, and not of CPanorama, as the Pedestal application's Pane is. CPicture is itself a subclass of CPanorama and provides the handling required to draw and display a picture on the screen. As was true for all these headers, CSimpleLWPane.h combines methods from both Pedestal and Starter. For now, this is all the alteration that is required.

Now you must create the CSimpleLWPane.c module. This provides the basic functions that are required to make your application run correctly. The complete code is shown in Appendix A, but the main methods that you must use are given here. The first of these is the **ISimpleLWPane** method, shown in Listing 3-4.

Listing 3-3. CSimpleLWPane.h header file

```
/**********************************************************************
 * CSimpleLWPane.h
 *
 *  Pane class header for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
 **********************************************************************/

#define _H_CSimpleLWPane    /* Include this file only once */

#include <Commands.h>
#include <CBartender.h>
#include <CDocument.h>
#include <CEditText.h>
#include <CPicture.h>

struct CSimpleLWPane : CPicture {

                    /** Contruction/Destruction **/
    void    ISimpleLWPane(CView *anEnclosure, CBureaucrat
*aSupervisor);

                    /** Draw **/
    void    Draw(Rect *theArea);

                    /** Mouse and Keystrokes **/
    void    DoClick(Point hitPt, short modifierKeys, long when);
    Boolean HitSamePart(Point pointA, Point pointB);

                    /** Cursor **/
    void    AdjustCursor(Point where, RgnHandle mouseRgn);

                    /** Scrolling **/
    void    ScrollToSelection(void);
};
```

Listing 3-4. Code for **ISimpleLWPane** method

```
/*****************************************************************
 * ISimpleLWPane
 *
 *  Initialize a LWPane object.
 *
 ***/

void CSimpleLWPane::ISimpleLWPane(
                    CView           *anEnclosure,
                    CBureaucrat     *aSupervisor
                    )
{
    Rect        margin;

    CPicture::IPicture(anEnclosure, aSupervisor, 1, 1, 0, 0,
                sizELASTIC, sizELASTIC);
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the pane a little margin.
         ** Each element of the margin rectangle
         ** specifies by how much to change that
         ** edge. Positive values are down and to the
         ** right, negative values are up and to
         ** the left.
         **
         **/

    SetRect(&margin, 2, 2, -2, -2);
    ChangeSize(&margin, FALSE);
}
```

Here you have adapted the approach that is used in the TinyEdit example to provide a Picture pane with some margin around it. I don't expect that you will have any problems with this method.

Next look at the **Draw** method. This is just like the **Draw** in Pedestal, but it is still worth looking at as a basic example. You will be doing virtually all of your work later in this chapter from variants of this **Draw** method. The code for **Draw** is shown in Listing 3-5.

Listing 3-5. Code for **Draw** method

```
/***
 * Draw {BASIC}
 *
 *  Draw the contents of the Pane. The area parameter, specified in
 *  the pane's Frame coordinates, indicates the portion of the Pane
 *  which needs to be drawn.
 ***/

void    CSimpleLWPane::Draw(
    Rect      *area)
{

    MoveTo(frame.left, frame.top);      /* XXX Draws an X from corner to */
    LineTo(frame.right, frame.bottom);  /*   corner in the Pane          */
    MoveTo(frame.right, frame.top);
    LineTo(frame.left, frame.bottom);

}
```

This is quite simple, just drawing two lines in the window: one from the top, left corner of the pane to the bottom, right and the other from the top, right to the bottom, left. Notice here that you are drawing directly into the pane, without any further complications. This is the most basic form of using QuickDraw.

▶ First example

We are now ready to compile and run this example. I would suggest, at least for the first test, that you use the Debugger while you run the example; in this way, if you have made any small errors, you will be able to see them happen and possibly recover. This is a simple precaution, but often a very useful one.

In any case, select Run from the Think C Project menu. This will probably require that you bring the project up to date and save it before running. If you are using the Debugger, press the Step button to execute the top-level *main()* method in SimpleLW.c. If all goes well, you should see something like Figure 3-1 on your screen.

When you see this screen, select Print... from the File menu. This will give you the standard LaserWriter job dialog box (remember, we discussed this in Chapter 1). Click OK to print the file. You should shortly see an output that looks like Figure 3-2.

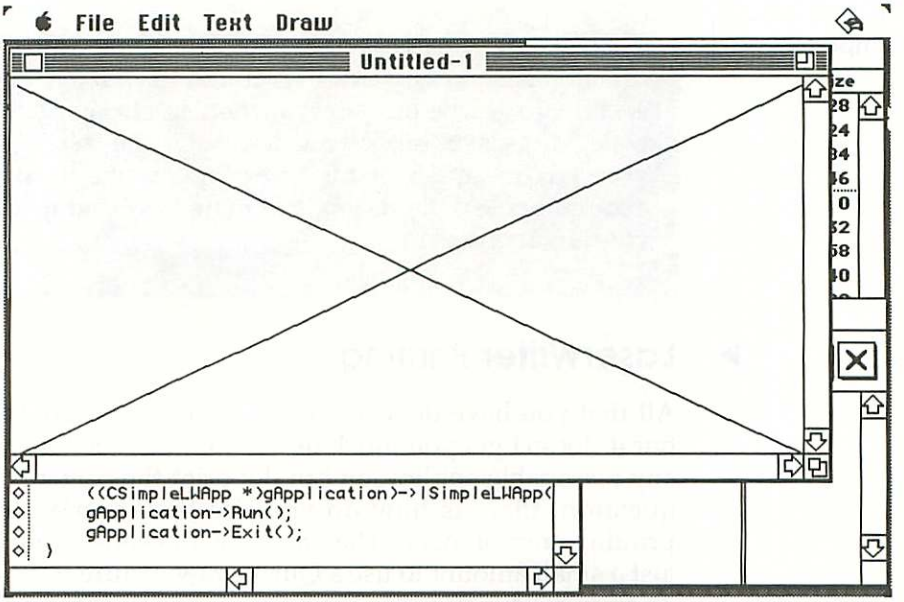

Figure 3-1. Screen output from the first example

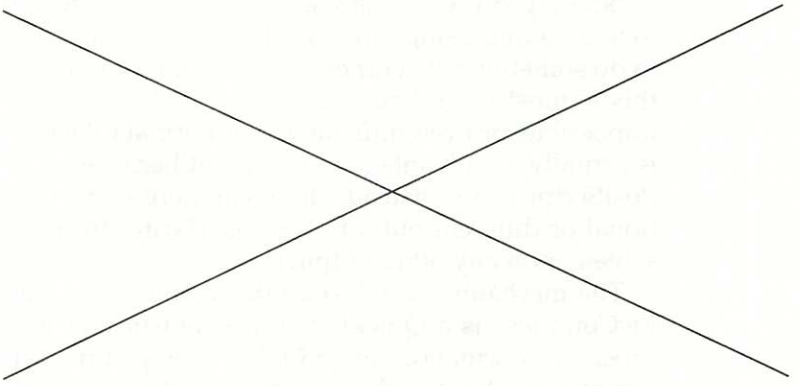

Figure 3-2. Page output from the first example

That's all there is to it. The various classes and objects already provided in the Think C library take care of the remainder of the problems of printing and so on for you. Once you are done with the printout, select Quit from the File menu to terminate the application. If you are using the Debugger, click on the Go button to finish the cleanup and termination process.

Important ▸

This brief demonstration has shown you how valuable object-oriented programming can be. You didn't have to do anything very exciting to get the full print functionality from this small portion of code. This makes our job much simpler and faster. From here on, you can concentrate on the LaserWriter without worrying too much about other less interesting, but more basic and no less essential, considerations.

▶ LaserWriter Printing

All that you have done so far has all been, no doubt, very interesting, but it doesn't get you much nearer to the LaserWriter. It is merely what any reasonable application can do: print the contents of the screen. The question, then, is how do you use the LaserWriter in this type of printing environment? The answer is to change your drawing function just a small amount to use a QuickDraw picture for the drawing instead of drawing directly onto the screen, as in the preceeding example. This requires a bit of reworking in the code, but what is needed is not difficult to implement.

Before proceeding, let's see what you are about to do. You are about to launch into using direct PostScript commands to get the LaserWriter to do something that other devices could not do. For the examples here, this is mostly a rabbit-out-of-a-hat trick; nothing that is done here is impossible, or even difficult, using normal QuickDraw techniques. This is actually an advantage at this point because you won't get lost in the PostScript code. Instead, the focus here is on how you can get additional or different output on a LaserWriter from what you get on your screen or on any other output device.

The mechanism used to do this is the QuickDraw PicComment. The PicComment is a QuickDraw mechanism that lets you embed certain types of information into a QuickDraw picture without using drawing commands. As you already know, a QuickDraw picture is a collection of calls that are recorded and then can be played back. Such a structure provides a number of benefits for the application programmer; you may already be quite familiar with them. A picture also provides the opportunity to add comment data that is "invisible" in the QuickDraw processing but is still contained within the picture and is available for processing by anyone else who understands the comments. The PicComment provides this type of communication.

In this case, the PicComment is coded to provide PostScript code. When the Printing Manager passes the QuickDraw calls along to the LaserWriter driver, these comments get translated according to a well-defined scheme into types of PostScript calls. The final effect is to allow you to embed actual PostScript code into your application without worrying too much about communications, translation, and so on.

The methods that you will see here demonstrate the most basic forms of PicComment. In Chapter 5, you will meet some more of these little gems and see how those are used. The complete definitions for all the relevant PicComments, along with some example code in Pascal (naturally) is provided in Technical Note #91, which is, frankly, essential reading for anyone who is contemplating doing this type of work. The code here should help you understand the points being made in Technical Note #91 (and others related to the same subject) and should allow you to work out most of the examples there using these techniques.

▶ Revised example

The revised example begins with the CSimpleLWPane.h header file.

Here you have made three additions to provide for your new drawing mechanism. First, you have added a new method, **Dispose.** You have also added two instance variables: myPSHandle and myMacPicture. These will be used in the drawing method and in the new **Dispose** method. The rest of the header remains the same. This code is shown in Listing 3-6.

The actual code class is next; the new heading to this version of CSimpleLWPane.c is shown in Listing 3-7.

Here you have added the new include file, CError.h. This header file provides links and definitions for error handling, which you will be using in this class. The first use of this information is to define the external name gError, which is the global error object. Then you define a series of class constants; all of these are simply mnemonic aids for coding, as you will see in the method definitions.

The major new part of this class is in the **Draw** method, which is quite changed. Listing 3-8 shows you the new code.

This code follows a standard pattern for creating a QuickDraw picture in a window. First, you set up the picture. Then you reduce the frame by two units and draw that rectangle (you will see why this is interesting when you compare the screen output and the printed output). For clarity, you create a message, using standard QuickDraw text methods, to illustrate what you are doing: in this case, preparing PostScript from QuickDraw text.

Listing 3-6. Revised CSimpleLWPane.h header

```
/**********************************************************************
 * CSimpleLWPane.h
 *
 *  Pane class header for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
 **********************************************************************/

#define _H_CSimpleLWPane    /* Include this file only once */

#include <Commands.h>
#include <CBartender.h>
#include <CDocument.h>
#include <CEditText.h>
#include <CPicture.h>

struct CSimpleLWPane : CPicture {

                    /** Instance Variables **/
    Handle      myPSHandle;
    PicHandle   myMacPicture

                    /** Contruction/Destruction **/
    void    ISimpleLWPane(CView *anEnclosure, CBureaucrat *aSupervisor);
    void    Dispose(void);

                    /** Draw **/
    void    Draw(Rect *theArea);

                    /** Mouse and Keystrokes **/
    void    DoClick(Point hitPt, short modifierKeys, long when);
    Boolean HitSamePart(Point pointA, Point pointB);

                    /** Cursor **/
    void    AdjustCursor(Point where, RgnHandle mouseRgn);

                    /** Scrolling **/
    void    ScrollToSelection(void);
};
```

Listing 3-7. Class constants and global variables from CSimpleLWPane.c

```
/*****************************************************
 * CSimpleLWPane.c
 *
 *  Pane methods for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
 *****************************************************/

#include "CSimpleLWPane.h"
#include <CError.h>

/*** Global Variables ***/
extern  CError      *gError;  /* Error handler  */

/*** Class Constants ***/
#define PS_BEGIN    190
#define PS_END      191
#define PS_HANDLE   192
#define PS_FILE     193
#define PS_TEXT     194
#define PS_RESOURCE 195

#define NIL_HAND    0L
```

Listing 3-8. **Draw** method code from revised CSimpleLWPane.c

```
/***
 * Draw {PicComment 192 Method}
 *
 *  In this method, you draw whatever you need to display in
 *  your pane. The area parameter gives the portion of the
 *  pane that needs to be redrawn. Area is in frame coordinates.
 *
 ***/
```

Listing 3-8. **Draw** method code from revised CSimpleLWPane.c (continued)

```
void CSimpleLWPane::Draw(Rect *area)

{
    char        *theString;
    char        *strPtr;
    long        strLength;
    OSErr       theError;
    Rect        picFrame;

    picFrame = *area;
    myMacPicture = OpenPicture( &picFrame );
    ClipRect( &picFrame );

    /**
        first draw a basic rectangle that outlines the frame
    **/
    InsetRect( area, 2, 2 );
    FrameRect( area );

    /**
        tell us (on the screen) what's happening
    **/
    TextFont( helvetica );
    MoveTo(20, 20);
    DrawString("\pPostScript from QuickDraw TextHandle.");

    /**
        set up for PostScript comment data
    **/
    theString = "\p0 761 translate 1 -1 scale /Times-Roman findfont
        20 scalefont setfont 50 650 moveto (Hello, world!) show\n";
    strLength = theString[0];
    strPtr = theString + 1;
    theError = PtrToHand( strPtr, &myPSHandle, strLength);
    if ( theError != noErr )
        gError->SevereMacError( theError );

    /**
        do PostScript printing
    **/
    PicComment(PS_BEGIN, 0, NIL_HAND);
```

Listing 3-8. **Draw** method code from revised CSimpleLWPane.c (continued)

```
    PicComment(PS_HANDLE, strLength, myPSHandle);
    PicComment(PS_END, 0, NIL_HAND);

    /**
        tell us (on the screen) that we've finished
    **/
    MoveTo(50, 50);
    DrawString("\pPostScript processing completed." );

    ClosePicture();

    /**
        store picture handle as instance variable
    **/
    inherited::SetMacPicture( myMacPicture );
    inherited::Draw( &picFrame );

}
```

Now you come to the heart of this process. You use PicComment 192, also known as PostScriptHandle, to actually send the PostScript data to the LaserWriter. This comment embeds PostScript code in a QuickDraw picture, taking the PostScript from a text handle and placing it into the picture definition. The text handle must be a handle to a generic text string, without a Pascal length byte at the front and terminated by a white-space character, here a carriage return (ASCII 13); in other words, a standard PostScript string. The format of the PicComment, then, is as follows.

```
PicComment( 192, (int) strLength, (Handle) PSHandle )
```

Before you can send this, however, you have two tasks to perform. First, you must set up your PostScript text into a text handle. Second, you must announce to the driver that you are going to send it some PostScript.

You set up the text handle by placing the PostScript code that you want to send in a simple Pascal string. The string here selects the Times-Roman font, scales it to 20 points, and sets it as the current PostScript font. It then moves out onto the page and displays the perennial string "Hello, world!" on the page. Finally, the string ends with a carriage return, encoded in C by the \n character. Since all Pascal strings begin

with a length byte, you pull that out as the length value. Then you reset the pointer past the length byte because you do not send that to the printer. Now you are ready to call the Toolbox utility routine PtrToHand, which returns you a handle to the text portion of the string as myPSHandle. After you do a quick test to make sure there were no memory errors, you're ready to send your PostScript.

Note ▸

You may wonder why this is a Pascal string since you can't use a Pascal string for the actual transmission. Actually, you could almost as easily use a C string, but there are certain advantages to the process shown here. The Think C compiler supports Pascal strings by using the \p code at the front of a standard C string. In this case, you need to know the length of the string before you can send it in the PicComment. If you used a C string, you would have to include the C string handling routines to find the length and you would have to eliminate the terminating null character from the string. The approach here seems a bit easier. However, if you want to use a C string, by all means do so. Do notice how much easier this technique is than using a native Pascal string, which doesn't have a convenient \n representation for the return character.

Now you announce your intentions to the driver by sending it a PicComment 190, more commonly known as PostScriptBegin. This comment requires no additional information, so the next two parameters are set to 0 and NIL_HAND, respectively. Since there are a wide variety of PicComments, you must always supply the correct number of parameters, even when the individual comment does not require any. Next you send your PostScript string, using the length and handle information that you developed earlier. Finally, you tell the driver that you are done sending PostScript by sending a PicComment 191, which is known, for obvious reasons, as PostScriptEnd.

You are now ready to process the picture. First, you must close it by sending a ClosePicture call to QuickDraw. Then, you set the instance variable to myMacPicture and call the inherited **Draw** method to display the picture on the screen.

You must provide one more method. Once you have done all this, you have allocated memory for a handle and for the picture; as a careful and professional programmer, you want to release these resources when you are done with the processing. You do this by creating a new **Dispose** method, as shown in Listing 3-9.

Listing 3-9. Code for new **Dispose** method

```
/***
 * Dispose
 *
 *   In this method you dispose of the picture and PS handles
 *   and print the picture.
 *
 ***/

void CSimpleLWPane::Dispose(void)

{
    /**
        clean up when the picture is done
    **/
    DisposHandle( myPSHandle );
    myMacPicture = inherited::GetMacPicture();
    KillPicture( &myMacPicture );
}
```

The code here simply disposes of the text handle that you created earlier and retrieves the picture handle and releases it. (See the note under the **SetMacPicture** method in CPicture if you have a question about this process.) All nice and tidy.

With this all accomplished, let's compile the code and run it. If all goes well, you will see something like Figure 3-3 on your screen.

In this output, note the rectangle around the inside of the window, showing you the limits of the frame that you are drawing into. You also see the text that you drew with the DrawString command, carefully placed where you might expect it. What you don't see, and shouldn't see, are the PicComment values that you placed into the picture. However, they are still there. To prove it, select Print... from the File menu. You will see output something like Figure 3-4.

Note the differences between the screen and printed outputs. The line of QuickDraw text that you first placed onto the picture is in the same position on both; but everything else is changed. First, the line around the frame now runs around the entire page; this is an important point, and it shows you that the frame coordinates have been enlarged to the page boundaries. Second, the PostScript text that you sent to the LaserWriter has been executed and has printed the string that you set onto the output page.

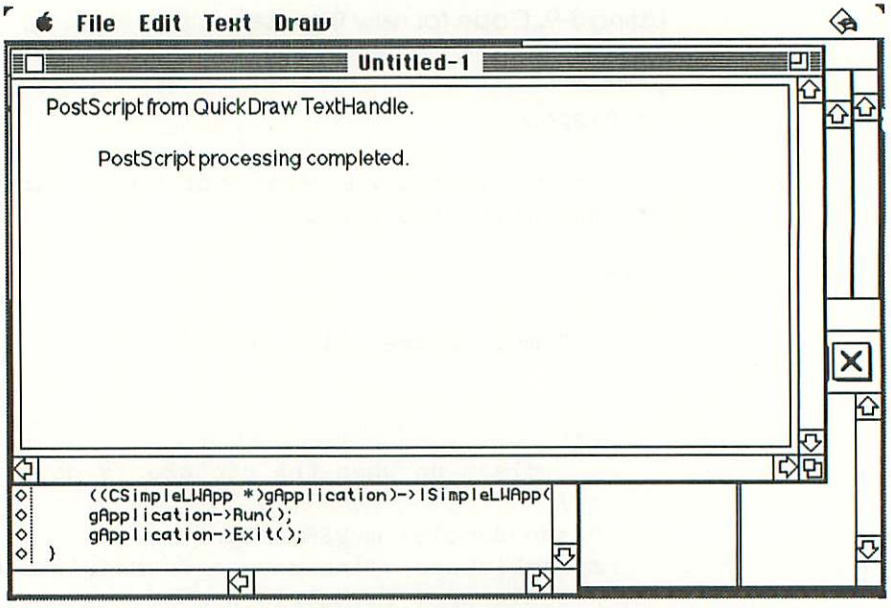

Figure 3-3. Screen output from PicComment 192 test

PostScript from QuickDraw TextHandle.

PostScript processing completed.

Hello, world!

Figure 3-4. Printed output from PicComment 192 test

There you have it! Your first PostScript program. It's small, but it works. As you will see in a short while, there are several other variations on this same theme, all using one form of PicComment or another.

▶ PostScript code for example

For a simple exercise such as this, you can easily just copy the text into your file. Although Chapter 4 discusses this code, and similar code, in much more detail, it is useful to briefly look here at your first PostScript

code. The code is placed in one string, which actually comprises about seven normal lines of PostScript code, as shown here.

```
0 760 translate
1 -1 scale
/Times-Roman findfont
20 scalefont
setfont
50 650 moveto
(Hello, world!) show
```

This is the same PostScript code that is used for all the exercises in this chapter. This code performs several functions. The first two lines move the page origin from the current QuickDraw (0, 0) back to approximately the PostScript (0, 0), and then they reverse the *y*-axis so that positive values move up the page (the PostScript default, remember) instead of down the page (the QuickDraw default). The next three lines find the Times-Roman font, scale it to 20 points, and set it as the current font. Recall that this process was discussed in Chapter 2. (I used Times-Roman here so that you could easily distinguish the PostScript text from the QuickDraw text, which is printed in Helvetica.) Then you move to the point (50, 650) in PostScript coordinates. This is the equivalent to the QuickDraw position (50, 110), which is just below the two lines of text that QuickDraw has placed on the page with DrawString. Finally, the **show** operator draws the string "Hello, world!" onto the page at the chosen point, using the given font. A simple, but effective test.

Note ▸

You may wonder why I say that this only "approximately" restores the PostScript coordinates. First, the QuickDraw translation from screen coordinates into PostScript coordinates changes depending on the choice for parameter settings in the style dialog. Second, the default translations from PostScript to QuickDraw move the origin along both the *x*- and *y*-axes. Here, you are only moving back down the *y*-axis, and not along the *x*-axis. The result is that the revised origin is close to the default PostScript origin, but is not exactly on it. You will examine these transformations in detail in Chapter 5, but, for now, you can just take my word that the conversion is only approximate.

▶ Device-independent printing calls

One of the most important things to notice about this process is that is device independent—as the Macintosh Printing Manager defines device independence. This does not mean what you might think: that you get the same results on any device, independent of the special device characteristics. The situation here is that neither the application nor the system cares whether you are connected to a LaserWriter or some other device. This code can be processed by any device driver without causing errors or problems. As you saw earlier, the screen output, with the PicComments embedded, worked just fine. The actual output, of course, is quite different on the LaserWriter than it was on the screen or than it would be on an ImageWriter.

Nevertheless, device independence is quite an important point, and an advantage not to be lightly dismissed. Effectively, any purely QuickDraw commands that occur between a PostScriptBegin comment and a PostScriptEnd comment are ignored by the PostScript device, whereas the PostScript commands that are employed in the PicComment are ignored by non-PostScript devices such as the screen or an ImageWriter. This allows you to create true device independence while still having the benefit of PostScript—subject, as all "device independence" is, to the limitations of the final device. You do that by providing two representations of your picture. The first one would be using QuickDraw commands, perhaps simulating or indicating any specialized features that can't be easily shown. Then you provide a series of PostScript commands, using PicComments, that provide the alternate representation of the picture for the LaserWriter. This means that, when you print to a PostScript device such as the LaserWriter, you get the full benefit of the PostScript drawing commands, but when you print to a non-PostScript device, you get the QuickDraw representation. In ancient times, this was known as eating your cake and still having it—a good trick; and all done without any concern on your part about the actual device that is being used for the final output.

▶ PicComment usage

PicComments tell the LaserWriter driver that the application intends to communicate directly with the LaserWriter using the application's own PostScript code. You have seen them in action and now should look at them in some detail to derive the full benefit out of this feature. The PicComment requirements (see *Inside Macintosh, Vol. I*) presented in C format (as they would be called from a C program) look like this.

```
PicComment( int kind, int dataSize, Handle dataHandle )
```

This general method can be used for a wide variety of information. The PicComment routine inserts the specified comment into the currently open picture definition. The kind integer specifies the type of the comment. When the comment has data, the dataHandle points to the data and the dataSize is the size of that data in bytes. Some PicComments, such as PostScriptBegin and PostScriptEnd, don't require any additional data. In these cases, the handle should be nil and the length value should be 0. Table 3-1 shows all the PostScript PicComment values.

Table 3-1. PicComment values and data

Type	*Kind*	*Data Size*	*Data*	*Description*
PostScriptBegin	190	0	NIL	Saves QuickDraw state, disables QuickDraw drawing commands, and sets driver to process PostScript
PostScriptEnd	191	0	NIL	Restores QuickDraw state
PostScriptHandle	192	-	PSData	PostScript data in Handle
PostScriptFile	193	-	FileName	FileName data in Handle
TextIsPostScript	194	0	NIL	QuickDraw text is sent as PostScript code
ResourcePS	195	0	Type/ID/ Index	PostScript data in a resource file

Note ▸

You may wonder how these comments get processed. The actual technique is documented in the QuickDraw section of *Inside Macintosh, Vol. I*, under the heading "Customizing QuickDraw Operations." This explains how the comments can be parsed by providing the address of a special procedure in the QDProcs record to process all picture comment data. The DrawPicture routine calls the routine specified at this address to process all comment data. The default procedure simply ignores the comments, but the LaserWriter driver directs the processing to its own routine, which then can execute the comments as requested. This process is also very well explained in Scott Knaster's excellent book, *Macintosh Programming Secrets* (Addison-Wesley 1988), which has lots of other good information as well.

There are some cautions and concerns to deal with before you continue with more PicComment magic. To begin with, you should turn off Background printing on your Macintosh if you are running MultiFinder (as you most probably are, since the Think C Debugger only works under MultiFinder). This is necessary because some of these comments only work when the application that uses them does not spool the output to the Print Monitor. The two PicComments that are restricted in this way are PostScriptFile and ResourcePS. When you use either of these two methods, there is a problem finding the requested resource when the Print Monitor goes to spool the file; the result is an error if Background printing is in effect.

Remember that you have inserted your commands within a QuickDraw picture. Therefore, the LaserWriter driver has performed a coordinate translation to set the coordinates to the QuickDraw standard instead of the default PostScript orientation by using standard functions defined in the Laser Prep header. This means, as we discussed in Chapter 2, that the page origin is at the top, left corner of the printable area, and that the positive *y*-axis runs down the page. In other words, the entire graphics state that the driver normally sets up for translating QuickDraw commands into PostScript is still set up and active when your comments begin processing. The driver does nothing to change or reset this. The header does, however, save the graphics state at the beginning of the comment processing and restore it at the end, so that you can alter the coordinates as you wish without worrying about getting them back for any further QuickDraw processing. Therefore, you will generally want to restore the original PostScript orientation before drawing; if you don't, your results may come out upside down. Hence the coordinate transformation that you used earlier.

It is both useful and important to have some idea of what the Laser Prep header is doing and what functions it provides. Because this is a changeable (sometimes *very* changeable) set of procedures, you will learn techniques for exploring the header functions and discovering what they do and how they do it in Chapter 5.

The PostScript code that you write and send to the printer should be "well behaved"; that means that it should not use certain functions or operators that cause your code to undo or change the state of the printer or of the current page image. The following list shows you the complete set of operators that you should avoid using.

banddevice	**copypage**	**erasepage**	**exitserver**
framedevice	**grestoreall**	**initclip**	**initgraphics**
initmatrix	**nulldevice**	**quit**	**setdevice**
setmatrix	**setsccbatch**	**setscreen**	**settransfer**
setcolortransfer			

In addition to the operators on this list, you should also avoid using the **showpage** operator to generate printed output. The standard bottlenecks automatically cause a page to be printed when you have completed your picture; if you issue a **showpage** command yourself, you cause an additional page to be printed, and one of the pages will be blank—not, presumably, what you had in mind. Of course, this only applies to output that fits on a single page, as all our examples do. If you are creating multipage output, you must take alternative steps. These steps are discussed with the advanced programming issues in Chapter 7.

▶ Additional PicComment examples

Now that you know some more about PicComments and are familiar with some of the concerns and cautions about using them, let's explore some of the alternatives so that you can see how you might implement these in your own applications.

The first example uses the PicComment type TextIsPostScript (194). In this comment mode, QuickDraw text strings are considered as PostScript program information and treated as programming instructions rather than as data to be printed. This can be quite useful, for two reasons. First, this type of comment can use any of the QuickDraw text calls. This makes it more flexible than the PostScriptHandle (192) call that you just used. Second, this mode has no restrictions regarding its use in printing; that is, there is no requirement that Background printing be turned off for this method to work as there is for the PostScriptFile (194) and ResourcePS (195) comment types. Let's look at a simple example of how this might work. Listing 3-10 shows you the code for a new version of the **Draw** method in the CSimpleLWPane.c routine.

Remember that you originally set up CSimpleLWPane.c with all the necessary values for the comment types, so no class constants need to be changed. Simply replace the previous **Draw** method with the one shown here. This is quite similar to the previous **Draw**, with the single exception that here we have added some processing to hide the pen during the PostScript portion of the picture. This is necessary unless you want to see the PostScript code on the screen as text as well as having it execute on the LaserWriter as code. Remember that the PostScript code here is contained in QuickDraw text calls and therefore will be drawn on the screen or other output medium by a QuickDraw device (which is busily ignoring the comment data that tells it that the following text should be interpreted as PostScript).

Listing 3-10. **Draw** method Code with PicComment 194

```
/***
 * Draw {PicComment 194 Method}
 *
 *  In this method, you draw whatever you need to display in
 *  your pane. The area parameter gives the portion of the
 *  pane that needs to be redrawn. Area is in frame coordinates.
 *
 ***/

void CSimpleLWPane::Draw(Rect *area)

{
    Rect        picFrame;

    picFrame = *area;
    myMacPicture = OpenPicture( &picFrame );
    ClipRect( &picFrame );

    /**
        first draw the basic rectangle
    **/
    InsetRect( area, 2, 2 );
    FrameRect( area );

    /**
        tell us (on the screen) what's happening
    **/
    TextFont( helvetica );
    MoveTo(20, 20);
    DrawString("\pPostScript from QuickDraw Text.");

    /**
        set up for PostScript comment data
    **/
    HidePen();
    MoveTo(20, 30);

    /**
        set up PostScript printing
    **/
```

```
    PicComment(PS_BEGIN, 0, NIL_HAND);
    PicComment(PS_TEXT, 0, NIL_HAND);
        DrawString( "\p0 728 translate" );
        DrawString( "\p1 -1 scale" );
        DrawString( "\p/Times-Roman findfont 20 scalefont setfont");
        DrawString( "\p150 650 moveto" );
        DrawString( "\p(Hello, world!!) show" );
    PicComment(PS_END, 0, NIL_HAND);

    /**
        tell us (on the screen) that we've finished
    **/
    ShowPen();
    MoveTo(50, 50);
    DrawString("\pPostScript processing completed." );

    ClosePicture();
    /**
        store picture handle as instance variable
    **/
    inherited::SetMacPicture( myMacPicture );
    inherited::Draw( &picFrame );

}
```

If you substitute this code in your application and rerun the project, you will get the screen of data shown in Figure 3-5, and the output page shown in Figure 3-6. The only changes here to your output are in the QuickDraw text strings, which tell you that you're using a different type of comment, and in the addition of another exclamation point to the PostScript output string.

Now let's progress to one more variation on this same theme. You may feel that this is getting to be old hat, but I assure you that this variation is one of the more important ones—at least from the viewpoint of this book. In Chapter 4, you are going to use this method to start coding and testing your own PostScript files, without any more fussing with the application! So follow along just once more before you move on.

This comment type is PostScriptFile (193). In the previous comments, the PostScript data was contained within the picture definition itself. This has both advantages and drawbacks. The advantages are that the picture is self-contained and, once created, does not depend on any outside information to be processed correctly. Also, the picture can be

processed whether Background printing is enabled or not. On the other hand, it has the disadvantage that the data must be specified at the time the picture is created. Also, pictures were limited to a maximum of 32K bytes of data prior to System 6, so you can only use a modestly sized PostScript program with earlier system versions.

The PostScriptFile comment tells the LaserWriter driver to draw the necessary PostScript code from the *resource fork* of another file. The PostScript data in that fork must be in a specific resource type and have

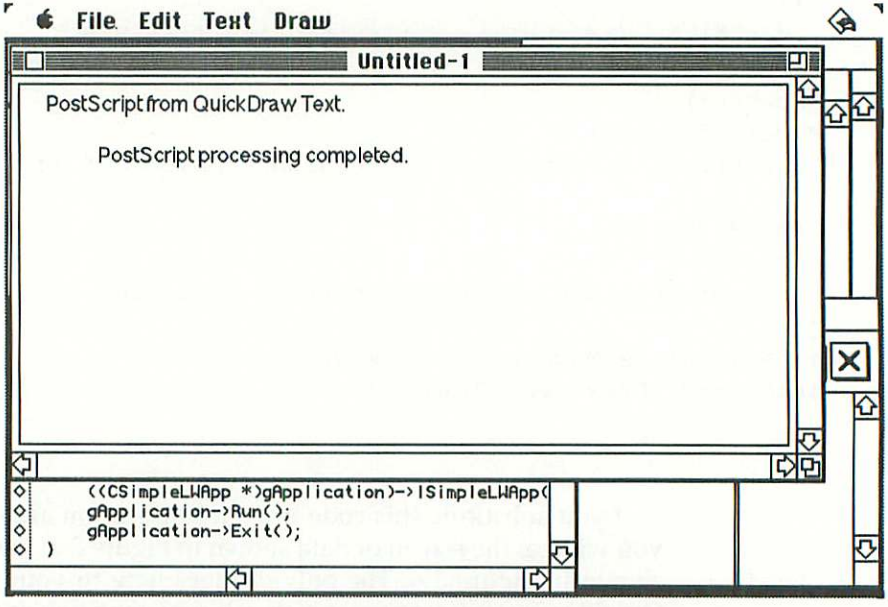

Figure 3-5. Screen output for PicComment 194

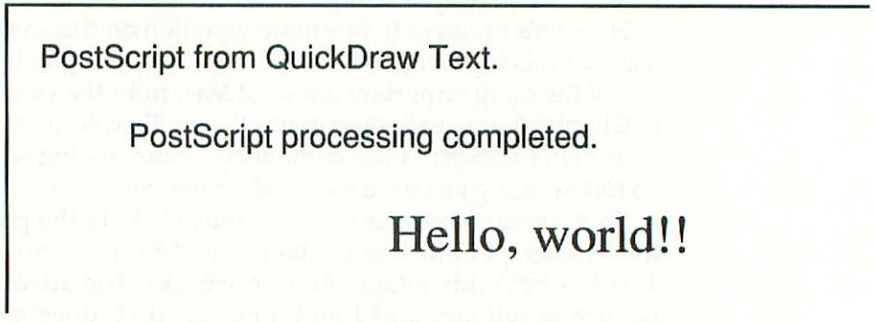

Figure 3-6. Print output for PicComment 194

a specific form and format. However, if you follow these rules, you can use arbitrary PostScript code in your picture. The drawback here is that you *must* have Background printing turned off or you will get an error. (See Note box in the section on "Device Independence.") With that small caution, let's look at the code that will use this technique, shown in Listing 3-11.

Listing 3-11. **Draw** method code using PicComment 193

```
/***
 * Draw {PicComment 193 Method}
 *
 *  In this method, you draw whatever you need to display in
 *  your pane. The area parameter gives the portion of the
 *  pane that needs to be redrawn. Area is in frame coordinates.
 *
 ***/

void CSimpleLWPane::Draw(Rect *area)

{
    Boolean     errSw;
    char        *theString;
    char        *strPtr;
    Handle      resHandle;
    int         handleSize;
    int         refNo;
    int         resErr;
    long        strLength;
    OSErr       theError;
    Rect        picFrame;

    picFrame = *area;
    myMacPicture = OpenPicture( &picFrame );
    ClipRect( &picFrame );

    /**
        first draw the basic rectangle
    **/
    InsetRect( area, 2, 2 );
    FrameRect( area );
```

Listing 3-11. **Draw** method code using PicComment 193 (continued)

```
    /**
        tell us (on the screen) what's happening
    **/
    TextFont( helvetica );
    MoveTo(20, 20);
    DrawString("\pPostScript from FileResource.");

    /**
        set up PostScript comment data
    **/
    theString = "\pMacSE30 HardDrive:PSRes193.rsrc";
    strLength = theString[0] + 1;
    strPtr = theString;
    theError = PtrToHand( strPtr, &myPSHandle, strLength);
    if ( theError != noErr )
        gError->SevereMacError( theError );

    /**
        do PostScript printing
    **/
    PicComment(PS_BEGIN, 0, NIL_HAND);
    PicComment(PS_FILE, GetHandleSize( myPSHandle ), myPSHandle);
    PicComment(PS_END, 0, NIL_HAND);

    /**
        tell us (on the screen) that we've finished
    **/
    MoveTo(50, 50);
    DrawString("\pPostScript processing completed." );

    ClosePicture();
    /**
        store picture handle as instance variable
    **/
    inherited::SetMacPicture( myMacPicture );
    inherited::Draw( &picFrame );

}
```

The code in Listing 3-11 looks more like our first example, and for good reason. Once again, you are allocating a handle for some data, rather than including text with direct QuickDraw calls. You begin here by setting up a string that points to the file that you will use. You must substitute your disk name and path information for this string. This time the string is intended to be a Pascal string, so that the length must be increased by one to account for the length byte. Then the string pointer is changed into a handle as before. The comment sequence is just the same, with the single difference being that the length here is the length of the handle, as returned by the GetHandleSize routine. The remainder of the code is just the same and performs the same functions as before.

However, the PostScript is nowhere to be seen. It must be created as the resource of a separate file. Listing 3-12 shows you the RMaker code that you can use to create the actual file.

Listing 3-12. Code routine for PSRes193.make.r

```
PSRes193.rsrc
rsrcRSED

* resource declarations follow ...

TYPE POST = GNRL
    ,501              ;; must begin with res no. 501
.H                    ;; this is ASCII text comment
0100
.S                    ;;
0 760 translate

    ,502
.H                    ;; this is ASCII text comment
0100
.S                    ;;
1 -1 scale

    ,503
.H                    ;; this is ASCII text comment
0100
.S                    ;;
/Times-Roman findfont 20 scalefont setfont
```

Listing 3-12. Code routine for PSRes193.make.r (continued)

```
    ,504
.H                      ;; this is ASCII text comment
0100
.S                      ;;
150 650 moveto

    ,505
.H                      ;; this is ASCII text comment
0100
.S                      ;;
(Hello, world!!!) show

    ,506
.H                      ;; this is EOF
0500

    ,507
.H                      ;; this is ASCII text comment
0100
.S                      ;;
(Never see this!!) show
```

This introduces you to a new type of resource, the 'POST' resource. This resource type is specifically intended for storing and transmitting PostScript data to the LaserWriter, and the LaserWriter driver is familiar with this type of resource. The PicComment PostScriptFile (193) tells the driver to look for the given file name and to use the 'POST' resources in that file to send to the LaserWriter as PostScript. If the file or the resource is not found, the driver closes the file without any additional error message. (How about them apples??) The first two bytes of the 'POST' resource must be coded as indicated in Table 3-2. This table is a complete list of all the values that can be used in a 'POST' resource. It shows you the value of the first byte of the resource which tells the driver what to do with the remaining bytes of data; the second byte is always 0x00.

Table 3-2. Values used in 'POST' resources

Value	*Significance*
00	The following data in the resource is a comment and should be ignored.
01	The following data in the resource is ASCII text.
02	The following data in the resource is binary and should be converted to ASCII text before being sent.
03	The first part of the data is an AppleTalk end-of-file. If there is any further data in the resource, it should be sent as ASCII text after the end-of-file.
04	The data fork of the current resource file is opened and the data in it is sent as ASCII text.
05	This marks the end of the resource file.

These resources should be limited to about 2K bytes of data at maximum since the entire resource must be loaded into memory before being transmitted. You should not mix ASCII text and binary data in a single resource (note that there is no code for such a mixture). 'POST' resources must begin with 501, with each successive resource value incremented by one. The driver stops processing the 'POST' resources when it cannot find the next sequential number, or after it executes a 'POST' resource with the code 0x05.

Since there is no RMaker template for 'POST' resources, you use the GNRL type as a model in the PSRes193.make.r file. All the 'POST' resources here are ASCII text, with the first byte being coded in hexadecimal according to Table 3-3. Run RMaker on this file and you will generate a new file called PSRes193.rsrc, which has its file type and owner set so that you can open it with ResEdit. The file consists of nothing but a resource fork, with only the 'POST' type resources being displayed. If you open this with ResEdit, you will see the seven lines of 'POST' resources that you have defined, as shown in Figure 3-7.

Next, move the PSRes193.rsrc file to the volume and folder that you have coded into your example. In the code here, this file has been placed onto the desktop of the hard disk labeled "MacSE30 HardDrive"; in your code, move it to wherever you have coded it to be. Remember that if the driver can't find your file or has a problem finding the resource, you won't get any error; it just skips the comment data and continues processing.

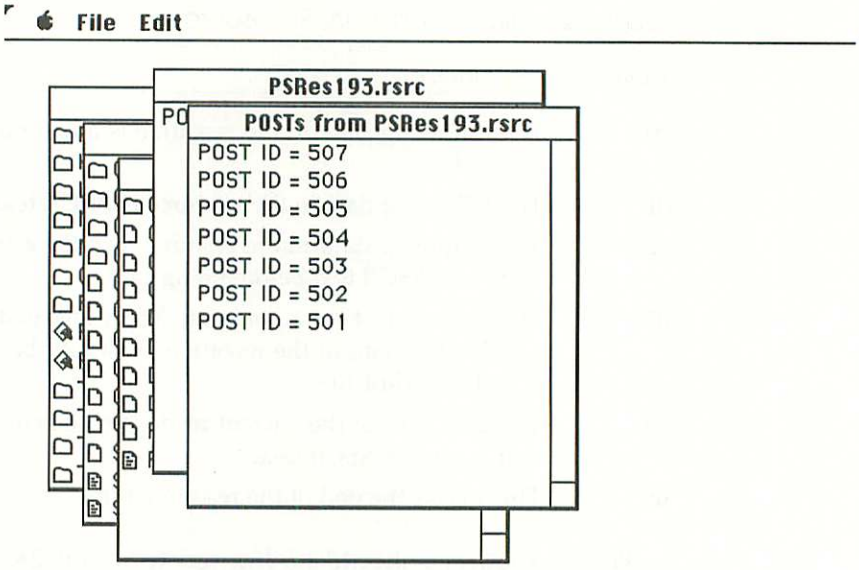

Figure 3-7. ResEdit display of PSRes193.rsrc

Note ▸

Obviously you should never use this coding technique in a real application! This is just a simple exercise so that you can test and execute such files. In a real application, you would use the File Manager to find the file, get the name, and so on. The problem with doing that here is that, since the LaserWriter driver must process the file at some point after the picture is created, you must have the full path name of the file for the driver to process. This means a lot of work, all about file handling and not about LaserWriter programming. So I didn't do it here.

In a normal application, you would want to test both that the file could be found and that the 'POST' resource 501 exists. However, in a normal application, you should never use this technique anyway because it is incompatible with Background printing.

You can now compile and run your application. You will see the usual screen output, as shown in Figure 3-8, and when you print, you should see output like that shown in Figure 3-9.

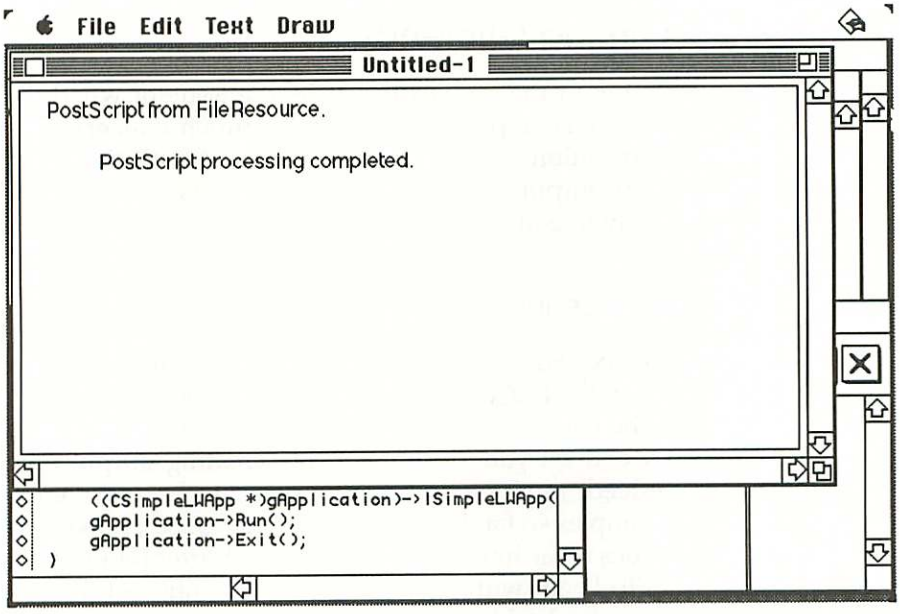

Figure 3-8. Screen output from PostScriptFile

PostScript from FileResource.

PostScript processing completed.

Hello, world!!!

Figure 3-9. Print output from PostScriptFile

If your printed output looks just like the screen, without any "Hello, world!!!", the driver couldn't find the file or the resource. Check that you have placed the file where you said it was, and use ResEdit to check that the resources are correctly entered into the file.

▶ Printing Efficiency

So far, in the application code, the issue of whether or not there was a LaserWriter present was not of much concern. Nevertheless, in any application you should be concerned with the speed and efficiency of your output, and there are some factors to consider when that output is likely to end up on a LaserWriter.

▶ Device independence

Device independence is the first issue. You have already seen that, by using the PicComment methods described here, you don't generally have to be concerned about what specific device your output will appear on. Of course, you are presumably creating output on a LaserWriter and, indeed, must be doing so in order to have gotten any printout of the examples so far. Under these circumstances, however, there are still some issues regarding the use of the LaserWriter for output.

To begin with, we have already mentioned the most important issue: that you disable Background printing on your system. You should also be connected directly to the LaserWriter; that is, you should not be sending your files to the LaserWriter through a spool/server device. Although the intention is to develop software that will, in fact, run in such situations and with the standard background or spool/server implementations, the fact is that, during the development process, such printing complications only slow down your processing and add potential problems to your programming and debugging. Sending files to the background Print Monitor or to a spool/server device slows down the printing process quite measurably, particularly for short jobs such as those created in these exercises. For ordinary users, this slowdown is offset by getting their application back as soon as the spooling process is complete, so that they don't have to wait for printing to complete. In your case, however, you will still be waiting since you need to see the results (or lack thereof) in order to determine whether your code is working correctly or not. Also, sending the print file through another handling program loses some of the information on errors and processing problems that you would like to see. Again, for average users, there isn't much that they can or should do to provide for these situations anyway. But you, as the programmer, can make changes and perform alternative processing to correct and recover from errors; honestly, if truth be told, you have probably caused the error. Therefore, you want to have direct connection with the LaserWriter so that you can get, first hand, any feedback or error information that is available.

Of course, some of these techniques won't work in such printing environments. We have already discussed that the PicComments PostScriptFile and ResourcePS are not available when Background printing is enabled. A little thought indicates the problem. In both of these cases, the information is being drawn from a resource file. The main problem here is that the user could, unwittingly, move or rename the resource file before the PrintMonitor can open it. This would obviously be a disastrous error, and, what is worse, would be one almost impossible to explain to the unfortunate user.

Note ▸

If you attempt to use these techniques when Background printing is enabled, you will get a System error –8130. As far as I can tell, this code is undocumented in any of the standard references; however, it always occurs if you try to use these PicComments with Background printing, and it never occurs otherwise. I therefore think it safe to say that this indicates that you made a mistake and should disable Background printing before running your job again. If you want to test this, enable Background printing and run the preceding exercise that uses PicComment 194 (PostScriptFile) with the Debugger on. You will get the standard Think C OSErr dialog alert, with the error code –8130.

You may wonder if you could not test for Background printing and simply avoid using these techniques when it is enabled. That's a good thought, but not practical. Technical Note #192 tells you that it is not possible to determine whether Background printing is enabled or not. Since you cannot determine what the current status is from within an application, you must not rely on these methods in most circumstances. For our exercises here, of course, you can easily and quickly set Background printing off and determine the state of the LaserWriter from your system. Therefore, this is quite acceptable for the exercise environment. Since you cannot control the user's environment, however, you should avoid these techniques in a general application.

Another issue to be concerned with, although not so important, is the level of the LaserWriter driver that is being used with your application. The PostScript PicComments are only available in the LaserWriter driver Versions 3.0 or later. Since this book assumes that you are using at least LaserWriter 5.2 or later, this isn't a major issue. Even in the real world of application programming, there are few systems these days that aren't past LaserWriter 3.0. In Chapter 7 you will learn how to test

for the LaserWriter driver and how you might test for version identification information. In any case, for this book be sure that you have (and are running!) at least Version 5.2 of the LaserWriter driver and the matching Laser Prep file.

▶ Optimizing printing functions

The previous section talked about using the LaserWriter. This section discusses some things to avoid when you are writing picture code that may use the LaserWriter as an output device. Note that many of these concerns are not specifically tied to whether you use the PicComment enhancements or not; these are general concerns that any application should be careful about. After all, with so many LaserWriters out there, you have to assume that at least some of your users will have them. If they do, following the advice here will make your application better and more efficient.

Here is a short list of things that you can do in QuickDraw that are not translated into PostScript by the bottlenecks. Since these are not translated, you should avoid them in code that may be printed on a LaserWriter.

- The XOR and NotXOR transfer modes are not supported.
- The grafverb Invert is not supported.
- Regions are not supported; use polygons or bitmaps to simulate regions if required.
- Clip regions should be limited to rectangles.
- There is a small difference in character widths between the screen representation of a font and the printer version due to the difference in resolution on the two devices. For this reason, the spacing between characters may be different on the two devices, although the endpoints of text strings will be the same.

There is also a small difference relating to older versions of the LaserWriter drivers. Release 1.0 of the driver (very ancient, at this point) does not support the QuickDraw call SetOrigin between PrOpenPage and PrClosePage calls; instead you must use OffsetRect. This is fixed in the later drivers.

Bitmaps are also affected to some degree by these differences in resolution. In particular, the ratio of the LaserWriter resolution to the screen resolution is 300 to 72, or 4.17. By choosing Precision Bitmap Alignment from the Options dialog selection in the style (Page Setup...)

dialog box, you make the driver use a ratio of exactly 4, which is both more precise and faster to use. It also, however, distorts the image by making it smaller by about 4% (the difference between the 4.17 magnification and 4).

You should consider some additional issues in order to speed up your printing. Basically, some approaches to printing and some types of printing activities cause measurable slowing in the printing process. So that your user won't think your application is related to Rip Van Winkle, avoid these techniques.

- Avoid erasing—On a screen, it is usually essential that you erase the area that you're working in to make a white background for text, graphics, or whatever. Since the LaserWriter works on a clean sheet of white paper, this isn't necessary. Moreover, making the driver erase an area takes a substantial amount of time. Generally, therefore, you should avoid any use of the QuickDraw erase verbs like EraseRect or EraseOval.

  Notice that TextBox (in TextEdit) performs an implicit call to EraseRect to clear the area before drawing each line of text. The result is quite slow on a LaserWriter. If you must print text in rectangles, consider using standard QuickDraw text methods such as DrawString or DrawText instead of TextBox.
- Avoid using individual DrawChar—Using DrawChar to place every character on the page is a waste of time and effort. As mentioned earlier, the characters on the printer and those on the screen differ slightly; the driver adjusts the printed text to achieve the best character spacing. If you try to place characters individually, you will conflict with the Printing Manager and the resulting line will look worse than it would if you had left it alone. In addition, you take a double hit in the efficiency department. First, each character must be sent through the bottlenecks separately, making your file longer than it would otherwise be and taking substantially more time to process on the Macintosh. Second, the PostScript interpreter must follow your instructions for each character, which involves going through the processing loop multiple times for each character, at an additional overhead in processing time. This is really a *bad idea*—please don't do it!
- Use standard patterns—Printing patterns have a lot of overhead associated with them. The standard patterns are optimized so that the bottlenecks take advantage of the PostScript **setgray** operator to translate black, white, and all gray patterns into shades of gray, which is very efficient. If you use other patterns,

you will find that they take significantly longer to reproduce. They will also be produced at the screen resolution (72 dpi), rather than at the full resolution of the output device.

- Keep the clipRect within rPage—If your clipRect in the printing grafPort falls outside of the rPage rectangle, calculating the intersection becomes very time consuming. To avoid this, make sure that clipRect falls entirely within rPage before printing; this improves the print speed by a factor of 4 or so.

Technical Note #72 suggests that you limit the number of fonts that you use to speed up printing. Well....certainly, the more fonts you have in a document, the longer it is likely to take to print. Actually, the biggest problem with a lot of fonts in a document is exhausting the virtual memory in the printer. Also, using a lot of fonts that are not already resident in the printer causes you to lose time as the driver downloads all of them. With those constraints out of the way, however, there isn't much to be gained in later versions of the LaserWriter printers by limiting font usage in a document. There are so many variables that giving a general rule regarding font use and efficient printing becomes almost impossible. Honestly, the PostScript font handling is quite efficient, particularly when you have plenty of memory or a hard disk to cache the fonts. Since the font cache is extremely efficient and the rendering algorithms are quite fast, any benefit from limiting the fonts in a document is likely to be minuscule, at best. (Now, if Apple would just figure out some way not to have to reencode the fonts all the time so that you could use the idle time font caching, that would really help.)

Another issue that is of some concern is the issue of applications using the spool-a-page/print-a-page philosophy. This is done to minimize the disk or memory requirements where spool printing is being used. Although efficient for an ImageWriter, this approach is both inefficient and ineffective on the LaserWriter. However, the CPrinter class does not do this, so you don't need to be concerned about it here. If you are writing your own print code, be sure to review Technical Note #72 regarding this issue.

▶ Resource management issues

Another concern that you should have involves managing LaserWriter resources, particularly virtual memory. Chapter 2 discussed how PostScript objects consume virtual memory in your LaserWriter. If you use too many procedures or download too many fonts, you run the risk of running out of memory, which is even more devastating in the LaserWriter than it is in the Macintosh, if you can believe that.

The main issue is how to manage virtual memory in the printer. The PostScript language provides a simple, and rather coarse, memory management tool; it's not elegant, folks, but it works—and it's the only game in town. This is implemented by the pair of PostScript operators **save** and **restore**. Its process works like this. You issue the **save** operator at certain logical points in your program, for example, at the start and every time you begin to compose a page of output. The **save** returns a *savestate* object to the operand stack, and you would normally name the object and save it in a dictionary. Then, whenever you are finished with some part of the processing and can remove that from the memory, usually because the processing is completed and the resulting image has been transferred to the output, you issue a matching **restore**, using the *savestate* that you had created earlier. This restores the memory, and all composite objects, to the state that they had at the moment of the **save**. This recovers all the virtual memory that you have used since the **save**, but it also removes all font definitions (for example), changes the current path back to what it was, removes all definitions that you may have made in the meantime, and so on. It is the very coarseness of this method that encourages you to use it at well-defined states of the page output, such as in between pages. Such planning keeps to the minimum any chance of losing something that you wanted or needed to keep for further use, like procedure definitions.

In fact, the current version of the Laser Prep bottlenecks does exactly this when you call it with the PostScriptBegin comment, and it restores to the state that it saved at the beginning of the comment processing when it reaches the PostScriptEnd comment. You, too, may want to use this mechanism to ensure that you don't gobble up too much memory. You will see some examples later in this book.

Downloaded fonts take up probably the most room in a typical PostScript document. Fonts are large, and many fonts are often used in a document. Here is where it behooves the user to be quite careful about errors because running out of virtual memory not only practically ensures that you will have a PostScript error, but also often causes the entire device to reset itself, with the loss of all information and fonts that had been loaded up to that time. Really annoying!

You may have noticed the Unlimited Downloadable Fonts in a Document check box in the Options from the style dialog box. Now you can begin to see how this works: It surrounds every use of a nonresident font with a **save**, **restore** pair of operators, which recovers the memory that the font used—at the cost of some processing time and some additional download time.

The LaserWriter driver is always looking to save you additional virtual memory if it can. As you remember from Chapter 1, much of the memory in the LaserWriter is being taken up by the frame buffer, which holds the image of the page as it is produced. One way, then, to get more memory is to reduce the size of the page on which you are working. The LaserWriter driver actually does this, setting the smallest page size that is available for the paper that you have specified in the style dialog box. If, in fact, you are not going to use very many fonts in your document, you can give up some virtual memory and get a larger page size, including much smaller margins. This is done by checking the Larger Print Area check box in the Options from the style dialog box. Chapter 5 explores this option in some detail to see how the page size changes.

▶ Permanent and temporary downloading

One aspect of memory management that you really cannot control is the use of the Laser Prep file. Obviously the Laser Prep procedures (also known as the **md** dictionary, you remember) must be present for any ordinary Macintosh document to print. How this is provided is an important issue, for two reasons. First, you need to understand this to truly control and use the procedures from your application. Second, since many other dictionaries and fonts can also be loaded in this same way, understanding this process will help you make judgments about other options as well.

Chapter 2 discussed the server loop and how a PostScript program could exit the server loop to leave information behind after the job was done. Recall that the server cleans up after most jobs and removes any leftover information when the job terminates. Any PostScript file may be executed outside the server loop, and, if it is, the information that it creates is left in the printer until it is turned off or until it is reset (which amounts to the same thing). In PostScript jargon, this is called *permanent downloading*, whereas the ordinary type of job processing is called *temporary downloading*. Note here that permanent downloads are only permanent until the device is turned off or reset; only a few special PostScript parameters are preserved when the device is turned off.

An application can exit the server loop by starting with the following sequence of instructions.

```
serverdict begin 0 exitserver
```

The 0 in that line is the password for your LaserWriter. The idea behind the password was to protect the device from tampering and to allow the interpreter to catch any errors before they could cause real problems or damage. Unfortunately Apple decided to load the Laser Prep permanently when you first print. The net result, of course, is that any other password will cause you some problems with downloading Laser Prep and running Apple software.

Note ▸

In System 7.0, this is no longer true. With System 7.0, the Laser Prep file is downloaded every time a document is printed. This takes a little more time than the process of not downloading, but it saves virtual memory and eliminates many complications that arise from having Laser Prep permanently resident on the LaserWriter. This is an improvement that should have happened sooner!

The same concerns apply to any personal dictionaries that you write. You should, generally, send the dictionary with the document and not download it once, outside the server loop. (Chapter 5 discusses exactly how to do this.) However, some applications provide for both alternatives in their code, and you might wish to consider this as well.

Fonts also can be permanently downloaded to the printer by using a utility such as Apple's LaserWriter Font Utility. This allows you to download a font to the printer's memory (that is, outside the server loop) or to an attached hard disk if one is available. For fonts, this option makes more sense. Fonts may get downloaded not once but many times, and fonts can be large and cumbersome. This option makes the most sense in an environment with just a few nonresident fonts that are used by multiple workstations in many documents. In this case, the savings can be substantial and the loss of flexibility is quite small. However, in the average environment, permanent downloading makes little or no sense.

▶ Conclusion

This has been an important chapter. It has introduced you to a variety of issues concerning LaserWriter programming. The fundamental thrust of this chapter has been to show you the structure of the ordinary printing process and how, using the Printing Manager, your application produces printed output. This is a very complex issue, and there is much more to be learned about it than has been covered here. Never-

theless, you have seen most of the various issues and concerns that arise when printing to a LaserWriter.

You have also had your first experience in accessing the LaserWriter directly and sending programs in the PostScript language. Creating PostScript programs is one of the most interesting activities that you can do when you have a LaserWriter as an output device. A world of graphic opportunities is available to you when you can tap the full versatility of the LaserWriter and PostScript, and I think that you will enjoy seeing how these work. In Chapter 4, you will learn more about the PostScript language and how it can be used; in Chapter 5, you will learn more details about using the Printing Manager and altering the way it behaves.

4 ▶ PostScript Program Construction

▶ Chapter Overview

This chapter gives you a quick overview of how to use PostScript. Let me emphasize that this is not intended as a tutorial on the PostScript language; that is a book in itself. Instead, this is an introduction to some practical coding techniques that you can use to build and test PostScript code. In the examples presented here, I have tried to explain the code that is used, both in C and in PostScript. However, the explanations, especially of the PostScript code, are rather terse and superficial. If you don't follow how the code works, or you have any other questions about the use of the operators and so on, I would suggest that you get a good primer on PostScript programming and that you have a PostScript language reference available. A list of reference books is provided in the Bibliography.

▶ Testing with PostScript

To learn PostScript, it is, naturally, very important that you be able to code with PostScript and so be able to learn by doing. As discussed in Chapter 1, one way to do this is by using the interactive mode of the PostScript interpreter in your LaserWriter. The drawback is that this requires either reconnecting your LaserWriter by the serial port and providing communication software, or purchasing specialized software that allows interactive access over AppleTalk. There is an alternative that, if not quite as good as the interactive facilities, does allow you to work with PostScript code and learn something about how this language

works, using the tools and techniques that you already have available. Let's look at how you might do this, and what the drawbacks and limitations of this method are.

▶ General example structure

Now that you have learned about using the PicComment facilities to provide PostScript access from your own application, you can easily work out an alternative that provides simple access to PostScript without any new software. You use the same **Draw** method that you have just completed, but with slightly different 'POST' resources. Here you code two 'POST' resources that allow you to enter and test PostScript code within your own application.

Recall the 'POST' resource types that you learned in Chapter 3. You only used type 1 because that was all you needed; however, there are several other types. In this case, you are going to use 'POST' resource types 4 and 5 to send your PostScript data. Type 4 tells the driver to look in the data fork of the file and send whatever is there as ASCII PostScript code. Type 5 tells the driver that you are through. Use the RMaker code shown in Listing 4-1 to create the necessary file.

Listing 4-1. Code for PSResTest.make.r

```
PSResTest
TEXTttxt

* resource declarations follow ...

TYPE POST = GNRL
    ,501                ;; must begin with res no. 501
.H                      ;; take PostScript from data fork
0400

    ,502
.H                      ;; this is EOF
0500
```

As you did in the earlier example, this creates a file with the given resources. Here, however, you see that you have set the file type to TEXT and the file's creator to ttxt, which is the identification of the Apple TeachText application. If you run this resource file through

RMaker, you get a file named PSResTest, which can be opened with TeachText. Either double-click on the file to start or open the file from within TeachText manually. In either case, you will see an empty TeachText file. Edit this file with TeachText and add the code, as shown in Figure 4-1.

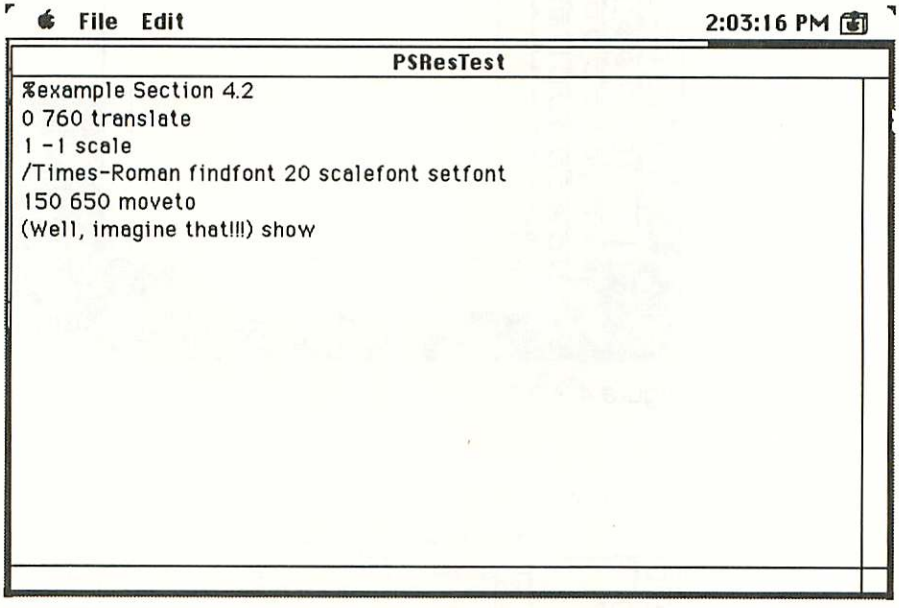

Figure 4-1. TeachText window with first example code

The code in Figure 4-1 is a simple variant on the code you produced earlier; it again simply shows the given string on the output page. Close and save this file. Open ResEdit and open the PSResTest file. You should see a single 'POST' resource in the resource fork of the file. Open that resource and you will see two 'POST' resources, as shown in Figure 4-2. As you see, these two resources are correctly numbered 501 and 502, as required by the 'POST' resource processing. Then open resource 501, as shown in Figure 4-3.

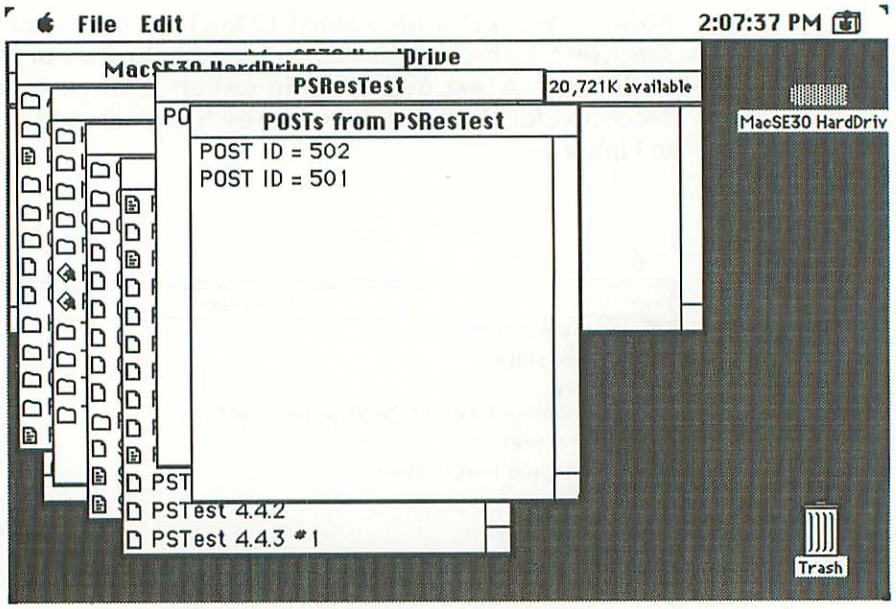

Figure 4-2. 'POST' resources in the PSResTest file

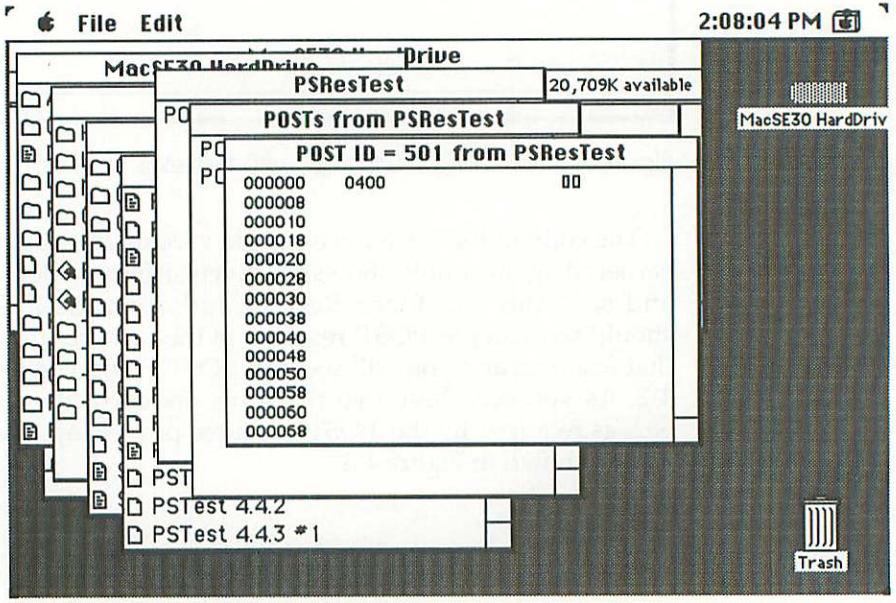

Figure 4-3. 'POST' resource 501

All that this resource contains is a single value, 0x0400, which tells the Printing Manager to send the information stored in the data fork of the file as ASCII text. This is exactly what you want. This sends the code that you just added, using TeachText, as PostScript to the LaserWriter. Anything that you type into this file is sent as PostScript ASCII text to the printer and executed. In the same way, resource 502 only consists of two bytes. In this case, the control value is 0x0500, indicating that this is the last 'POST' resource to be sent to the printer. Effectively, this terminates the 'POST' processing and allows the Printing Manager to complete the picture that it is processing. Then it can print the file for you, using the standard QuickDraw bottlenecks from Laser Prep.

Now let's set up the necessary **Draw** method to use our neat file. The code to do this is shown in Listing 4-2.

Listing 4-2. Code for **Draw** method

```
/***
 * Draw {PicComment 193 Method}
 *
 *  In this method, you draw whatever you need to
 *  display in your pane. The area parameter gives the
 *  portion of the pane that needs to be redrawn.
 *  Area is in frame coordinates.
 *
 ***/

void CSimpleLWPane::Draw(Rect *area)

{
    char        *theString;
    char        *strPtr;
    Handle      rsrcHandle;
    long        strLength;
    OSErr       theError;
    Ptr         rsrcPtr;
    Rect        picFrame;
    struct
    {
        ResType     resType;
        int         resID;
        int         resIndex;
    }           rsrcData;
```

Listing 4-2. Code for **Draw** method (continued)

```
picFrame = *area;
myMacPicture = OpenPicture( &picFrame );
ClipRect( &picFrame );

/**
    first draw the basic rectangle
**/
InsetRect( area, 2, 2 );
FrameRect( area );

/**
    tell us (on the screen) what's happening
**/
TextFont( helvetica );
MoveTo(20, 20);
DrawString("\pPostScript from FileResource.");

/**
    set up PostScript comment data
    DON'T DO THIS IN A REAL APPLICATION!!
**/
theString = "\pMacSE30 HardDrive:PSResTest";
strLength = theString[0] + 1;
strPtr = theString;
theError = PtrToHand( strPtr, &myPSHandle,
strLength);
if ( theError != noErr )
    gError->SevereMacError( theError );

/**
    do PostScript printing
**/
PicComment(PS_BEGIN, 0, NIL_HAND);
PicComment(PS_FILE, GetHandleSize( myPSHandle ),
myPSHandle);
PicComment(PS_END, 0, NIL_HAND);

/**
    tell us (on the screen) that we've finished
**/
```

Listing 4-2. Code for **Draw** method (continued)

```
    MoveTo(50, 50);
    DrawString("\pPostScript processing completed." );

    ClosePicture();
    /**
        store picture handle as instance variable
    **/
    inherited::SetMacPicture( myMacPicture );
    inherited::Draw( &picFrame );

}
```

This is essentially the same technique that you used earlier for PostScript from a file resource; indeed, it is PostScript from a file resource. The only change here is in the name of the desired file and in the 'POST' resource values within the resource itself. To run this, move the PSResTest file out onto your desktop or wherever you have coded the location to be. Then execute the file as before. You should see a screen like Figure 4-4. Now select Print from the File menu. You should see a page like Figure 4-5.

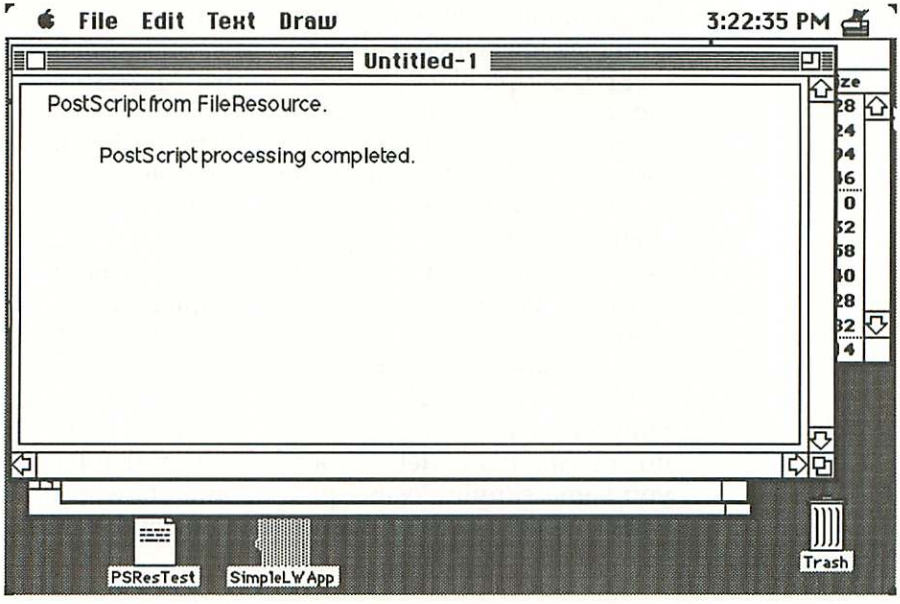

Figure 4-4. Screen output from the first PSResTest

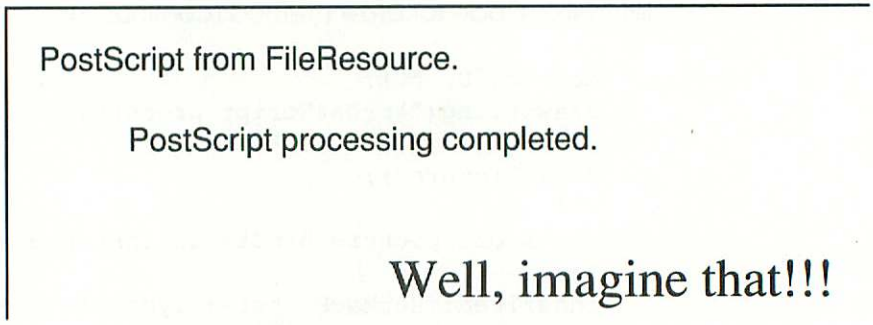

Figure 4-5. Page output from the first PSResTest

▶ Debugging and error handling

So, if this is so easy, why doesn't everyone do it? Well, there are major drawbacks to this process, as you may have already found out. To begin with, as was discussed in Chapter 3, this technique only works with Background printing disabled. Next is the problem of PostScript errors. If you didn't get the desired output, what happened? Probably nothing! In fact, if you didn't get any output, you most likely have some coding error in your PostScript code. Check your code against Figure 4-1 *exactly*. In particular, make sure that there is a blank line at the end of the data; this is necessary so that your code does not become mixed up with the QuickDraw bottlenecks. All you Pascal folks should remember that PostScript, like C, is case sensitive. Make sure that you have exactly the same code and try, try again.

This discussion points up and illustrates the biggest drawback to this form of PostScript programming. If you have an error, you can't easily tell what it was. The PostScript interpreter, in fact, sends a simple but useful error message back to the Printing Manager, but if the application doesn't test for it (as it doesn't here!), the message, and even the fact that there is an error, is lost. Chapter 5 shows you how to capture these errors and let your users know that something went wrong. For now, you just have to be careful.

The second problem is related to the first, but it is more subtle. Suppose that you suspect—or even *know*—that you have an error. How do you find it and debug your code? This chapter is going to both teach you some simple PostScript code and show you how to see some of what's going on behind the scenes. Look and learn PostScript.

Important ▸

For the rest of this chapter, you will be creating and running small PostScript programs. To do this, simply open the PSResTest file with TeachText, erase the previous code, and enter the code shown in the examples. Then you have two choices: Run the project as you did earlier, using the Debugger or not as you choose, or make and run a standalone application. Personally, I found the latter form to be both faster and easier. I will show the expected print output (but not the screen, which will be the same) for each test. I will not repeat these instructions every time since each test is exactly identical except for the PostScript code. The important point here is that there are no more changes to the C code for this chapter.

▶ Creating Procedures

The first topic that you need to become acquainted with is the use of PostScript procedures. Like all computer languages, PostScript uses procedures to allow you to do repetitive tasks or to encapsulate complex operations into a single routine. Such procedures are an important part of PostScript programming. Here, you focus on rather simple procedures that test and print various status data from the interpreter and the PostScript objects that it controls. This teaches you to use procedures in PostScript and teaches debugging techniques that you can use as you do other programs.

▶ Procedure definitions

Recall that procedures are collections of PostScript commands enclosed in braces, { and }. They are defined like any other object, with a name literal and the **def** operator. Once defined, the procedure can be invoked anywhere in the program simply by using its name, as long as the dictionary that contains the definition is on the dictionary stack. (If you have any questions, review Chapter 2 on PostScript language concepts.)

With this short refresher, let's look at our next example for you to use. The complete code is shown in Listing 4-3.

Listing 4-3. Code routine for PSText1

```
% example PSText1
% procedure variables
/LeftMargin 150 def
/LineLead 14 def

/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord --
%no requirements
{
    0 760 translate
    1 -1 scale
}
bind def

/newline
%procedure to advance a line
%called as:     -- newline --
%requires:  LineLead (for leading value)
%           LeftMargin (for margin)
{
    currentpoint
    LineLead sub
    exch pop
    LeftMargin
    exch
    moveto
}
bind def

%now run a test of this
restoreCoord
/Times-Roman findfont 12 scalefont setfont
150 650 moveto
(Hello, world - one time!) show
newline
(Hello, again - two times!!) show
newline
(Third time's a charm!!! - Goodbye, now.) show
```

This example defines two procedures, **restoreCoord** and **newline**. The procedure definitions begin with the name literal—/restoreCoord or /newline—and end with the line **bind def**. As a matter of convenience, I always start procedure definitions with some comment lines that tell what the procedure does, how it is called, and which variables or other procedures it requires to work correctly. The procedure commands themselves are contained between the two braces and are indented for easier viewing. All the procedures here are bound, using the **bind** operator, before they are defined. This speeds up the operation of the procedure.

The **restoreCoord** procedure returns the QuickDraw coordinates to PostScript coordinates. You saw in Chapter 3 that this is usually necessary and, since it is an operation that you will perform in each of these exercises, you should have a procedure that performs this task. The procedure itself is identical to the sequence of steps that you have used before.

The **newline** procedure advances to a new line. It does this by using two constants, **LeftMargin** and **LineLead**, which give the position of the left margin and the distance between the lines. The procedure gets the position of the current point and subtracts the distance for the next line—remember that in PostScript coordinates, unlike QuickDraw coordinates, the y-axis starts at the bottom of the page and is positive upwards. Therefore, moving down the page requires subtracting from the existing y coordinate, rather than adding to it. Then the old x value is removed from the stack and replaced by the fixed value, **LeftMargin**. Finally, the procedure moves the current point to the new location, thus effectively moving down a line.

The body of the code simply sets the font, as you have seen before, moves to the beginning point for the display, and shows three lines of text. All quite straightforward. When you execute this, you should get the output shown in Figure 4-6.

PostScript from FileResource.

PostScript processing completed.

Hello, world - one time!
Hello, again - two times!!
Third time's a charm!!! - Goodbye, now.

Figure 4-6. Page output from example

This code shows you how to define procedures and execute them within your PostScript program. However, it doesn't provide much of the promised help with internal information from the interpreter. Therefore, let's modify these procedures a bit and do another example that shows you how to get some internal information down on paper. Look at the code in Listing 4-4.

Listing 4-4. Code routine for PSText 2

```
% example PSText2
% Global variables
/myStr 255 string def
/LeftMargin 50 def
/TopMargin 650 def
/LineLead 12 def

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord --
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/newline
%procedure to advance a line
%called as:     -- newline --
%requires:  LineLead (for leading value)
%           LeftMargin (for margin)
%           TopMargin (for initialization)
{
    {currentpoint}
    stopped
    {
        LeftMargin TopMargin moveto
    }
```

Listing 4-4. Code routine for PSText 2 (continued)

```
    {
        LineLead sub
        exch pop
        LeftMargin
        exch
        moveto
    }
    ifelse
}
bind def

%now run a test of this
restoreCoord
/Times-Roman findfont 10 scalefont setfont
newline
(Current Memory Use is: ) show
newline
vmstatus
(Maximum memory installed: ) show myStr cvs show
newline
(Current memory used: ) show myStr cvs show
newline
(Current save level is: ) show myStr cvs show
```

This example uses the same two procedures, but each of them has been slightly modified. The **restoreCoord** procedure now includes a **newpath** operator. The conversion from QuickDraw to PostScript coordinates leaves the current point (and possibly other path elements) still in the document. To ensure that you start from a clean path, you issue a **newpath** here to remove any elements that have been left behind.

The **newline** procedure also now has a new feature. It uses the **TopMargin** variable to initialize the position if there is no current point already defined. Since you have done a **newpath** in the **restoreCoord** procedure, the first time around there will not be any current point. In that case, the **currentpoint** operator reports an error, which is caught by the **stopped** operator. If an error occurs, the **stopped** operator places **true** on the stack; in this case, this executes the procedure code within the braces after the **stopped**

```
LeftMargin TopMargin moveto
```

which moves to the predefined position given by these two constants. In this way, you can use **newline** without any concern about whether the current point is already set or not. If the current point is set, the **newline** procedure moves down one line from that point; if it is not set, the **newline** moves to the point given by (LeftMargin, TopMargin).

The body of the exercise gives you a new operator, **vmstatus**. This operator returns three values to the operand stack: the maximum memory available in your device; how much memory you have used so far; and the number of unmatched saves (called the *save level*) that are presently outstanding. The exercise uses the global string variable **myStr** as working storage for the **cvs** operator, which converts these numbers into string values for printing. The exercise simply prints the three values on the sheet, as shown in Figure 4-7.

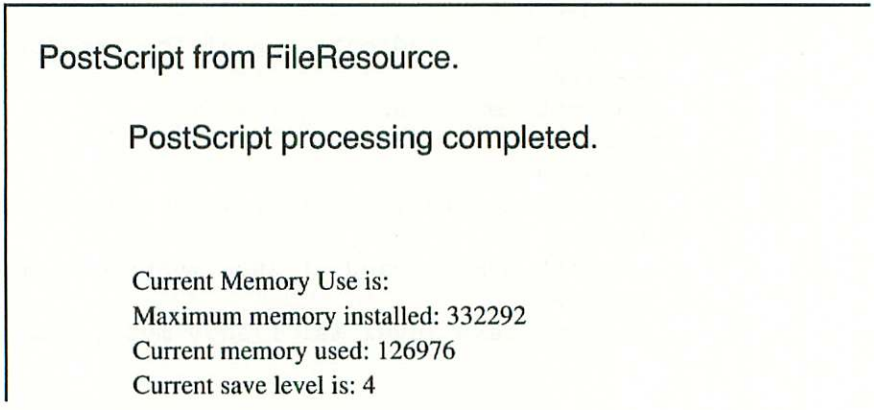

Figure 4-7. Page output from example

Remember that the values reported will vary from device to device, and they will vary depending on the internal state of the device itself. Therefore, your numbers are likely to be different from those shown in Figure 4-7.

Obviously there is much more to procedures than these short examples can show. This does give you some flavor of how procedures are created and used in PostScript and how you might use them yourself in a program.

▶ Procedures and dictionaries

You now need to look at how you can provide your own set of procedures in your own dictionary. This is a common process in PostScript programming and is one that is highly recommended. Let's look at the example code in Listing 4-5 to see how you can do this.

Listing 4-5. Code routine for PSText 3

```
% example PSText3
/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables
/myStr 255 string def
/LeftMargin 50 def
/TopMargin 650 def
/LineLead 12 def

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord --
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/newline
%procedure to advance a line
%called as:     -- newline --
%requires:  LineLead (for leading value)
%           LeftMargin (for margin)
%           TopMargin (for initialization)
{
    {currentpoint}
    stopped
    {
        LeftMargin TopMargin moveto
    }
```

Listing 4-5. Code routine for PSText 3 (continued)

```
    {
        LineLead sub
        exch pop
        LeftMargin
        exch
        moveto
    }
    ifelse
}
bind def

%now some tests for dictionaries
restoreCoord
/Times-Roman findfont 10 scalefont setfont
newline
(Current dictionary values are:) show
newline
(Maximum size (key, dict pairs) is: ) show
currentdict
dup maxlength myStr cvs show
newline
(Current size (key, dict pairs) is: ) show
length myStr cvs show

newline newline
(userdict dictionary values are: ) show
newline
(Maximum size / Current size: ) show
userdict maxlength myStr cvs show
( / ) show
userdict length myStr cvs show

newline newline
(Laser Prep dictionary values are: ) show
newline
(Maximum size / Current size: ) show
md maxlength myStr cvs show
( / ) show
md length myStr cvs show

end     %SimpleLWdict
```

The procedures and global variables here are identical to those in the last example, but some code is added to the top of the example to define and use our own dictionary, called **SimpleLWdict**. The dictionary is defined to hold a maximum of 10 items. All of the procedure definitions and global variables are now defined in this private dictionary instead of being defined in **userdict** as they were earlier.

The body of the example shows you new operators, **length** and **maxlength**, that tell you how many entries are contained in a dictionary. Each dictionary has a maximum number of entries that it can hold—for **SimpleLWdict**, for example, that number is 10—which is reported by the **maxlength** operator. Exceeding the maximum number of entries causes a PostScript error. The **length** operator, on the other hand, reports the number of items currently defined in a given dictionary. Knowing these two numbers tells you a lot about a dictionary. The code in the example determines these two values for three dictionaries: our own **SimpleLWdict**; the internal dictionary, **userdict**; and the Laser Prep dictionary, **md**, which is downloaded by the LaserWriter driver. The results of this example are shown in Figure 4-8.

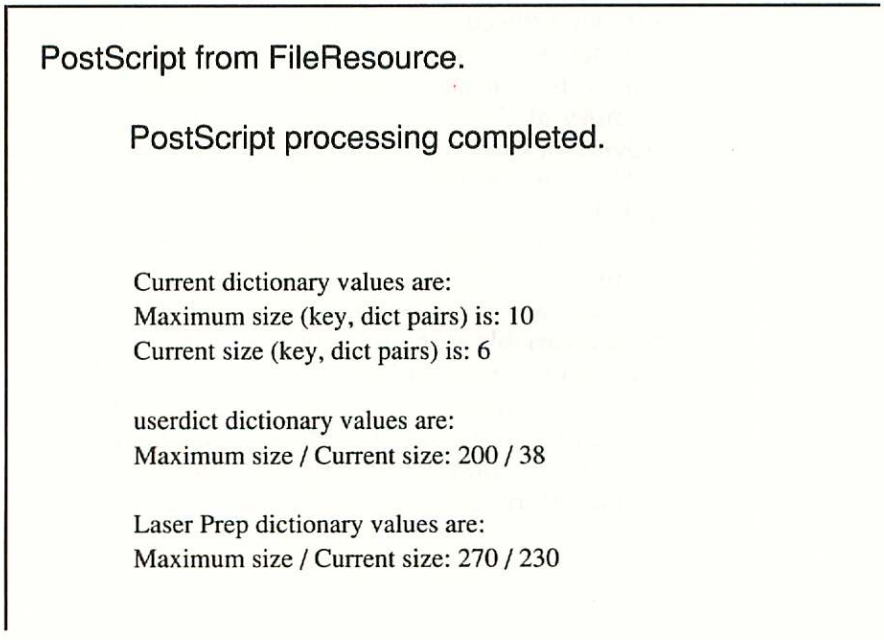

Figure 4-8. Page output from example

The code ends with a new operator, **end**, which removes our **SimpleLWdict** dictionary from the dictionary stack. As before, the exact values for **userdict** and **md** may vary from device to device; don't be concerned if your numbers are different from those shown here. The numbers for **SimpleLWdict**, on the other hand, should be exactly the same (unless you have changed something) because it was created and used here.

In addition to using your own dictionary for processing, you may want to provide your own memory management as well. This is not strictly necessary since the Laser Prep routines that begin and end the PostScript comments in your picture automatically provide similar processing. However, there are times and circumstances when you need to handle this yourself, and you should be able to determine both how much memory you have used and how you can retrieve that memory if required. The next example shows you the procedure for this and discusses some consequences of using these operators.

First, let's look at the code in Listing 4-6. This code is again identical to the last example as far as dictionary definition, variables, and procedures are concerned. The intention here is to determine how much room you use in this code segment, and, to do that, you begin by retrieving the current status of memory at the beginning of the program and storing these values into **userdict** as the **Vmax**, **Vused**, and **SVlvl** constants. Then you use the **save** operator to save a picture of the memory at this point. This is stored as the **_mySave** constant. This provides a known point to return to at the end of your processing.

The body of the example displays the current use of virtual memory and the previous use, and then subtracts one from the other to show how much memory these routines use. Notice that it saves the amount in current use by a **dup** operator, which then remains on the stack for later use as one of the operands to the **sub**. Using the stack for temporary variable storage is good programming because it allows you to save information without having to explicitly name and store it in a dictionary. The only drawback, of course, is that you must keep track of the number and position of the stack elements at all times—but, if you're going to program in PostScript, you have to do that anyway. The results of your efforts should look something like Figure 4-9.

Listing 4-6. Code routine for PSText 4

```
% example PSText4
vmstatus
    /Vmax exch def
    /Vused exch def
    /SVlvl exch def
/_mySave save def

/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables
/myStr 255 string def
/LeftMargin 50 def
/TopMargin 650 def
/LineLead 12 def

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord --
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/newline
%procedure to advance a line
%called as:     -- newline --
%requires:  LineLead (for leading value)
%           LeftMargin (for margin)
%           TopMargin (for initialization)
{
    {currentpoint}
    stopped
    {
        LeftMargin TopMargin moveto
    }
```

Listing 4-6. Code routine for PSText 4 (continued)

```
    {
        LineLead sub
        exch pop
        LeftMargin
        exch
        moveto
    }
    ifelse
}
bind def

%now some tests for memory
restoreCoord
/Times-Roman findfont 10 scalefont setfont
newline
(Previous memory was: ) show
newline
(Maximum: ) show Vmax myStr cvs show
(, Used: ) show Vused myStr cvs show

newline newline
(Current memory is:) show
newline
vmstatus
(Maximum: ) show myStr cvs show
dup
(, Used: ) show myStr cvs show

newline newline
(Amount used by my code is: ) show
Vused sub myStr cvs show
pop

end     %SimpleLWdict
_mySave restore
```

The last line of the code returns memory to the state that it was at the moment the matching **save** was done, when the **_mySave** dictionary was created. What that means is that you have retrieved all the memory used by your routines, as shown in the printed output. The cost of this,

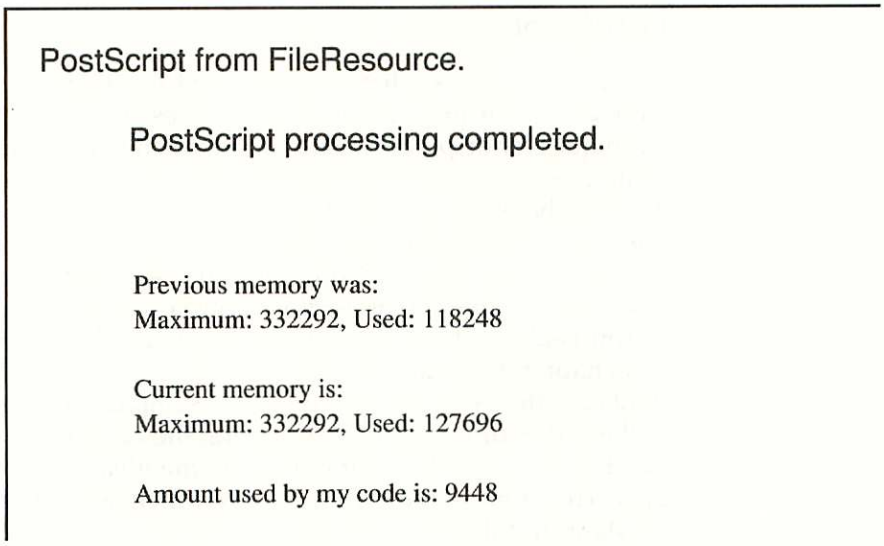

Figure 4-9. Page output from example

however, is that all your procedures and variables, and even your **SimpleLWdict** dictionary, are now erased from memory and are no longer available.

This raises an important point. This technique is essentially the same one that Laser Prep uses to bracket your PostScript code as generated by the PicComments. Since these items are no longer available, you must be sure that no references to them are still active in the system. In particular, you must be sure that you have removed your dictionary from the dictionary stack and that you have removed any composite objects from the operand stack.

Note ▸

Remember that composite objects are strings, arrays, procedures, and dictionaries. All of these consume memory and will be lost when you restore to a saved state that was created by a **save** operator before the object was created. You must remove all references to these objects from the dictionary and operand stacks before you do the **restore** or else you will get a PostScript error when you attempt the restore operation. Also note that you don't have to restore in the same sequence that you did the saves; since you restore to a given save level, you can pick any one that has not been previously used. However, any save levels from later **save** operations will be lost and cannot be used once you have used a previous one.

▶ Transfer of control

So far you have seen how to create and use your own dictionary, along with your own procedures and variables. You have also learned some techniques for display of critical information within your application. This is the type of processing that usually includes a goto statement or something of that sort. PostScript does not provide any type of direct branch, nor does it implement statement labels of any kind. All PostScript programs use structured programming techniques to control program flow. The examples in this section show you some of these options.

You begin with an example based, as usual, on the preceding one in this chapter. Look at the code in Listing 4-7. The procedures and constants are the same as the preceding example; only the code in the body of the text is different. After changing the coordinates, this code defines two fonts. This is done for efficiency; the idea here is to find and create the correct font in the correct size and then store that as your own font constant. In this way, as you use multiple fonts, you will have much of the font overhead out of the way and can switch fonts much more quickly and efficiently. Here, for the sake of the example, you use both Times-Roman and Times-Bold.

The first section of the example code loads the coordinate transformation matrix (CTM) from the interpreter and displays the six values on one line. The CTM is loaded by creating an empty matrix with the **matrix** operator and then filling it with the current transformation matrix, which is accessed by the **currentmatrix** operator. Then the matrix is converted into its individual components by using the **aload** operator, which places all the elements of the matrix onto the operand stack, with the matrix itself on the top—hence the **pop** to remove the matrix, which is no longer useful to us. This conversion is essential because, if you apply the **cvs** operator to the matrix directly, you will not get a printable string; instead, the result is a default string (--nostringval--), which tells you that the item you tried to convert cannot be represented as a simple string. Once the matrix is broken up into its components, however, the individual values can be displayed correctly. The display technique here uses the **repeat** operator, which executes the procedure in front of it

```
myStr cvs show ( ) show
```

the number of times indicated by the integer in front of the procedure—here, 6. Since there are six elements in the CTM, this shows you all of them.

Listing 4-7. Code routine for PSText 5

```
% example PSText5
/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables
/myStr 255 string def
/LeftMargin 50 def
/TopMargin 650 def
/LineLead 12 def

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord --
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/newline
%procedure to advance a line
%called as:     -- newline --
%requires:  LineLead (for leading value)
%           LeftMargin (for margin)
%           TopMargin (for initialization)
{
    {currentpoint}
    stopped
    {
        LeftMargin TopMargin moveto
    }
    {
        LineLead sub
        exch pop
        LeftMargin
        exch
        moveto
    }
```

Listing 4-7. Code routine for PSText 5 (continued)

```
    ifelse
}
bind def

%now some tests for program controls
restoreCoord
/TR10 /Times-Roman findfont 10 scalefont def
/TB10 /Times-Bold findfont 10 scalefont def
newline
TR10 setfont
(Coordinate Transformation Matrix values: ) show
newline
matrix currentmatrix aload pop
6 { myStr cvs show ( ) show } repeat

newline newline
(Loop variables during ) show
TB10 setfont
(for) show
TR10 setfont
( execution:) show
newline
0 1 10 { myStr cvs show ( ) show } for

newline newline
(CTM: ) show
matrix currentmatrix
([) show
0 1 5 { 1 index exch get myStr cvs show ( ) show } for pop
(]) show

end     %SimpleLWdict
```

The example continues by displaying the loop variables as provided by the **for** operator during processing. The **for** operator executes a procedure for a given number of times; the processing is controlled by a loop variable, which is defined by the three numbers that precede the procedure: the beginning value, the increment, and the ending value. The processing is quite similar to **for** processing in C, but it is not as flexible. The loop variable is initialized to the beginning value and is

incremented for each iteration of the loop until it is greater than the ending value. At each iteration, the current value of the loop variable is available to the procedure on the operand stack. This is quite useful, but one of the problems in using the **for** is understanding how it presents the loop variable to the procedure during processing. Here you have designed a procedure that simply prints out the value on the top of the stack, which is the value of the loop variable for each iteration. Notice that the values of both 0 and 10 are in the processing loop, so this executes eleven times in all. You also see, in this part of the example, how the two fonts that you created can be used to switch between the normal font and the bold font and back again. This is the most efficient technique that you can use for switching fonts without performing additional calculations.

The earlier display of the CTM was an example of how to use the **aload** and the **repeat** operators; however, the technique there had two drawbacks. First, the values are printed out in reverse order, since they are printed from the operand stack and **aload** places them on the stack in reverse order, with the first element of the array on the bottom of the stack and the last on the top. Second, it would be nice to see the matrix as an array, rather than as a sequence of numbers. If you were displaying several different types of information, you would certainly want to see an array displayed together. Since PostScript arrays are normally written with brackets, that is the way you will display the values of the array in the next part of this example.

Here you again get the CTM as before, but you don't unload it onto the stack. Instead, you set up a **for** loop that retrieves each element from the array in succession and displays it. The display begins with an open bracket, [, to indicate the beginning of the array; then the **for** loop executes this procedure

```
1 index exch get myStr cvs show ( ) show
```

This process uses the current value of the loop variable as an index to retrieve one element of the array and display it. The procedure first retrieves a copy of the matrix, which is underneath the loop variable on the operand stack. Then this duplicate of the matrix and the loop variable are exchanged, which prepares them for the **get** operator. This retrieves one element from the array and removes both the index value and the copy of the array from the stack, replacing them with the desired element. Then you print this in the usual way. At the end of the loop processing, you still have the matrix on the stack, so you **pop** that off and print a closing bracket,]. Now the array is printed as an array should be: in correct order and with brackets around it. The entire exercise produces a page that looks like Figure 4-10.

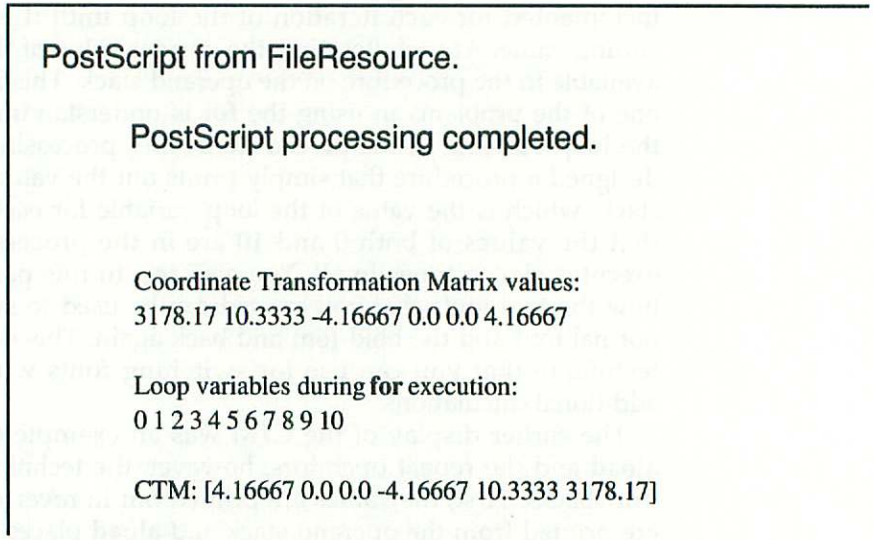

Figure 4-10. Page output from example

The next examples—in fact, the next two examples—are simply variations on this theme. Their major improvement is to develop a method to print data in multiple columns. This involves less work than you might suppose. Since these examples are quite similar, let's look at the first one briefly, and then just describe the second. The first column print code is shown in Listing 4-8.

Listing 4-8. Code routine for PSText 6

```
% example PSText6
/SimpleLWdict 15 dict def
SimpleLWdict begin

% Global variables
/myStr 255 string def
/CM [72 252 432] def
/NowMargin 72 def
/TopMargin 650 def
/BottomMargin 50 def
/LineLead 12 def
```

Listing 4-8. Code routine for PSText 6 (continued)

```
% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/prStr
%procedure to print a value as a string
%called as: obj print --
%requires:  myStr (for working storage)
{
    myStr cvs show
    ( ) show
}
bind def

/newline
%procedure to advance a line
%called as:     -- newline --
%requires:  LineLead (for leading value)
%           NowMargin (for current col margin)
%           TopMargin (for initialization)
{
    {currentpoint}
    stopped
    {
        NowMargin TopMargin moveto
    }
    {
        LineLead sub
        dup BottomMargin le
        {
            nextColumn
        }
```

Listing 4-8. Code routine for PSText 6 (continued)

```
        if
        exch pop
        NowMargin
        exch
        moveto
    }
    ifelse
}
bind def

/nextColumn
%procedure to move over 1 column
%called as:     -- nextcolumn --
%requires:  CM (for column positions)
%           NowMargin ( for current col margin )
%           TopMargin (for top of column)
{
    0 1 CM length 1 sub
    {
        dup
        CM exch get
        NowMargin le
        {
            CM length 1 sub eq
            {
                /NowMargin CM 0 get def
            }
            if
        }
        {
            CM exch get
            /NowMargin exch def
            exit
        }
        ifelse
    }
    for
    pop
    TopMargin
}
bind def
```

Listing 4-8. Code routine for PSText 6 (continued)

```
%now some tests for program controls
restoreCoord
/TR10 /Times-Roman findfont 10 scalefont def
/TB10 /Times-Bold findfont 10 scalefont def
newline
TR10 setfont
(Dictionary display for ) show
TB10 setfont
(SimpleLWdict) show
TR10 setfont
(:) show
/TopMargin TopMargin LineLead sub def
/NowMargin CM 0 get def
newline
SimpleLWdict
{
    pop
    prStr
    newline
}
forall

end     %SimpleLWdict
```

The first thing you notice about this code segment is that your dictionary now requires 15 elements rather than the previous 10. As you will see, several more definitions are now to be placed into the dictionary; if you left the maximum dictionary size at 10, you would run out of room. In the global variables, you now have two new definitions: the **CM** array, which is an array of column positions, and the new **BottomMargin** constant. In this case, **CM** defines three columns with left margins of 72, 252, and 432 units each. In addition, the old constant **LeftMargin** has been renamed **NowMargin** since it changes during processing, as you will see.

I don't know about you, but I was getting tired of writing

```
myStr cvs show ( ) show
```

several times in every program. Anytime that you see the same sequence of instructions several times in the same program, you know that these instructions are a good candidate for becoming a procedure. So here you have the new procedure **prStr** that simply converts the item on the top of the operand stack into a string and displays it, followed by a space because this operation is so common in all our examples.

Next are some small changes to the **newline** procedure. After the line leading is subtracted from the current point, the result is tested to see if it is below the **BottomMargin**; if not, processing continues as before. If it is below, however, the new procedure, **nextColumn**, is executed to move over to the top of the next column, as defined by the **CM** array.

The **nextColumn** procedure is quite long but not really too complex. There isn't enough room here for detailed analysis so let's just look at the overall structure. The procedure uses a **for** loop, as you have seen before, to access each of the elements of the **CM** array. The current position along the *x*-axis, which is defined by the **NowMargin** variable, is tested against each element of **CM** in turn. If the element is less than or equal to the margin, you go on to the next element in turn. If this is the last element in the array, you automatically return to the first column. (Here is where you would add next page processing if you wanted to use multiple pages.) If the element is greater than the current margin, then you set it as the current margin by changing the **NowMargin** value. Once you have found a new value from the array, you terminate the loop processing by using the **exit** operator. The **nextColumn** procedure was called by **newline** with two values on the stack: the current *x* and *y* coordinate values, in that order. So, to force a return to the top of the stack, you must replace the old *y* value with the **TopMargin** variable before you return to **newline**.

The example's code itself simply uses our own **SimpleLWdict** as a subject for the listing. The code uses a new operator, **forall**, which is quite useful for such tasks. When you execute **forall**, it places every element of the dictionary onto the operand stack: first the key and then the associated value. Then it executes the procedure

```
pop prStr newline
```

which pops off the value—which you don't want to use here—and prints the key as a string, followed by advancing to the next line. With the new **newline** procedure, if you run off the bottom of the page, you simply move onto another column. In this case, of course, the few items in **SimpleLWdict** won't be enough to fill even one column. The resulting output is shown in Figure 4-11.

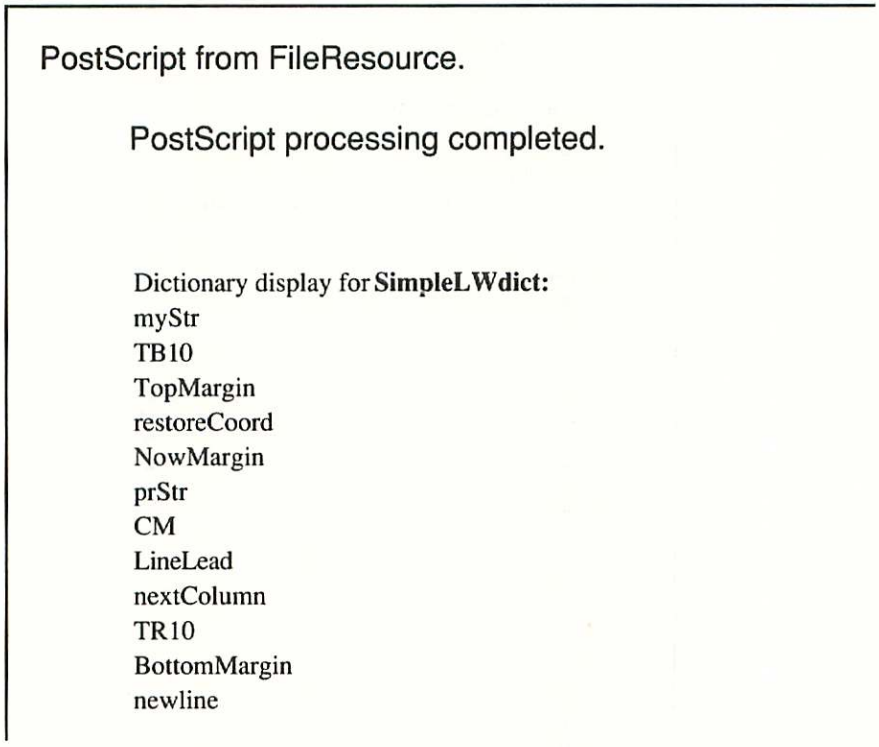

Figure 4-11. Page output from example

To demonstrate that these techniques actually work, let's run this again with two changes: change **CM** to provide five columns and list the Laser Prep dictionary, **md**. To do this, change **CM** to **[60 168 276 384 492]**, and replace **SimpleLWdict** with the name **Laser Prep** in the name display and with **md** in the line before the **forall** procedure. If you do this and run again, you should see the output page shown in Figure 4-12.

Overall, you now have a good grasp of the use of procedures and dictionaries in PostScript and you have some experience in how to use various techniques to display internal information on your output. This is very helpful as you work in PostScript, particularly when you are using the PicComment as a basis for your code, as you are here.

PostScript from FileResource.

PostScript processing completed.

Dictionary display for **Laser Prep:**

fc	PaintBlack	SimpleLWdict	gc	nc
ov	cd	lnop	scaleby96	wd
dv	kp	xp	pen	savescreen
srot	us	tt	cdb	smalls
fp	xdf	av	mdf	bwi
te	SwToSym	nlw	x8	qC
pgr	ma	smc	dbinvertflag	invertflag
sm	bdf	setTxMode	rf	xb
pa	oldsettransfer	sc8	sa	mf
tc	3q	jn	bf	sfd
sl	qa	bt	rh	mup
3a	vrb	gr	sb8	2a
th	bp	kif	dd	dsc
pl	sd8	concatprocs	@t	qq
sg	mx3	kwn	ef	fr
bmbc	2t	fBitStretch	ppr	sfl
bb	se	pys	qx	rc
bu	@2	noflips	pat	sos
qA	ndf2	bcarray	fm	fg
qp	mtx	ul	@1	fnt
lshow	wi	sob	tab	ts
it	db	bn	ao	gl
sfreq	bwsc	F	udf	4colors
freq	3colors	doop	pnsh	psu
the	qm	fs	sn	lw
por	rot	tp	ns	scs
fillflag	rm	mo	2colors	su
obl	pgs	mfont	ep	c
gw	aa3Size	xflip	setcmykcolor	ih
cpat	pxs	spf	ms	qs
sis	pr	barc	vs	fz
st	settransfer	as	lm	psb
qn	6a	lin	pse	sspf
qi	d8	dlin	xl	pm
qf	restorescreen	pnsv	sa8	
ou	wtkey	mx1	pop4	
oc	txpose	yflip	dc	
mc	psuedo	di	xc	
px	od	mx2	newmm	
nop	pnm	blank	qc	
sss	rb	psuedo1	ar	
ps	tso	d4	macvec	
qB	rr	T	s75	
cdf	cf	fNote	tu	
dh	vo	routines	min	
dl	op	sbs	ct	
mb	cp	so	aps	
pt	eu	fmv	gm	
sgt	bigs	x4	max	

Figure 4-12. Page output listing Laser Prep

Important ▸

Note that the techniques you are using here, particularly those for features such as multicolumn display, are only appropriate in general for debugging purposes. PostScript is a wonderful, very flexible language, and it provides full language features that allow you to do things like this when required. In normal processing, however, particularly on the Macintosh, you want to display your data on the screen as well as print it. Using such techniques would make your screen display different from the printed output and would cause you to do the same work twice: once inside your application, for screen display, and once in the printer, for printed output. It is better to do these calculations of columns and so on once for both purposes in your application code and then use PostScript commands to make the printed output match your display. In this case, you are displaying internal data that is not available to your application, and, of course, you can only do the calculations and display in PostScript, as you have done here.

Obviously you might want to make many refinements to these routines. For example, for a long list such as **md**, it would be very nice if the list were presented in a sorted order, so that you could find a particular procedure more easily. You might also want to see what type of object each of these names represents; or you might wish to see the value associated with it. In addition, the code here provides no internal error checking—to allow for keys that are not names, for example—and is generally not of "industrial strength" as it would have to be for a real-world application.

There are two obvious (I hope) reasons for these limitations: space and complexity. The space issue is quite simply that there is no room in this book for all the additional code and explanation that would be required to bring this code up to that standard. The concern over complexity is, perhaps, more subtle but just as real. To add so much code to these examples would make them much more complex to follow, even with explanations. It would obscure the point of the examples by burying the essential code in a mass of supporting routines, and, not incidentally, it would require much more knowledge of PostScript to follow. Therefore, the routines are as you see them: small and compact; useful as they stand but not robust; in short, waiting for you to take them as a starting point and make them your own.

▶ Performing Graphics Operations

The real strength of the PostScript language is in its handling of graphics operations. PostScript contains a complete set of graphics operators that allow you to create all types of complex graphics. You have already read, in Chapter 2, something about how PostScript constructs and manages graphic objects. In this, it takes an approach that is subtly, but quite distinctly, different from QuickDraw. In this section, you have a chance to experiment with PostScript graphics in their native state (so to speak) and to explore some of the major differences and similarities between PostScript and QuickDraw. In many ways, QuickDraw and PostScript are, as Churchill said of the English and the Americans, "One people, divided by a common language."

At the risk of sounding like a broken record, once again the graphics techniques demonstrated here are not very complex and are oriented toward debugging and understanding PostScript concepts rather than toward producing elegant pictures. PostScript is a most elegant graphics tool; in the hands of the talented, it can produce wonderful drawings and images. Here, however, the focus is on the most simple issues and operators so that you will understand how to use basic PostScript operations for ordinary tasks. Then, if you have the time and inclination, you can proceed to more advanced graphics with confidence.

▶ Line operations

To begin at the beginning, let's look at creating lines and curves. The best and easiest way to do so is to start with some code. Listing 4-9 shows your first line exercise.

Once again, you define your own dictionary and add the usual procedure to remap the coordinates. This example has no global variables and no other procedures, so you proceed directly to the example code itself. This code begins by resetting the coordinates to PostScript standard and defining the current font to be 20-point Helvetica. Then you draw an *X*. The sequence is quite clear: first, **moveto** the top, left corner (150, 650) of the figure and draw a line, using the **lineto** operator, to the bottom, right corner (350, 450); then move to the top, right corner (350, 650) and draw a second line down to the point (150, 450).

Listing 4-9. Code routine for PSText 7

```
% example PSText7
/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

% now let's do lines
restoreCoord
/Helvetica findfont 20 scalefont setfont

%draw an X
150 650 moveto
350 450 lineto
350 650 moveto
150 450 lineto
stroke

%and give it a title
250 430 moveto
(X marks the spot!)
dup stringwidth pop 2 div
neg 0 rmoveto
show

end     %SimpleLWdict
```

Does this actually "draw" a line? No. As you remember from Chapter 2, in PostScript this only constructs a path; in this case, a path consisting of two, unjoined straight lines. To make the path visible, you must issue one of the painting operators. Here you use the **stroke** operator, which strokes the path with the current line values. Now the path is visible on the page, and, once the stroke is completed, the path no longer exists.

When this is completed, you can add a little caption by moving to a designated point and showing a string. In the code here, the string is centered on the designated point. This is a short but interesting piece of code. It begins by moving to the point that will become the center of the string. Then you enter the string—here, the string is (X marks the spot!). This is duplicated and the **stringwidth** operator is used to find the width of the string. The function and use of this operator are similar to the QuickDraw function StringWidth. This operator returns the dimensions of the string in both *x*- and *y*-axes; for roman alphabets, the *y* component is always zero, but that is not necessarily true with other alphabets. Since this is English, the *y* component is zero and is discarded by a **pop** operator, leaving only the *x* width on the stack. You divide this by 2, change its sign to a negative, and then do a *relative move*, using the **rmoveto** operator. This operator bears the same relation to an ordinary **moveto** as the QuickDraw Move routine does to MoveTo; in both cases, the coordinates are supplied as distances from the current point rather than as absolute coordinate values. Here, the **rmoveto** operation moves the current point directly back along the *x*-axis by one-half the width of the string, thus effectively placing the center of the string at the point that you want. The string, which was left on the stack for this purpose, is now rendered onto the page by the **show** operator.

Note ▶

Here, again, note that this technique is appropriate only when you are not coordinating the printed output with a screen display. In fact, there is a substantial overhead to this process since the **stringwidth** operator must perform the same tasks as **show** in order to calculate the actual width of the string. This process is more efficient within your application because you can use the font width information to calculate the width of the string, rather than actually going through the full font access mechanism. However, there are some tricks that can speed this process up in some cases, and they are discussed in the next section on text handling and fonts.

When you run this example, you should get an output that looks like Figure 4-13. This is a simple and rather basic display, but it shows you some of the features of PostScript and draws your attention to how these both differ from and are similar to QuickDraw functions that you have used in the past.

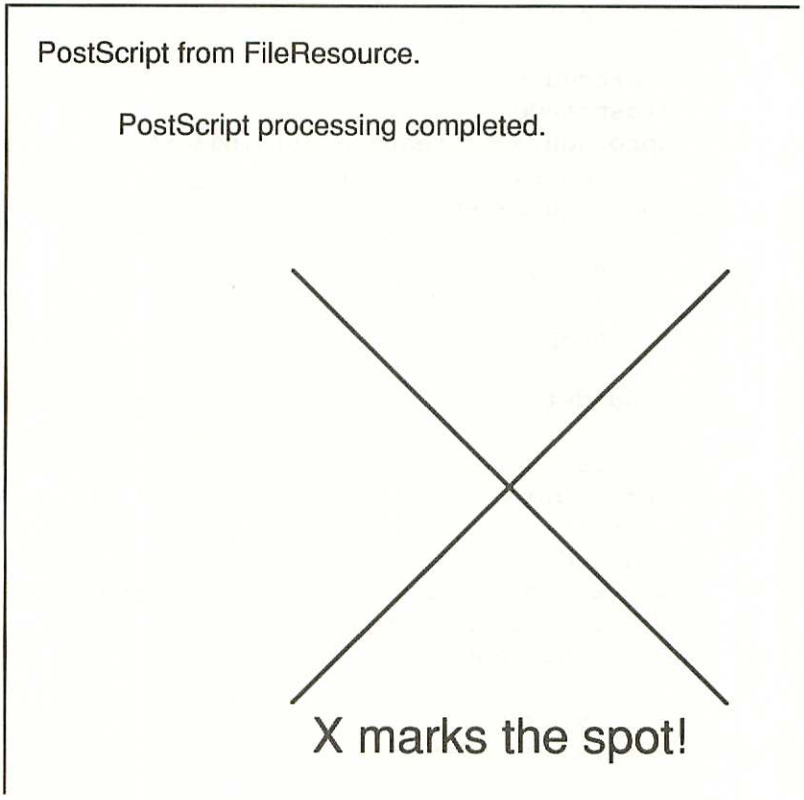

Figure 4-13. Page output from exercise

▶ Graphics state

Let's continue with another example that demonstrates some additional, important PostScript concepts. Look at the code in Listing 4-10. This example creates a simple procedure to draw a cross, with the center of the cross (where the arms join) at the current point and the length of each arm given by an *x* and a *y* operand. Such a procedure can be very useful in making drawings or graphs, for example, where you want to precisely locate and label a given point.

Listing 4-10. Code routine for PSText 8

```
% example PSText8
/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/cross
%procedure to draw a cross centered at the current
%point
%called as: x y cross --
%   where: x and y are the length of the arms in
%current coordinates
%no requirements
{
    /y exch def
    /x exch def
    gsave
        x 2 div neg 0 rmoveto
        x 0 rlineto
        stroke
    grestore
    gsave
        0 y 2 div neg rmoveto
        0 y rlineto
        stroke
    grestore
}
bind def
```

Listing 4-10. Code routine for PSText 8 (continued)

```
% now let's do lines
restoreCoord
/Helvetica findfont 20 scalefont setfont

%draw a cross
300 650 moveto
20 30 cross

250 550 moveto
30 30 cross
45 0 rmoveto
30 30 cross
45 0 rmoveto
30 30 cross

%and give it a title
295 500 moveto
(Very Good!)
dup stringwidth pop 2 div
neg 0 rmoveto
show

end     %SimpleLWdict
```

Given, then, that you want to have the intersection of the two arms at the current point, you must preserve the current point while you draw the two arms of the cross. In PostScript, as indeed in all computer languages, you might accomplish such a task in several ways: for example, get and save the current point coordinates, or move back to the center after each drawing operation. In fact, PostScript provides a fast, easy way to save the current point and return there. You can do this by saving and restoring the PostScript graphics state. Recall from Chapter 2 that the graphics state is a collection of all the current settings for PostScript graphics: the current font, the current path, and so on. In particular, therefore, the graphics state contains the current point.

Now look at the **cross** procedure. Here you begin by saving the two operand values, **y** and **x**. Notice that you save them in the *reverse* of the order in which they were placed on the operand stack; since the stack is LIFO, the last operand is on the top of the stack and must be removed first. Also recall that the name, or key, for a dictionary entry must come

before the value; hence, the use of the **exch** operator to place the name and the operand in the correct sequence for saving. Once these are safely tucked away, you then draw the horizontal (or **x**) arm. First you issue the **gsave** command, which saves the current graphics state on the graphics stack—note that this includes your current point. Next you divide the **x** value by 2 and do an **rmoveto** to that position, moving horizontally, as indicated by the **0** value for the *y*-axis in the operation. Then you draw a horizontal line that is **x** units long. To do this, you use the **rlineto** operator, which draws a line from the current point to the point described by the operands, which are given as displacements from the current point. Here, the displacements are x and 0, respectively, making the line horizontal. Then you **stroke** the line that you have drawn.

Now comes the neat part. You next issue the **grestore** operator. This restores the graphics state to the one that you had saved earlier, and, by that one stroke, also restores the current point to the position that it had at that time, which is the center of the cross. Now you are ready to start again. The next code section simply does the same thing in the vertical, rather than the horizontal, direction, again encased in **gsave** and **grestore**, so that the current point, at the end of drawing the cross, is still exactly where it was when you began: at the center of the cross.

Let's think about this for a moment. This is really quite a useful technique. It allows you to simplify your procedure by using relative movements from a known point, and it also allows you to return to a given point at the end of the routine without any further movement or calculations on your part. For a simple cross, of course, these calculations would not be difficult; imagine, however, the potential problems if you had been drawing a star, or some circular figure. Moreover, the **gsave** and **grestore** pair preserve all the characteristics of the current graphics state. (The next example gives another way to exploit this useful feature.)

The body of this example simply uses the **cross** procedure to draw crosses at several location along the page. First, you draw a cross with a horizontal arm of 20 units and a vertical arm of 30 units, centered at the point (300, 650). Then you draw three successive crosses, each 30 units in both arms, beginning at (250, 550). Notice how you can set the spacing exactly here, using an **rmoveto**, because you know both the size of the cross and that the current point after the drawing is exactly the same as when you started. Finally, there is a short title centered at the middle of the line of crosses using the technique that you used before—I figured that, if one is good, three must be very good. This all results in the page shown in Figure 4-14.

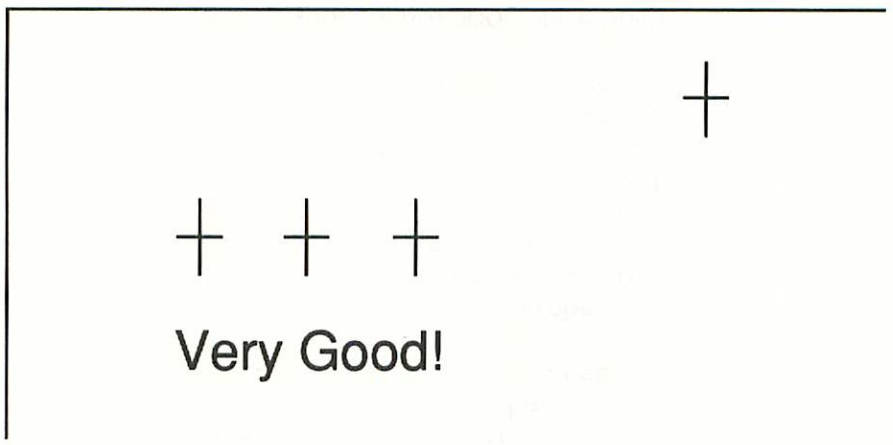

Figure 4-14. Page output from exercise

The **cross** procedure that you used here is not what a typical PostScript programmer would write, since it saves the *x* and *y* operand values from the stack as variables. For such a simple routine, this is not necessary; and, in general, you wouldn't do it. I did it here to show you how to define variables within a procedure and to remind you about the ordering of the presentation of these operands. Let's look, for a moment, at the more usual alternative that does not save and define these operands; the code is shown in Listing 4-11.

Here you have removed the two definitions that you used earlier. This leaves the two operands on the stack, in reverse order. Now, it doesn't matter to you or to PostScript which arm you draw first; therefore, since the **y** value is on the top of the operand stack, you can begin with the vertical arm. First you must duplicate the value because it is used twice: once for movement and once for the line. You divide the value by 2, as before, and then you place the **0** onto the stack and swap the two values to place them in the correct order for processing, by using the **exch** operator. In the same way, the second copy of the **y** operand is swapped and then used for the line drawing.

With the **y** done, you can proceed to the horizontal arm, described by the **x** value. Here you do the same things as before, without the **exch**, which is unnecessary for movement in the horizontal direction. The net result is exactly the same output, without having used any definitions. This saves you dictionary space and generally speeds up your processing. The code, however, is slightly more difficult to follow since you must remember what values are on the operand stack and how they are being used. As you become proficient in PostScript programming, you will find that such tasks become almost automatic.

Listing 4-11. Code routine for PSText 9

```
% example PSText9
/cross
%procedure to draw a cross centered at the current
%point
%called as: x y cross --
%   where: x and y are the length of the arms in
%current coordinates
%no requirements
{
    gsave
        dup
        2 div neg 0 exch rmoveto
        0 exch rlineto
        stroke
    grestore
    gsave
        dup
        2 div neg
        0
        rmoveto
        0 rlineto
        stroke
    grestore
}
bind def
```

▶ Closed paths

In PostScript you can define two types of paths: an *open path* or a *closed path*. All the paths that you have drawn up until now have been open paths; that is, the last point of the path does not come back to the first point of the path. The X and the cross are simple examples of open paths. The example code in Listing 4–12 shows you a closed path; in this case, a rectangle that is defined by the **frameRect** procedure, named in imitation of the similar QuickDraw routine.

Listing 4-12. Code routine for PSText 10

```
% example PSText10
/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/frameRect
%procedure to draw a rectangle
%called as: width height frameRect --
%   where: width and height are dimensions of rect in x
%and y axes
%no requirements
{
    /h exch def
    /w exch def
    gsave
        w 0 rlineto
        0 h rlineto
        w neg 0 rlineto
        closepath
        stroke
    grestore
}
bind def

% now let's do closed paths
restoreCoord
/Helvetica findfont 15 scalefont setfont
```

Listing 4-12. Code routine for PSText 10 (continued)

```
%draw a rectangle (not closed!)
200 400 moveto
150 0 rlineto
0 250 rlineto
-150 0 rlineto
0 -250 rlineto
20 setlinewidth
stroke
220 500 moveto
(Not closed!) show

200 100 moveto
150 250 frameRect
220 200 moveto
(Closed!) show

end     %SimpleLWdict
```

This example provides some graphic (excuse the pun) evidence of an important difference between PostScript and QuickDraw, so I want to run the code now so that you can see the output, which is shown in Figure 4-15. As you see, you have drawn two identical rectangles on the page. One, however, has a rather glaring error in one corner: a small notch that should not be there. The point of this exercise is to explain how that notch came to be there, and how you can avoid such errors.

In this example, let's begin our discussion with the body of the example rather than with the procedure—you will see why in a moment. In the body, after the usual coordinate reversal and after setting up a font, you begin to draw a simple rectangle as you might with QuickDraw commands. You move to a point and draw a path that is 150 units along the *x*-axis and 250 along the *y*-axis. As you see, the operator used here is **rlineto**, which you have met before. The motion is quite simple, and you can see that you end up at the same point where you began. Now you set the width of the line with the **setlinewidth** operator to 20 units and **stroke** the path to make it visible. However, unlike QuickDraw, this is not a closed path in PostScript even though it returns to the starting point, and so you add a small comment within the rectangle to remind you of this fact.

Now look at the procedure definition of **frameRect**. Here you see that the same technique is used to create the first three sides of the rectangle:

an **rlineto** for each side. The final side, however, is created with a new PostScript operator, **closepath**. This operator adds to the current path a straight line segment that extends from the current point back to the starting point of the path, and it *explicitly closes the path.* In QuickDraw, when you draw a polygon, for example, you close the drawing by returning to the starting point. In PostScript, however, returning to the starting point does *not* close the path; only using the **closepath** operator actually closes the path. Does this make a difference? As you see from the example output, it really does!

Why does this difference happen? A PostScript path consists of an imaginary line, with no real dimension. When you **stroke** a path, the

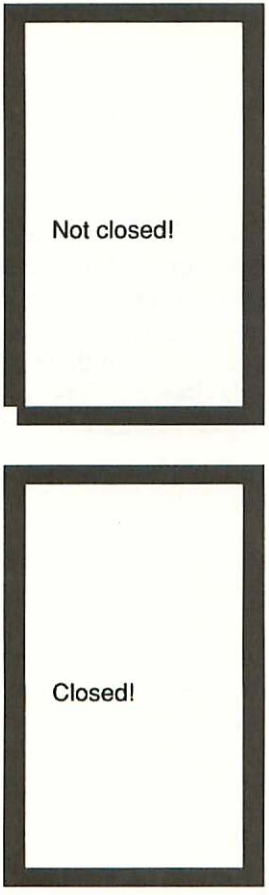

Figure 4-15. Output of example

pixels around the line are turned on to the defined width of the path, which is set by the **setlinewidth** operator, with one-half of the width on either side of the path line. You may have noticed that, here, you have set the line width to 20 units so that the difference between an open and a closed path is quite obvious. Now, in the ordinary course of events, any line is painted from its beginning point up to its end point and then it stops. When that happens, the ends of the lines are cut off like a paint stripe right at the point where the line segment begins and ends.

When the line returns to its starting point, there is an issue about how the line should be painted. If the line is not closed, the painting stops at the end, with the result that the outside corner, between the line that ends the path and the one that begins the path, remains unpainted, as you see in the first rectangle in Figure 4-15. However, if the path is closed, PostScript automatically fills in this corner, as you see in the second rectangle. This is a major difference between PostScript and QuickDraw and one that you should keep in mind as you are creating drawings.

Note ▸

As you see, I have here once again defined the width **w** and height **h** of the rectangle within the procedure rather than leave them on the stack. As in the earlier example, you would normally be better off leaving them on the stack and using them directly from it. However, for clarity, I have chosen to define them and use them rather than use the stack operands directly. This is purely for illustrative purposes, and you would not be likely to use this technique in an actual program.

So far you have stayed with straight line segments for our graphics because they are by far the easiest and most straightforward elements with which to work. PostScript would not be much use as a graphics environment if it were limited to straight lines, however, and it does, in fact, allow you full flexibility to use both segments of circles and arbitrary curves. Making arbitrary curved line segments is somewhat beyond what can reasonably be accomplished here, but this next example shows you how to make and use circular path segments. Let's look at the code in Listing 4-13.

Listing 4-13. Code routine for PSText 11

```
% example PSText11
/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/frameRoundRect
%procedure to draw a rectangle
%called as: llx lly urx ury oval frameRect --
%   where: (llx, lly) are the coordinates of the lower
%          lefthand corner of the rect
%       and (urx, ury) are coordinates of the upper
%          righthand corner
%       and oval is the radius of the desired corner circle
%no requirements
{
    /oval exch def
    /ury exch def
    /urx exch def
    /lly exch def
    /llx exch def
    gsave
        llx lly moveto
        oval 0 rmoveto
        urx lly urx ury oval
            arcto
            4 {pop} repeat
```

Listing 4-13. Code routine for PSText 11 (continued)

```
        urx ury llx ury oval
            arcto
            4 {pop} repeat
        llx ury llx lly oval
            arcto
            4 {pop} repeat
        llx lly urx lly oval
            arcto
            4 {pop} repeat
        closepath
        stroke
    grestore
}
bind def

% now let's do closed paths
restoreCoord
/Helvetica findfont 15 scalefont setfont

%draw a pacMac figure
350 500 moveto
currentpoint 100 20 340 arc
closepath
stroke

100 100 moveto
200 200 500 400 50
frameRoundRect

%now some text for the frame
220 300 moveto
(Very MacLike.) show

end     %SimpleLWdict
```

Here again let us begin with the body code and then discuss the procedure. As always, you begin by redefining the coordinates and setting a font for titles and display. Next you draw a fanciful little picture of a part of a circle, with a small wedge removed from it. I call

this a "pacMac," since it looks so much like one of the classic PacMan figures. The necessary path is easily created with a single PostScript operator, **arc**. The code begins by moving to the center point of the figure; here, it is the point (350, 500). The **arc** operator requires five operands. In order from left to right (that is, bottom of the stack to the top) they are: x and y coordinates of the starting point of the arc; the radius of the arc; the beginning angle for the arc, measured counterclockwise from the horizontal (or x) axis; and the ending angle, measured in the same way. Here this is given by the code line

```
currentpoint 100 20 340 arc
```

This code makes the beginning of the arc the current point and draws a segment with a radius of 100 units, beginning at the angle of 20 degrees and ending at the angle of 340 degrees. This makes a gap in the arc of 40 degrees; 20 above the horizontal axis and 20 below it. When the **arc** is finished, the endpoint of the path is at the end of the arc. To finish your figure, you simply use the **closepath** operator to make a straight line segment that returns to the starting point and closes the path as we discussed earlier. Then you **stroke** the path.

Next you use a new procedure that is defined here, **frameRoundRect**, to make a rectangle with curved corners. This is deliberately designed to mimic, in so far as possible, the QuickDraw routine of the same name. The operands are identical, except that the two oval coordinates in the QuickDraw call are replaced here by a single circular radius—a small concession to our tutorial approach.

Here again the procedure begins by removing the operands and storing them. In this case, however, this is not simply for clarity, as it was before. Here you will be using and reusing these values several times. In such cases, it is generally best to save the operands rather than trying to reuse them from the stack because the stack handling becomes overly complex and requires many duplications and stack movement commands. In such cases, any efficiencies that might come from using the stack directly are quickly lost—to say nothing of the problems of reading the code.

Note here how you have defined the rectangle that encloses the figure. As in QuickDraw, you are using the rectangle coordinates for placement; the difference here is that, instead of the usual QuickDraw "top left bottom right" sequence, you have used the PostScript standard, which—naturally enough, given the difference in coordinates—is "bottom left top right." The names here are the standard ones used in most PostScript applications: llx, lly, urx, and ury. The actual numbers,

of course, look just like their QuickDraw counterparts because the top coordinate is lower than the bottom in QuickDraw, whereas the bottom is lower than the top in PostScript. The net effect is the same, however: to define the rectangle that encloses the figure. In PostScript, this rectangle is known as the *bounding box*, and it occurs very frequently in PostScript processing—almost as frequently as rectangles occur in QuickDraw.

Once the operands have been stored, you can then proceed to making the desired figure. For this work, you use the PostScript operator **arcto**, which effectively draws a curved corner, starting from the current point, with the curve being defined by the radius of the curve (here the **oval** operand) and the next two corner points of the rectangle. Since teaching PostScript is not the primary objective, a complete description of how the **arcto** works is not given here. Note, however, that it returns four values to the stack when it is done. As these are not necessary for processing here, you discard them with the code sequence

```
4 {pop} repeat
```

which simply pops them off the stack. The result of the procedure is shown in Figure 4-16.

As the title says, this rectangle represents a very Mac-like performance by our clone of FrameRoundRect.

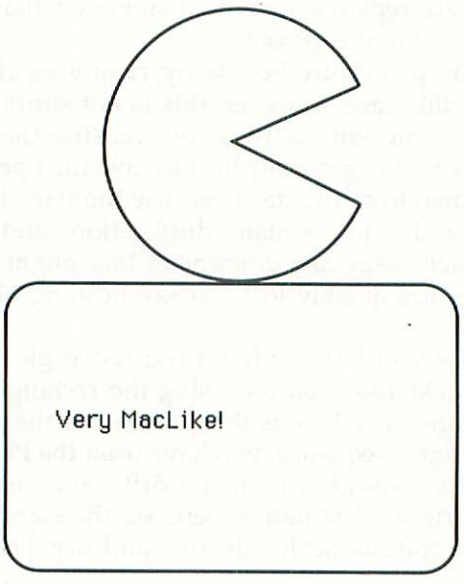

Figure 4-16. Page output from example

▶ Shading and patterns

PostScript has a variety of methods for handling the issues of generating colors, shades of gray, and patterns, in fact, many more than can be discussed or demonstrated here. This section covers some of the basic points about PostScript handling of shading and briefly discusses patterns.

Chapter 2 explained the concept of the current color, which is part of the current graphics state. The current color may be set to any color and PostScript supports several color models. All PostScript devices support the rgb (red, green, blue) and hsb (hue, saturation, brightness) models and full-color PostScript output printers also support the cmyk (cyan, magenta, yellow and black) model. (Honestly, if this doesn't mean anything to you, don't bother yourself about it—you aren't going to use this here anyway!) However, issues of color reproduction are quite complex and require a lot of technical expertise and space to cover in even minimally adequate detail; moreover, there are few color devices in the LaserWriter category. You do not deal with color in this book, and so nothing more needs to be said except to remind you that PostScript automatically converts color values into gray values for black and white output.

Black and white, however, is another story. All PostScript devices can produce output that is black, white, or shades of gray by using *halftone* methods. For programming purposes, it suffices to say that the PostScript interpreter can render various shades of gray on any PostScript device without much attention on the part of the average programmer. Let's look at a code sample in Listing 4-14 to see how this works.

This uses a variant on our old friend **frameRoundRect**, with a slightly revised name, **frRndRect**, so that you can tell the two procedures apart in a listing (or in the index). This now no longer has the **gsave, grestore** pair in it, and it has dropped the **stroke** at the end. The reasoning here is that you would like to both stroke this path and fill it with a shade of gray. Since **stroke** would erase the path, you cannot leave that in the procedure; in the same way, once you restore back to the graphics state that you saved at the beginning of the procedure, you would lose the path—which is, as you remember, part of the graphics state. Therefore the **gsave** and the matching **grestore** must be eliminated from the procedure. Other than these two changes, the procedure remains the same.

The example code however, shows you some different approaches to PostScript graphics. In the earlier examples, you created a path in a procedure and used it, and the design was to leave the state of the interpreter and the page just as it was when the procedure was executed. This is generally a good approach, but sometimes there are alternatives that are more suited to the needs of the program at hand. Here is one such example.

Listing 4-14. Code routine for PSText 12

```
% example PSText12
/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/frRndRect
%procedure to draw a rectangle (like previous frameRoundRect
%called as: llx lly urx ury oval frameRect --
%   where: (llx, lly) are the coordinates of the lower
%          lefthand corner of the rect
%       and (urx, ury) are coordinates of the upper
%          righthand corner
%       and oval is the radius of the desired corner
%          circle
%no requirements
{
    /oval exch def
    /ury exch def
    /urx exch def
    /lly exch def
    /llx exch def
        llx lly moveto
        oval 0 rmoveto
        urx lly urx ury oval
            arcto
            4 {pop} repeat
```

Listing 4-14. Code routine for PSText 12 (continued)

```
        urx ury llx ury oval
            arcto
            4 {pop} repeat
        llx ury llx lly oval
            arcto
            4 {pop} repeat
        llx lly urx lly oval
            arcto
            4 {pop} repeat
        closepath
}
bind def

% now let's do some mixed fill and stroke
restoreCoord

%draw a frameRoundRect
gsave
    100 400 translate
    0 0 350 250 50 frRndRect
    gsave
        .5 setgray
        fill
    grestore
    2 setlinewidth
    stroke
grestore
gsave
    150 450 translate
    .715 .715 scale
    0 0 350 250 50 frRndRect
    gsave
        .7 setgray
        fill
    grestore
    2 setlinewidth
    stroke
grestore
gsave
```

Listing 4-14. Code routine for PSText 12 (continued)

```
    115 480 translate
    .51 .51 scale
    0 0 350 250 50 frRndRect
    gsave
        .2 setgray
        fill
    grestore
    2 setlinewidth
    stroke
grestore

end     %SimpleLWdict
```

You want to stroke and fill the same path. Look at the code in the body of the example. Here you have adopted a quite different approach to the process of creating these graphics. Each invocation of the procedure is contained within its own **gsave**, **grestore** pair. Once the **gsave** is done, you translate the page origin to the point where you want the graphic to start. Then you invoke the **frRndRect**, starting from the origin (0, 0) and with the desired width and height as the coordinates of the upper right-hand corner of the rectangle. Now you do a **gsave** again. This stores the current graphics state, which includes the path for the rounded rectangle, as the top of the graphics stack. You set the current color to .5 (or 50 percent) gray and fill the path with that. This removes the path from the current graphics state, but you get it back with a **grestore**. Now you can set the line width to 2 and stroke the path. Voilà! The same path is filled with a color and then stroked in black. Finally, you **grestore** again to remove the effects of the translation and restore the original coordinate system. This also, incidentally, returns the current color to black.

The code proceeds to do this two more times, using different transformation factors—both scaling and translation—and different shades of gray. The final result is shown in Figure 4-17.

You should notice two things about this output. First, notice that the gray values do not proceed as you might have intuitively expected. The third gray value that you used, .2, produces a darker shade than the second that you used, which was .7. This shows you that the number that you give to the **setgray** operator, which sets the current color to a shade of gray, is not a percentage of gray; in fact, it is the exact reverse, with .2 being an 80 percent gray and .7 being a 30 percent representation. PostScript views colors in the sense of light on a page, so that 0 is no light (or black) and 1 is full light (or white). The same reasoning is applied to all the color values. Therefore .2 represents 20 percent of the light on the page, or an 80 percent gray, and so on.

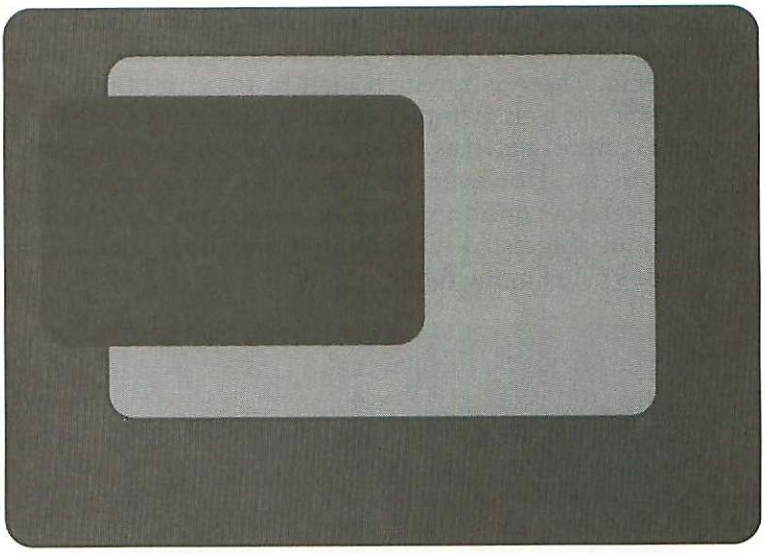

Figure 4-17. Page output from example

The second thing to notice is something that you read about in Chapter 2, but may possibly have forgotten at this point: PostScript colors are opaque and are laid down in layers, with the last layer completely overlaying and obscuring any underlying layers. Here you have created three overlapping windows, with different colors. However, as you see, the last one painted completely obscures the ones underneath, whether they are darker or lighter in color.

Note ▸

One additional point needs to be made about the sequence of this code. Notice that you first fill the path and then stroke it. It may seem to you that it doesn't matter in what order you do these. However it does matter to some extent—to precisely one-half of the stroke width to be exact. Why is that? Well, if you stroke the path first and then fill it, the fill goes out to the path boundary. Remember that the stoke width is painted with one-half inside the path and one-half outside. The result is that the fill erases—or, more precisely, overlays—the half of the stroke that is inside the path, so that the stroke width appears to be only one-half the size that you have set. If you do your operations in the order shown here, however, the stroke overlays the fill and so is full size. Of course, this loses a bit of the fill, but the visual effect is correct.

This raises the issue of patterns and the QuickDraw transfer modes that allow you to combine different regions to create merged or mottled effects. There is no such facility for combination in PostScript, and PostScript as it is implemented in currently available LaserWriters does not support patterns. There are ways to simulate both of these effects in PostScript. However, the techniques are complex and difficult; as such, they are beyond the scope of a book on LaserWriter programming. If you are interested in learning more about such techniques, look at the books listed in the Bibliography.

Note ▸

The restriction on patterns should not exist in future PostScript printers. PostScript Level 2 has support for patterns built into the language. Unfortunately, for most of us, it will be a while before we can get our hands on a Level 2 device. Until then, you will have to make your own patterns by using the present, rather complex, methods.

I don't want you to leave this section, however, on such a negative note, so let's look at something that PostScript does very well indeed, and that is quite difficult in QuickDraw. This is drawing what artists call a fountain, which goes in an even way from light to dark or vice versa. Here, in Listing 4-15, is an example that creates a simple fountain effect.

Listing 4-15. Code routine for PSText 13

```
% example PSText13
/SimpleLWdict 10 dict def
SimpleLWdict begin

% Global variables

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
```

Listing 4-15. Code routine for PSText 13 (continued)

```
bind def

/fountain
%procedure to draw a fountain
%called as: gray fountain --
%   where: gray is the beginning gray value (expected
%          to be either 0 or 1)
%       the fountain is drawn on a unit square,
%          positioned at (0, 0)
%       with 100 shades of gray in the square;
%       it goes from black (0) to white if gray is 0,
%       or from white (1) to black if gray is 1
%no requirements
{
    dup
    /gray exch def
    /incr
        exch 0 eq
            {.01}
            {-.01}
        ifelse
    def
    /pos 0 def
    .01 setlinewidth
    100
    {
        gray setgray
        pos 0 moveto
        pos 1 lineto
        stroke
        /pos pos .01 add def
        /gray gray incr add def
    }
    repeat
}
bind def

% now let's do a fountain
restoreCoord
```

Listing 4-15. Code routine for PSText 13 (continued)

```
%draw a fountain
gsave
    100 500 translate
    6 72 mul 144 scale
    1 fountain
grestore

%and give it a title
/Helvetica findfont 24 scalefont setfont
120 560 moveto
(POSTSCRIPT) show

end     %SimpleLWdict
```

This example has a simple **fountain** procedure that makes a series of small stripes across a unit square, that is, a square with a bounding box of (0, 0) and (1, 1). There are 100 lines drawn vertically, beginning at the left of the square, and each is colored with a 1 percent difference from its neighbors. The direction of the fountain, from dark to light or vice versa, is controlled by its single operand, which is set to 0 or 1 to determine the starting color value. If the starting value is 0, the fountain proceeds from black to white; if it is any other value, it proceeds from that value to black.

The body of the code shows a simple use of the **fountain** procedure. You start with a **gsave** so that you can get back to the current coordinates after the necessary transformations. Then you **translate** the origin to the lower, left corner of where you want to begin the fountain. Next, you stretch the coordinates, so that a unit square spans 6 inches across the page and is 2 inches high, by using the code

```
6 72 mul 144 scale
```

Now you make a fountain, using the 1 operand to get it to go from white to black. As a final feature, you print the word "POSTSCRIPT" across the fountain, as a reminder of how you created this rather neat effect. The result is shown in Figure 4-18.

This completes our brief look at how PostScript handles colors, shading, and patterns. There is much more to discuss here, but I must leave you to explore this interesting subject on your own time, as we move on to further PostScript topics.

Figure 4-18. Page output from example

▶ Clipping

One last topic regarding PostScript graphics is the issue of clipping. PostScript and QuickDraw both support clipping to arbitrary regions, but they do it in very different ways. Since you are familiar with QuickDraw clipping, you will cover the PostScript form of clipping here, with an example to illustrate several points at once. Look at the code shown in Listing 4-16. This routine uses the familiar routine **restoreCoord** to begin. It then proceeds directly into the body code to allow you to work with clipping.

Listing 4-16. Code routine for PSText 14

```
% example PSText14
/SimpleLWdict 15 dict def
SimpleLWdict begin

% Global variables

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def
```

Listing 4-16. Code routine for PSText 14 (continued)

```
%now some tests for clipping
restoreCoord

clippath
2 setlinewidth
stroke

clippath
pathbbox
newpath
    /ury exch def
    /urx exch def
    /lly exch def
    /llx exch def
    urx llx sub 2 div lly moveto
    urx ury lly sub 2 div lineto
    urx llx sub 2 div ury lineto
    llx ury lly sub 2 div lineto
    closepath
clip
newpath

clippath
.8 setgray
fill

end     %SimpleLWdict
```

Before you begin to analyze this code, however, you need to know some of the theory behind clipping in PostScript. Clipping in PostScript works conceptually in exactly the same fashion as it does in QuickDraw: It restricts the area where drawing occurs. However, the PostScript interpreter uses clipping itself to define the size of the output page. This is different than QuickDraw, where the normal output area is given by the visRgn, and the default clipRgn is very large. In PostScript, the *clipping path*, which is part of the current graphics state, defines where on the page drawing occurs; it is set to the size of the output page at the beginning of a job, so that the interpreter uses clipping to determine what is visible on your output.

By the same token, you can expand the clipRgn in QuickDraw to be very large if you want, and you can both shrink and expand it as you require; if your clipRgn is larger than the visRgn of the grafPort, only the portion of a drawing that lies within the visRgn is shown. In PostScript, on the other hand, the clipping path cannot be expanded; it defines the edges of the printable area on the page at the beginning of the job and it can only be made smaller within the job. The net result is that you cannot have a clipping path that is larger than the printable area of the page. Now, let's look at some examples of clipping in PostScript.

The body code in this example begins as usual with the **restoreCoord** procedure. Now you begin the exploration of clipping by executing the **clippath** operator. This operator makes the current clipping path into the current path. As we just discussed, the clipping path initially defines the printable area of the page; the result here is to make the current path a rectangle that precisely outlines the boundary of the printable page. Having converted this into the current path, you can use it as you would any other path; specifically you can **stroke** the path as you do here. This provides a visible outline of the margins of the current printable area.

Next you define a new clipping path based on the original one. The code begins by retrieving the clipping path again, using **clippath** as before. Next you issue a **pathbbox** operator, which returns the bounding box of the current path as the coordinates of the lower left and upper right corners of the bounding box for the path. In this case, this gives you the lower left corner and the upper right corner of the imageable area. Now you erase this path and save these coordinates. Then you create a new path that forms a diamond whose four corners are the center of the bottom margin, the center of the right margin, the center of the top margin, and the center of the left margin. Notice that the path is closed by the **closepath** operator at the end. Once this path is created, you make it the current clipping path by issuing the **clip** operator. This takes the current path, intersects it with the old clipping path, and makes the interior of the result the new clipping region. In this case, the interior is the interior of the diamond because, by design, all points inside the diamond lie within the previous clipping path.

As a last point of cleanup, you issue the **newpath** operator to remove the current path. This is necessary because, unlike **stroke** or **fill**, the **clip** operator does not erase the current path when it executes. Therefore, to avoid unwanted or unexpected complications, it is always best to clear the path after a **clip** has been issued. To show what the new clipping region looks like, simply recall it using a **clippath** and fill it with a light (20 percent) gray. The resulting output looks like Figure 4-19.

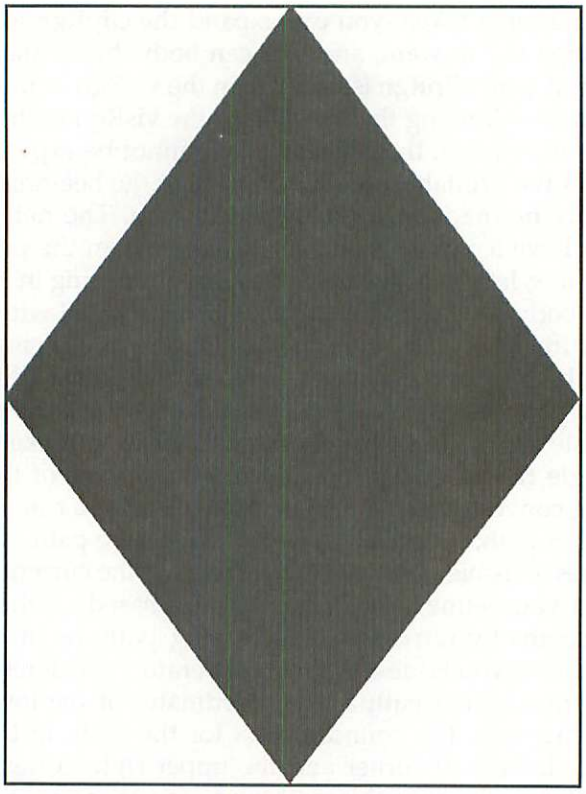

Figure 4-19. Page output from example

So that's basic clipping. These operations can provide you with some interesting information about the page and, if used incorrectly, can cause some (perhaps) unexpected behavior. Let's look at a variant of the previous example, shown in Listing 4-17, that illustrates some of these points.

This example adds routines and variables that you have seen before: **prStr, newline**, and their associated global variables. The body code is also almost identical to the previous example; the differences, however, are quite enlightening. Before you proceed, you might want to imagine what you think the results of executing this code will be.

The beginning is the same as the preceding example: You get the current clipping path and stroke it. So far, so good. Then there is a new code segment that simply displays the current memory available and used; this is very similar to the code that you have used previously for

this same type of display. There's nothing to amaze you there either. Then you once again draw the same diamond, using the same information as you did before, and you make a new clipping path with the results. Then fill it with gray and print the result, as shown in Figure 4-20.

Listing 4-17. Code routine for PSText 15

```
% example PSText15
/SimpleLWdict 15 dict def
SimpleLWdict begin

% Global variables
/myStr 255 string def
/LeftMargin 100 def
/TopMargin 500 def
/LineLead 12 def

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

/prStr
%procedure to print a value as a string
%called as: obj print --
%requires:  myStr (for working storage)
{
    myStr cvs show
    ( ) show
}
bind def
```

Listing 4-17. Code routine for PSText 15 (continued)

```
/newline
%procedure to advance a line
%called as:     -- newline --
%requires:  LineLead (for leading value)
%           LeftMargin (for margin)
%           TopMargin (for initialization)
{
    {currentpoint}
    stopped
    {
        LeftMargin TopMargin moveto
    }
    {
        LineLead sub
        exch pop
        LeftMargin
        exch
        moveto
    }
    ifelse
}
bind def

%now some tests for dictionaries
restoreCoord
/Helvetica findfont 10 scalefont setfont

clippath
stroke

100 500 moveto
(Standard QuickDraw page size memory:) show
newline
vmstatus
(Maximum: ) show prStr
(/ Used: ) show prStr
pop

clippath
pathbbox
    /ury exch def
```

Listing 4-17. Code routine for PSText 15 (continued)

```
    /urx exch def
    /lly exch def
    /llx exch def
    urx llx sub 2 div lly moveto
    urx ury lly sub 2 div lineto
    urx llx sub 2 div ury lineto
    llx ury lly sub 2 div lineto
    closepath
clip
newpath

clippath
.8 setgray
fill

end     %SimpleLWdict
```

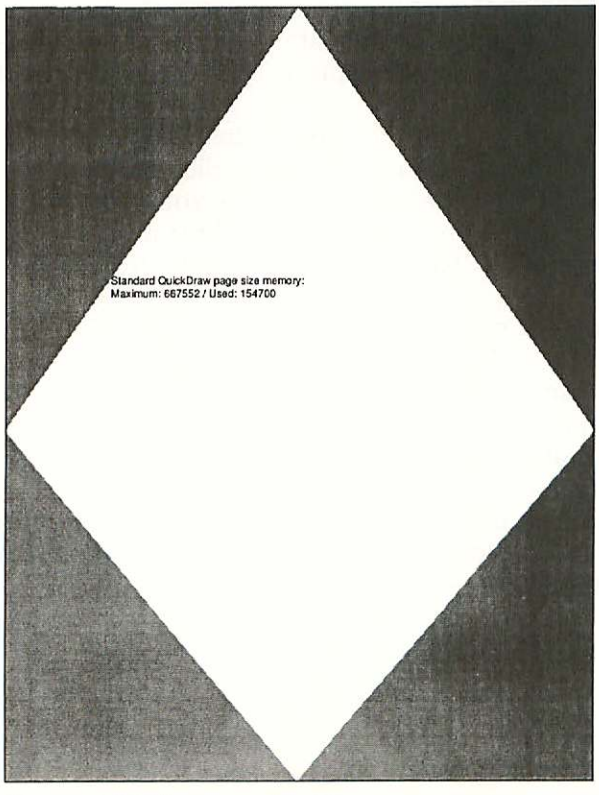

Figure 4-20. Page output from example

I'll bet that Figure 4-20 doesn't look like you expected it to, right? The change is subtle and one that is easy to make when using and setting clipping paths. The difference here is that you have removed the **newpath** at the beginning of the code that constructs the diamond. This leaves the original clipping path, around the edge of the page, as part of the current path when you finally execute the new **clip** operator. In this case, therefore, the interior of the combination of paths is not what is inside the diamond, but what is outside it, between it and the original path. This is a neat effect if that's what you wanted, but this type of result can give you real debugging nightmares if it's unexpected.

Also notice that the text that you printed before you reset the path still shows quite plainly on the page. The fact that you alter the clipping path does not hide or remove marks that you have already made on the page. Finally, if you look closely, you'll see that the line that you originally made around the page is now gone. It has been overlayed by the gray fill that you put into place. If you didn't use a clipping function, you would still see one-half of the line because the fill would only cover exactly up to the path boundary, whereas the line was stroked with one-half of it visible on either side of the path. (Remember our earlier discussion on this point.) Here, however, the path is the clipping path, so that the half of the stroke that would normally be outside the filled area is clipped off and doesn't print. The net result is that the gray fill erases the line entirely.

This code gives us the opportunity to explore one of the options in the Page Setup dialog box. When you click on Options in this dialog box, you get a new dialog that allows you to set, among other things, whether you want a Larger Print Area and, as a result, fewer downloadable fonts. Change the caption of the string in the example to

```
(Larger QuickDraw page size memory:) show
```

and rerun the job. When you are ready to print, select the Larger Page Size option from the dialog box. Now your printout shows you the exact tradeoff that you made: The new page size is shown by the gray fill and the reduction in available memory is shown on the printed output. This is an interesting example of how you can use these tools to make better judgments about your printing requirements. The result is shown in Figure 4-21.

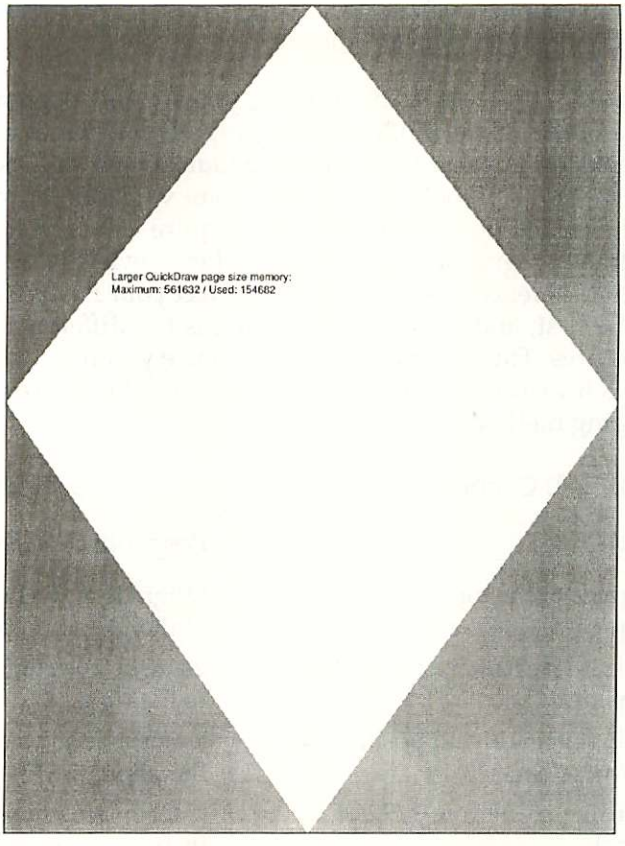

Figure 4-21. Page output from example

Note ▸

You may be wondering why I keep saying "units" instead of points, when the default PostScript coordinates are in fact, as we discussed in Chapter 2, 1/72nd of an inch—almost exactly a printer's point and identical to the QuickDraw coordinate units. The answer is quite simple, and it serves as a reminder about PostScript coordinates. The PostScript coordinate system is flexible and can easily be modified by a **scale** operator, or some other coordinate transformation. When you execute PostScript operators, the numbers that you give them are in current coordinate units, whatever those happen to be. As an exaggerated example, suppose you changed **restoreCoord** to say

```
72 -72 scale
```

In that case, each unit in our examples would be 1 inch rather than 1 point and all the drawings and text characters would be magnified by that factor. For this reason, I encourage you to think of PostScript coordinates as units rather than points; that way you are less likely to be fooled when coordinate transformations occur.

▶ PostScript and QuickDraw

With all of this PostScript drawing, you are now beginning, I hope, to see some of the subtle and, when not properly taken into account, dangerous differences between PostScript and QuickDraw. As you have seen, these two methods of looking at graphics are very similar in many ways; yet, in the end, they are different and require different habits of mind and different programming approaches. Let's review now what some of the major differences are and how they affect your PostScript coding.

The first, and most obvious point, is the difference in the coordinate structures. This is certainly obvious since you have to run **restoreCoord** in each example. Table 4-1 shows you the comparison of the two imaging methods.

Table 4-1. Comparison of QuickDraw and PostScript

QuickDraw	*PostScript Defaults*
Origin is at the top, left corner of the printable area.	Origin is at the bottom, left corner of the physical page.
x-axis is horizontal and positive to the right.	*x*-axis is horizontal and positive to the right.
y-axis is vertical and positive down the page.	*y*-axis is vertical and positive up the page.
Coordinate units are 1/72 of an inch.	Coordinate units are 1/72 of an inch.

Even within these two lists, there are some nonobvious variations. First, the PostScript coordinate system begins at the edge of the page, not at the edge of the printable area. The result is that, if you are in native PostScript coordinates, the point (0, 0) is almost never within the printable area. In QuickDraw, however, (0, 0) is the topmost printable position. The bottleneck transformation that converts the default PostScript coordinates into QuickDraw compatible form moves the origin inside the page to accommodate this requirement. The result can be, as they say, a source of unpredictable behavior.

Also, the PostScript coordinate system is infinitely malleable; it can be transformed underneath you in a multitude of ways. QuickDraw, on the other hand, is essentially fixed. One QuickDraw unit is 1/72 of an inch (unless you modify it by a PrGeneral comment); one PostScript unit may be whatever you want. This leads to another point: PostScript coordinates are real numbers; that is, they can and do have decimal

points. QuickDraw coordinates, since they are tied to the device pixels, are inevitably integers. There can be 0.5 or 0.05, or 0.005 PostScript units; there cannot be one-half a pixel—it's an impossibility.

As you have already seen, there are a number of similarities between the PostScript drawing environment and a QuickDraw grafPort: an independent coordinate system; graphics controls, such as pen size; and so on. One noticeable difference is that there are few defaults in the PostScript environment, but many in QuickDraw. You must explicitly establish position and font information in PostScript, for example, before you begin work. In QuickDraw, most of these are set by default and you change them only if you need some other values. PostScript also has the notion of a current path, which must be made visible by painting operators. In QuickDraw, the act of making a line implies painting the line with the current pen values; it all happens in one function. QuickDraw also has a number of functions that support screen output, such as the transfer mode, that have no direct counterpart in the PostScript environment. The most notable side effect of these differences is that you generally will erase an area in QuickDraw before writing to it by filling it with white (or the current screen color). In PostScript, the page is already white, so no erasing is required; in fact, using erasing to clear an area materially slows down your output. Refer to Technical Note #91 for more information about QuickDraw functions that should be avoided when writing to a LaserWriter.

An important difference, discussed earlier, is that QuickDraw provides patterns for the current pen; PostScript (at least until Level 2) does not. However, PostScript does support colors and gray scales that can be used in many of the same ways that patterns are in QuickDraw. As a part if this, when you draw in PostScript, you automatically use the current color, linewidth, and other variables from the current graphics state.

Finally, both QuickDraw and PostScript use rectangles around graphics. In QuickDraw, these rectangles are often used directly for placement and calculation. In PostScript, the rectangle around a graphic is called the *bounding box* and is not used so directly. However, it can play an important part in the handling and placement of graphics as you will see shortly. The major issue here is that QuickDraw graphics are in a box described by the rectangle values (top, left, bottom, right), whereas PostScript bounding boxes are given as the coordinates of two points: the lower left corner, written as (llx, lly); and the upper right corner, written as (urx, ury).

From the previous discussion, you know that QuickDraw coordinates are always integers since they represent pixels, whereas PostScript

coordinates may be fractional. But, you may say, obviously the final output of a PostScript device must use integer coordinates. Even at 300 dpi, there are no half pixels. This is very correct and it raises an important issue for LaserWriter programming in some circumstances. In general, the PostScript interpreter transforms your PostScript user coordinates, which are real numbers, into device coordinates through the CTM; then, when you go to actually make a mark, say, through a **stroke** command, the interpreter decides what pixels must be turned on to create the actual line. Under certain circumstances, when this happens on a medium-resolution device like a LaserWriter, it can cause an annoying visual effect: If you are making a series of grid lines, you may notice that some of the lines are thicker than others, even though you have defined them all to be the same width in PostScript. This occurs because the size of the pixels interacts with the placement of the lines. When the interpreter draws the line, it tries to make the line the width that you have selected at the point that you have chosen. However, the edge of the line may not lie along a pixel boundary. If it does not, the interpreter must choose which pixels to turn on to draw the line. Look at Figure 4-22.

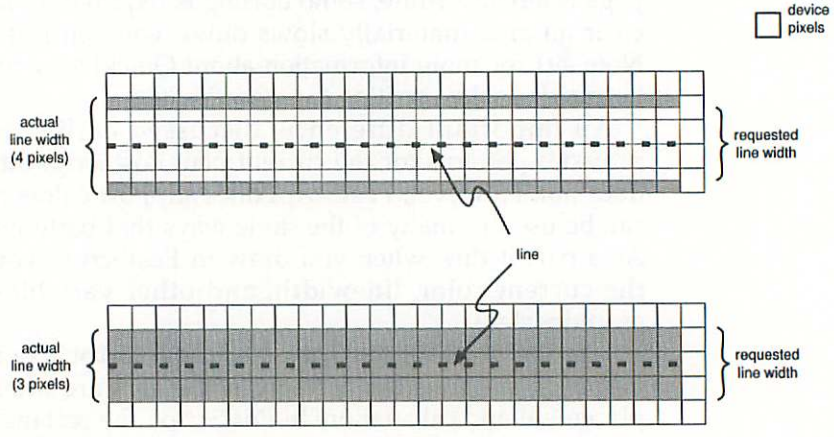

Figure 4-22. Lines and pixels

Here you have two lines, defined to have a desired width as shown. The width, as you see, is either three or four pixels, depending on the exact position of the starting point. On a high-resolution device, such variations do not matter because a difference of a single pixel is invisible to the naked eye. On a 300-dpi device like a LaserWriter, however, the eye can see a difference of one pixel. This can lead to having a grid with lines of a varying width; not at all what you want.

The solution (which is used by the Laser Prep file) is to convert the PostScript coordinates into device coordinates and then make sure that you start on a precise relative device coordinate for each line. Then you convert the device coordinate back into PostScript and proceed on your merry way. The actual code used in the Laser Prep file looks like this:

```
transform round .5 sub exch round .5 sub exch itransform
```

This routine takes two values on the operand stack: an *x* and a *y* coordinate. It first changes these coordinates into the equivalent device coordinate values, by using the **transform** operator. Then it rounds each device coordinate to an integer and subtracts exactly one-half. This positions you exactly in the center of a pixel on the device. Finally, it reverses the transformation back into user coordinates. This gives you nice, uniform straight lines at a possible cost of moving the starting point of the line by one-half pixel. Just as you can't know the position and the size of an atomic particle, you can't have both precise line width and precise placement at the same time. If you want to know more about these issues, I recommend reading *PostScript Language Program Design*, published by Addison-Wesley.

▶ Performing Text Operations

It may surprise you that this section focuses on text operations; after all, you have been using text for almost all of the examples since the very first one. You may think, therefore, that using text has already been covered quite adequately. And you would be, to some extent, correct; for basic text handling in PostScript is quite easy (as you have seen) and quite intuitive.

However, there is more to using text in PostScript than simply placing a single line or two of characters onto a page. PostScript also provides a number of features that allow you to use text as a graphic element. There are also internal complexities in fonts and text handling that you should know something about. This section, then, moves beyond your previous work with text and fonts to show you some more of PostScript text manipulation.

▶ Text display

Since you already know how to show basic text strings, let's do something a little different that will allow you to experiment with some of the typical PostScript text display enhancements. The code in Listing 4-18 shows two enhancements used together.

Listing 4-18. Code routine for PSText 16

```
% example PSText16
/SimpleLWdict 15 dict def
SimpleLWdict begin

% Global variables

% Procedures
/restoreCoord
%procedure to restore original PS coordinates from QD
%called as: -- restoreCoord __
%no requirements
{
    0 760 translate
    1 -1 scale
    newpath
}
bind def

%some text display options
restoreCoord
/T36 /Times-Roman findfont 36 scalefont def
/T12 /Times-Roman findfont 12 scalefont def

T36 setfont
150 340 moveto

(S)
gsave
    dup false charpath
    stroke
grestore

stringwidth
pop
0 add 8 rmoveto
T12 setfont
(an) show

T36 setfont
( ) stringwidth
pop -8
```

Listing 4-18. Code routine for PSText 16 (continued)

```
rmoveto
(F)
gsave
    dup false charpath
    stroke
grestore

stringwidth
pop
0 add 8 rmoveto
T12 setfont
(rancisco) show

150 320 moveto
(where little cable cars ) show

-1 5 (climb halfway) ashow

( to the stars.) show

end     %SimpleLWdict
```

This example uses only the **restoreCoord** procedure that you already know. The example begins by defining the two fonts that you will use in the display; here, you define one Times-Roman as 36 point, another Times-Roman as 12 point, and store these for later use. Then you set the Times-Roman as a 36-point font and move to a defined position on the page. (You could use any position, of course, but the relative positions must remain the same for the effect to work.) Then you push the single letter, **(S)**, onto the stack. You perform a **gsave** to store the current point and then execute these two lines of code:

```
dup false charpath
stroke
```

This begins by duplicating the letter on the stack, and then uses the **charpath** operator to convert that letter to an outline and add the outline to the current path. The **false** boolean is a signal to **charpath** that you want an outline for stroking, not filling or clipping. Once the path is

present, you **stroke** it to make it visible. Then you **grestore** to return to the current point that you had before.

At this point, you have drawn the outline of a large S onto the page at the designated point. However, the stroke consumed the path, as usual, and so the **grestore** was necessary to return you to the current point. However, this is the current point that was in front of the letter. If you want to continue printing (and you do here), you must move past the letter to the next print position. You do that by getting the **stringwidth** of the letter that you painted. This gives you the *x* and *y* dimensions of the letter, which is the distance that you have to move to get past the printed output. Remember that the *y* dimension is always 0 for a roman alphabet, so you throw that away with a **pop**. Since the letter is quite large, you move up (in the positive *y* direction) by 8 points while keeping the correct *x* position, using an **rmoveto**, and switch to the Times-Roman 12-point font. Then you show the next two letters: **(an)**. You switch back to the 36-point font and do the same trick again, with one additional wrinkle. You need to space past the end of the **(an)** that you just made and move back down to the baseline for the original S. If you simply use the 12-point space, the output will not look proportionate. Therefore, switch back into the 36-point font, get the stringwidth of a space in that font, and use that as an *x* value with the required –8 for the *y* value as operands for another **rmoveto**. Once in position, the code technique is exactly the same; this time, using an **(F)** and **(rancisco)**, respectively.

So now you have San Francisco printed onto the output page with fancy first capitals. Let's add one more touch to that, from the famous song. Move a little below the city name (I did the spacing by trial-and-error, actually) and display the string **(where little cable cars)** in the 12-point Times-Roman font using the **show** operator. Now comes the fancy stuff. Next you push a **–1** and a **5** onto the stack, followed by the string **(climb halfway)** and use a new operator, **ashow**, to display the string. **ashow** takes two numeric operands, representing distances in the *x*- and *y*-axes, in addition to the string to be shown. These values are added to the coordinates of each character as the string is drawn onto the page. In this case, the result is that each letter of the string is moved 6 units up the page and 1 unit back (the backward motion makes the effect a little steeper) to create a staircase effect. Then the phrase is finished off with the string **(to the stars.)**. The resulting output is shown in Figure 4-23.

There are several things to note about this example. First, you have a space at the end of the first string and at the beginning of the last one. If you place the space in the string in the middle, that space becomes part

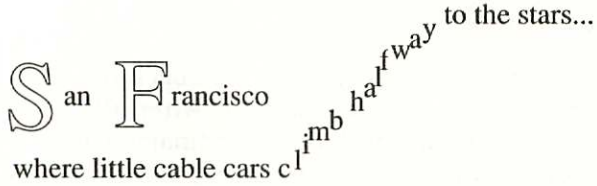

Figure 4-23. Page output from example

of the staircase effect, which is not what you want. Second, note that the staircase is somewhat irregular, because each letter moves based on the width of the preceding one, as is always true in proportional fonts. There are ways around this, but for the effect here I don't think that you need them—hills in San Francisco, after all, are somewhat irregular in shape. Finally, the choice of –1 and 5 was purely aesthetic; I encourage you to play with other values to observe the results. As an aside, if you make any major changes, you may also want to adjust the positioning of the city name as well.

The **ashow** operator is one of a group of operators that all perform a variation on the basic **show**; the others are **widthshow**, **awidthshow**, and **kshow**. There is no time or space here to discuss these, but if you want to know more about how to precisely place text in a PostScript environment, look up how to use these operators in any PostScript language reference manual.

Note ▸

In addition to these operators, PostScript Level 2 has added three more variants of **show**, taken from Display PostScript. These are the **xshow**, **yshow**, and **xyshow** operators. These allow a programmer to perform precise placement of characters in a string by providing the exact location of each character in the *x*, *y*, or both *x* and *y* directions, respectively. These new operators will be a great advantage if your application has already worked out the exact character placement on-screen for page layout or other advanced text processing, since you will now be able to transfer those calculations directly to the printed output. These new operators probably won't be available in most devices for some time, but when they are, they will be very welcome.

▶ Text controls

It may have occurred to you a while back that a lot of this coordinate changing seems rather redundant. After all, what difference does it make if the default PostScript coordinates have been reversed by the bottleneck routines? Why not use QuickDraw coordinates for your PostScript as well, instead of reversing them every time? A good question, and one with a good answer—and a neat trick inside it. The simple answer to why you need to fix the coordinates is that, if you don't, all your text will come out upside down and backwards, like mirror writing. Let's do a simple, short example where you don't use **reverseCoord** or anything like it and see the result. The code is based on the first example that you did and is shown in Listing 4-19 with the resulting output in Figure 4-24.

Listing 4-19. Code routine for PSText 17

```
% example PSText17
%example - reverse text
/Times-Roman findfont 20 scalefont setfont
150 650 moveto
(Well, imagine that!!!) show
```

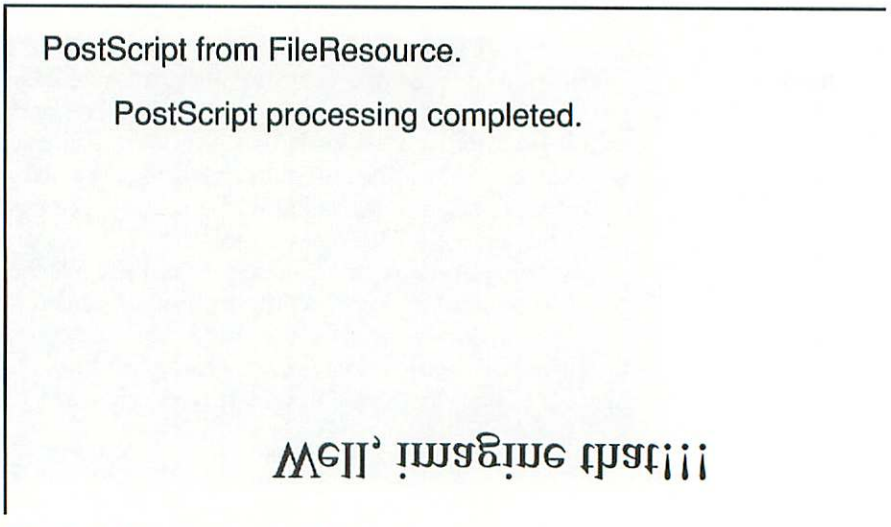

Figure 4-24. Page output (upside down)

Figure 4-24 is certainly not an acceptable output option; and that's why you have been converting the coordinates in each exercise. However, there is a simple and tidy way to reverse this effect without resetting the coordinates themselves. Basically, it involves setting the transformation matrix for the font so that the letters draw themselves right side up. This requires using another PostScript operator, **makefont**, in place of the more familiar **scalefont**. With a bit of work, in fact, you can define a new procedure that will do all the nasty bits for you, so that you can simply use the QuickDraw coordinates and fonts in the regular way. Let's look at the code in Listing 4-20.

Listing 4-20. Code routine for PSText 18

```
% example PSText18
/QDscalefont
%procedure to do a scalefont when coordinates are in
%   QuickDraw
%called as:     font pointsize QDscalefont font
%where all operands and results are the same as scalefont
%no requirements
{
    dup neg
    matrix scale
    makefont
}
bind def

%example - reverse text
/Times-Roman findfont 20 QDscalefont setfont
150 110 moveto
(Well, imagine that!!!) show
```

For simplicity, this uses the same code as you saw earlier, in Listing 4-19, but with one new procedure, **QDscalefont**. This procedure reverses the transformation of the characters and is called in the same way that the normal **scalefont** is called. Let's just review the code here for a moment. I'm not going to dwell on the details; just point out a few of the features that I think you may be able to transfer to your other code in the future. The procedure is called with the same two operands as a **scalefont**: the font dictionary and the desired size of the font, and it

returns the same result: a new font dictionary properly scaled. The heart of this is the **makefont** operator, which does the same job as a **scalefont**, except that it requires a complete transformation matrix, which is then applied to the font. This makes the **makefont** operator much more general than the simple **scalefont**, since it allows you to perform any linear transformation on the font.

All very nice, no doubt, for those to whom matrix algebra is as nothing; however, that's not me and may not be you either. If you can't simply whip out the required transformation matrix and write it down in the program, how can you derive it inside your procedure? Actually, PostScript (sneaky dog that it is!) has a simple way to generate transformation matrices by using the same coordinate transformation operators that you already know. The secret here is that, if you give a transformation operator, such as **scale**, a second operand in the form of a matrix, it applies the requested transformation to that matrix rather than to the coordinate system.

Clear as mud, right? Well, it sounds more complex than it is. You know what transformation you have to make to the coordinates to make this all work: You have to turn them upside down by the command

```
1 -1 scale
```

which reverses the change made for QuickDraw. If you could just apply that scaling when you drew a letter, all would be well. And you can, using the code shown in Listing 4-20. First, you take the single point size that you got as an operand and duplicate it—remember that **scale** requires two operands: one for the x dimension and one for the y. In this case, of course, both are identical. Now you use the **neg** operator to reverse the sign of the y scale; this is exactly equivalent to multiplying it by –1. Next you generate an *identity matrix* by using the **matrix** operator. This receives the result of the **scale** operator that comes next and can then be applied to the **makefont** operator to generate the desired font transformation. And that's all there is to it. Putting this code into practice gives you the desired result, as shown in Figure 4-25.

With this new routine, you can throw out the previous **restoreCoord** procedure. The major benefit is that the coordinates from your QuickDraw procedures remain good in your PostScript world. Notice in the example how you now move to (150, 110) to show your output; and also notice that the positioning, both horizontally and vertically, matches that of the strings that you have drawn with QuickDraw. This integration allows you more flexibility in the placement of graphics and text when you want to mix and match PostScript effects and QuickDraw graphics.

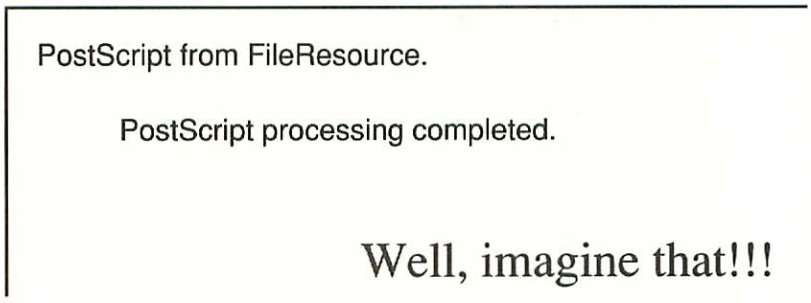

Figure 4-25. Page output from example

▶ Font operations

You have now learned something about how to use and display strings, using the fonts that are provided by your PostScript program. Where do these fonts come from, and how are they structured and used to generate output? Chapter 6 is entirely devoted to these interesting and complex questions, but before you get to that, you should know something about how to find and use basic font information. This section shows you how to list the fonts available on your system (assuming that you don't have an attached hard disk—those are covered in Chapter 6) and how to display simple information from within the font. Our code is shown in Listing 4-21.

Listing 4-21. Code routine for PSText 19

```
% example PSText19
/SimpleLWdict 15 dict def
SimpleLWdict begin

% Global variables
/myStr 255 string def
/LeftMargin 150 def
/TopMargin 100 def
/LineLead 12 def

% Procedures
/QDscalefont
%procedure to do a scalefont when coordinates are in
%   QuickDraw
```

Listing 4-21. Code routine for PSText 19 (continued)

```
%called as:     font pointsize QDscalefont font
%where all operands and results are the same as scalefont
%no requirements
{
    dup neg
    matrix scale
    makefont
}
bind def

/prStr
%procedure to print a value as a string
%called as: obj print --
%requires:  myStr (for working storage)
{
    myStr cvs show
    ( ) show
}
bind def

/newline
%procedure to advance a line
%called as:     -- newline --
%requires:  LineLead (for leading value)
%           LeftMargin (for margin)
%           TopMargin (for initialization)
{
    {currentpoint}
    stopped
    {
        LeftMargin TopMargin moveto
    }
    {
        LineLead add
        exch pop
        LeftMargin
        exch
        moveto
    }
```

Listing 4-21. Code routine for PSText 19 (continued)

```
    ifelse
}
bind def

%now some work with text
/Helvetica findfont 10 QDscalefont setfont
newpath

/FontDirectory load
{newline pop prStr} forall

300 100 def
currentfont
/FontName get
prStr

end     %SimpleLWdict
```

This code is the usual collection of routines and procedures, with two notable changes. First, you have replaced the **restoreCoord** procedure with your new **QDscalefont** and have revised the **TopMargin** variable to reflect the reversed QuickDraw coordinates, so that it is now 100 instead of the previous 650. Second, you have changed the line that calculates the new line position to

```
LineLead add
```

which now adds the line leading instead of subtracting it. Remember that QuickDraw coordinates increase down the page instead of up it. The body code, too, is quite familiar in form, if not in function. You start by setting the current font to 10-point Helvetica, using the **QDscalefont** procedure to ensure that the characters come out right side up and issuing a **newpath** to make sure that no QuickDraw path elements are still around when you begin. This was done before, inside the **restoreCoord** procedure, you remember.

The names of all currently loaded and available fonts are listed in the **FontDirectory** dictionary. To list the fonts, you load the **FontDirectory** and apply some of the techniques that you have used before, using a **forall** to enumerate the entries. The procedure here displays the names

down the page. When that's done, you load the Helvetica font and retrieve the name from it. This is printed by the **prStr** procedure in the usual fashion. The output is shown in Figure 4-26.

If some of these names don't look entirely familiar, don't be alarmed. You will find out all about these entries in Chapter 6. This example demonstrates two things. First, it shows how to get a basic list of the fonts in the system. You will see this again in Chapter 7 when you will use it to generate a list of fonts and return it to our application. Second, you have gotten some data—the name of the current font—directly from the font itself. We look at more of this information in Chapter 6; you will want to know how to retrieve such information for your work there.

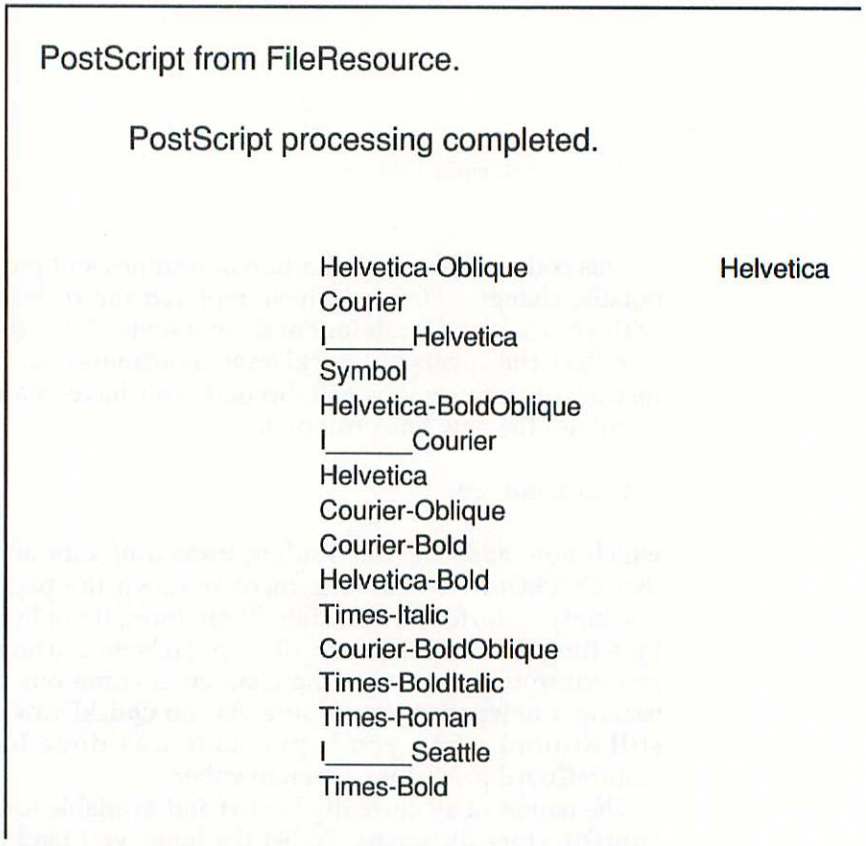

Figure 4-26. Page output from exercise

▶ Understanding Program Structure

This section discusses how a complete PostScript program is structured and what components go into its makeup. This is an important topic for the application programmer for several reasons. To begin with, when you understand how PostScript programs are constructed, you will understand how the LaserWriter driver builds an output file and how the Laser Prep functions are incorporated into that file. From this, you can see at least how to begin to build your own driver output, if required. Finally, the discussion lays the basis for the techniques that allow you to incorporate other files into your own documents, using the Encapsulated PostScript (EPS) format, which is discussed in the next section.

PostScript is an interpreted language that, in common with most interpreted languages, imposes virtually no internal structure on the programmer. Specifically, it has no reserved words and it has neither record formats nor standard headers and trailers to give some fixed shape to a program. However, if you want to produce page descriptions (which are PostScript programs) that can be used by other applications or by other programmers, you must follow some conventions about the structure of your programs.

Adobe Systems, the creators of the PostScript language, have developed and published guidelines about what correctly structured PostScript page descriptions should look like and how they should be put together. Following these guidelines allows you to both produce and use PostScript code that is easily portable and understandable in a variety of settings. These conventions allow the output of one application to be incorporated into another; they allow page descriptions to be spooled and managed in a reasonable fashion without parsing the actual PostScript code contained in the program; and they allow network control and management software to route page descriptions without too much concern about network resources and addresses. The complete set of these conventions is documented by Adobe Systems as the Document Structuring Conventions, which are currently at Version 2.1.

Note ▸

The terms "document" and "program" are used synonymously here and in most PostScript literature. Since PostScript is a general-purpose computer language, a PostScript program can be constructed to perform virtually any task; however, in general, PostScript programs are written to generate documents for display. On the other hand, all documents that are produced by a PostScript device must have been generated by a sequence of PostScript instructions, or, in other words, by a PostScript program. For these reasons, these two words are most often used interchangeably, as here.

▶ Basic components

A properly structured PostScript program has four basic components.

1. Header
2. Prologue
3. Script
4. Trailer

The header and trailer are made up of comments that have a special format so that they can be read and understood by other software or by humans, without any special knowledge of PostScript. The great advantage of using comments in this fashion is that they are ignored by the PostScript interpreter. In this, they are quite reminiscent of the PicComments that you have been using in QuickDraw, which convey PostScript information that QuickDraw ignores. In the same way, these *structural comments* convey information about the document that the PostScript interpreter ignores. The next section discusses the exact format and specific examples of these comments.

The prologue and script contain all the executable PostScript code in the document. These two components complement one another, together providing all the information that is required for creating the given document. The *prologue* contains all the procedures and definitions that are required in the program, and it never executes any code that makes marks on the output page. In short, if you were to execute a **showpage** operator at the end of the prologue, the page produced would be blank. The Laser Prep file is a good example of a typical PostScript prologue. The *script*, on the other hand, contains only data and procedure calls. It has no procedure definitions, but it makes all the marks on the page.

This division of labor has important consequences. First, the prologue and the script may be created and stored separately. The prologue is normally created by a programmer to carry out certain tasks. Very often, the prologue is stored in a private dictionary that then contains all the variables and procedures that are required to produce a document. The script, however, is usually produced by an application, which inserts data from a user—perhaps from the screen or from a file—and combines it with calls to the procedures that have been provided in the prologue. The prologue can be either included in the document file itself or downloaded separately into the output device. The most common approach is to send both of them together to ensure that the required information is available at the time of execution and to conserve device

resources. If the prologue is loaded separately (as Laser Prep is, for example), it must be placed outside the server loop, as discussed in Chapter 3, so that it remains in the device until a reset occurs.

When you create a structured document, you should take an additional consideration into account. It has to do with how you organize the script of the document. This requirement relates to any document that produces multiple pages; that is, any document that has more than one **showpage** operator. In such documents, the script should be set up so that each page is entirely independent of all the other pages in the document. Thus each page could be created without any of the code from other pages and would only require the information within the page itself and the prologue to print correctly. This has some important benefits. It allows document management software such as spoolers and print servers to break the document into pages or sections and print these in any required order, without any concern that information and procedures on one page depend on some other page. The most obvious application of this is in print spoolers, which might print pages in reverse order when printing on an original LaserWriter. Since the original LaserWriter produces its pages face up, printing pages from first to last leaves the output in reverse order; a common task for print spoolers is to print these pages last to first, thus making the output appear in correct, collated order. Such tricks, however, require that the pages of the document be structured so that *each page* is independent of all the others and so each page only requires the prologue to print.

If you consider how you have been approaching even the short examples that you have done here, you will immediately see how this approach works. In the examples, you have been creating a personal dictionary and placing your variables and procedures into that dictionary. This represents the prologue. Then, once that is complete, you send data, such as strings or coordinate values, and execute the desired procedures from the dictionary to create the output that you want. This is the script. As you see, this is a simple but effective structure that allows both flexibility and control.

▶ Conventions

The method for dividing your program up into these segments is through use of the Document Structuring Conventions. These conventions specify the exact form and content of the structure comments that break your program up into the four divisions that we have just discussed, and they provide certain critical information that allows the document to be managed by other software without requiring an actual

interpretation of the PostScript code. This section gives you a quick overview of the required structure and discusses a few of the comments in detail so that you will understand how they are used to place EPS files. A complete definition of the Document Structuring Conventions and a discussion of how they are used is available from Adobe Systems.

All document structuring information is provided by PostScript comments that are inserted into the document at appropriate locations. This method of defining structure has three major benefits. First, the comments have a specific format that can be recognized and processed by document management programs such as spoolers. Second, the comments can be read and understood by humans, so that the structure of a document is apparent to anyone reading the code. Finally, since these are PostScript comments, they are ignored by the interpreter, which continues to process the PostScript code that makes up the document in the ordinary way. This means that the structural comments have no effect on the actual page output, and whether or not a document executes (that is, prints) correctly has no relation to its structure.

Each section of the document has an individual format that provides the necessary global information regarding structure. This format information is provided by PostScript comments that follow a specific form and sequence, so that the document structure is controlled and maintained without any intervention on the part of the interpreter. This is an important point and should be stressed once more. The structure of a PostScript document is neither understood nor enforced by the interpreter. Documents that are not structured, or documents that violate the structuring conventions, may print perfectly correctly on the printer. Where they are likely to cause trouble is in networks or when used with document management software.

A complete discussion of all the document structuring comments would be quite lengthy, and is not necessary in order to deal with most PostScript output. This discussion focuses on only the most basic comments—those that are generated by almost all applications (including Laser Prep). In this way you can learn how to read and understand these comments without becoming mired down in all of the various possibilities. If you want to read about all the possible comments or to find out more about the ones described here, you should review the Adobe publication *Document Structuring Conventions, Version 2.1.*

The document structuring comments are a special form of the general PostScript comments that indicate the section of the document and provide other structural information within the document. These comments all begin with the characters %%, except the first one, which begins %!. Recall that all PostScript comments begin with a % and end

with the next newline. The structural comments do the same. PostScript comments, however, are free form: You may enter any information into them that you want. The structural comments, on the other hand, have a strict format, to allow document management software or other applications to parse the comments for specific information. (An example of this is given in the next section on EPS files.) Because the structural comments begin with these specific characters, %%, they can easily be separated from all other lines in the program, either by an application or by you.

Now you are ready to learn the exact format of the structure comments. Note that these must be entered exactly as shown; they are not free form like the standard PostScript comments. All these comments begin with the characters %%, except the first one, which must begin with %!, and these two characters are followed by a keyword. There are no spaces between the %% and the keyword. The keywords are case sensitive, as is all PostScript; you must enter and test for the keywords exactly as shown. In comments that require data, the keyword is followed by a : with no space between the keyword and the colon; the data elements are separated from the colon and from other data elements by a single space. The required newline must appear immediately after the last data element or after the keyword if there is no data.

All documents that use these conventions are called *conforming* documents, whereas all documents that do not follow these conventions are called *non-conforming*. A document may choose to use only as much of the structure as it needs for its own processing; the only requirement is that if the document does use the structuring conventions, it should use them correctly and consistently.

Four constraints apply to any conforming PostScript document. The first, and most important, is that the document be divided into a prologue and a script, as described earlier. This means that you should be able to divide the prologue from the remainder of the document and download it separately into the printer if desired. The second constraint is that the **%%Page** comment, which you will soon read about, must only be used for pages that are independent of all other pages in the script. This means that these pages can be printed by downloading the prologue and then the individual page, without requiring any other information. This is an important part of PostScript document structure, for the reasons that were discussed earlier. The third constraint is that comment lines be limited to 256 characters. This is not, as you know, a PostScript limitation, but is done so that the comments can be parsed by document manager software that has to allocate buffers and strings and so on in order to read the comments. If a comment must extend over

more than this maximum, you can use the following *continuation comments* to continue the comment data onto additional lines.

```
%%DocumentFonts: Helvetica Helvetica-Bold
%%+ Times-Roman Times-Italic Times-BoldItalic
%%+ Symbol
```

Continuation comments come immediately after the keyword comment line and begin with the characters %%+, followed by the additional data elements. The preceding example describes the fonts required in a document. Since a document can contain any number of fonts, many applications use these continuation comments to provide the required list of fonts. The fourth constraint is that certain PostScript operators must not be used, or must be used in specific ways, in conforming documents. The operators that must not be used are listed below.

banddevice	**copypage**	**erasepage**	**exitserver**
framedevice	**grestoreall**	**initclip**	**initgraphics**
initmatrix	**nulldevice**	**quit**	**setdevice**
setmatrix	**setsccbatch**	**settransfer**	**setcolortransfer**

Also, the **showpage** and **setscreen** operators must be used in specific ways and with certain controls in a conforming document. You may remember that these are the same operators that you were to avoid when producing code for use within a QuickDraw document; and that's not surprising. All these restrictions are to ensure that any conforming document can be placed inside another PostScript document and both of them will still print correctly. You might be surprised to learn that the output produced by the Printing Manager and the LaserWriter driver is not, strictly speaking, conforming. This occurs because the Laser Prep dictionary uses **settransfer** (correctly) to provide reverse image processing. The transfer function in PostScript is a sensitive tool and generally should not be used in a document because it affects the output of all parts of the page; in this case, however, this is the correct technique to create the desired effect, and Apple properly concatenates their changes to the transfer function to the current function. The net result is a document that should be able to be used without any problems wherever a conforming document is required.

The structure comments may be divided into three groups.

1. Header Comments—These occur only once in the document and, as the name implies, come before any PostScript code in the document.
2. Body Comments—These may appear anywhere inside a document and generally locate and limit the use of specific features and components that are used in the document.
3. Page Comments—These are similar to Header comments in that they appear in the script before each page to define page-level structure. Trailer comments are considered part of the page comments because they mark the end of the last page of output.

▶ Header comments

All PostScript programs that conform to these structuring conventions must begin with a comment line that starts with the characters %!. This has certain advantages in some specific environments (such as UNIX); you'll know if you are working in such an environment. In the typical micro-based environment, there is no advantage to following this recommendation if you're not going to follow the other structural conventions. In a conforming program, this first line gives information regarding the structuring of the program and is called the *version identifier*. Its format is as follows.

```
%!PS-Adobe-2.0
```

This comment may also contain one of three *keywords* following the 2.0. These keywords indicate to a document manager that the following document belongs to certain special classes of documents, so that it can change modes of processing, if required. These keywords are as follows.

- **EPSF**—This indicates that the file that follows is Encapsulated PostScript. A version number can follow the EPSF keyword. Use of this keyword is illustrated in the next section.
- **Query**—This indicates that the entire job that follows consists of PostScript query commands.
- **ExitServer**—This indicates that the job that follows will modify the persistent information within PostScript by issuing an **exitserver** operator.

The *header* comments themselves begin immediately after the version identifier and end with the comment **%%EndComments** or with any line that doesn't begin with the characters %%. Sometimes the information required for a comment is not easily available when the header comments are generated. In such cases, the information can be deferred to the trailer section of the comments. Where specific information is postponed to the trailer section, the appropriate header keyword must be followed by the value (atend) instead of by one of the following data value or values.

The following comments are the most common and basic header comments that you will see.

%%Title: title
gives the title of the document. This can be any text and is only terminated by a newline. The title is used for identifying documents; in some environments, this might be a file name or it might be formatted to be machine-readable.

%%Creator: name
the name of the person, user, or program (or maybe all three) who generated the PostScript document. This may be different from the person or user designated to receive the document, which may be designated by the **%%For** comment. The name consists of any arbitrary text, terminated by a newline.

%%CreationDate: text
gives the date and time of the creation of the document. There is no specific format for this information; all that is expected is that the text will be able to be read as a date and a time by humans. Generally, this comes from the system in whatever the standard system format may be.

%%For: userid
this is the identification of the person or user who gets the output; if this comment is absent, the output destination is presumed to be the same as the **%%Creator** comment.

%%Pages: number [page order]
this is a nonnegative, decimal integer that represents the number of pages expected to be produced on the output device. If no pages will be output, the number 0 should be inserted. The optional page order argument is provided to allow document managers to reorder

pages if appropriate. Allowable values are: -1, indicating that the document pages have been produced in *descending* (last page to first page) order; 1, indicating that the document pages are in *ascending* (first page to last page) order; and 0, indicating a special order. The 0 value indicates that the pages are in some defined order that the document manager should not change, for example, signature order. In the absence of a page order value, the document manager may, at its option, reorder pages within the document; however, if the 0 value is present, the document manager must respect the current page sequence and not alter it. This comment also can be deferred to the trailer section by the use of (atend).

%%BoundingBox: llx lly urx ury
this gives the dimensions of the box that bounds all the marks on a PostScript page. If the PostScript program produces more than one page, this is the largest box that bounds any page. The values are the *x* and *y* coordinates of the lower left and upper right corners of the page, given in the default user coordinate system. This comment may be deferred to the trailer section by the use of (atend). This comment is required for EPSF files.

%%DocumentFonts: font font font ...
lists the fonts used in the document by their PostScript names. This comment can be deferred to a trailer by the use of (atend).

%%EndComments
this comment explicitly ends the header section of the PostScript program. It is not required because any line that does not begin with the characters %% or %! also automatically terminates the header, as explained earlier.

▶ Body comments

The body comments break up the executable portion of a PostScript program into the prologue and the individual pages of the script. If a utility program acts on a structured PostScript program, it must respect these markers and keep the structure intact as it operates on the program text. In particular, it must retain the prologue at the beginning (since the pages in the script portion depend on it), and it must retain any trailer information at the end.

The *prologue* begins with the first line of the PostScript program that does not begin with the characters %% or %!, or it begins with the first

line after the **%%EndComments** statement, which terminates the header section. The only required comment in the prologue is **%%EndProlog**.

%%EndProlog
explicitly terminates the prologue section of the PostScript program. This comment is required in conforming programs.

Body comments may appear anywhere in a document and are designed to provide structural information about the organization of the document. In particular, many body comments are designed to match related information provided in the header comments section.

%%BeginDocument: name [version] [type]
%%EndDocument
These comments delimit an entire document file when it is included within another document description.

%%BeginFont: fontname [printername]
%%EndFont
These comments delimit a downloaded font that is included within a document. The optional *printername* is intended for use in networked environments where fonts may be tied to certain printers, by license or other registration arrangements.

%%BeginProcSet: name version revision
%%EndProcSet
These comments delimit procedure sets contained within the document.

%%BeginSetup
%%EndSetup
These comments delimit information that performs device setup functions. This comment is unusual because it must come immediately after the prologue, that is, after the **%%EndProlog** comment. It forms the first part of the script, before any page output is generated.

▶ Page comments

The *script* begins with the first line of the PostScript program following the **%%EndProlog** comment. The comments listed in this section are placed in the script code to mark page boundaries and to provide page information.

The following page-level comments are all self-explanatory; they mirror, at the page level, the information provided at a document level. The most common are as follows.

%%Page: label number
marks the beginning of an individual page within the document. The *label* and the *number* identify the page according to two methods. The *label* is a text string that gives the page identification according to the document's internal numbering or labeling scheme. For example, this might be page ix of the introduction, or page 2-4 (meaning the fourth page of Chapter 2, for example). The *number*, on the other hand, is a positive integer that gives the position of this page within the normal document output. This number begins at 1 and runs through *n* for an *n*-page document. This information is intended to be useful to utility programs; using this information, they can retrieve pages by either the internal page descriptions, for example, "pages 2-4 through 2-9," or they can retrieve pages by position, for example, "the last 10 pages." It also allows pages to be handled in nonsequential order, for example, to produce pages in folio order, for book binding.

%%PageFonts: font font font ....
this comment lists the fonts required on the current page. These will be a subset of the fonts listed in the %%DocumentFonts header.

The end of the last page of the script marks the beginning of the trailer processing, which begins with the **%%Trailer** comment.

%%Trailer
ends the script portion of the program and marks the beginning of the trailer section (if any). Any non-comment PostScript commands that follow this comment are presumed to be cleanup or otherwise not part of the page output.

The *trailer* section begins immediately after the **%%Trailer** comment that terminates the script. The trailer section consists of cleanup procedures, such as restoring the state of the printer, removing dictionaries used during processing from the dictionary stack, and so on, as well as information that has been deferred from the header section by use of one or more (atend) values in the header comments.

The order of the header and trailer comments is not generally significant. It only becomes important if there is more than one comment

with the same keyword. In that case, the data from the *first* header with the duplicated keyword is retained and used; for a trailer, the data is taken from the *last* keyword. This allows a utility program to modify the header and trailer information by simply placing the new header at the front of the PostScript program—after the version identifier, of course—and placing a new trailer at the end of the program, without having to delete any of the previous structural data. Remember, however, that the trailer data is used only if there is a header with the same keyword and the (atend) value.

Note ▸

These are by no means all the structural comments that can be used in a PostScript document. I have chosen only those that are most common or are produced by the LaserWriter driver, or those that you will encounter as you work with EPS files. The intention here is not to teach you all about document structure, since that would be a chapter in itself, but rather to give you enough information to read and use structured PostScript output. If you are going to be generating PostScript yourself, that is, without going through the LaserWriter driver as we have here, you should learn all the Document Structuring Conventions in some detail. The complete description is given in the Adobe documentation mentioned in this section and in the reference materials that are listed in the Bibliography.

▶ Program Integration

As was discussed in the preceding section, document structure is important for several reasons. One of the most important of those reasons is that correctly structured documents may be included within other documents as an entire unit, without the receiving document having any direct information about the actual contents or processing within the included document. This ability of PostScript documents was understood from the very inception of the language to be of the utmost importance, but it took some cooperation and standards to turn the ability into a practical reality. These standards are embodied in the Encapsulated PostScript (EPS) format, which you have, I'm sure, heard about before now.

Before discussing EPS files, however, some terminology should be established to help us keep these matters clear; EPS structure and

processing can be confusing enough without adding thickets of verbiage to them. As you learned in Chapter 2, all PostScript output may be considered to be "pages," even when the actual object is not a full page. For EPS purposes, the discussion focuses on placing pages within other pages; you are to understand by this that the pages may be fully or partially filled with some output. With EPS files, however, you should know about one limitation: An EPS file must represent only a single page of output. In other words, no EPS file can contain more than one **showpage** operator. Second, an EPS file is designed to be placed in another document file, and both of these files are generally created by applications. Here, the application that creates the EPS file is called the *source* and the application that accepts the file and places it into another document is called the *destination*. So, to review, an EPS file is a single PostScript page that is created by a source application for inclusion in another PostScript page by the destination application.

The EPS format was designed to solve two related, but not directly connected, problems. The first of these was how to format a PostScript page that contains an embedded page when the destination application cannot itself directly understand PostScript. Think about this for a moment, and you'll see the problem. The source application has generated some page of output as PostScript code. It knows what the page looks like and what the exact contents of the PostScript file must be. However, the destination application, unless it is itself a PostScript interpreter, cannot tell what the code that it has received will do. It cannot determine the size or shape of the page—remember that a PostScript page might be of any physical size, small or large. If it wants to place text around this page, for example, or place another page onto the same physical output, it can't do that without some information about the size and shape of the imported document. Further, if it wants to fulfill the requirements of the structuring conventions, it needs to know what resources this new document requires: fonts, paper color or forms, device features, and so on.

The second, related, problem is how to display this embedded page on the output screen. Although not essential, it would certainly be nice if the user could see the embedded document so that he or she would at least know it was there and that it was going to show on the PostScript output. Again, the destination application doesn't know PostScript, so it cannot simply draw the output onto the screen (unless the system is using Display PostScript, of course—but that's another issue, with its own can of worms).

These two issues are answered by the EPS file format. An EPS file consists of two pieces: a PostScript page description with a special

structure in the data fork of the file and an optional 'PICT' representation of the page in the resource fork of the file. In some ways (*imagine* my surprise!!) this is very reminiscent of the work that you have been doing in this chapter and the last. As in the examples in this book, the two forks work together to provide the information required by the destination application to both display the file on the screen and produce elegant finished PostScript output.

▶ Basic considerations

The Encapsulated PostScript format is designed, as the name implies, to allow documents, which generally are single graphic elements, to be successfully included as graphic elements, or "capsules," in other PostScript documents. EPS files are not designed to interact in most ways with the page output from the destination application. Specifically, the destination application does not modify the internal PostScript code to affect the appearance of the graphic, nor does it alter the internal 'PICT' representation. However, the destination application can perform simple placement and sizing of the graphic without any more information than that provided by the required structure comments. It is the destination application's responsibility to maintain the graphic environment to allow this: to create the necessary coordinate transformations to display the graphic correctly and to restore the original coordinates when it's done. Think of the EPS graphic as a photograph; like a photo, it can be enlarged, shrunk, rotated, or cropped by the destination application, but the actual image in the photo cannot be changed.

Three basic aspects make this process successful. First, the EPS page of the document being included must depend only on its own header information, making it independent of all other information or files. Second, the EPS page must have a bounding box, given in the correct comment format, that gives the dimensions of the graphic in PostScript coordinates. Third, the EPS page must have correct comments to define the fonts (if any) required by the document for correct processing. With this information, the destination application can correctly place and process the EPS graphic.

▶ EPS file format

To provide this essential information, all EPS files must conform to the document structuring conventions discussed in the last section. Specifically, EPS files must have certain structural comments and they must follow the restrictions on operator use and page structure described earlier.

As is true for all conforming files, an EPS file must begin with a correctly formed version comment as the first comment in the header. It should look like this:

```
%!PS-Adobe-2.0 EPSF-1.2
```

This comment tells the destination application that the following file conforms both to the document structuring conventions, at Level 2.0, and to the EPS file format conventions, at Level 1.2. Of course, either of these numbers might be larger if the file conforms to more recent versions. This comment also indicates to the destination application that the file is intended for EPS use; without this comment, the destination application may refuse to import the file, even if the file otherwise has the correct format.

The second required comment in a EPS file is the **%%BoundingBox** comment that specifies the area of the page that is covered by the image created by the PostScript program and that is represented by the 'PICT' graphic. For review, here is the exact format of this comment:

```
%%BoundingBox: llx lly urx ury
```

where the points (llx, lly) and (urx, ury) define the rectangular bounding box as described earlier. The dimensions are given in standard, default PostScript coordinates. Notice that these *must* be in default coordinates so that the destination application can use them for placement and sizing.

If the EPS graphic contains text, then the **%%DocumentFonts** comment must also be included. For review, this comment has the following format:

```
%%DocumentFonts: font1 font2 ...
```

and may be continued as described in the previous section. This comment tells the destination application what fonts must be downloaded for correct output of the graphic. It is the responsibility of the destination application to coordinate all the fonts and make sure that they are loaded as required, so this comment is used to provide the information about what fonts need to be coordinated.

The following comments are optional, in that they are not required for a correctly structured EPS file, but they are often used by destination applications to provide additional onscreen information when the graphic is displayed. This can be of special significance when the 'PICT' resource is absent or cannot be displayed for some reason.

```
%%Creator: name
%%Title: file name
%%CreationDate: date and time
```

In such cases, the destination application can use these comments and the bounding box information to produce an identification block that can still be represented onscreen for the user. These comments contain the data suggested by their names, although there is no specific format requirement for the data.

In addition to maintaining a conforming document structure, you should follow certain coding rules to ensure that an EPS file doesn't cause problems in the destination application's output. First, all stacks should be returned to their original state at the end of the processing in the file. This means, specifically, that the operand and dictionary stacks should have nothing on them that was not present at the time you began processing. No information should be left on any stack after the EPS file finishes execution. This is essential so that **save** and **restore** can function correctly. Second, the EPS file should not change any global information. The best and safest method of doing this is to create and use your own dictionary for the file, just as you have been doing in these examples. Obviously, in keeping with the first rule, this means that you have to remove the dictionary (or dictionaries, if you have used more than one) by a matching **end** operator when you are done processing. Finally, if you require any special information or setup in the EPS file, you should provide these resources yourself and not presume that they are already available. These rules are simply good PostScript coding for any document, but they are especially important for EPS files since, by design, these are included within other documents.

In addition to the PostScript code that generates the actual printed output, most EPS files contain the optional 'PICT' resource that gives you a screen representation of the graphic as well. This 'PICT' resource must have the ID number 256 and should contain all the information that is in the bounding box rectangle of the printed output. In this way, what shows on the screen will accurately represent the final output. This also allows the user to modify the image by scaling, rotating, cropping, and so on (assuming that the destination application supports this), while being comfortable that the result will look like what they expect.

Remember that the 'PICT' image is not required for a valid EPS file, it is only recommended. If you are creating PostScript files by hand and wish to make them EPS files, you may have to forgo the 'PICT' resource, unless you have a screen representation of the graphic already available.

▶ EPS transformations

An EPS file is designed to be included in another output page as a unit. This means that the source application expects that you will take the entire file, including the header and all, and include it within some other PostScript code. To do this successfully, or to print the file by itself for that matter, you need to determine several things: where on the output page you want to place the EPS graphic, what coordinate structure it requires, and what size you want it to be. In addition, you need to provide a **showpage** operator (if one is missing, as it often is) and a proper context for execution that includes saving the current coordinates before the graphic is drawn and restoring them afterward.

There is only space here for a brief overview of this process, without a real example, which would take quite a bit of explanation and code. The information here should be enough, however, to allow you to construct your own code for a simple EPS display if you want. We will concentrate on the PostScript code that is required for the display since the C code is almost identical to what you have already been doing or to familiar things such as reading and parsing a file.

Suppose you have an EPS file that has been displayed on the screen. From this display process you now have two important pieces of information. From the EPS file's **%%BoundingBox** comment, you have the coordinates of the bounding box as four numbers: llx, lly, urx, and ury. Similarly, when you display the 'PICT' representation of the graphic on the screen, the user is able to position it on the final output as shown on the screen, usually by moving the mouse. In any case, the position and scaling of the screen representation of the EPS graphic generates a QuickDraw Rect which gives you the screen coordinates for positioning as the four numbers: top, left, bottom, and right. The first set (llx, lly, urx, ury) is in default PostScript coordinates, whereas the second set (top, left, bottom, right) is in QuickDraw coordinates. The differences between these two should, by now, be quite familiar to you.

With this information, you can now generate the PostScript commands that place the EPS file where you want and at the size that you want with the code in Listing 4-22.

The code in Listing 4-22 establishes the required coordinate transformation by moving the origin of the EPS graphic from its original starting point of (llx, lly) to the new starting point at (left, top). Notice here that since you are working in the revised QuickDraw coordinates, you must scale and translate the coordinates from the EPS file in order to make the display come out right side up in the QuickDraw coordinates. If you had previously restored the original PostScript coordinates, for

example, by using a procedure such as **restoreCoord**, then you would have to make the appropriate reversals of the coordinates presented here.

Listing 4-22. Code to place an EPS graphic

```
/mySv save def
[llx lly urx ury]
[top left bottom right]
%first set small dictionary for our working space
2 dict begin
%now save the source and destination bbox
/dest exch def
/src exch def
%calculate the new origin
dest 1 get dest 0 get translate
%then calculate scale factor in x dimension
dest 3 get dest 1 get sub
src 2 get src 0 get sub
div
%now calculate scale in y dimension
dest 2 get dest 0 get sub
src 1 get src 3 get sub
div
%and scale the graphic
scale
%calculate the new origin
src 0 get neg src 3 get neg translate
end   % our small dict
%insert EPS code after this ...
mySv restore
```

The assumption then is that you have gotten the required eight numbers and placed them into two arrays, as shown in the beginning of the code. The code starts by doing a **save** so that you do not lose any virtual memory in this process; if you called this code by a PicComment, this step is superfluous because the PostScriptBegin and PostScriptEnd do this for you automatically. In that case, or if you are doing additional processing in this code block, you should replace the **save** with a **gsave**, and the corresponding **restore** with a **grestore**. This brings you back to the original coordinates for any remaining processing.

Obviously, this code deals only with the issue of placement and scaling of the EPS graphic in the destination. If you want, you can use similar methods to add code to provide rotation and cropping.

The remaining issue is what to do with the text requirement of the EPS file. The problem here is that since you are sending the EPS file through the Printing Manager by means of PicComments, the fonts that the document needs are not requested and downloaded as necessary and therefore may not be present in the printer. The solution to this is to force the Printing Manager to download the required font or fonts. One technique for doing this is given in Chapter 6, when we discuss all forms of fonts in more detail.

This brief look at EPS file use and control gives you some insight into how the PostScript document structuring conventions are used to provide services that would otherwise be almost impossible to work out. You see that this is all done without any direct intervention on the part of the PostScript interpreter. This allows these conventions to change and be expanded as necessary without any changes in the PostScript document itself. The flexibility and economy of the language are also evident here; you can see how simply you can perform powerful transformations to include one document inside another.

▶ Conclusion

This chapter has been a short look at PostScript programs and programming. It first gives you a simple tool for creating and testing PostScript code. Although not really adequate for full-scale development, this process is quite enough for you to explore some of the basic concepts of the PostScript language and their use in actual documents.

The chapter discusses some of the basic PostScript language structure, including procedures and dictionaries. It covers simple PostScript graphic and text effects to show you some of the power of the PostScript language and to illustrate some of the subtle, but very real and important, differences between PostScript and QuickDraw graphics. Along the way, as you are working through these examples, you also learn some simple tricks for debugging PostScript programs as you create them. In this way you can generate a small but useful toolbox of PostScript routines that you can apply over and over to your programming.

The chapter concludes with two sections on PostScript program global structure and how that structure is used in Encapsulated PostScript files. Although these sections don't have much code in them, they do give you the concepts and background for handling general PostScript files. You are now ready to proceed, with a little effort, to creating and debugging full-scale PostScript programs.

5 Basic LaserWriter Programming

Chapter Overview

In this chapter you begin the first excursions into using the LaserWriter as a LaserWriter. Up to now, you have been letting the Printing Manager determine whether or not you are writing to a LaserWriter. The code that you have generated will run correctly on any type of printer, even though, as you have learned, it will not produce identical results. This actually is a rather unsatisfactory situation, in that you are investing time and energy in code that cannot produce the desired effect unless it is run on a LaserWriter, and yet you don't know whether the LaserWriter is there or not. This is the situation that you resolve in this chapter.

The chapter involves all the basic elements of direct programming on the LaserWriter. It begins with an example that shows you how to determine whether you are writing to a LaserWriter or LaserWriter-compatible device. This is quite an interesting example, and it introduces you to techniques that allow you not only to identify the type of device, but also allow you to determine the device name and the user's name. Although you don't make use of these names here, they play an important role in future examples.

Once you know that you have a LaserWriter, you can begin to put that information to constructive use. To do so, however, you very often also need some feedback from the user regarding certain choices and options that only the user can really make. Therefore, the next section discusses how to insert new items into the job and style dialog boxes. This allows you to customize your user interface and solicit some

information from the user as you require it. In this section, you have an example of adding a single check box that allows the user to specify that a marking of DRAFT be placed behind the text being printed, if he or she desires it. This example is done in two parts, with the first part being the addition of the check box to the dialog box, and the second part being the use of this check box, in conjunction with the test for a LaserWriter, to determine how to mark the output page. This is a clear, if simple, example of how you can use these techniques.

This section, on the LaserWriter device driver, also includes a discussion of the various resources that are located in the driver that you can modify. The main purpose of modifying these resources is to change various messages from their default state into more useful and understandable forms. For example, you could modify these resources to provide information in another language, if desired.

The chapter continues with an exploration of the Laser Prep dictionary that is used for all printing to the LaserWriter. The examples here show you how to intercept both document files and the dictionary itself, and how to use that information to explore the dictionary and see how it performs. This enables you to make some judgments about how to interact with the dictionary in your own programs. This section is designed not to teach you what specific Laser Prep functions do, but rather to teach you how to discover for yourself how these functions work. The point here is that the Laser Prep file changes regularly, as Apple updates its printing software and as new features or devices come along. If you learn only about a specific version of the Laser Prep file, that information, sooner or later, must become obsolete. However, if you learn how to examine it for yourself, you will always be able to find out what you want to know. "Give a man a fish and you feed him for a day; teach a man to fish and you feed him for a lifetime"—well, as long as the fish don't run out, anyway.

The chapter next explores the various methods of sending PostScript code to the LaserWriter. There are more of these than you probably expect, and, although you have already used several of the most useful methods, this section provides a fairly comprehensive introduction to these options. But the best, like dessert, is saved for last. The last section of the chapter shows you how to create and load your own PostScript dictionary for use with your documents, and it discusses some of the pros and cons of this process. The example in this section is identical to the previous example, so you can see the changes that this process brings into your code and you can test it easily.

▶ Identifying the LaserWriter

The first task you must undertake, if you are going to use the LaserWriter, is how to determine that the printing device that has been selected by your user is, in fact, a LaserWriter or LaserWriter-compatible device. The correct strategy might seem to be to test something for the name "LaserWriter," and the device driver would seem to be the obvious place to start. Alas, things are not so simple.

You must remember that a wide variety of devices use AppleTalk and PostScript, and many of them are not, in fact, a LaserWriter. Indeed, they are so far from being LaserWriters that they must have their own device drivers installed in the System folder. As one good example, if you or your user have an HP LaserJet III printer with an AppleTalk interface and a PostScript upgrade, that printer works just like the LaserWriter, from an application and user viewpoint. However, because the device has different mechanical characteristics and internal functions than a LaserWriter, it must have its own device driver. Following the practice that you already have seen, this device driver is labeled "LaserJet III". And correctly so, because if you tried to label it "LaserWriter," you might be able to print to a properly configured LaserJet, but you'd never get it to work with a real LaserWriter. Moreover, it would be highly confusing to users that have more than one type of device when they all appeared under the same name but worked differently.

So you see the problem. To conform to the Printing Manager and Chooser requirements, the device driver must be named according to the actual device. From the application's view, however, all these devices work in the same way: They all speak PostScript and they are all on AppleTalk. Interestingly enough, there is no direct method to get out of this dilemma. You might expect there to be a system global, say, gPrDevice, that you could simply interrogate to determine whether the current printer is a LaserWriter or compatible device or some other type of device. However, remember that the party line on printing devices is to make your application work with all devices, regardless of the actual features of the specific chosen device. In that case, you should never need to know what the actual device is.

Of course, somebody has to know which printer has been selected because the Printing Manager and other Toolbox software must make decisions based on the actual device being used. So, the answer to the dilemma is to go to the same place that the system software goes to find out this information. Moreover, it's almost as easy as simply checking a global variable. The information is stored, logically enough, as a part of the print record, but a part that is a bit difficult to get and test. The actual

device information is stored as part of the wDev field of the prStl section of the print record. The actual device code is the high-order byte of this field.

This byte is set to show the generic type of device that the actual printer represents. The byte is set to 3 to indicate a LaserWriter-type device: that is, a PostScript-equipped, AppleTalk device. Other devices have other numbers: For example, the ImageWriter is 1 and the LaserWriter IISC is 4. The set of device numbers changes from time to time, as new devices are released. However, for our purposes here, 3 is the only relevant device number.

Note ▶

Technical Note #72 strongly advises against using this method to determine the current printing device. It tells you that this is an unsupported method of determining the current device, "meaning that at some point in the future it will no longer work." Moreover, Technical Note #72 warns you that new Apple devices and devices from other manufacturers may not set this byte correctly.

I'm really not sure what that all means. This is, after all, the method that is documented in *Inside Macintosh, Volume II*, and it does use a global variable that is well known and documented in a variety of publications, some official and others not. In point of fact, many developers, including major application developers, use this information, and it would be foolish for any manufacturers to cut themselves off from the wide variety of applications that can work with PostScript devices without a good reason. At the present time, to the best of my knowledge and research, all PostScript or compatible devices on an AppleTalk network set this number in wDev to 3. No doubt, some day, Apple may change this. But it isn't likely to happen in the near future, and it may never happen at all.

With this information, you can easily identify a LaserWriter device. You only need to retrieve the current print record. Make sure to call PrValidate before you use this method to ensure that the print record is compatible with the current device selected by the user in the Chooser. Then look at the prStl subrecord and check the wDev high-order byte for the value 3, conveniently presented as the constant bDevLaser in the Printing Manager.

▶ Basic device code

With the theory out of the way, you can proceed to the actual code to implement this technique in an example. However, it is always best to consider what you want to accomplish before you start to code, and this rule is even more important in object-oriented code than in traditional programming. Here, you want to test the print record to establish the current device, as described earlier, and then communicate that information to the rest of the program modules. The first task clearly should be done in the code that sets up and uses the printer. In the classes that come with Think C, this task is done by the CPrinter class. Since this is a Think C class, you don't want to modify CPrinter directly, so you create a new subclass of CPrinter, called CLaser.

Note ▸

You could, of course, directly modify the CPrinter class. The source code is easily available and, in fact, provides the guidelines for creating much of CLaser. You don't want to do that, however, for two reasons. First, as new releases of Think C are made available, you would have to implement these changes each time for every class that you have changed. Second, when you call the class from other programs, you might have changed the actual operation of the class in a way that is not compatible with the new use. By simply adding a subclass, you can easily add or modify methods and still not change the basic class behavior.

In addition to creating CLaser, you also have to modify your SimpleLWDoc class to set the printer object to the CLaser printer class rather than the default class. You also create and set a class instance variable, the isLaserWriter Boolean, that tells whether or not the current device is a LaserWriter. You could also, if you'd prefer, make a new method that tests the isLaserWriter Boolean and reports it. I don't see any real benefit to that (although there is no evident drawback, either), so here it is just tested directly.

You use the same code that has been used in the previous examples. Set up CSimpleLWPane to use the original PicComment (PicComment 192) so that you will have some appropriate output for printing. Copy all the rest of the modules into a new folder so that you can make the required modifications. Let's begin by looking at the required modifications to CSimpleLWDoc.

First, the class definition CSimpleLWDoc.c must have the initialization method changed. In the previous version, this method simply called the inherited initialization method and passed on the two parameters, namely, the supervisor and the Boolean that determined whether the document was printable. It is this Boolean that you must now trap. If you send the message to the supervisor that the document is printable, you automatically generate a call to CPrinter, which creates an instance of the printer class with its definitions. You don't want that; instead, you want to override those definitions with your new CLaser class. Therefore, you must create an instance of the CLaser class instead of an instance of CPrinter. Since CLaser is a subclass of CPrinter, you can still use the instance of CLaser anywhere that you used CPrinter, but you can also invoke your additional methods in it.

Listing 5-1. New **ISimpleLWDoc** method

```
/*****************************************************
 * ISimpleLWDoc
 *
 *  This is the document's initialization method.
 *  If your document has its own instance variables,
 *  initialize them here.
 *  The least you need to do is invoke the default method.
 *
 ***/

void CSimpleLWDoc::ISimpleLWDoc(CBureaucrat
*aSupervisor, Boolean printable)

{
    THPrint     macTPrint;
    short       devNo     = 0;      /* default value    */

    CDocument::IDocument(aSupervisor, FALSE);

    if (printable) {
        myPrinter = new(CLaser);
        myPrinter->ILaser(this, NULL);
        macTPrint = myPrinter->GetPrintRecord();
        pageWidth = (**macTPrint).prInfo.rPage.right;
        pageHeight = (**macTPrint).prInfo.rPage.bottom;
```

Listing 5-1. New **ISimpleLWDoc** method (continued)

```
        myPrinter->GetDeviceInfo(&devNo);
        itsPrinter = myPrinter;
    } else {
        myPrinter = NULL;
    }


    if (devNo == bDevLaser)
        isLaserWriter = TRUE;
    else
        isLaserWriter = FALSE;
}
```

The code in Listing 5-1 shows you the revised **ISimpleLWDoc** method. As you can see, the first thing you do here is to send the CDocument class a message to initialize the document, but with the Boolean value for printing forced to FALSE. This ensures that no printer object is created for the document in the default method, but everything else is set up correctly. Now your method takes over and tests the Boolean, printable, that you were sent with the document. If this is not true, you are done because the document is already set up as not printing. However, if it is TRUE, you must create an instance of CLaser and make that the printer instance both in this method and in the default supervisor method. The code block here is basically like that in CDocument: It creates an instance of the printer class called myPrinter, which is an instance of CLaser; it initializes it by sending it an initialization message; and it gets the current print record from the class and sets the page width and height from the page rectangle. Then it sends the new message **GetDeviceInfo** to the class, which returns the device number in devNo.

The very last step is to set the default instance of the printer to the correct, new instance by equating the default instance, itsPrinter, which is defined as an instance variable in CDocument.h, to the instance of CLaser that you have created. You need to do this last step so that any external method that references the printer class can send messages to the correct responder and will not be confused. If you did not set this, although your classes would know to send the message to myPrinter in CSimpleLWDoc, everyone else would test the default itsPrinter, which would be NULL, and therefore they would conclude that the document

was unprintable. On the other hand, if the document is, in fact, not printable (because the printable Boolean is FALSE), you should set your internal instance of the printer, myPrinter, to NULL and let everyone else use the default instance in CDocument. This finishes the setup of the instance of the printer class.

Since you retrieved the information about the device as devNo, you can now test it to see if your current printer is, in fact, a LaserWriter. If it is, you set the global variable isLaserWriter to TRUE; if it is any other value, you set the variable to FALSE. Note that, if the document was not printable, the default value of devNo is used, which results in isLaserWriter being set to the default value, FALSE. Note also the #include that provides the required reference to the CLaser.h header, so that the compiler will know what type of object the new myPrinter actually is.

The changes to CSimpleLWDoc.h are shown in Listing 5-2. This listing shows you the two new instance variables: myPrinter and isLaserWriter.

Listing 5-2. CSimpleLWDoc.h

```
/*****************************************************************
 * CSimpleLWDoc.h
 *
 *  Document class header for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
*****************************************************************/

#define _H_CSimpleLWDoc      /* Include this file only once */

#include <Global.h>
#include <Commands.h>
#include <CApplication.h>
#include <CBartender.h>
#include <CDataFile.h>
#include <CDesktop.h>
#include <CDecorator.h>
#include <CDesktop.h>
```

Listing 5-2. CSimpleLWDoc.h (continued)

```
#include <CDocument.h>
#include <CError.h>
#include <CPanorama.h>
#include <CScrollPane.h>
#include <TBUtilities.h>

#define BASE_RES_ID    400

#define  BASE_WINDOW    BASE_RES_ID     /* Resource ID for WIND template */
#define  BASE_PANE      BASE_RES_ID     /* Resource ID for ScPn template */

struct CSimpleLWDoc : CDocument {

                                /** Instance Variables **/
    struct CLaser    *myPrinter;
    Boolean          isLaserWriter;
    Handle           lwFileDataH;

                            /** Class Methods **/
                                /** Construction/Destruction **/
    void     ISimpleLWDoc(CBureaucrat *aSupervisor, Boolean printable);
    void     Dispose();

    void     DoCommand(long theCommand);

    void     Activate(void);
    void     Deactivate(void);

    void     NewFile(void);
    void     OpenFile(SFReply *macSFReply);
    void     BuildWindow(Handle theData);

                                /** Filing **/
    Boolean  DoSave(void);
    Boolean  DoSaveAs(SFReply *macSFReply);
    void     DoRevert(void);
};
```

▶ Code for CLaser class

Now you must define the new CLaser class. This class will be a subclass of CPrinter. Look first at the header file, CLaser.h, shown in Listing 5-3. The header is in the standard format, and shows that CLaser is a subclass of CPrinter. There are two instance variables: gLaserName, which is the AppleTalk name of the device that you have accessed; and gUserName, which is the name of the workstation that is presently active (that is, your computer).

Listing 5-3. Header file for new class, CLaser.h

```
/**********************************************************************
 * CLaser.h
 *
 *      Interface for the LaserWriter Printer Class
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#define _H_CLaser

#include "CPrinter.h"           /* Interface for its superclass */

struct CLaser : CPrinter {      /* Class Declaration            */
                                /** Instance Variables **/
    char                *gLaserName;
    char                gUserName[32];

                                /** Instance Methods    **/
                                    /** Contruction/Destruction **/
    Boolean     ILaser(CDocument *aDocument, THPrint aMacTPrint);

                                    /** Accessing   **/
    void        GetDeviceInfo(short *devNum);
                                    /**     Simple Function     **/
    void        Cur_Prntr( void );

};
```

Three methods are also defined. Two of them you already know about, from the calls in CSimpleLWDoc.c: **ILaser** and **GetDeviceInfo**. To simply get the current device number, these two would suffice. However, while you're at it, it is useful to retrieve some additional information about the selected printer if it is a LaserWriter, for example, its AppleTalk name. To do this, you create the auxiliary method **Cur_Prntr**, which finds and records this information. Although you don't require this information for this chapter, you will require it in later chapters, and now is the best time to retrieve it. This will save you revisiting this module in later chapters when you will require this information.

The heart of the matter is in the class methods defined in CLaser.c and shown in Listing 5-4. The beginning is quite straightforward. There are two constants, which we will discuss as they arise in the code segments; other than that, the beginning information is just like what you have seen and used before. The first method is **ILaser**, which initializes a new CLaser object. There isn't anything special that you have to do here, so it simply calls the CPrinter initialization method.

Listing 5-4. Header and initialization code for CLaser.c

```
/**********************************************************************
 * CLaser.c
 *
 *                          The LaserWriter Printer Class
 *
 *  SUPERCLASS = CPrinter
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#include "CError.h"
#include "CPrinter.h"
#include "CLaser.h"

#define CURRENT_VERSION     1
#define LAS_NOT_AVAL        415

extern CError       *gError;    /* Error handler                    */
```

Listing 5-4. Header and initialization code for CLaser.c (continued)

```
/**********************************************************************
 ILaser

        Initialize a Laser Printer object. Printer can be passed a
        handle to an existing Toolbox print record. If handle is NULL,
        a newprint record with default values is created.

        Boolean result indicates whether the print record passed as a
        parameter was changed because it was incompatible with the
        currently installed printer. This allows documents which
        associate a printer record with a file to update itself if
        desired.

**********************************************************************/

Boolean     CLaser::ILaser(
    CDocument       *aDocument,
    THPrint         aMacTPrint)
{
    Boolean     changed = FALSE;/* Has print record changed?     */

    /* set instance variables   */
    gLaserName = "";

    changed = CPrinter::IPrinter(aDocument, aMacTPrint);
    return( changed );

}
```

You also set the instance variable gLaserName here to a null string. You might note that, in this case, there is really no need to do so. Only one method can be called externally, and that is the method that sets gLaserName itself. However, in later work, you may want to have additional methods that can be called externally. When that happens, you need to ensure that some natural value is in gLaserName at all times because you cannot control the sequence of messages. On the other hand, it isn't ever necessary to set gUserName because you would only reference it if gLaserName exists. Therefore, any new methods test gLaserName for a valid name and exit (or return some default value, or whatever) if it is a null string, without ever looking at gUserName.

The next method is **GetDeviceInfo**, which is shown in Listing 5-5. This is the method that you would call to establish the nature of the device (is it a LaserWriter or something else?). The code is quite simple. It is called with a space for the device number as a parameter, and it performs the exact steps that you read about earlier. You begin by opening the printer driver. If there is no error, you validate the class print record, macTPrint, which is stored in CPrinter. Naturally, that too must be checked for an error when it returns. If there is no error, you close the printer and continue.

Listing 5-5. Code for **GetDeviceInfo**

```
/*****************************************************************
 * GetDeviceInfo
 *
 *      Returns information about the Device .
 *
 *      Note that this method calls PrValidate before returning the
 *      print record. This is important because the user could have
 *      picked another printer via the Chooser since the last time
 *      PrValidate was called.
 *
 *      This method extracts the pertinent information from the print
 *      record.The programmer could also use the GetPrintRecord
 *      command and access the information directly.
 ***/

void    CLaser::GetDeviceInfo(
        short           *devNum)        /* Device Number              */
{
        register TPPrint        recPtr; /* Deference of print record
                                           handle    */

        PrOpen();                       /* Open the printer driver    */
        if ( !PrError() ) {

                                        /* Check validity of print record */
             PrValidate(macTPrint);
             gError->CheckOSError( PrError() );
        }
```

Listing 5-5. Code for **GetDeviceInfo** (continued)

```
    PrClose();                      /* Close the printer driver          */

    if (macTPrint != NULL) {        /* Extract device from print record*/
        recPtr = *macTPrint;
        *devNum = (recPtr->prStl.wDev) >> 8;

    } else {                        /* There is no device info           */
        *devNum = 0;
    }

    if (*devNum == 3) {             /* If this is an LW type device      */
        Cur_Prntr();                /* get current printer name etc..    */
    }
}
```

Next you check macTPrint to see if, in fact, a print handle is in place. If the document had originally been set up as not printable, this would be null and all of this is irrelevant. If it's not null, however, then you have a good print record that reflects the current device that the user has chosen. From that, you can now determine the wDev value, in the next two lines of code, by looking at the wDev information and shifting it right so that only the high-order byte is saved into the devNum variable. If there is no print record, the devNum is automatically set to 0, which represents the Macintosh screen. This sets the return value automatically.

The last thing in the method is to call the auxiliary method **Cur_Prntr** to establish some additional information about the device if it is a LaserWriter. The code for **Cur_Prntr** is shown in Listing 5-6.

Listing 5-6. Code listing for **Cur_Prntr**

```
/*****************************************************************
 * Cur_Prntr
 *      checks Chooser to see what type of printer is selected
 *          and sets the global (instance) variables:
 *      gLaserName = name of the LaserWriter on AppleTalk
 *      gUserName  = name of the workstation (i.e. user)
 *      NB: This is just a way to get a bunch of code out of the way
 *          in GetDeviceInfo()
 ***/
```

Listing 5-6. Code listing for **Cur_Prntr** (continued)

```
void    CLaser::Cur_Prntr(void)
{
    int         filNum;
    Handle      laserNameH;
    int         loop;
    OSErr       osErr;
    int         prFoldRefNum;
    char        *pPrinterType;
    Handle      prTypeH;
    int         resFileNum;
    FInfo       temp;
    SysEnvRec   theWorld;
    Handle      usrTypeH;

    resFileNum = CurResFile();
    UseResFile( 0 );
    prTypeH = GetResource( 'STR ', -8192 );
        CheckResource( prTypeH );
    pPrinterType = *prTypeH;
    DetachResource( prTypeH );

    usrTypeH = GetResource( 'STR ', -16096 );
        CheckResource( usrTypeH );
    for (loop=0; loop<(*usrTypeH)[0]; loop++)
        gUserName[loop]=(*usrTypeH)[loop+1];
    gUserName[loop]=0;
    ReleaseResource( usrTypeH );
    UseResFile( resFileNum );

    osErr = SysEnvirons( CURRENT_VERSION, &theWorld );
    if (gError->CheckOSError( osErr ) )
        prFoldRefNum = theWorld.sysVRefNum;

    filNum = OpenResFile( pPrinterType );
        gError->CheckOSError( ResError() );

    GetFInfo( pPrinterType, prFoldRefNum, &temp );
    if ( temp.fdType != 'PRER' )
        gError->CheckOSError( LAS_NOT_AVAL );
```

Listing 5-6. Code listing for **Cur_Prntr** (continued)

```
        laserNameH = GetResource( 'PAPA', -8192 );
          CheckResource( laserNameH );
    HLock( laserNameH );
    gLaserName = *laserNameH;
    HUnlock( laserNameH );
    DetachResource( laserNameH );

    CloseResFile( filNum );
}
```

The **Cur_Prntr** method is really the most interesting code in this class. It uses some of the internal information inside the system to retrieve the name of the LaserWriter and of the user, and it stores this information in the instance variables that you have already defined. You start by saving the current resource file number so that you don't destroy your own program. Then you get the System resource file for the current system. In this resource file are two 'STR ' resources. The first one is –8192, which gives you the name of the current device type. This name defines the device driver that is being used by the Printing Manager. This isn't quite what you need yet, but it is the first step to getting the device name. The second one is –16096, which gives you the user name directly, so that you can simply store the name into your global variable, gUserName. When you're done with the System resources, you restore the natural order of things by reinstalling the original resource file that you saved earlier.

Next you get the current System folder from the SysEnvirons record. The request to SysEnvirons only asks for Version 4.1 or later, by setting the requested version in the CURRENT_VERSION constant to 1, so that this isn't a very onerous test; the main purpose here is to get the current System folder so that you can safely retrieve the device driver. If you were really being good (or if this were a commercial application), you would have some checking and error handling code here to ensure that everything went all right even for older versions of the system. As usual, that sort of complication is omitted here in favor of clarity.

The next line of code is the critical one. It gets you the file information record for the device driver that was named in the System resource file. The device driver must be in the current System folder if the printing process is going to work correctly, so that's where you look. The file you get must have a file type of PRER if it is an AppleTalk device, so you first check for that. If it is not the correct type, you force

an error with error code 415 (an arbitrary number) which was set by the LAS_NOT_AVAL constant. We will discuss how to handle this error code in a moment, but for now just notice that the default behavior of CError puts up a dialog box with this error number as a result code, which is quite satisfactory for a first test.

If this is an AppleTalk device driver, you proceed to open the resource fork attached to the driver file. From this, you retrieve the resource 'PAPA' –8192, which is the PAP (Printer Access Protocol) address of the device that this resource looks for when it prints. This is the name of the LaserWriter that you want to save. And that's what the next four lines of code do: lock the name handle, store the name, unlock the handle, and release the handle. Nice and tidy. Finally, you close the device driver resource file and you're done with this process. You should also know that, if no printer had been selected yet, then the name string would be empty. That's why you could set the name to an empty string as a default. Then if you need to test later, in some other method, to see if the device has been chosen, you'll have the same test for a null string and get the correct results either way.

So now you have established several pieces of information. You have determined the actual type of the current printing device and returned that to the calling method. If the device is a LaserWriter (or a compatible device), you have also retrieved the current user's name and the AppleTalk name for the device and stored those as instance variables.

If you run this project now, with the Debugger on, you can see that the device type is correctly returned to the document and that the required global variables are set. For now, that's all that you're going to do. In the next example, you will see how to alter Printing Manager Job dialogs so that the user can tell you something about the document being printed. The example after that shows you how to combine these two techniques to make use of this information in a more concrete manner.

▶ Error handling

You can and should make one more small improvement here: to provide improved error messages for this project. Now that you are beginning to work more directly with the LaserWriter, it is more important that you tell your users exactly what is going on, particularly if there is an error. Let's start by providing for the error that you created in the previous code and also by providing for the most common LaserWriter error: when the printer is not on the network.

CError has a simple, general, and very effective mechanism for handling all errors. It puts up a standard dialog box that simply tells the user that an error has occurred and shows the result code that caused the error. Before it does, however, it checks a special set of string resources, called 'Estr' resources, to see if there is a string with the same number as the error result code. If there is, it uses that string as the information for the dialog box. If not, you get the default result.

You have deliberately forced an error, number 415, if the device driver is not AppleTalk. This was generated by the LAS_NOT_AVAL constant in the CLaser code. So what you want to do is add an 'Estr' resource with the number 415 to tell the user what has happened when this error occurs. At the same time, you can provide a special error message to cover the most common occurrence when you are printing to an AppleTalk network: that the device is not available on the network for some reason. This error has a result code of –4101.

You could, of course, use ResEdit to add these resources to the SimpleLW.π.rsrc file. That's fairly easy, but it doesn't provide you with any real backup if you have a problem later. Instead, let's build an RMaker file that automatically adds these strings to the current version of SimpleLW.π.rsrc. This file is shown in Listing 5-7.

Listing 5-7. Estr.Make.r to add error messages

```
!SimpleLW.π.rsrc

* added resource declarations follow ...

TYPE Estr = GNRL
PrinterNotFound,-4101    ;; Error #410
.P                       ;; this is ASCII text comment
Printer not found or closed! ++
Check that printer is on and connected to the network.

LWNotAvailable,415       ;; Program Error 415
.P
Requested printer is not available! ++
Use CHOOSER to select a printer.
```

The code in Listing 5-7 shows you two new resources being added to SimpleLW.π.r. The code is added to the file because the file name, at the first line of the code, begins with an !, which tells RMaker to combine

these resources with the existing resource file of the given name. The resources must be cloned from the 'GNRL' type because they are special to Think C, but that's quite easy since the resources are simply Pascal strings. Then you define each of the resources and give them more informative dialog than the simple result code. This dialog is much preferable for users since it lets them know what happened more clearly than a cryptic result code. In the case of error -4101, it is most important, since effective corrective action is usually to turn on the device or make another Chooser selection. In the case of error 415, it is even more essential because this error is an application-generated result code that is not documented in any of the usual places.

There are a number of errors that are only returned for the LaserWriter or similar devices. Table 5-1 shows all the Printing Manager errors that you might see at one point or another in your coding.

Table 5-1. Printing Manager error codes for the LaserWriter

Error Value	*Description*
–4096	No free Connect Control Blocks (CCBs) available.
–4097	Bad connection refNum.
–4098	Request already active.
–4099	Write request too big.
–4100	Connection just closed.
–4101	Printer closed or not found.
–8132	Timeout has occurred on the AppleTalk link. This usually occurs when the document requires an extremely long time to image on the device.
–8133	A PostScript error has occurred in the code being executed by the device.

Some additional error codes are defined, but you should never see them. They are handled by the Printing Manager itself as it is processing. They occur between a PrOpenDoc and a PrCloseDoc, and they are the reason that Technical Note #118 warns you not to test and handle errors until you're done with your print loop, since the Printing Manager itself will correct these errors before you end the printing process. The set is listed in Table 5-2. Of all these errors, the only one that is likely to occur is the one you have just provided for: –4101. That error, however, happens quite frequently—at least it does in my office, since I regularly and absentmindedly forget to turn on the printer before I start to work.

Table 5-2. LaserWriter errors that are automatically corrected by the Printing Manager

Error Value	*Description*
–8149	Version mismatch between the Macintosh system and the chosen LaserWriter. No software reset of the printer is possible.
–8150	No LaserWriter has been chosen.
–8158	Version mismatch between the Macintosh system and the chosen LaserWriter. A software reset of the printer is possible.
–8159	No Laser Prep file is installed on the LaserWriter.
–8160	Zoom scale factor is outside of the possible range.

Since these error codes are handled internally, they are subject to change without notice by Apple as the software changes. You really will never need to use these codes; they are listed here solely for reference.

▶ Using the LaserWriter Driver

There are a number of features in the LaserWriter driver that a programmer can use, and some that you should certainly be aware of when you are programming a LaserWriter. There isn't room here for all the possible features and issues concerning LaserWriter, so you will concentrate on one major point and cover a few minor points as you proceed.

The major point is how to add information to the style and job setup dialogs and how to use the information that you get back from them as a part of your application. The most important minor point is a discussion of the customizable and localizable resources that you can change in the LaserWriter driver software to make it more friendly in other climes and cultures. This type of *localization*, as it is called, is one of the real strengths of the Macintosh. Moreover, understanding what resources need localization can also help you understand what these resources do and how they are used.

▶ Dialog box choices and options

As you read in Chapter 1, two basic dialogs are used to set up the information required for printing. The *style dialog* is the dialog that you get when you choose Page Setup... from the File menu. This dialog

allows the user to set information that affects the page size and layout for the entire document. Figure 5-1 shows a typical LaserWriter style dialog.

The intention is to allow the user to use this dialog box to set any specific information that you need to match the document to the selected printer. The *job dialog* is the dialog that you get when you choose the Print command from the File menu. This allows the user to set information about this specific print job, for example, paper source and number of pages to print. Figure 5-2 shows a typical job dialog box.

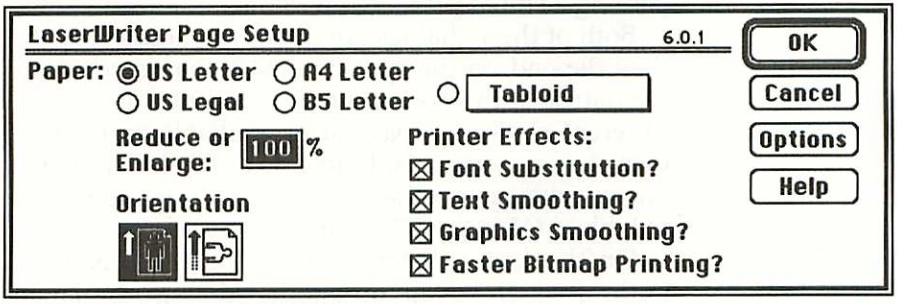

Figure 5-1. The style dialog box

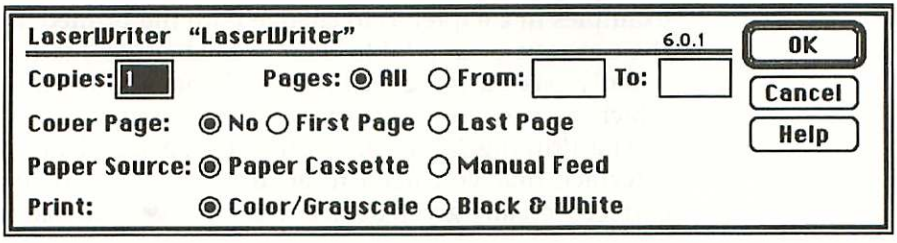

Figure 5-2. The job dialog box

As you see, each of these dialogs has a place in the printing process. The style dialog sets information that depends on both the nature of the printer being used for output and the exact features of the document. This information will remain useful as long as the structure of the document and the chosen printer remain the same. The job dialog, on the other hand, contains information that is relevant only at the moment of printing and has no persistent value.

Since the information in the style dialog is specific to the document and remains generally the same from one printing of the document to the next, it is stored by the Printing Manager as part of the print record.

For this reason, most applications store the print record with the document in some form and restore it when the document is called back into the program. Then the stored print record can be checked, using a call to PrValidate, when the document is called back into the application. This checks for compatibility with both the current printer and the current version of the Printing Manager, and it resets any fields that require it to appropriate default values. It also returns a Boolean value that indicates whether it made changes; you could use this to request that the user select Page Setup... to check the new options. If you noticed, this can be done in **IPrinter** (and in **ILaser** which calls it) by passing a stored print record to the initialization method.

Both of these dialogs are directly generated by the device driver, and they depend on the specific device that is currently chosen as well as the nature of the document. For example, different output devices have different choices of page sizes available and support different media. One device may be able to print only letter size (8 1/2 in. by 11 in.) pages, whereas another supports a complete range of sizes from letter to tabloid (11 in. by 17 in.) and so on.

On the LaserWriter style dialog there is an additional Options button, which brings up another dialog, similar to the one shown in Figure 5-3. This dialog provides additional controls over the printed output, mostly by setting controls in the Laser Prep file. Here, for example, is where you previously set the Larger Print Area box (see the examples in Chapter 4) to change both the printed area of the page and the amount of available memory. Changes made in this dialog are usually passed to Laser Prep as changes to certain control variables, which are used as part of the document setup.

The Printing Manager is carefully designed to provide most of the interface that you need to ensure that your documents are printed correctly on any given device. The job and style dialogs make it easy for

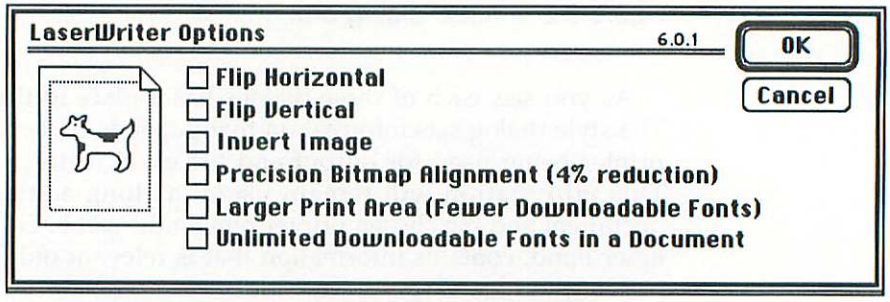

Figure 5-3. Options dialog box for a LaserWriter

the user to modify the printing parameters without causing too much concern or special coding inside the application. Nevertheless, sometimes it is both useful and necessary that you be able to make changes to these dialogs.

▶ Altering Printing Manager dialogs

When your application provides features that depend on a specific output device, you very often need to add information to the standard style dialog and job dialog boxes. You need to do this so that the user has one consistent place to go to for the same type of information. It would be very inconvenient and disorienting (although not unusual in *other* computer environments) to force the user to set document information or printing options in two different places, for example.

Let's just briefly review the construction and use of dialogs generally, with specific attention to the style and job dialogs. These two dialogs are stored in the device driver file for a specific device. Thus the LaserWriter dialogs are stored in the LaserWriter files as standard 'DLOG' resources. Each 'DLOG' has a 'DITL' resource associated with it; and each 'DITL' contains a series of items that are numbered and that each represent a single element or control in the dialog. When you read the dialog into memory in your application, these items are presented in a list that you can read and manipulate. In particular, each item is represented by a number that corresponds to its position on the item list both in the resource file and in memory. All of this is covered in *Inside Macintosh, Volume I,* in the chapter on the Dialog Manager. Changing and controlling your own entries in the style and job dialogs requires accessing and modifying the item list for the given dialog.

Before you start to modify these dialogs, however, you need to consider a few ground rules. Following these rules will help make your application compatible with any future changes to the printing process.

- Only add new 'DITL' entries at the end of the current list. This also means that you should not try to remove items from the middle of the list or change the order of the current list *in any way.*
- Don't count on the position of any items remaining the same. This means that you can't rely on the Option button being the same in every version of the style dialog, for example. It also means that you can't rely on the number of items in the list remaining the same. When you add items to the list, be sure to use a method that doesn't rely on there being some exact number already there.

- Apple reserves the top half of the screen for the current dialogs (and future extensions). Therefore, don't use more than half a standard screen for your extensions.

If you are going to add very much information to a job or style dialog, you may want to consider adding your own dialog that is accessible from the related Printing Manager dialog, by a button, for example. In this way you would avoid confusing the user with a variety of choices and making the screen look quite different than normal. This is obviously the correct, indeed the only, alternative if your requirements take up more than half of the standard screen.

To add things to these dialogs, you first have to understand how they work. This is covered in Technical Note #95, which gives you the basic information on how to work with these dialogs. The whole process hinges on the access and use of the TPrDlg record. This record provides all the necessary information for the control and display of the dialog records. It has the following structure in C.

```
struct tPrDlg {
  DialogRecord  Dlg;        /* the Dialog window */
  ProcPtr       PFltrProc;  /* the Filter Proc */
  ProcPtr       pITemProc;  /* item evaluation Proc */
  THPrint       hPrintUsr;  /* the print handle */
  Boolean       fDoIt;
  Boolean       fDone;
  /* four long integers -- reserved by Apple */
  long          lUser1;
  long          lUser2;
  long          lUser3;
  long          lUser4;
  int           iNumFst;    /* numeric edit items */
  int           iNumLst;
  /* and more stuff as required by the specific dialog */
  } TPrDlg;
  typedef struct tPrDlg    *TPPrDlg;
```

This structure provides you with the necessary information to modify the dialog items.

Both the job dialog and the style dialog return the same structure with different pointers and information. Of course, in the ordinary course of printing, you don't access this information directly at all. You issue calls to PrStlDialog and PrJobDialog in response to the user's command choices from the menu. These routines, in turn, actually call a common routine called PrDlgMain, with the following Pascal definition:

```
FUNCTION PrDlgMain (hprint: THPrint; pDlgInit: ProcPtr)
BOOLEAN;
```

PrDlgMain in turn calls the pDlgInit routine to set up the appropriate dialog (either the Job or Stl dialog, as required) in the Dlg entry of tPrDlg; the item evaluation procedure, or *item hook*, in pItemProc; and the dialog event filter procedure in pFilterProc. The dialog initialization routine that is passed as a parameter to PrDlgMain is set according to the type of dialog being used: PrStlInit for the style dialog and PrJobInit for the job dialog. These are the routines that you must replace in order to modify the given dialogs. Their definitions, in the usual Pascal format, are as follows.

```
FUNCTION PrStlInit (hPrint: THPrint): TPPrDlg;
FUNCTION PrJobInit (hPrint: THPrint): TPPrDlg;
```

As you can see, these are functionally identical.

Once the initialization is complete, PrDlgMain calls ShowWindow to make the window (which is initially invisible) visible. Then it calls ModalDialog using the filter procedure given in the dialog record. When an item is hit in the dialog, the routine pointed to by pItemProc is called to process the hit. This routine must handle the hits correctly, and it is another procedure that you have to modify in order to handle new entries into the dialog boxes. When the OK button is pressed (either by clicking on it or pressing Return or Enter), the entire print record is validated using PrValidate. Of course, if the Cancel button is pressed, no validation takes place.

So the entire process is essentially identical for each of the dialog boxes. The differences are so minor that you can, quite effectively, substitute one for the other by simply changing Job to Stl every place. Although the following discussion concentrates on the job dialog, the exact same routines can be used for the style dialog simply by substitution of Stl for Job in every routine.

▶ Adding information to the job dialog

To modify a dialog box then, you must add an item or items to the item list after the other items but before the dialog box is displayed. Then, once the box is displayed, you must intercept the hits in order to screen out the ones on your new items and pass the others along to the system procedures, which will handle them. You do that by replacing the standard item evaluation procedure in TPrDlg with a new one. Having done this, you can let the dialog processing proceed normally.

Let's just recapitulate what you must do to provide your own additions to the job dialog box (remember, the steps are identical for the style dialog, but with the names changed). The steps are as follows.

1. Insert your own procedure for the usual call to PrJobDialog.
2. Use that procedure to call PrDlgMain with your own initialization process.
3. In your initialization process you
   - add additional items to the end of the item list.
   - save the last item number of the current list for future processing.
   - insert your own processing routine for the standard item processing routine in TPrDlg.
   - save the address of the standard item processing routine for future processing.
4. In your item processing routine you
   - test for your items, using the saved value of the last normal item number.
   - handle hits on your items with appropriate processing.
   - call the standard routine to process any other item hits.

See how simple it is? Typical common modifications to the job dialog are boxes that allow the user to specify that pages are to be printed in reverse order, for example—an extremely useful feature when printing long documents for those still using the original LaserWriter, which stacks the pages face up.

That gives you the theory, so now let's look at an example. This example adds a simple check box to the job dialog that determines whether or not the user wants this output marked as a draft. Ultimately you will use this information to print the word "DRAFT" at an angle behind each page. This is intended simply as illustrative code, not as

code that would really be used in a commercial application. Ultimately there will be three segments to this example. You must modify the CLaser class to add the processing outlined earlier. You must modify the header for the CLaser class to support these new procedures. Finally, you must create the required 'DITL' resource to add the check box to the job dialog.

Before beginning, however, you must determine where to put your new item: the job dialog or the style dialog. In this case, the logic is that this is something that may change with each printing of the document. The first few times, the document is labeled "DRAFT," but, after some time, the document will no longer be a draft. Since this is something that changes with each printing, it should be part of the job dialog box. On the other hand, if this were something like registration marks, say, that require a certain page size to be effective and that are likely to persist in the document for its entire life, then it might make more sense to place the control into the style dialog.

The new procedures must be placed into CLaser.c. First, you need to override the default **DoPrint** method. This is the method that calls PrJobDialog, which you must replace with your own code. Listing 5-8 shows you the new **DoPrint**. This method is just like the default method except that you have replaced the call to PrJobDialog with a call to **MyJobDialog**. Listing 5-9 shows you the new **MyJobDialog**.

Listing 5-8. The revised **DoPrint** method in CLaser.c

```
/**********************************************************************
 * DoPrint (OVERRIDES DEFAULT METHOD in CPrinter)
 *
 *      Respond to the Print command by displaying the standard
 *      printer job dialog. Then print the associated Document if the
 *      user confirms the print request.
 ***/

void    CLaser::DoPrint()
{
    Boolean     wantsToPrint = FALSE;

    if (OpenPrintMgr()) {           /* Open printer driver      */
        wantsToPrint = MyJobDialog(macTPrint);
        gError->CheckOSError( PrError() );
    }
```

Listing 5-8. The revised **DoPrint** method in CLaser.c (continued)

```
    PrClose();                  /* Close the printer driver     */

    if (wantsToPrint) {
        SetCursor(*gWatchCursor);
        PrintPageRange((**macTPrint).prJob.iFstPage,
                       (**macTPrint).prJob.iLstPage);
    }
}
```

Listing 5-9. **MyJobDialog** method from CLaser.c

```
/*****************************************************************
 * MyJobDialog
 *
 *      Modify the standard job dialog to add our new item.
 *      Then continue by displaying the standard printer
 *      job dialog.
 ***/

Boolean CLaser::MyJobDialog (
    THPrint aMacTPrint)
{
    Boolean     recOK   = FALSE;

    extern  pascal  Boolean     PrDlgMain ();
    extern  pascal  TPPrDlg     PrJobInit();

    gPrtJobDialog = PrJobInit(aMacTPrint);
        gError->CheckOSError( PrError() );

    recOK = PrDlgMain(aMacTPrint, &MyJobDlgInit);

    return(recOK);

 }
```

This routine starts by defining the Boolean variable recOK to hold the results from the dialog validation. The results here are the same as those returned by PrValidate, so a return of FALSE, as you have as a

default here, means that the record has not changed. You don't have to define the calls to PrDlgMain and PrJobInit that you will be making because these are Toolbox calls and are defined in the Think C libraries.

The code simply follows the structure that you read about earlier. It calls PrJobInit to get the job dialog record before it is displayed. Then it calls PrDlgMain with the address of your own job dialog initialization procedure instead of the default one. The first real work begins, as you could see in the earlier step outline, in the job dialog initialization, **MyJobDlgInit**, which is shown in Listing 5-10.

Listing 5-10. **MyJobDlgInit** procedure

```
/*****************************************************************
 * MyJobDlgInit
 *
 *      Set up our dialog items before the big boys get here.
 *      Then display the standard printer job dialog.
 *      Note that this is NOT a method declaration.
 ***/

pascal TPPrDlg  MyJobDlgInit (
    THPrint aMacTPrint)
{
    int     firstItem;
    int     itemType;
    Handle  itemH;
    Rect    itemBox;

    firstItem = AppendDITL((DialogPtr)gPrtJobDialog, DRAFT_DITL);
    prFirstItem = firstItem;

    /* here is where you set the owned dialog items     */


    /* when you're done, save the old item handler
        and patch in the new item handler for future use */
    prPItemProc = gPrtJobDialog->pItemProc;
    gPrtJobDialog->pItemProc = (ProcPtr) &MyJobItems;

    /* return a pointer to the modified dialog routine  */
    return gPrtJobDialog;
 }
```

Notice that **MyJobDlgInit** is *not* a method, but is simply a normal procedure declaration. This is required because this procedure will be passed to the operating system for processing and not used internally to your own program. For the same reason, this procedure must be declared with the 'pascal' keyword since it will be receiving its operands in Pascal order. It also is impossible to define this procedure within the class structure in CLaser.h, as you would quickly see if you tried. For the same reason, the global variables that are required here, which normally would be set as instance variables, must now be declared in the usual fashion, at the top of the procedures and outside of all procedure boundaries, so that they remain available to all these routines throughout the life of the application. You can mark these as static to prevent them from being visible outside the CLaser.c context, if you want. Also, the check box to be added to the job dialog is defined as a 'DITL' resource, with a resource number defined by the DRAFT_DITL constant, which is defined in the CLaser setup. The additions to the top of CLaser.c are shown in Listing 5-11.

Listing 5-11. Startup information added to CLaser.c

```
#include "CError.h"
#include "CPrinter.h"
#include "CLaser.h"

#define CURRENT_VERSION        1
#define LAS_NOT_AVAL           415
#define DRAFT_DITL             401

extern      CError      *gError;        /* Error handler              */
extern      CursHandle  gWatchCursor;   /* Watch cursor for waiting*/
            TPPrDlg     gPrtJobDialog;  /* Job Dialog pointer         */
            int         prFirstItem;    /* Box item number            */
            ProcPtr     prPItemProc;    /* Default Job Dialog Item
                                           handler                    */
```

The routine begins by calling an auxiliary procedure to add the desired item (or items, if you wanted) to the dialog box. The **AppendDITL** procedure returns the value of the first item that you have added to the previous item list. You will use this in **MyJobItems** to determine when one of your items is hit. For that reason, the number is saved in the setup area as a static variable, prFirstItem.

Normally, you would next perform some setup on the dialog items that you have added. In this case, I have deferred that to the next section, when you add the routines to process the item. For now, you simply add the check box to the dialog; this seems complex enough for a first effort. The next step, then, is to retrieve and save the original item processing procedure, pItemProc, from the gPrtJobDialog job dialog record. This, too, must be stored for later use in **MyJobItems**, so it is also placed in the setup area as a static variable, prPItemProc. With that done, you can insert your own item processor, **MyJobItems**, into the job dialog record.

Note ▸

In all of this code, both here and in other parts of the book, you may notice that I don't generally try to make the code as compact as possible, as many C programmers would. For example I almost never nest procedure calls within other procedure calls, even when this requires adding another variable to the code as a temporary holder for the result from the first call. There are two reasons for this. First, I think the expanded code is somewhat easier to follow than more compact (and therefore more convoluted) code. Second, spacing the information out allows you to use the Debugger to trace the results of each step more completely than might otherwise be possible.

Next, look at the auxiliary procedure **AppendDITL**. This is an adaptation and simplification of the routine outlined in Technical Note #95, which is more general than the one here—not to mention that it is also more complex and in Pascal, of course. You only want to add a single check box in this case, which is what this routine, shown in Listing 5-12, does.

The code starts with the definitions of two related structures. The first structure, DITLItem, is the actual structure of an individual 'DITL' item. The second structure, ItemList, is the structure that contains the list of 'DITL' items. Since ItemList depends on the definition of DITLItem, these definitions must be presented in this order in the code, that is, DITLItem before ItemList. Also note the definitions of pointers and handles to each of these items after the items themselves. After these two important structures, there are definitions for a series of internal variables.

Listing 5-12. **AppendDITL** code routine

```
/*****************************************************************************
 * AppendDITL
 *
 *      Add an element to the Dialog box from a DITL resource.
 *      A simplified version of the routine in TN #95.
 *      Note that this is NOT a method declaration.
 ***/
int AppendDITL (
    DialogPtr theDialog,
    int theDITL)
{
    int     firstItem = 400;

    struct  DITLItem{
        Handle          itemHndl;
        Rect            itemRect;
        char            itemType;
        char            itemData[];
        } DITLItem;
    typedef struct DITLItem     *pDITLItem;
    typedef struct DITLItem     **hDITLItem;

    struct  ItemList{
        int                 dlgMaxItems;
        struct DITLItem     theDITLItems[];
        } ItemList;
    typedef struct ItemList     *pItemList;
    typedef struct ItemList     **hItemList;

    DialogPeek      pTheDialog;
    Rect            maxRect;
    Point           offset;
    hItemList       hDITL;
    pDITLItem       pItem;
    hItemList       hItems;
    pItemList       pItems;
    Handle          resH;
    OSErr           osErr;

    pTheDialog = (DialogPeek)theDialog;
    maxRect = pTheDialog->window.port.portRect;
    offset.v = maxRect.bottom;
    offset.h = 0;
```

Listing 5-12. **AppendDITL** code routine (continued)

```
    maxRect.bottom -= 5;
    maxRect.right -= 5;

    hItems = (hItemList)pTheDialog->items;
    firstItem = (*hItems)->dlgMaxItems;
    firstItem += 2;

    resH = GetResource( 'DITL', theDITL );
        CheckResource( resH );

    hDITL = (hItemList)resH;
    HLock(hDITL);
    pItem =  (*hDITL)->theDITLItems;

    OffsetRect( &(pItem->itemRect), offset.h, offset.v );
    UnionRect( &(pItem->itemRect), &maxRect, &maxRect );

    /* run some tests here to see how this all works...
    */
    pItem->itemHndl = (Handle) NewControl(
                    theDialog,
                    &(pItem->itemRect),
                    &(pItem->itemData),
                    true,
                    0, 0, 1,
                    1,
                    (long) 0 );

    osErr = PtrAndHand( pItem, hItems, GetHandleSize(hItems) );
        gError->CheckOSError( osErr );

    (*hItems)->dlgMaxItems += 1;

    HUnlock(hDITL);
    DetachResource(hDITL);

    maxRect.bottom += 5;
    maxRect.right += 5;
    SizeWindow( theDialog, maxRect.right, maxRect.bottom, true );

    /* just a dummy for now.... */
    return firstItem;
}
```

The processing begins with accessing the print dialog as a pointer, using DialogPeek to cast the pointer from a window pointer to a dialog pointer. The exact structure of the dialog record is given in the Dialog Manager chapter of *Inside Macintosh, Volume I*. You retrieve the portRect from the dialog window and store the information for later use since you are likely to have to expand the dialog window when you add the new items. Next you retrieve the current number of dialog items from the dialog record. Note that this is the actual number minus 1, as noted in *Inside Macintosh*. Therefore, the number for your first item must be two greater than the number that you retrieve.

Now you begin the process of retrieving the dialog items. First, you get the handle to the 'DITL' resource that has your own new items. Through a series of steps, you end up with the pointer pItem which points to the internal item or items (at this point you don't care how many there are) in your 'DITL' resource. Next, you offset the rectangle for this 'DITL' item by the amount required to move it to the bottom of the dialog box, and then you grow the rectangle from the original dialog box by that amount. This gives you the correct positioning for the new item and enough room to put it into the dialog box—once you make the dialog box larger, of course. Next you create the required handle to the new dialog item and place that into the item list. Notice that you retain the basic item information from the 'DITL'; you just create a new reference handle to it. Since you only have one control and you know what type of 'DITL' that control represents, you can eliminate a lot of overhead here; but, as usual, at the cost of some generality. Look at the code in Technical Note #95 for a more general method of handling this process.

Once the pointer to the new items is completely updated, it is inserted into the previous list with the PtrAndHand function. The last task for updating the old item list is to add the number of the new items (in this case, 1) to the previous item count.

With the item list taken care of, you can release the locked resources and restore the adjustment for the dialog box border to the window rectangle. The last task, before returning, is to resize the window to accommodate the new additions with the SizeWindow function. Then you return to the calling program by passing the new first item number.

The standard printing process uses ModalDialog to return the number of the item hit in the dialog box. Then it passes that number, along with the dialog pointer, to the procedure specified in pItemProc. Therefore, the last new procedure that you have to add is the one to handle hits on your owned dialog items. This code is given in **MyJobItems**, shown in Listing 5-13. For this iteration, you won't do anything but simply filter out hits on the box and pass the information for any other hits on to the standard processing routine. Simple enough.

Listing 5-13. **MyJobItems** code module

```
/*****************************************************************
 * MyJobItems
 *
 *      Check up on our dialog items before the big boys get here.
 *      If its not one of ours, we pass it on to the default handler.
 *      Note that this is NOT a method declaration.
 ***/

pascal void MyJobItems (
    TPPrDlg theDialog,
    int     itemNo)
{
    int     myItem;
    int     firstItem;

    int     itemType;
    Handle  itemH;
    Rect    itemBox;

    firstItem = prFirstItem;  /* saved from previous MyJobDlgInit */
    myItem = itemNo - firstItem + 1;

    /* here is where you test and reset the owned dialog items */
    if (myItem > 0)      /* is this a hit in our box? */
        {
            /* yes, do nothing for now      */
        }
    else
        {
            /* if its not an owned item, call the old item handler */
            CallPascal(theDialog, itemNo, prPItemProc);
        }

    /* and exit now */
    return;
}
```

This completes the first, and most onerous step in adding information to the job dialog. Next you have to add the new procedures to CLaser.h, as shown in Listing 5-14.

Listing 5-14. New header CLaser.h

```
/**********************************************************************
 * CLaser.h
 *
 *      Interface for the LaserWriter Printer Class
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#define _H_CLaser

#include "CPrinter.h"       /* Interface for its superclass      */

struct CLaser : CPrinter {  /* Class Declaration                 */
                        /** Instance Variables **/
    char            *gLaserName;
    char            gUserName[32];

                        /** Instance Methods     **/
                            /** Contruction/Destruction **/
    Boolean     ILaser(CDocument *aDocument, THPrint aMacTPrint);

                            /** Accessing    **/
    void    GetDeviceInfo(short *devNum);

                            /** Printing **/
    void    DoPrint(void);

                            /** Special **/
    void        Cur_Prntr( void );
    Boolean     MyJobDialog (THPrint aMacTPrint);

};
```

Listing 5-14. New header CLaser.h (continued)

```
/*  these are aux fcns that are NOT methods                          */
    pascal  TPPrDlg     MyJobDlgInit (THPrint aMacTPrint);
    pascal  void        MyJobItems(TPPrDlg theDialog, int itemNo);
            int         AppendDITL(DialogPtr theDialog, int theDITL);
```

Last, but by no means least, you must create a new 'DITL' resource containing the desired check box and add it to SimpleLW.π.rsrc. As before, you can use RMaker with a simple file, shown in Listing 5-15, to perform this chore. This code simply adds a check box 'DITL' to the existing SimpleLW.π.rsrc. The check box is enabled by default, and the position is given in terms of an empty dialog box. Remember that, in the code in Listing 5-12, you already adjusted this to position the new item correctly within the dialog box.

Listing 5-15. DITL.make.r code for new 'DITL' resource

```
!SimpleLW.π.rsrc

* added resource declarations follow ...

TYPE DITL
PRJobAdds,401       ;; New Resource #401
1                       ;; one item in list

checkBox
8 0 25 126
Mark as Draft
```

Now compile and run this new set of modules. When you have your standard screen, select Print from the File menu. You should see something like Figure 5-4. That's really all there is to this. As noted before, if you want to add to the style dialog, simply substitute Stl for Job everywhere in these modules.

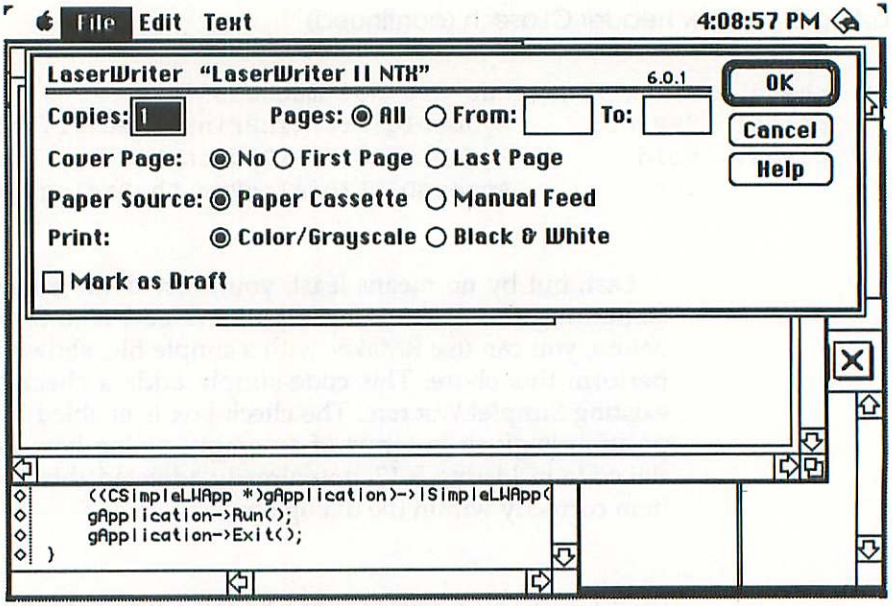

Figure 5-4. Modified job dialog box

▶ Working with the LaserWriter driver

Obviously you don't go to all the trouble of adding an item to a dialog unless you intend to use it. In this section, therefore, you use the new check box that you added to the job dialog to place a DRAFT message behind the page as you print, if the draft box was checked.

To use the check box is quite easy, but, naturally, it involves adding some code to the modules that you have already written. You need to make additions to three modules. Logically, you need to test for the draft option before you print the page or pages, so that you can add the DRAFT message to the page when you print. You do that in CSimpleLWPane. To test for the draft message option, you need to add code to CLaser that performs three tasks:

1. Sets the status of the draft message box when you start.
2. Checks the status of the draft message box when there is a hit.
3. Provides a method for checking the current status of the draft message box.

The first two of these were provided for, you remember, in the code that you produced earlier by placing comments where the actual code must now go. The third is clearly essential so that external classes can find out whether the box is set or not. Specifically, the CSimpleLWPane class methods must be able to interrogate the current status of the box. As it turns out, this is most easily done by creating an additional method in CSimpleLWDoc, called **TestDraftStatus**. This method allows the CSimpleLWPane to easily access the draft variable that is stored in CLaser.

You can begin this proces at either the drawing end or the printer end. Since you have just finished with CLaser.c, you may as well return to that first and make the necessary changes at the printer end. The first change, as just outlined, is to set the status of the draft at the start. This means adding code to the **MyJobDlgInit** method, as shown in Listing 5-16. The new code here is the two lines where you get the handle to your dialog item, as set by the firstItem variable that was returned by **AppendDITL**. Then you set the control value for that item to the global variable, gMarkDraft. And that's all there is to setting the value.

Listing 5-16. New code in **MyJobDlgInit**

```
/*****************************************************************
 * MyJobDlgInit
 *
 *      Set up our dialog items before the big boys get here.
 *      Then display the standard printer job dialog.
 *      Note that this is NOT a method declaration.
 ***/

pascal TPPrDlg  MyJobDlgInit (
    THPrint aMacTPrint)
{
    int     itemType;
    Handle  itemH;
    Rect    itemBox;
    int     draftValue;

    prFirstItem = AppendDITL((DialogPtr)gPrtJobDialog, DRAFT_DITL);

    /* here is where you set the owned dialog items     */
    GetDItem( gPrtJobDialog, firstItem, &itemType, &itemH, &itemBox);
    SetCtlValue( itemH, gMarkDraft );
```

Listing 5-16. New code in **MyJobDlgInit** (continued)

```
    /* when you're done, save the old item handler
       and patch in the new item handler for future use */
    prPItemProc = gPrtJobDialog->pItemProc;
    gPrtJobDialog->pItemProc = (ProcPtr) &MyJobItems;

    /* return a pointer to the modified dialog routine  */
    return gPrtJobDialog;
}
```

Retrieving and resetting the check box are equally straightforward. Listing 5-17 shows you the code from **MyJobItems** that handles this task. As before, you test the item number provided to see if it is your own item or one of the standard items. If it is one of the standard items, this code passes the call on to the standard item handler. If it is your own item (or items—you could easily generalize this), you take three steps to set and display the new value. First, you get the handle to the item. Then you reverse the current value of the global variable, gMarkDraft. Since a check box is essentially a Boolean, this either checks or unchecks the box, depending on its former status. Then the reversed gMarkDraft is used to set the dialog item to the correct display.

Listing 5-17. New code for **MyJobItems**

```
/*****************************************************************
 * MyJobItems
 *
 *      Check up on our dialog items before the big boys get here.
 *      If its not one of ours, we pass it on to the default handler.
 *      Note that this is NOT a method declaration.
 ***/

pascal void MyJobItems (
    TPPrDlg theDialog,
    int     itemNo)
{
    int     myItem;
    int     firstItem;
```

Listing 5-17. New code for **MyJobItems** (continued)

```
    int     itemType;
    Handle  itemH;
    Rect    itemBox;

    firstItem = prFirstItem;    /* saved from previous MyJobDlgInit */
    myItem = itemNo - firstItem + 1;

    /* here is where you test and reset the owned dialog items */
    if (myItem > 0)      /* is this a hit in our box? */
        {
            GetDItem( theDialog, itemNo, &itemType, &itemH, &itemBox );
            gMarkDraft = !gMarkDraft;
            SetCtlValue( itemH, gMarkDraft);
        }
    else
        {
            /* if its not an owned item, call the old item handler   */
            CallPascal(theDialog, itemNo, prPItemProc);
        }

    /* and exit now */
    return;
}
```

The last task that you must perform is to provide the global variable, gMarkDraft, and initialize it in **ILaser**. The code to do that is shown in Listing 5-18.

Listing 5-18. Definition and initialization of global variable, gMarkDraft

```
/*****************************************************************
 * ILaser
 *
 *      Initialize a Laser Printer object. Printer can be passed a
 *      handle to an existing Toolbox print record. If handle is
 *      NULL, a new print record with default values is created.
 *
 *      Boolean result indicates whether the print record passed as a
 *      parameter was changed because it was incompatible with the
 *      currently installed printer. This allows documents which
 *      associate a printer record with a file to update itself if
 *      desired.
 ***/
```

Listing 5-18. Definition and initialization of global variable, gMarkDraft (continued)

```
Boolean     CLaser::ILaser(
    CDocument       *aDocument,
    THPrint         aMacTPrint)
{
    Boolean     changed = FALSE;    /* Has print record changed?    */

    /* set instance variables    */
    gMarkDraft = FALSE;
    gLaserName = "";

    changed = CPrinter::IPrinter(aDocument, aMacTPrint);
    return( changed );

}
```

At this point you can check and uncheck the box, and it will work as you expect. However, it still doesn't actually do anything since you haven't set up any code to take advantage of the setting. To do that, you need to add another method to CLaser.c, **GetDraftStatus**. This code is shown in Listing 5-19.

Listing 5-19. New code for **GetDraftStatus** method

```
/**********************************************************************
 * GetDraftStatus
 *
 *  Returns information about the Print Job Dialog .
 *
 *  This method is provided for all other methods to inquire about
 *  the current setting of the MarkDraft check box in the PrJob
 *  Dialog. This allows printing pages to determine what to do.
 *
 ***/

Boolean     CLaser::GetDraftStatus(void)
{

    return(gMarkDraft);

}
```

This is amost trivially simple. You just return the current value of gMarkDraft to the calling class or method. However, this simple method is the only correct way for anyone outside the class to access the current value of the variable, and it does it in a nice, structured way. You also have to add this new method to CLaser.h. This revision is shown in Listing 5-20.

Listing 5-20. Revised CLaser.h

```
/**********************************************************************
 * CLaser.h
 *
 *      Interface for the LaserWriter Printer Class
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#define _H_CLaser

#include "CPrinter.h"           /* Interface for its superclass */

struct CLaser : CPrinter {      /* Class Declaration            */
                            /** Instance Variables **/
    char                *gLaserName;
    char                gUserName[32];

                            /** Instance Methods    **/
                                /** Contruction/Destruction **/
    Boolean     ILaser(CDocument *aDocument, THPrint aMacTPrint);

                                /** Accessing    **/
    void        GetDeviceInfo(short *devNum);
    Boolean     GetDraftStatus(void);

                                /** Printing **/
    void        DoPrint(void);
```

Listing 5-20. Revised CLaser.h (continued)

```
                                    /** Special **/
    void        Cur_Prntr( void );
    Boolean     MyJobDialog(THPrint aMacTPrint);

};

/*  these are aux fcns that are NOT methods                 */
    pascal  TPPrDlg     MyJobDlgInit (THPrint aMacTPrint);
    pascal  void        MyJobItems(TPPrDlg theDialog, int itemNo);
            int         AppendDITL(DialogPtr theDialog, int theDITL);
```

Now that you have set up the printer part of the code, you can proceed to the drawing part. All the drawing takes place in the CSimpleLWPane.c code module. Here I have done something a bit different. I have added the new method, **PrintPage**, which overrides the default **PrintPage**. **PrintPage** is called by the printer function before each page is printed, and overriding the default method allows you to insert your own code or setup functions prior to the start of printing. This new method is not strictly necessary. Although it might seem that this is the correct place to test for the draft box, it is not appropriate here because you want to alter, not the printing process (which is what you can alter at this point), but the actual page output, which is produced for the printer, as it is for the screen, by the **Draw** method. However, there is a good reason to add this code for debugging. **Draw** gets called for every refresh of the screen as well as for the printed output. For that reason, placing breakpoints in **Draw** is rather troublesome and annoying because you have to step by them every time the screen is drawn. By adding this method, however, you can set a breakpoint here. Then, when you stop at that point, you can set breakpoints in **Draw** because the next time **Draw** is executed, it will be to create the printed output. The method is shown in Listing 5-21.

Listing 5-21. The new method, **PrintPage**

```
/*****************************************************************
 * PrintPage
 *
 *  When your page is ready to be printed, this method
 *  can make control the actual printing process.
 *  Specifically, this is the point to add the Draft background,
```

Listing 5-21. The new method, **PrintPage** (continued)

```
 *  if the JobDialog asks for Draft markings.
 *
 ***/

void WSimpleLWPane::PrintPage(short pageNum, short pageWidth, short
                              pageHeight)

{
    Boolean     testBool;

    testBool = ((WSimpleLWDoc *)itsSupervisor)->TestDraftStatus();
    if (testBool )
        {
            /* probably don't need any PicComments here    */
        }

    /* then continue with printing process */
    inherited::PrintPage(pageNum, pageWidth, pageHeight);
}
```

The dummy method shown here does test the draft status variable, but it does nothing with that information. It simply illustrates the use of the added dialog items in this context. In another application, with a different set of options, you might want to check the option selections here. For example, if you had a check box to print in reverse order, this is where you would test it so you could control the sequence of the output.

Notice how the test for draft status proceeds. The CSimpleLWPane class does not have direct access to the printer object, which represents the current printer and was created by and is stored in the CSimpleLWDoc class. Therefore, you send a message to CSimpleLWDoc, which is the supervisor of CSimpleLWPane, and ask it to use the **TestDraftStatus** method to find out the current setting and to report that setting back. Note that you have not yet added **TestDraftStatus** to the methods of CSimpleLWDoc.

Now you are ready to move to the heart of the process and add code to the **Draw** method to print DRAFT behind the text on a page. Listing 5-22 shows the code for the revised method.

Listing 5-22. Code for **Draw** method revised to add DRAFT lettering

```
/**
    set up for PostScript comment data
**/
theString = "\p0 761 translate 1 -1 scale /Times-Roman findfont 20
            scalefont setfont 50 650 moveto (Hello, world!) show\n";
strLength = theString[0];
strPtr = theString + 1;
theError = PtrToHand( strPtr, &myPSHandle, strLength);
if ( theError != noErr )
    gError->SevereMacError( theError );

/**
    do PostScript stuff for LaserWriter
**/
PicComment(PS_BEGIN, 0, NIL_HAND);
PicComment(PS_HANDLE, strLength, myPSHandle);
PicComment(PS_END, 0, NIL_HAND);

/**
    tell us (on the screen) that we've finished
**/
MoveTo(50, 50);
DrawString("\pPostScript processing completed." );

/* test for printing status ... if so, add DRAFT if requested  */
isDraft = ((WSimpleLWDoc *)itsSupervisor)->TestDraftStatus();
if (printing & isDraft)
    {
        if (((WSimpleLWDoc *)itsSupervisor)->isLaserWriter)
        {
            PicComment(PS_BEGIN, 0, NIL_HAND);
            PicComment(PS_TEXT, 0, NIL_HAND);
                DrawString( "\p0 760 translate 1 -1 scale" );
                DrawString( "\p/Helvetica findfont 72 scalefont
                              setfont");
                DrawString( "\p45 rotate 250 50 moveto" );
                DrawString( "\p(DRAFT) false charpath" );
                DrawString( "\p0 setgray stroke" );
            PicComment(PS_END, 0, NIL_HAND);
        }
        else
        {
            /* do something less elegant to show draft status */
        }
    }

ClosePicture();
```

The revised code begins by setting an internal Boolean, isDraft, using the same technique that you just saw in **PrintPage**. Then you test for the combination of printing status and the draft box being checked. If both are set, you want to add the DRAFT message. Next you check to see if you are printing on a LaserWriter, using the global instance variable isLaserWriter that was set in CSimpleLWDoc. Since this is a book about LaserWriter programming, the code here doesn't try to do anything if the output device is not a LaserWriter, but you see how you would handle that case.

Take a good look at this code. Notice that this is a fairly device-independent way to handle the Mark as Draft box setting. If you were running on some other device, you might choose to add a small header or footer, saying DRAFT, or whatever. In any case, the code here could be set up to run on any device, using two different techniques depending on the capabilities of the output device. This is exactly the way that you are supposed to use this device-dependent information.

The actual code that you execute if it is a LaserWriter is very similar to code that you have created before. You use PicComment PostScript to reverse the QuickDraw coordinates, rotate the coordinates by 45 degrees, and then move to a suitable point toward the center of the page. Then you turn the word "DRAFT" into an outline, using the techniques from the preceding chapter, and then stroke it in black.

That finishes the processing in CSimpleLWPane.c. If you have chosen (as I did) to add the new method, **PrintPage**, you also have to update the Pane header file, CSimpleLWPane.h. Since that is exactly like all the other updates to add methods, it is not shown here.

The last change that you need to make is to update CSimpleLWDoc.c by adding the new method, **TestDraftStatus**. This is used by CSimpleLWPane.c, and could be used in the same way by any pane that fell under this document. The code is shown in Listing 5-23.

Listing 5-23. The new method, **TestDraftStatus**

```
/**********************************************************************
 * TestDraftStatus
 *
 *  This method simply tests the printer's Draft status routine
 *  whenever requested (usually by a pane just prior to printing).
 *
 ***/

Boolean WSimpleLWDoc::TestDraftStatus(void)

{
    return( myPrinter->GetDraftStatus() );
}
```

This method simply redirects the query to the **GetDraftStatus** method of the global printer object that was defined and stored in **ISimpleLWDoc**. The Boolean return value is then simply returned to the calling method as a response. Neither difficult nor confusing. Once again, you must update the header file, CSimpleLWDoc.h, to reflect the new method.

With all this code done, you can now run the project. Compile all the modules to bring the project up to date, and then run it. You will get the usual window display. When you select Print from the File menu, check the Mark as Draft dialog box, as shown in Figure 5-5. When you run this with the box checked, you should get the output page shown in Figure 5-6.

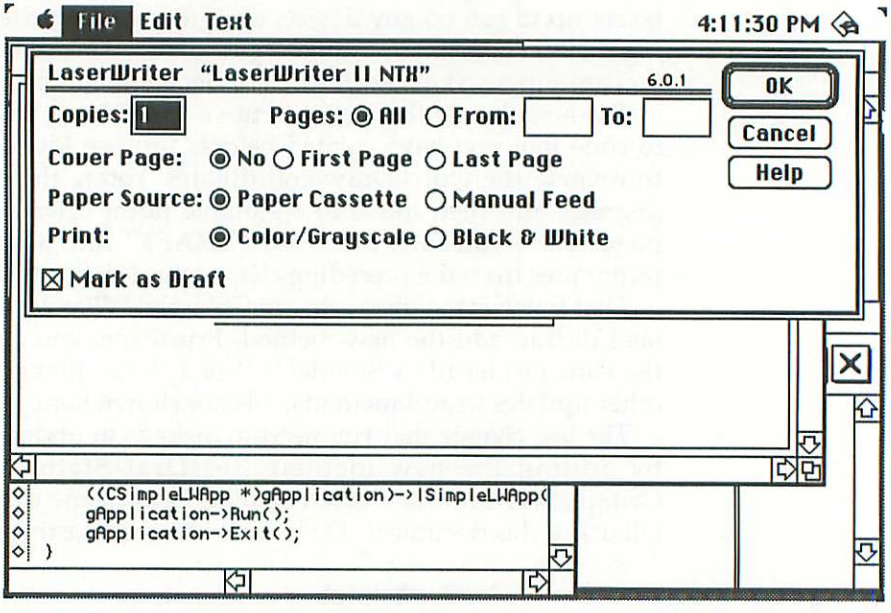

Figure 5-5. Mark as Draft box checked

This is a simple but useful example of what you can do when you know your output device. Of course, once you know how to create new items and add them to the job and style dialogs, you can easily see how you might go about using them. Also, now that you have some grasp of PostScript processing in QuickDraw, you can use the information that you get back to provide special output features, as you have done here.

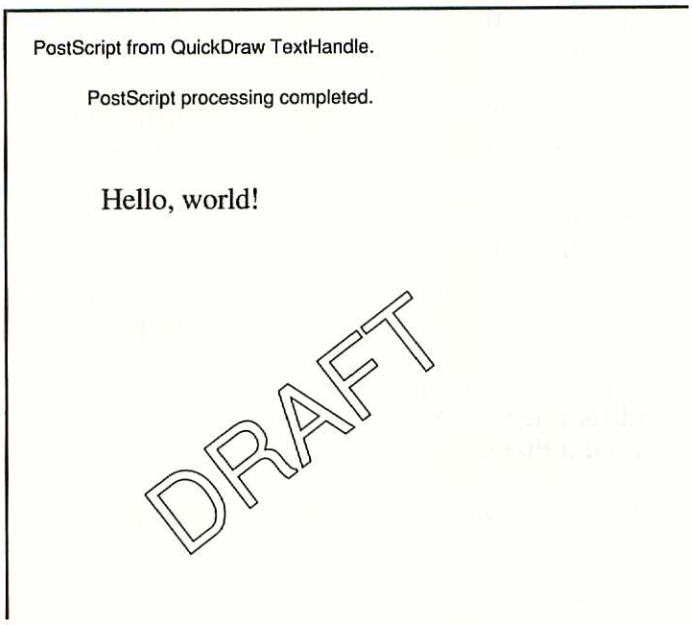

Figure 5-6. Output with Mark as Draft box checked

▶ LaserWriter driver resources

This section describes some of the resources that you may want to change in order to make the error messages, dialogs, and other displays that come from the LaserWriter a little more approachable in other languages. You can do this, as you would in your own application, by editing and changing the values of various resources that are stored in the LaserWriter driver file.

Important ▸

You're a professional, right? So I don't have to say this more than once: Make a copy of the driver before you touch the file and work only on the copy. After all, the LaserWriter driver is the connection between your Macintosh and your LaserWriter printer, so it wouldn't do to destroy it or damage it so that it doesn't print. Also, never edit the driver while MultiFinder is running, as you can severely damage it. Note that what you will be modifying here are the strings and other resources that are used for display. If you blow these off, you may not get anything but messages in Martian. So be warned, be professional, and **ALWAYS HAVE A BACKUP.**

There are three types of resources that you may want to modify to create a local version of the LaserWriter driver.

- The 'STR ' resources contain strings that the Chooser uses to notify the user about printing options and that contain the name of the current printer file.
- The 'DITL' resources contain the strings used in printing the dialog boxes and alerts.
- The resource 'PREC' 109 contains strings that translate the standard LaserWriter error and status information.

Table 5-3 shows you the type of resource, the number of the resource, and its purpose. You can change any of these to another language or format if that is required.

Table 5-3. Resources in the LaserWriter driver file that can be changed to reflect local language or other requirements

Resource Type	*Id*	*Explanation*
'STR '	–4091	Select string for the Chooser dialog that asks the user to select a printer of a specific type; the type requested will be this string.
'STR '	–4094	Plural form of the printer device name for the Chooser.
'STR '	–4095	Singular form of the printer device name for the Chooser.
'STR '	–4096	The printer device name for the Chooser. Generally, this is identical to the form given in –4095.
'STR '	–8159	"Creating PostScript(r) File." This string is used in the status dialog box when the user forces a PostScript version of the printed document to be created as a file.
'STR '	–8160	"Looking for LaserWriter <*name*>". The sequence <*name*> must not be moved from the end of the string, even though this may be ungrammatical in some languages, since the driver expects the string to be in this form.
'DITL'	–8149	A message that the LaserWriter is initialized with a different version of the Laser Prep file than is currently in use on the host computer. This message appears when the LaserWriter cannot be reset from the host.

Table 5-3. Resources in the LaserWriter driver file that can be changed to reflect local language or other requirements (continued)

Resource Type	*Id*	*Explanation*
'DITL'	–8150	A message that no LaserWriter has been selected in the Chooser dialog. The user is requested to select a LaserWriter.
'DITL'	–8152	This message cannot be used or changed.
'DITL'	–8158	A message that the LaserWriter is initialized with a different version of the Laser Prep file than is currently in use on the host computer. This message appears when the LaserWriter can be reset from the host and offers the option to reset the device.
'DITL'	–8159	A message that no Laser Prep file has been downloaded to the device. The user must supply a valid Laser Prep file or cancel the printing. Activation of this message will also automatically eject the disk in the internal drive.
'DITL'	–8160	A message that the reduction or zoom value is out of the allowed range. The appropriate maximum or minimum value is used automatically, and this message notifies the user of the substitution.
'DITL'	–8161	The help screen for the LaserWriter job dialog. All text in this dialog may be modified as required. Each string is in a separate box for ease in formatting.
'DITL'	–8179	A message that the page setup selections in the current print record do not match the currently selected printer.
'DITL'	–8181	A message that printing is terminated because the document exceeds allowable resource limits.
'DITL'	–8191	The job dialog box with all the associated dialog items. Text for any items in this box can be redefined, but the items should not be moved.
'DITL'	–8192	The style dialog box with all the associated dialog items. Text for any items in this box can be redefined, but the items should not be moved.
'DITL'	–8138	The style options dialog box. Text for any items can be redefined, but the items must not be moved.
'PREC'	109	Pascal string pairs that redefine messages from the printer for display.

Most of the resources listed in Table 5-3 are quite standard, and you probably are quite familiar with them. The sole unusual resource in this group is the 'PREC' resource. 'PREC' 109 is a resource that consists of pairs of Pascal strings, with the first entry of the pair indicating the string that is returned by the printer and the second string, if any, indicating the string that should be displayed on the screen instead of the first string. The set of string pairs is terminated by the first pair whose first element is an empty string (that is, where the length byte is 0x00). Pairs where the second element is an empty string use the first element for display, but they do not terminate the set of string pairs.

I know all that sounds a bit confusing, so let's look at the first entry in the 'PREC' 109 resource as an example. Here is the pair of strings, in standard C notation for a Pascal string:

```
\pbusy
\pprocessing job
```

When the LaserWriter driver receives a status message back from the LaserWriter printer, it parses the return to find the status. (We will discuss this process in a moment; for now, just accept that it can determine what the status is.) When it finds the status, it looks that word up in the table of string pairs that is generated from 'PREC' 109. If it finds a match, it substitutes the second string for the first in the dialog box that reports the message. In this case, if the status returned was *busy*, then the message in the dialog box would be *processing job*.

This all depends, of course, on the fact that these messages can be parsed by a software program into appropriate and identifiable segments for use in this process. Actually, this isn't as hard as it seems, since PostScript returns messages that have a very precise format for exactly this purpose. All PostScript messages follow a rigid format, which looks like this:

```
%%[keyword: value; keyword: value; ...  ]%%
```

These messages all begin with the string %%[and end with]%%. In this way, the beginning and ending are quite well defined and can be easily identified by standard string comparisons. Within the message, all information is presented as a series of keyword: value pairs, where the keyword and the value are separated by a colon (:) and the successive pairs are separated by a semicolon (;). In this way, the entire message can be broken down into segments and the desired element or elements easily extracted by standard string handling techniques.

Generally, PostScript messages fall into one of several standard categories. The first of these is the standard status response. This has the following format.

```
%%[job: name; status: string; source: AppleTalk]%%
```

This status response has three components. The job name may be any arbitrary string. This string is the current value of the PostScript string **jobname**, which is set by the driver when the document is first sent to the device. The status string may be any string, but the possible values are set internally for each version of the PostScript interpreter. Common status values are *busy*, *waiting*, *idle*, and *printing*. The source is the current communications channel. For our purposes, this is AppleTalk, but, if you were working with the device over other communication channels, it might be *serial 25*, *serial 9*, or *parallel*, for example.

Other standard messages that are produced by the interpreter provide information about hardware and software errors and about the current state of the interpreter. These messages are as follows:

```
%%[Error: errorname; OffendingCommand: object]%%
%%[PrinterError: reason]%%
%%[Flushing: rest of job (to EOF) will be ignored]%%
%%[exitserver: permanent state may be changed]%%
```

Now you see how the LaserWriter driver can parse the return information to find the desired value for replacement. In the case that you were looking at earlier, the message would be something like this:

```
%%[job: Untitled-1; status: busy; source: Appletalk]%%
```

The driver looks for the status keyword and then extracts the following string to the next semicolon. This value, *busy*, is then compared to the values in the table from 'PREC' 109. When the value is found, then the replacement of the strings is made and the dialog window displays the new message,

```
status: processing job
```

which is what the user sees on the screen.

This also explains how you can use the 'PREC' 109 resource to modify the message that is displayed. The string returned by the printer, of course, remains the same: *busy* or whatever. But you can change the resource value to be any new string that you want, and in

any language. If you want to do this, look at the current string pairs in 'PREC' 109 and decide which ones you want to modify. There are two important considerations for you when you modify these strings. First, you can add more items to the end of the list, but you should not remove items from the list. Second, you should not move the string beginning "out of paper" from its position in the list.

There are also some messages that the Laser Prep file itself generates. These messages do not follow the preceding format, which is for messages that originate within the PostScript controller itself. Laser Prep messages take the form

```
|0|, |1|, |2|, ...
```

and so on for as many messages as are required. These strings are returned and translated like the other strings, using the 'PREC' 109 table for conversion. Therefore, you can translate the second element of the string in the same way that you would for the other entries. This discussion of the Laser Prep messages leads quite naturally to a consideration of the entire range of processing that the Laser Prep file provides, which is the subject of the next section.

▶ Understanding the Laser Prep File

So far, you have read about and worked with the features that the LaserWriter driver provides. This is quite natural, since all the work to this point has really concentrated on the requirements and interface for Macintosh processing. In this section, however, you cross over into the processing that is taking place within the LaserWriter printer itself. That processing is done by the functions that are loaded into the LaserWriter as a part of the Laser Prep file.

At this point, you are well equipped to understand exactly how the Laser Prep functions work. In fact, at various points in the text, various facets of how this structure works have been discussed. Laser Prep is a PostScript prologue that is downloaded to the printer outside of the server loop, so that it remains resident from the time that you first print to the time that the printer is turned off or reset. The Laser Prep file provides definitions of PostScript procedures that are used to convert the QuickDraw commands in your printed output into PostScript commands, and it also provides the necessary setup and cleanup operations for individual pages and for the document as a whole. These procedures are kept in a private dictionary, called **md**, that is created and loaded by the Laser Prep code. All this simply reviews what you already know.

Note ▸

In System 7.0 much of this will change. The Laser Prep file is reduced to a very small size and now consists of a skeleton **md** that contains only the definition of the **av** variable. The rest of the code previously in Laser Prep is incorporated into the LaserWriter driver, and the remainder of the **md** dictionary will be loaded with each job, instead of only being loaded once, as it is now. This adds some code to the individual job, as you could see by the difference between the document with and without the **md** dictionary. However, it minimizes the dialog required between the driver and the device, and it should eliminate the annoying message that your colleagues or co-workers have used a different version of Laser Prep and so your printer requires reinitialization.

Now you want to look into the actual structure and definitions that Laser Prep uses. This has two purposes. First, it allows you to understand in more detail exactly how the commands that you generate are transformed inside the LaserWriter into PostScript code. Second, it gives you a look at some PostScript code and coding techniques that are used in the Laser Prep file. These are useful since, as you'll realize if you think about it, Laser Prep is probably the single collection of PostScript code most often executed in the LaserWriter. This code gives you an example of how you might structure your own dictionary and procedures later.

▶ Translations and functions

Now you want to make a more detailed analysis of the code that is generated when you create and print a document. The easiest method for doing this is to intercept an existing document and have it stored as a PostScript file. You can then examine the resulting file with any text editor, such as the editor in Think C. Recall that Chapter 1 discussed how to intercept PostScript files in the section entitled "Intercepting Printing Manager output." Here is a brief review:

- Turn off Background printing if it is not already off.
- Select the Print command from the File menu.
- When the job dialog appears and you are ready to print, hold down the Command (or Apple) key and the F key and click OK.
- You should see a dialog box confirming that a PostScript file is being created.

Pressing Command-F saves just the document, without the associated Laser Prep file. If you use Command-K instead, you will get a larger file that includes the Laser Prep definitions.

Since you have just finished an exercise that creates special LaserWriter output, let's look at that as an example of typical translated code. Enter the project file again and check the Mark as Draft box. This time, however, when you are ready to print, hold down the Command (⌘) or Apple key and the F key as well. Do not hold down the Shift key. You should get a message ('STR ' –8159, remember?) that tells you that a PostScript file is being created. Usually this file is created in the same folder as your project file. Look in there for a file called PostScript0 and open it with the Think C editor. Your file should look something like Listing 5-24.

Listing 5-24. PostScript0 file from example

```
%!PS-Adobe-2.0
%%Title: Untitled-1
%%Creator: SimpleLW.π
%%CreationDate: Wednesday, September 19, 1990
%%Pages: (atend)
%%BoundingBox: ? ? ? ?
%%PageBoundingBox: 30 31 582 761
%%For: David
%%IncludeProcSet: "(AppleDict md)" 70 0
%%EndComments
%%EndProlog
%%BeginDocumentSetup
md begin

T T 0 0 730 552 -31 -30 761 582 100 72 72 1 F F F F T T T F psu
(David; document: Untitled-1)jn
0 mf
od
%%EndDocumentSetup
%%Page: ? 1
op
-2 -2 xl
1 1 pen
20 20 gm
(nc 2 2 728 550 6 rc)kp
0 gr
```

Listing 5-24. PostScript0 file from example (continued)

```
T 1 setTxMode
0 fs
bu fc
{}mark T /Helvetica /|______Helvetica 0 rf
bn
12 fz
bu fc
2 F /|______Helvetica fnt
bn
0.45379 0. 32 0.04537 0.(PostScript from QuickDraw
TextHandle.)awidthshow
psb
0 761 translate 1 -1 scale /Times-Roman findfont 20 scalefont setfont
50 650 moveto (Hello, world!) show
pse
50 50 gm
0.77819 0. 32 0.07781 0.(PostScript processing completed.)awidthshow
psb
0 760 translate 1 -1 scale
/Helvetica findfont 72 scalefont setfont
45 rotate 250 50 moveto
(DRAFT) false charpath
0 setgray stroke
pse
0 0 gm
(nc 2 2 728 550 6 rc)kp
0 gr
2.5 2.5 727.5 549.5 0 rc
F T cp
%%Trailer
cd
end
%%Pages: 1 0
%%EOF
```

Listing 5-24 has several notable features. To begin with, you see that it follows the document structuring conventions that you learned in Chapter 4. Also notice that the prologue calls for "(AppleDict md)" and a specific version and revision number; however, the actual code for **md** is not included here.

The document begins with a standard group of setup functions. These are delimited by the **%%BeginDocumentSetup** and **%%EndDocumentSetup** structure comments. Then the actual document processing begins; this is marked by the **%%Page:** comment. Every page in your document will begin with a comment like this; of course, in this case, you only have a single page. The end of the document starts at the **%%Trailer** comment, which contains the cleanup functions and the final trailer comments.

In the middle of the document you can find your own PostScript code; both the "Hello, world!" code and the DRAFT code, in two separate segments. However, if you look around these segments, you will see that most of the code is nothing at all like the PostScript that you were creating in Chapter 4. The code here looks rather cryptic and uninformative, compared to standard PostScript operators. In fact, the only PostScript operators that are recognizable in this document, other than your own code, are the lines where the text is actually drawn. These lines include the PostScript operator **awidthshow**, which aligns text strings as discussed briefly in Chapter 4. Other than that, the commands here are all short two- or three-letter names.

These, of course, are the procedure names that are defined in **md**, the Laser Prep dictionary. These function just like the procedures that you defined in Chapter 4, such as **restoreCoord**, except that the names are very short and cryptic—although they do bear some relation to what the functions do, believe it or not. Actually, there are three very good reasons for making these names as short as possible. First, this code is not designed to be read by humans, but instead is both created and read by application software. The result is that longer names, or names with more mnemonic value, are of little use in this environment. Second, the longer the name is, the more characters it requires and the more time it takes to transmit that name over the network. This may not seem like much, but consider that these names form the bulk of the script of a document, which may be many pages and consist of a large number of procedure calls. In such documents, the savings represented by four or five characters per name amounts to many thousands of bytes of code. Third, the PostScript interpreter must translate and look up each name as it reaches the LaserWriter and is executed. Although PostScript is optimized for such dictionary searches, nevertheless the shorter the name, the fewer characters must be converted and looked up to retrieve the associated procedure. For these reasons, people who create device drivers very often attempt to minimize the length of the names that are used.

This shows you, without too much complication, just what a document looks like when it comes out of the Printing Manager and the LaserWriter bound, over an AppleTalk network, for your favorite printer.

▶ Laser Prep format

Now that you have seen the cart, so to speak, you need to look at the horse. Let's do the same process again, but this time use the Command (⌘) key and the K. This results in a much longer file. If you haven't renamed your first file, this one will be called PostScript1, and it, too, will be in the same folder that the project file is in (or, at least it will be in the same place that PostScript0 was in).

Open the PostScript1 file with the Think C editor, and you will see that the file is essentially in two pieces: a first segment that begins with the **%%Title: "Laser Prep -- The Apple PostScript Dictionary (md)"** comment and a second segment, down toward the bottom of the file, that looks just like the document file that you looked at before. Indeed, it is identical to the previous document file because you printed the same document. The second segment is preceded by the **%%EndProcSet** and **%%EOF** comments, which mark the end of the Laser Prep file. This entire file is much too large and complex to print here. Instead, let's look at some individual segments and procedures so that you can see how to find procedures in this dictionary.

The best way to do this is to look at your document and identify certain functions that you want or need to analyze. As an example, let's begin by looking at the document code that surrounds the PostScript code that you inserted into your document with PicComments. If you look back at Listing 5-24, you see that the first line of PostScript code that you inserted has a single line of code both before and after it, and that the same lines are included just before and after the next PostScript code also. These lines look like this:

```
psb
... your PostScript code is here
pse
```

Let's find out what these lines of code do. Open the file that has the **md** dictionary in it and use the Search function to find the line that contains the string

```
/psb
```

This line will be the definition of the **psb** procedure, and the next line (in version 70 of **md**) contains the definition for **pse**. These two definitions are shown in Listing 5-25.

Listing 5-25. Definitions of **psb** and **pse** from the **md** dictionary

```
/psb{/us save def}bdf
/pse{us restore}bdf
```

As you might expect, these two procedures work together. **psb** creates a save object and stores it as the name **us**; then **pse** restores that object to return the interpreter to its original state when your code segment is finished. Of course, you already knew (from Chapter 3) that there was a **save, restore** pair of operators around your code. However, if you had been working strictly on your own, this would be of some interest and would help you understand why you have to clean up all dictionaries and so on before you return to ordinary QuickDraw processing. This is a simple sample of how you can go about looking up code in the **md** dictionary.

Next, let's look at a more complex example, taken from the setup section of the document. This is an important part of the document, where a number of the page settings and other document information is saved or created for future reference. This section of the document in Listing 5-24 consists of about five lines of code. The first line simply starts using **md** by placing it onto the dictionary stack:

```
md begin
```

The third line sets the job name to the name of the document, using the procedure **jn**. As you see, this is a compound name, including both the name of the user (David) and the name of the window being printed (Untitled-1). The second line, however, is more complex. This presents a long list of information and ends with the cryptic command **psu**. Let's analyze this command.

To speed up the process, I will tell you that the letters T and F are defined in **md** as the PostScript Boolean values **true** and **false**, respectively. Of course, you can use the same lookup technique that you just used to find out what any of these objects are. Given that, what does **psu** do? Well, look up

```
/psu
```

in the dictionary with the Search function. You will find a definition that looks something like Listing 5-26.

Listing 5-26. Definition of **psu** in the **md** dictionary

```
/psu
{
    /udf xdf
    /tso xdf
    /fNote xdf
    /fBitStretch xdf
    /scaleby96 xdf
    /yflip xdf
    /xflip xdf
    /invertflag xdf
    /dbinvertflag
        invertflag
        statusdict begin
        version cvr 47.0 ge product (LaserWriter) eq
            not and
        end
        invertflag and
        {not}if
    def
    xflip yflip or
        {/noflips false def}
        if
    /pgs xdf
    2 index .72 mul exch div /pys xdf
    div .72 mul /pxs xdf
    ppr astore pop
    pgr astore pop
    /por xdf
    sn and /so xdf
}
bdf
```

The actual code that you see does not look so spread out, of course. I have reorganized it here for ease of display and understanding in what is called, in the common terminology of programming, "pretty printing." In any case, you see that this consists of a series of names followed by the procedure **xdf**. As you might suspect, both from the

name and the common occurrence of the requirement within a procedure, the procedure **xdf** is defined in **md** as the sequence **exch def**. You can verify this by looking the sequence

```
/xdf
```

up in the dictionary. When you do, note that immediately before it is a procedure, **bdf**, which is defined as the operators **bind def**, another common definition pair. Both of these are used in the definition of the **psu** procedure.

With this you can begin to see how **psu** is put together and what the definitions are for your document. Remember that the operands are removed from the stack in reverse order of how they are put on. That means that the first definition inside **psu**, for the variable named **/udf**, is **false**, as set by the final F in the set of operands that occur before **psu**, and not **true**, which it would be if the operands were taken from the front of the line.

Note that you cannot look up **udf** as you have the other procedures up to this point, by searching for the string

```
/udf
```

because that would lead you right back to this definition. If you want to see what **udf** does, you must look it up without the initial /. Also be sure to put a space either before or after it so that you don't get hits on strings that have the letters "udf" as a part of them.

I leave it to you to follow the chain for each of these variables and procedures through the entire dictionary. Some may be of more interest to you than others, but I strongly recommend that you check at least a few of them so that you feel comfortable with the process of examining the **md** structure.

Two definitions within **psu** deserve special attention. These are the definitions for **ppr** and **pgr**. If you look these up in **md**, you will find that they are defined as follows.

```
/ppr [-32 -29.52 762 582.48] def
/pgr [0 0 0 0] def
```

In the **psu** function, these two arrays are redefined by the two sequences of four numbers each, near the beginning of the line of code—and therefore near the end of the definitions in **psu**. The **ppr** array is defined first, by the sequence **-31 -30 761 582**. The effective process could be written as

```
-31 -30 761 582 ppr aload pop
```

which stores the four numbers into the array. These are the coordinates for the default page before the QuickDraw transformation. The **pgr** array is defined in the same way as **0 0 730 552**, and it represents the coordinates after the QuickDraw transformation.

Now you can understand why I said earlier that the transformation that you have been using in the examples,

```
0 760 translate
```

does not return exactly to the origin of the PostScript page. If you follow the transformations in the **md** dictionary (in the procedure **txpose**, which is called by the **od** procedure in the setup section), you will find that the actual translation performed by **md** is

```
30 761 translate
```

To exactly reverse that would require the command

```
-30 761 translate
```

so the transformation that you have been using is off by 30 points in the x direction and 1 point in the y direction. This kind of analysis is what you can do when you understand how to open up **md** and work with it.

▶ Downloading Laser Prep

When you select Print from the File menu, the LaserWriter driver conducts a dialog over the AppleTalk network with the chosen printer to determine whether or not the Laser Prep dictionary, **md**, is already present in the device. If it is not present, the driver sends the dictionary down to the printer to prepare it for processing. It does this outside the server loop, so that the dictionary remains in the printer until you turn it off or it is reset. This generates a message of the form

```
%%[exitserver: permanent state may be changed]%%
```

from the printer, which is turned into the screen message

```
status: initializing printer
```

by the 'PREC' 109 resource transformation to avoid alarming the user.

As part of the same dialog process, the LaserWriter driver confirms that the version of the Laser Prep dictionary that is loaded corresponds to the version that is resident on the Macintosh. This ensures that all the procedures and so forth that are expected to be present are, in fact, there and accessible.

▶ Using PostScript in QuickDraw

So far in this book, you have used PicComment to add PostScript code directly into your document. There are, in fact, other methods that you can use or that you may have heard about. This section lists all the ways you can add PostScript processing to your document and briefly explains how each one is used.

Note ▸

I do not propose to give examples, or even to provide much discussion, of all these methods. You have, so far, learned some very useful techniques for adding PostScript to your documents. However, when it comes to putting PostScript to work in your application, the Macintosh provides an enormous number of alternatives—well, more than three, anyway. These mostly do similar functions in slightly different ways. If I tried to cover all these techniques in the same depth, this book would grow, like Alice, very large indeed. Instead, I have concentrated on the few methods that seem to be most useful and, I might add, that are most commonly used in the development of Macintosh software.

▶ PicComment use

You already know how to use the basic PicComment commands to place PostScript code directly into a document. In addition to these forms of PicComment, however, a number of additional possible values of the PicComment perform a wide variety of services, some very useful and others, in my opinion, rather less so. Table 5-4 lists all the varieties of PicComment that are defined for the LaserWriter driver. All of these comments are available in drivers with a version of 3.1 or later, which is most drivers at this time. The complete specifications and use of these comments is given in Technical Note #91, which also provides some examples of how to use them.

Table 5-4. Valid PicComments that can be used with the LaserWriter driver

Type	*Kind*	*Data Size*	*Data*	*Description*
TextBegin	150	6	TTxtPicRec	Begin text function
TextEnd	151	0	NIL	End text function
StringBegin	152	0	NIL	Begin pieces of original string
StringEnd	153	0	NIL	End pieces of original string
TextCenter	154	8	TTxtCenter	Offset center of rotation
LineLayoutOff	155	0	NIL	Turns LaserWriter line layout off
LineLayoutOn	156	0	NIL	Turns LaserWriter line layout on
PolyBegin	160	0	NIL	Begin special polygon
PolyEnd	161	0	NIL	End special polygon
PolyIgnore	163	0	NIL	Ignore following polygon data
PolySmooth	164	1	PolyVerb	Close, Fill, Frame
PicPlyClo	165	0	NIL	Close the polygon
DashedLine	180	–	TDashedLine	Draw following lines as dashed
DashedStop	181	0	NIL	End dashed lines
SetLineWidth	182	4	Point	Set fractional line widths
PostScriptBegin	190	0	NIL	Set driver state to PostScript
PostScriptEnd	191	0	NIL	Restore QuickDraw status
PostScriptHandle	192	–	PSData	PostScript data in handle
PostScriptFile	193	–	FileName	File name in data handle
TextIsPostScript	194	0	NIL	QuickDraw text is sent as PostScript
ResourcePS	195	8	Type/ID/ Index	PostScript data in a resource file

Table 5-4. Valid PicComments that can be used with the LaserWriter driver (continued)

Type	*Kind*	*Data Size*	*Data*	*Description*
RotateBegin	200	4	TRotation	Begin rotated port
RotateEnd	201	0	NIL	End rotation
RotateCenter	202	8	Center	Offset center of rotation
FormsPrinting	210	0	NIL	Don't clear print buffer after each page
EndFormsPrinting	211	0	NIL	End forms printing after PrClosePage

A few points need to be made about the comments listed in Table 5-4 that may not be quite obvious. The LineLayoutOff and LineLayoutOn comments perform the same function, in a slightly different way, as the FractEnable Boolean that can be set for all Macintoshes since the Macintosh Plus; these days, that's most Macintosh computers. The FractEnable Boolean and the routine that sets it are described in *Inside Macintosh, Volume IV*. The dashed line settings are, not surprisingly, used exactly like their PostScript counterpart, **setdash**. You will want to use the dashed line setting and the line width settings when you are working in QuickDraw and want to include these features in the lines you draw without having to go into PostScript. These can be really useful features when you need them. The rotation mechanism, on the other hand, is somewhat cumbersome and difficult to use. If you use it, notice in particular that the direction of rotation is reversed from the PostScript standard; incorrect use can be the source of many pictures lost in space.

Finally, note that the comments that set forms processing implement that processing by using the PostScript **copypage** operator. Use of this operator can have very negative effects on the throughput of your documents, and, generally, it is not recommended for use in generating forms. Before you use this feature, you should be familiar with the "Logos, Grids, Forms, and Special Fonts" section in *PostScript Language Program Design*. This gives you a variety of alternative methods for creating forms that are more efficient than the PicComment method here and, thanks to your work in this book, you will now know how to apply those methods from within your application.

▶ 'POST' resources

You have already been introduced to 'POST' resources and how they are used by PicComment 193 (PostScriptFile). This is a common mechanism for using 'POST' resources. In fact, it is the mechanism that PostScript fonts use to load the font definitions into the output device. However, many other programs, both system programs and applications, also use 'POST' resources that do not fall within the structure that you were introduced to earlier. In fact, it is quite customary for applications to place all their PostScript resource code into 'POST' resources with resource numbers other than 501 and higher, in order to control the use of and access to the resources.

Following this convention, while not required, can be of great benefit to you. If you store your PostScript text in a resource file, as most applications do, you may well want to access and control this information yourself rather than allowing the driver to do the job for you. Often, the first thought is to create a 'STR ' or 'STR#' resource for this purpose, and, of course, that works. However, you could just as well use a 'POST' resource and thereby make your code more accessible and maintainable because the resource would be one of those commonly used for storing PostScript information. Of course, such decisions are up to you.

▶ PostScript Escape font

The LaserWriter driver contains a special 'STR ' string resource -8188, which is the name of a font. The default name is PostScript Escape, but you can change that to any name that you want. When you change that name to an existing font that has been installed in the system, or create a font with the name PostScript Escape and install it, any text placed into a document in that font is transferred directly to the printer with no further conversion. In other words, everything that you enter in the PostScript Escape font is treated as actual PostScript code and passed directly to the printer for processing.

▶ PtrCtlCall

At one point in its life, the Macintosh supported low-level printer control calls that you could use in place of the normal, high-level calls that form the basis of most of the definitions in *Inside Macintosh, Volume II.* However, these calls were not supported from Version 5.0 to Version 6.0.1 of the LaserWriter driver, although current drivers do support them. Technical Note #192 spells out why these calls were discontinued

and alternatives for processing with these calls. In general, it is not good practice to use these calls, and using them for PostScript is, and has always been, very difficult to do correctly. For these reasons, using PtrCtlCall is not discussed here. Instead, you should use PicComment and QuickDraw with the high-level commands as you have seen throughout this book.

▶ Writing PostScript Headers

One last method of sending PostScript to the device has not yet been touched upon, but it is one of the most common and most important methods. This is the use of the 'PREC' resource to hold PostScript header or dictionary data. As you have seen, the common and correct method of providing procedures for your PostScript code is to create a personal dictionary that holds all the procedure definitions that you require. This is also consistent with the structural requirements when you are creating a PostScript page that conforms to the established PostScript document structure. The question is: What is the best way to provide your own PostScript prologue or dictionary?

So far, you have simply embedded the PostScript that you want to use in the actual page being printed. This works quite well if there is only one page (as is true in most of our examples), but works less well if you are printing multiple pages. Consider the example you worked on earlier where you were adding the word "DRAFT" to every page of a document being printed if a special draft check box was set in the job dialog. The code in the example simply adds all the required code to each page; admittedly not much code in itself, but still quite a bit of redundancy if you were printing many pages. Moreover, if you define procedures within the code that you are creating, then you are violating one of the structuring conventions. You remember that these require that all procedure definitions should be in the prologue, with none in the script. Adding code to each page, however, introduces procedure definitions into the script, in violation of that rule. For all these reasons, therefore, you would like to have a method to define and load your own dictionary.

As you see, a high overhead is associated with putting all the code separately into each page; it increases transmission time for the code and increases the time required to execute the code. The better way is to create your own private dictionary and load that every time that you want to perform your own code, such as add the DRAFT message to the page. In this way, you simply load the code one time and then execute it as many times as required for your processing. This also allows you to

have many procedures if you require them, which can be selectively executed as you determine within the application that they are needed. This section tells you how to use the 'PREC' resource to provide such a dictionary.

▶ Using 'PREC' 103

The process of using the 'PREC' resource depends on a system feature of the Macintosh: that the driver looks for a specific resource in the application and includes it as part of the document as it formats the document for output. The actual resource required is type 'PREC', number 103. This resource consists of ASCII text data, including all required linefeeds for separation. If this resource exists in the application, it is placed into the document at the point immediately after the **md** dictionary is loaded onto the dictionary stack. Strictly speaking, this makes it part of the document setup rather than part of the prologue; however, that placement works perfectly well both in basic processing and with spoolers since the document setup code, like the prologue, is executed prior to anything else in the document.

The changes required in the application to take advantage of this are really quite minimal. You need to create the 'PREC' resource with the necessary dictionary definition and procedures, and you need to modify the code in **Draw** to use the new dictionary and its code. As always, you could simply use ResEdit to create the resource, but here you create an RMaker file that will add the desired resource to your present SimpleLW.π.rsrc file. The required code is shown in Listing 5-27.

RMaker does not have a template for a 'PREC' resource, so you create a new type from the 'GNRL' type. Most of this is simply ASCII strings; however, as you see, you have to allow for one, perhaps surprising, wrinkle. Remember that the resource must contain all the required code, including linefeeds or returns, as necessary, to separate lines. Unfortunately RMaker does not automatically insert linefeeds after each line that you code here. To insert a newline character into your text, you must code it in hexadecimal. (There may be some other way to do this, but I haven't found it!) The hexadecimal code for a linefeed is 0x0D. So, at every point where you want a new line, you insert that code. This is not difficult, just a bit surprising. You should remember that, although PostScript itself does not care how long a line is, there are many common systems and document processors, such as spoolers, that expect lines to be not longer than 255 characters. I would recommend that you break your code into reasonable logical segments, as you have here, that are no longer than that.

Listing 5-27. Code to create the 'PREC' resource

```
!SimpleLW.π.rsrc

* added resource declarations follow ...

TYPE PREC = GNRL
    ,103            ;; must be res no.103
.S                  ;; this is ASCII text comment
/SmallLW 10 dict def
.H
0D
.S
SmallLW begin
.H
0D
.S
/restoreCoord
{
    0 760 translate
    1 -1 scale
} bind def
.H
0D
.S
/draftFont /Helvetica findfont 72 scalefont def
.H
0D
.S
/prDraft
{
    gsave
        currentpoint translate
        newpath
        draftFont setfont
        45 rotate
        0 0 moveto
        (DRAFT) false charpath
        0 setgray
        stroke
    grestore
} bind def
.H
0D
.S
end  %SmallLW
```

Note ▶

There is one place where PostScript does care, in some sense, about line length, and that is when it is processing comments. If you recall, a PostScript comment begins with the % character and continues *to the end of the line*. That means that, if you insert a comment into the text stream here, be sure that you follow it with a linefeed at the proper point, or everything after that point in the text will be considered a comment. Notice that the only comment here is at the very end of the text, where there is no more executable code to follow. Generally, I recommend that you avoid comments in your 'PREC' resource. The comments simply take up space and do nothing to improve the readability of the resource.

The code in Listing 5-27 simply defines a new dictionary, **SmallLW**, and inserts the required procedures and definitions into it. You should be familiar enough by now with this process to be able to follow the code here without much difficulty.

In addition to the 'PREC' resource, you need to make some minor alterations to the **Draw** method in CSimpleLWPane.c. The changes to the code are shown in Listing 5-28.

Listing 5-28. New code in the **Draw** method to use the **SmallLW** dictionary

```
/**********************************************************************
 * Draw {PicComment 192 Method}
 *
 *  In this method, you draw whatever you need to display in
 *  your pane. The area parameter gives the portion of the
 *  pane that needs to be redrawn. Area is in frame coordinates.
 *
 ***/

void CSimpleLWPane::Draw(Rect *area)

{
    char        *theString;
    char        *strPtr;
    long        strLength;
    OSErr       theError;
    Rect        picFrame;
    Boolean     isDraft;
```

Listing 5-28. New code in the **Draw** method to use the **SmallLW** dictionary (continued)

```
picFrame = *area;
myMacPicture = OpenPicture( &picFrame );
ClipRect( &picFrame );

/**
    first draw a basic rectangle that outlines the frame
**/
InsetRect( area, 2, 2 );
FrameRect( area );

/**
    tell us (on the screen) what's happening
**/
TextFont( helvetica );
MoveTo(20, 20);
DrawString("\pPostScript from QuickDraw TextHandle.");

/**
    set up for PostScript comment data
**/
theString = "\p0 761 translate 1 -1 scale /Times-Roman findfont
    20 scalefont setfont 50 650 moveto (Hello, world!) show\n";
strLength = theString[0];
strPtr = theString + 1;
theError = PtrToHand( strPtr, &myPSHandle, strLength);
if ( theError != noErr )
    gError->SevereMacError( theError );

/**
    do PostScript stuff for LaserWriter
**/
PicComment(PS_BEGIN, 0, NIL_HAND);
PicComment(PS_HANDLE, strLength, myPSHandle);
PicComment(PS_END, 0, NIL_HAND);

/**
    tell us (on the screen) that we've finished
**/
MoveTo(50, 50);
DrawString("\pPostScript processing completed." );
```

Listing 5-28. New code in the **Draw** method to use the **SmallLW** dictionary (continued)

```
    /* test for printing status ... if so, add DRAFT if requested  */
    isDraft = ((CSimpleLWDoc *)itsSupervisor)->TestDraftStatus();
    if (printing & isDraft)
        {
            if (((CSimpleLWDoc *)itsSupervisor)->isLaserWriter)
            {
                PicComment(PS_BEGIN, 0, NIL_HAND);
                PicComment(PS_TEXT, 0, NIL_HAND);
                    DrawString( "\pSmallLW begin" );
                    DrawString( "\prestoreCoord");
                    DrawString( "\p200 300 moveto" );
                    DrawString( "\pprDraft" );
                    DrawString( "\pend" );  /* SmallLW  */
                PicComment(PS_END, 0, NIL_HAND);
            }
            else
            {
                /* if not a LaserWriter,
                    do something less elegant to show draft status */
            }
        }

    ClosePicture();

    /**
        store picture handle as instance variable
    **/
    inherited::SetMacPicture( myMacPicture );
    inherited::Draw( &picFrame );

}
```

The PostScript code now simply places **SmallLW** onto the dictionary stack and executes the required procedures. Notice that you could have combined these two procedures if you wanted, but it is generally preferable to keep procedures such as these, which do two different things, as separate entities.

There are several points to notice about this new code. First, you might just as well have placed the **SmallLW** dictionary onto the

dictionary stack during the document setup code, from 'PREC' 103, as doing it here. That would be consistent with how **md** is handled, and it would save you two lines of code on each page. If you were going to do much processing—for example, if your application was generating many pages of complex code that relied on your dictionary—that would certainly be the correct approach. As it is here, you are doing a small amount of conditional processing, so you put the dictionary on the stack at the start of processing and remove it at the end. This seems to me to be slightly clearer and more tidy than the alternative, but the choice is quite subjective. Try the code the other way and see how you like it.

Second, you notice that the DRAFT code is positioned on the page in a separate step, with a **moveto** command, instead of having that code inside the **prDraft** procedure. This was done so that you could move the word around, if you like, to accommodate different size pages, for example, or to use the procedure in some other way. This, too, is mostly a matter of programming taste. You could embed the movement command into the procedure if you wanted.

Now you should run RMaker to update the resource file and recompile the project. When you run the project and select Print, you get an output that looks like Figure 5-7. This is, as it should be, exactly the same as the previous output, shown in Figure 5-6. The differences are entirely internal and are invisible to the user.

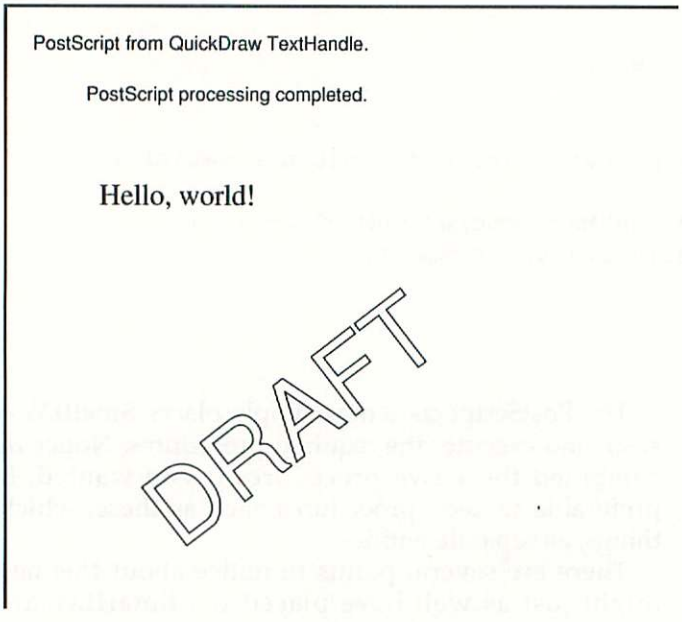

Figure 5-7. Output with private dictionary

Now that you know how to add your own dictionary to the document, there is an additional piece of code that can be very useful: an error reporter. As you recall, any PostScript errors in your document are reported back to the Macintosh over the AppleTalk network. However, they are only reported to you if the application is still printing at the time the error occurs. Even in that case, the error message is often replaced by a rather uninformative dialog box.

To ensure that you see any errors that occur during processing, you can insert some simple error reporting code that will print out the error message if one occurs at any time while the page is being processed. Although error handling and error reporting in PostScript are well worth discussing, they are too far from our present focus to allow much explanation of the code shown here. If you are interested in these topics, you should consult some of the reference works listed in the Bibliography, which will provide a more complete description of what this type of code does. With that understanding, Listing 5-29 shows the revised 'PREC' code that creates and installs a simple error reporter.

Listing 5-29. Revised 'PREC' including error reporting

```
!SimpleLW.π.rsrc

* added resource declarations follow ...

TYPE PREC = GNRL
    ,103            ;; must be res no.103
.S                  ;; this is ASCII text comment
/SmallLW 10 dict def
.H
0D
.S
SmallLW begin
.H
0D
.S
/restoreCoord
{
    0 760 translate
    1 -1 scale
} bind def
```

Listing 5-29. Revised 'PREC' including error reporting (continued)

```
.H
0D
.S
/draftFont /Helvetica findfont 72 scalefont def
.H
0D
.S
/prDraft
{
    gsave
        currentpoint translate
        newpath
        draftFont setfont
        45 rotate
        0 0 moveto
        (DRAFT) false charpath
        0 setgray
        stroke
    grestore
} bind def
.H
0D
.S
/errorPrint
{
    errordict begin
    userdict /PSerror /handleerror load put
    /handleerror
    {
        gsave
        /Courier findfont 12 scalefont setfont
        newpath
        100 360 moveto
        gsave
            -10 -10 rmoveto
            425 0 rlineto
            0 45 rlineto
            -425 0 rlineto
            closepath
            1 setgray
```

Listing 5-29. Revised 'PREC' including error reporting (continued)

```
            fill
        grestore
        0 setgray
        (Error in job: ) show
        statusdict /jobname get show
        100 372 moveto
        (Error: ) show
        $error /errorname get 255 string cvs show
        (  Offending Command: ) show
        $error /command get 255 string cvs show
        systemdict /showpage get exec
        /PSerror load exec
        grestore
    }
    bind def
    end
}
def
.H
0D
.S
errorPrint
.H
0D
.S
end  %SmallLW
```

The revised code simply replaces the default PostScript error processing, a procedure called **handleerror**, with a new procedure that prints a message if an error occurs and then executes the default processing, which is saved in **userdict** with the new name **PSerror**. The new error reporter erases a portion of the page, and prints a message in Courier, a font that is always available. The message consists of two lines: the name of the current job, if there is one; and the nature of the error, using the standard PostScript descriptions for the error name and the offending object.

To use this error reporter, simply run RMaker again to include this code into the 'PREC' resource. Notice that the error reporter is set up as a procedure that can be executed and then is executed immediately

after it is defined. This allows you to disable the error reporting very easily, by simply going into the resource file with ResEdit and inserting a % in front of the **errorPrint** procedure call at the end of the file. Also, if you turn off the reporter here, you can still activate it inside your document at any point by simply executing the **errorPrint** procedure.

Once you have created your new resource file, you can test the error reporter by opening CSimpleLWPane.c and changing any of the PostScript code to cause an error. For the example here, the **moveto** operator was misspelled as **mxveto** in the draft processing section. Running this causes the output shown in Figure 5-8.

Figure 5-8. Output from the error test with error reporting installed

Let me emphasize that this is a very simple error reporter; like the other code in this book, it is intended for illustrative purposes and is not of commercial quality. If you are going to do much PostScript processing, you may want to get a commercial error reporter. One such program is the *PinPoint Error Reporter*, which is a commercial product from the Cheshire Group. This product provides additional information about the type of error, what caused it, and where it occurred in your program. You can reach the Cheshire Group at the address listed in the Bibliography.

▶ Controlling dictionary contexts

Generally, you don't need to worry much about which dictionaries are on the dictionary stack when you are executing. If you want a special dictionary on the stack, you normally place it there yourself and remove it as required. You are already familiar with this process.

You can always assume that the Laser Prep dictionary, **md**, is on the dictionary stack since it is placed onto the stack as part of the document setup processing. As you saw earlier, you can place your own dictionary onto the stack at that same time or you can place it on when you execute your code—being sure, of course, to remove it at the end or you will get an **invalidrestore** error when **md** resumes processing for the remainder of your document. Remember that this occurs because the **pse** procedure that terminates your PostScript processing does a **restore** to the state saved previously by the **psb** procedure that began the processing.

In general, therefore, there is no problem to using dictionaries in a very natural way during your PostScript processing. The single possible area of concern arises if you want to place a dictionary into service, depending on some document parameter, and not have it be automatically loaded as part of the document setup.

You can see the potential problem here. Anything that you place in the 'PREC' 103 code resource will be executed at the beginning of the document, without letting you have any further control over the processing. On the other hand, if you are processing within the document and decide at that point to use a specific dictionary, you will have a problem since the **psb**, **pse** operations will generally cause loss of data and potential errors if you have installed a new dictionary during your PostScript processing.

There is a way around this problem. The solution is to issue the **pse** command in your own code as soon as you enter, create, or install your dictionary, do the work that you need to save, and then call **psb** yourself. This performs the **restore**, adds your new data and procedures (or whatever), and then performs the **save** that will match the **restore** that happens automatically when you exit. Everyone is happy. You have satisfied the processing requirements for the Laser Prep file while still installing your own dictionary into the process. From that point in the document on, your dictionary will continue to be available and will contain the new values that you have created.

The only possible drawback here would be if **md** were changed so that these two procedures no longer existed, or performed some other, incompatible functions. To guard against that, you need to test the version of **md** that is installed in the printer. You will see how to do that in Chapter 7.

▶ Exiting the server loop

The final issue that may occur to you is how you could download your dictionary permanently, like **md** itself, so that it would always be present in the device. Let me say, right at the start, that this is seldom necessary for ordinary programs. There are very few situations where you can justify taking up a permanent part of the device memory for your own procedures. The single exception to this is if you are installing some utility software, for example, an error reporter or accounting software, that must, by its nature, be part of the processing cycle at all times.

Once upon a time, nevertheless, such an action was possible using a variant of the process described earlier. If you loaded your code into a 'PREC' with an id number of 201, the system would download that software outside the server loop, just as it does **md**. However, with the advent of multiprocessing and spooled printing, this feature has been discontinued, as noted in Technical Note #192. You can no longer use this feature to guarantee that your code will be downloaded outside the server loop. If you require code to be loaded outside the server loop, then that code must be downloaded manually and must contain the correct instructions for exiting the server loop, which you read about earlier.

At some risk of belaboring a point already made, there is almost never a need to exit the server loop in ordinary processing. To begin with, when you are outside the server loop, you have the potential to damage the interpreter and to inadvertently modify important system variables, such as the communications settings and the password, that may have severe negative repercussions on further processing. All such changes should be made with the greatest care and only after significant and extensive testing. The necessary precautions are beyond the scope of this book, and I urge you to be very careful before you undertake any such changes without a fair amount of PostScript programming experience. The small possible benefit that you gain from having the procedures already loaded is offset by the amount of testing that you have to provide to ensure that the code is, in fact, already there when you begin. You should also note that even Apple has decided that, beginning with System 7.0, it is not necessary to load all of **md** that way anymore.

▶ Conclusion

In this chapter, you have gotten much closer to the LaserWriter. You have learned how to identify it in the print record. You have also gotten much more familiar with the interior processing that is generally hidden inside the LaserWriter driver and the Laser Prep code. The chapter has brought

you through these necessary but complex modules in a way that I hope has been both illuminating and satisfying. At least you now see why I said, in the Introduction, that this book would make you a better programmer, even if you never use these tools. You certainly understand the printing process now in a way that is not possible if you have not worked through this process in the kind of detail that you have followed here. As always, the Macintosh system reveals unexpected delights and complexities as you penetrate further into it.

You may think that you've "gone about as far as you can go"—to paraphrase the song a bit. However, if you remember the outline that we started with back in the Introduction and expanded in Chapter 1, you will remember that there is still one further layer to be uncovered. That is the process of interacting directly with the device itself, as you control the communications as well as the processing code. When you do that, you begin to bypass the Printing Manager and the LaserWriter driver altogether.

Before you can do that successfully, however, you must understand some more about one of the most critical interactions that takes place between the driver software and the device: the matter of fonts and font handling. Once you begin managing your own device communications, you must also accept the responsibility for providing the required fonts as well. This is quite a complex subject; so much so, that the entire next chapter is dedicated to it. After that, Chapter 7 takes you into direct communication with the LaserWriter.

6 ▶ Font Handling

▶ Chapter Overview

This chapter deals with all the different varieties of fonts that are a part of the Macintosh printing structure. With fonts, as with everything else about printing on the LaserWriter, there are two different types of fonts to understand: *screen fonts* and *printer fonts*. The screen fonts, as you might suppose, are created and managed by the Macintosh Toolbox software, particularly the Font Manager and QuickDraw. The printer fonts, naturally, are created and managed by the PostScript interpreter; and your application is somewhere in between these two. Each environment is quite distinct, yet both must work together to provide your application with printed output that looks like what you see on the screen.

This is not an attempt to present a comprehensive look at font handling in either of the two font environments; to do so for either would require at least an entire chapter, and to integrate the information would take at least another chapter. This one chapter simply focuses on how these two systems interact and communicate. Of course, this requires some brief background and review on each of these dissimilar systems, but the primary concern is on how the two pieces interrelate.

The interaction between the printer and screen environments is essentially divided into two parts. First, you need to follow how the Macintosh translates user requests for a given font and style in a document into a PostScript font selection. This is by no means an obvious or simple translation, and it involves several steps that you

should understand. Second, there are the issues of how to determine what fonts you have on the printer and how to manage printer resources for those fonts that are requested in a document. (This section ties directly into Chapter 7 where you discover how to access the LaserWriter directly for just this type of information.)

Like Chapter 2, this chapter is all discussion and information; it has only two examples of code. This chapter is designed to give you the minimum amount of information you need to be able to work comfortably with fonts. Font management is one area where all application developers need to be concerned and watchful, because if you don't work with this correctly, your users will not be able to see what they are going to print. I assume that some or possibly most of this information is at least generally familiar to you from previous programming. If you want to explore fonts in more depth, you can consult the tutorial materials listed in the Bibliography. You can use the techniques that you used in Chapter 4 along with the PostScript tutorial materials to work with PostScript fonts.

▶ Font Concepts

This section quickly reviews font types and how fonts are used, on both the Macintosh and the LaserWriter. As you have seen in earlier chapters, the differences between QuickDraw and PostScript create some small but important distinctions that you must understand to fully exploit the LaserWriter. How these two environments create, access, and display fonts is an important component of the printing process and, like the other areas you have become familiar with, it is also an area where they differ in both concept and execution.

Much of this material is probably familiar to you in some degree, and some of it, particularly the PostScript material, has been covered in earlier chapters. Nevertheless, this material is included here to refresh your memory of these two basic processes and to set the stage, as it were, for your journey to join these two mechanisms into parts of a unified system.

Note ▶

At the time of this writing, System 7.0 has not yet been released. Unfortunately the nature of the process of writing and producing books precludes "late-breaking" information. Nevertheless, I will try to indicate in this section how the new TrueType fonts have changed this process; understand, however, that this information is based on early documentation and experimentation with unstable, pre-release versions of complex system software. If the description here differs a bit from the actual implementation, please bear with me.

▶ Bitmapped and outline fonts

For our purposes, fonts are stored on a digital device in two basic formats: *bitmapped fonts* and *outline fonts*. Bitmapped fonts essentially consist of little "pictures," made out of dots, that represent the chosen character. This set of dots is then displayed on the output device to represent the character. Outline fonts, on the other hand, are stored as drawings of the outline of the desired character. These outlines, like a drawing that you might create with a drawing package on your computer, consist of lines and curves, joined together to draw an outline of the desired character.

These two types of fonts differ fundamentally in how they are stored and used by the system. They also each offer certain advantages and drawbacks in processing and storage. Bitmapped fonts are faster to display because the bitmap already exists and simply needs to be placed on the display surface. Outline fonts take longer to display because they require some calculation to turn the curves and lines into a visible character. Bitmapped fonts are also generally of better quality (surprise!) at the *exact* size that they were designed to be displayed, since each dot has been carefully chosen, often by a graphic artist or other professional, to represent the character image. However, bitmapped fonts lose their quality, and even their readability, when they are changed in size. That is, a bitmapped font that is designed to be exactly 10 points looks very good at 10 points, looks rather bad at 11 or 12, and may be virtually unrecognizable when expanded to 36 points. Outline fonts, on the other hand, are not necessarily so elegant at any given point size, but they retain their quality at all possible point sizes; moreover, since they are drawings, they can easily be scaled to any desired size, even a fractional one. Bitmapped fonts also require a lot of storage because you must have one bit (at least) for each pixel that you

have in a character. Outline fonts, however, are relatively compact, especially since one outline can be used for all sizes of the font, whereas a bitmapped font must have separate storage for each desired size.

At one level, all fonts for raster-output devices—which includes the Macintosh screen and the LaserWriter printer—are bitmapped fonts. This means that, at the end, the outlines must also be reduced to bitmaps for display purposes. This general process is called *scan conversion*, and it is quite important that it be done correctly and accurately. When outline fonts are converted into bitmaps, it is important that specific features of the characters be kept so that the character is clearly recognizable. If the outline is being scan-converted on a low- or medium-resolution device, this conversion can become very difficult to do using just the information in the outline itself. Many forms of outline fonts use additional *hints*, which are rules that allow the conversion process to be adjusted to make a better bitmap from the given outline on a given device at the given size.

At the present time, the Macintosh system uses bitmapped fonts for screen display and outline fonts when you print your output on a LaserWriter. How this process works is the primary subject of this chapter. The Macintosh System 7.0 software uses outline fonts, called TrueType, for the screen, although the outline format is not the same as the PostScript outline format used in the LaserWriter printer. Note also that Adobe Systems has released a commercial program, called Adobe Type Manager, or ATM, that does create screen font characters from PostScript outline fonts. The bottom line of all this is that, in a short time—and possibly by the time you read this book—most bitmapped fonts will have disappeared from the Macintosh screen and been replaced by outline fonts. Even with this change, however, most of what you read here will still be true, with the conversion of the outlines into bitmaps being transparent to both the user and the application to a great extent.

▶ Macintosh fonts

When you want to draw text on a page, whether that page will be displayed on the screen or printed onto a piece of paper, the Macintosh system needs three items of information to determine what font you want: the font or family *name*, the *style*, and the *size*. Generally, a user selects this information from a menu or series of menus. The Macintosh system provides many fonts and font families, and it provides all of them in seven varieties: bold, italic, underlined, outline, shadowed, condensed, and extended. These font styles may be built into the font itself, or they may be derived from the basic font. A font whose combi-

nation of size and style is built into the font itself is called an *intrinsic font*, whereas a font whose size or style must be generated by modifications to the basic intrinsic font is called a *derived font*.

When you select a specific font, style, and size from the type menus, QuickDraw asks the Font Manager to identify the desired font. In all Macintosh computers since the Mac Plus, this is done by looking in the System resources for a 'FOND' resource. The 'FOND' resource provides a variety of information for the Font Manager, including pointers to other resources that actually specify the information required to draw the font characters. (This process is clearly set out in *Inside Macintosh, Volumes I* and *IV*.) The 'FOND' resource also specifies which combinations of style and size are intrinsic; then, for all forms of the font that are not intrinsic, a derived form is created when that selection is requested. Note also that the 'FOND' resource determines font characteristics, and that there are intrinsic and derived font formats within the bitmapped format.

▶ PostScript fonts

You have already been introduced, in earlier chapters, to the fundamentals of PostScript font processing. Although it has many nuances, font processing in PostScript is basically quite an elegant and simple process. A PostScript font is a dictionary that defines the outlines, or shapes, for each character in a given typeface and style. Notice here that PostScript fonts provide only one style with each font dictionary. This arrangement is similar to that of typesetting, where an italic font or a bold font, for example, is independent of, although having common characteristics with, the original typeface. Characters from a PostScript font are created by executing ordinary PostScript routines that generate the character image from the outline description.

A font specifies the shape of its characters at one standard size. This is generally one unit, which would be one point in the default coordinate system. You then use the font machinery to convert this outline to the desired size and orientation, as was done in earlier examples. The font exists, as it were, in its own character space, which is independent of the ordinary user space. In this way, the font can be scaled, rotated, or otherwise transformed without altering the user space coordinates in any way.

The dictionary that holds the information for a given font is an ordinary PostScript dictionary, but certain key-value pairs must be present in it to be a valid font. Although the nature and function of these entries are well explained in the reference materials given in the Bibliography, you should be aware of two important entries:

Encoding	array	This array is stored in the font and contains 256 elements. Each element in this array is a name that identifies a routine to draw the character shape equivalent to that position in the array. The elements are accessed by using the character codes from the keys that you press as the index.
CharStrings	dictionary	This dictionary associates character names, as defined in the **Encoding** array, with the outline descriptions that generate the characters. These descriptions are often stored in a protected, compact format.

To request a PostScript font, you use the **findfont** operator to look up the desired name in the global **FontDirectory**. Remember that a PostScript font name typically includes style information; for example, Times-Roman is distinct from Times-Italic. However, as you saw in earlier examples, there are general methods for modifying PostScript font coordinates and for generating special effects with PostScript fonts. For this reason, it is possible to derive some styles from PostScript fonts in a manner similar to that used on the screen for derived fonts.

If your LaserWriter or other output device has an external storage mechanism such as a hard disk, this process has one additional wrinkle. If the requested font dictionary is not found in the **FontDirectory**, the **findfont** operator will search the attached disk to see if it can find the font there. If it does find the font, the font is brought into memory and the font name is added to the **FontDirectory**.

▶ Font metrics

One important issue in handling fonts is the size of the characters. This not only includes the height, which is determined by the point size, but it also includes the width of the characters. Basically, there are two types of fonts: monospaced fonts and proportional fonts. Fonts whose characters are all the same width are called *monospaced fonts*, whereas fonts in which each character has an individual width are called *proportional fonts*. Once, all computer output looked as though it had been created on a typewriter, where the widths of all the characters were the same. The Macintosh and the LaserWriter, however, changed all that, and now most of us regularly use proportional fonts.

Using a font, any font, to display text requires that the display software, which means QuickDraw on the Macintosh and PostScript on the LaserWriter, must know how wide each character is. When you were using monospaced fonts, this determination was quite simple because all letters required the same distance. With the rise of proportional fonts, however, the process becomes a little more complex because the software has to know the width of each individual letter.

There is one more issue that arises with proportional fonts. Some pairs of letters, like AV, can and should be placed closer together than either one would normally be placed to another letter; moving the two characters together makes the line of text cleaner and more readable. This process is called *kerning*, and the two letters that are moved in this way are known as a *kerning pair*. So, in addition to handling variable character spacing, high-quality text manipulation requires the use of kerning as well.

All of this information—the widths of each character, the kerning pairs, and the amount of kerning for each pair—is known collectively as the *font metrics*.

Macintosh internal font metrics

The Macintosh system maintains font metric information for the font. This information is collected and maintained by the Font Manager for use in the system. One important point to note here is that character widths on the screen must be integer values, just as all QuickDraw coordinates are integers. However, as you already know about coordinates in general, the coordinates in PostScript user space are real numbers and can therefore have a fractional part. This same process can be applied to font placement. In the original Macintosh, only whole numbers were allowed as widths. In the later Macintosh systems, however, the Font Manager maintains more precise fractional widths that can be used to improve the spacing and alignment for text on the LaserWriter. Of course, this precision has no measurable effect on the screen because QuickDraw still must position characters on pixel boundaries; however, on the LaserWriter and on higher resolution devices, this can make the text more elegant by taking advantage of more precise spacing.

The 'FONT' and 'NFNT' resources also maintain information about kerning the characters. Like the character widths and style information, all this is maintained and used by the Font Manager to pass the correct information along to QuickDraw to make the screen display look the way you want.

PostScript internal font metrics

PostScript fonts derive their character widths from the instructions that draw the characters in the **CharStrings** dictionary. This provides the basic information for each character's position in relation to the next character. In order to adjust intercharacter spacing, to do kerning, for example, you must provide the adjustment factor in the PostScript code that is generated, using something like the **kshow** or **awidthshow** operators, depending on the desired effect.

Note ▸

In PostScript Level 2, this all becomes much easier. Level 2 has new operators, such as **xshow**, that allow you to position every character in a string precisely where you want it on the page. The result is that, once you have adjusted the characters on the screen to the user's satisfaction, you can simply export that line of text, with the same exact placement, to the printer without doing anything further.

Adobe Font Metric files

Since the exact widths of the PostScript font characters are stored in the printer, where you normally can't get to them during processing, there is another method defined to retrieve the character metrics information for use in your application. You do this by using the information stored in the Adobe Font Metrics (AFM) file for your font. This file has a very rigid format that allows a program to access and parse the information easily. The file contains information about the metrics for each character in the font, as well as information about how to kern various character pairs and recommendations about what pairs should be kerned. The complete description of this file, along with suggestions regarding its use, is contained in the publication, *Adobe Font Metric Files Specification, Version 3.0*, which is available from Adobe Systems; see the Bibliography for a complete reference. *Display PostScript Programming* (Addison-Wesley, 1990) provides a discussion of how to load and use AFM files from a C program along with some example code that uses these files for standard text processing tasks, such as centering a string.

▶ Font encoding and character mapping

When you press a key on your keyboard to enter a character, what you get is, as is so often the case with the computer, a number that represents the key that was pressed. Each key on the keyboard generates a specific number and these numbers must be transformed into the characters that you see on the screen. This process is called *character mapping*.

When you print, a virtually identical process must take place. In this case, the numbers are inside the machine, as character codes, but they must be transformed into actual character output before they are useful. There are two factors that complicate this process. First, there are a variety of languages, and hence alphabets, that must be supported if you want to sell into the global market. Second, many of the languages that share an alphabet still have different requirements for certain characters. As a result, there is a mechanism, called *character set encoding*, that provides the mapping from the internal character codes to the outlines that define the actual character.

This process is a bit complex, but it is essential to obtain the full flexibility of the output. The following is a description of this process for roman alphabets; that is, alphabets that use fonts with a maximum of only 256 characters. For Asian or non-roman alphabets, which may have many more characters, the process is similar but even more complex. Thus it is beyond the scope of what is to be covered here.

The process inside the LaserWriter begins with the character code in a string to be displayed. This is a number from 0 through 255 (remember, this is only for roman alphabets). This number is then used as an index into the **Encoding** array that you read about earlier. The name that is at that position of the array is extracted, and that name is used as a key into the **CharStrings** dictionary. The name in **CharStrings** has an associated value that is the outline procedure required to render that character in the given font. The outline code is then executed to draw the character.

This structure gives the application enormous flexibility. In particular, it allows you to map any character code, and therefore any key, into any character outline that is defined in **CharStrings.** The only limitations are that you can only have 256 character definitions in action at any one time (although **CharStrings** might have more definitions than that, the **Encoding** array only has that many places), and you cannot use a character from one **CharStrings** dictionary in another font.

All of this processing, including the more complex processing required by non-roman alphabets, is well documented in the reference books that are described in the Bibliography.

▶ Font Processing

You are ready to look at the process of using fonts on the Macintosh and the LaserWriter. In this process, the driver software must solve two major issues. First, the driver must find and identify the correct font name to map the requested Macintosh font into a PostScript font. Second, it must then make the requested font available on the printer so that it can be accessed and used.

▶ Font selection

Once the user has selected a specific font name, style, and size from the text menus, QuickDraw and the Font Manager work together to generate a screen representation of the desired character. All of this selection and processing works off of resources and resource numbers. However, the PostScript interpreter in the LaserWriter must be given the name of a specific font before that font can be used on output. This section discusses how the LaserWriter driver converts the internal font identification into a PostScript font name.

PostScript font names

The process begins with the determination of the correct PostScript name for the desired font selection. Note that for every font family in PostScript, as in the Macintosh, some combinations of typeface and style are not defined. In this case, the Laser Prep file contains procedures that the driver can invoke to create the desired style variations. This is always the case, for example, for the shadow and outline font styles. As an example, you saw yourself in Chapter 5 that it is quite easy to use PostScript operations to create an outline style, as you did for the word "DRAFT."

The process of determining a name for the desired PostScript font, therefore, begins with the style-mapping table in the 'FOND' resource. The first two bytes of this table define whether the various styles of the font are represented by intrinsic fonts or whether they should be derived. This is followed by a series of indices, one for each style, that point to a style-name table, which is also part of the 'FOND' resource. The style-name table consists of a series of fragments of the font name: first the

base font name, and then name fragments that are used to build the entire PostScript font name. Thus, if you select Times, for example, and then select the Italic style, the driver constructs the correct name, Times-Italic. On the other hand, if you stay in the plain style, the driver constructs the name, Times-Roman. Once the correct name is constructed, that name is passed to the document and inserted with the correct procedures, drawn from Laser Prep, to use the font in the document.

Style mapping

Font name selection is a complex process, but it works very well. The only complication arises when the font does not support all the possible styles that are listed in the text menus. For some styles, this is never a problem. Underline, Shadow, and Outline are always generated by commands in the PostScript file. For these features, the base font name is inserted into the file along with calls to the procedures that are defined in Laser Prep that provide these attributes. Italic or Bold, on the other hand, generally require a distinct font name. In these cases, the correct font name is constructed using the name strings described earlier. Some fonts are designed only to be used as headlines, for example. Such fonts may not support any of the style features. In such cases, the standard font is substituted for the requested font and the base font is not adjusted.

This discussion should answer the question of what happens when you have a soft font, like StempelSchneider, which has both standard and italic versions that appear in the font menu, and you choose the base font with the italic style instead of using the italic version. The driver constructs a font name as required, which maps into the correct PostScript font name, whether you specifically ask for it or not. So you get the same, correct font in either case.

Another misunderstood point of terminology concerns italic and oblique fonts. You have seen how the font names are constructed when there is an italic version of the font, for example, and you know that, if an italic version is not available, the driver can use Laser Prep procedures to make the font appear italic. There is, however, a real difference in these two methods. The italic font is crafted, from birth, so to speak, to be printed at an angle. Each letter is actually designed to look as it does. The mathematical process of slanting the letters, on the other hand, is not the same at all. That process simply angles the original, upright letter to make an oblique version. And that tells you the difference between oblique and italic fonts. An italic font is designed and built from the beginning to be displayed at an angle, in the same way

that a script font is designed to look like handwriting. An oblique font, on the other hand, is simply a slanted version of the base font. This is, of course, the only type of italicizing that can be done for fonts that don't have an italic version.

Font coordination

The character-set encoding that is used by the Macintosh and the one that is native to the LaserWriter are slightly different. The default encoding in the LaserWriter, for example, does not include many of the accented letters that are provided on the Macintosh keyboard, and it also does not automatically include special symbols such as the copyright and trademark. Nevertheless, since the Macintosh keyboard has keystrokes for these characters, there must be some method to match these character values with the desired character output.

There is indeed such a mechanism, and it is performed by reencoding the PostScript font to include these characters. Each number represents a specific character in the encoding, and, for positions 0 through 127, it generally follows a standard structure called the ASCII encoding. However, the ASCII standard encoding does not extend above character position 127. Therefore, to map the keyboard characters into the character shapes that match the Macintosh screen display and to add those additional characters that are not present in the standard PostScript encoding, the LaserWriter driver uses Laser Prep procedures to *reencode* the requested font to match the Macintosh requirements. Fonts that have been reencoded to provide the new characters and to match the keyboard character codes are called *coordinated fonts.*

Obviously a font that has been reencoded must have a different name than the original font, or else you would never be sure what character you might get because only one **Encoding** array can be in use at a time. The LaserWriter driver uses this fact to establish a standard method of naming for coordinated fonts, so that it can tell by examining the font list that is returned from the printer which fonts that are already present have been coordinated. The rule is quite simple. Coordinated fonts begin their names with a vertical bar | followed by six underscore characters _ and then the original font name. For example, if you request the Helvetica font, the coordinated name is "|_____Helvetica."

Macintosh font names

One last point should be covered before moving into the process of locating and using printer fonts. As you already know, some of these fonts are stored on disk and loaded into the printer as requested.

However, the PostScript font names that are constructed by the driver are not in a format that can be used for file names. PostScript allows names that are quite long and names that may have different types of punctuation, but it does not allow spaces in valid names. Therefore, the driver must construct two names for each font: the internal PostScript name and the external file name. Instead of using the internal name directly, the LaserWriter driver constructs an external file name. This is based on the PostScript font name, but is not identical to it. The name is derived from the various parts of the PostScript name, using the capitalization of the name as a dividing mechanism, and ignoring the punctuation marks, such as the hyphen.

The rule for constructing the external file name is to take the first five letters or less of the first segment, and the first three letters or less of the remaining segments, and put them together to form the file name. If the name begins with the characters ITC-, then those characters are eliminated and the first segment begins after that. Therefore, a font name like StempelSchneider-BlackItalic becomes StempSchBlaIta, and so on. This name is important in the next section, when you read about downloading fonts.

Font use

Let's just put this all together for one font request.

- You select StempelSchneider Medium, Italic style
- PostScript name is /StempelSchneider-MediumItalic
- File name is StempSchMedIta
- Coordinated font name is / | _____StempelSchneider-MediumItalic

Once the font names have been determined, the driver can proceed to use the selected fonts in the document. First, however, there is the question of how the driver knows what fonts you already have in the LaserWriter. Is the StempelSchneider-MediumItalic font already available in the LaserWriter or not? The answer is that the driver queries the LaserWriter and asks it to list the fonts that are currently present. Then the driver stores that information in a table, which it maintains, adding additional fonts as they are loaded into the printer. This allows the driver to know at all times what fonts are present in the LaserWriter and what fonts it may need to provide in order to print your document.

With these points in mind, when you ask for a printed copy of the current page, the following process occurs. First, the LaserWriter driver constructs the font names as described earlier. Then it looks in the list of fonts currently on the printer to see if the font already exists on the printer. If so, no further action is required; the font name is simply inserted into the page description at the point where you want to use it. If the font is not already present, the driver uses the file name, constructed as described previously, to see if the desired font outlines are present on the Macintosh system. If the font is located on the system, the font is included with the document for processing and is used as requested. If the font is not found on the system, however, the screen bitmaps are converted into PostScript bitmaps and are placed into the file and used. This last situation accounts for the message that you will occasionally see, warning you that a screen version of the font is being used in your document.

This font strategy is quite reasonable and gives you the best possible font with a minimum overhead. Here is the strategy in a nutshell:

- If the font is already present on the printer, then the font is used without any further processing.
- If the font is not already on the printer, but is on the Macintosh system, then the font is loaded with the document and sent to the printer and used.
- If the font is not present either on the printer or on the system, a bitmapped version of the font is constructed and sent to the printer with the document and used.

The next point, quite clearly, is how to get these fonts loaded into the printer and where to store them. This is covered in the next section, where you will look at the download process for PostScript fonts.

▶ Font downloading

In the most precise sense, downloading is the process of sending font or other information and data to the printer. This section takes a somewhat broader view and looks at the more general process of determining which fonts need to be sent to the printer and when they are sent, as well as how they are formatted and sent.

Fonts that are required in a document fall quite naturally into two groups: fonts that must be included with the document and fonts that are already on the printer or are accessible to the printer. Fonts that

must be included with the document are called *transient* fonts. The previous section described how these fonts are determined and how the LaserWriter driver includes them into the actual document. These fonts are called transient because they only are available in the printer for as long as the document is being processed. When the document is finished processing, the fonts are removed from the printer memory to make room for the next document and its fonts.

Fonts that are already resident in the printer are called *permanent* fonts, but this name is a little misleading. These fonts may fall into any one of three categories. Table 6-1 lists these three types along with the distinctive qualities for each of them.

Table 6-1. Types of permanent fonts

Font Type	*Description*
ROM resident fonts	These fonts are stored in read-only memory inside the LaserWriter. These fonts are always present in the LaserWriter and never need to be downloaded, even if the printer is turned off or reset.
File resident fonts	If the LaserWriter has an external storage disk, these fonts are downloaded one time to the disk and remain on the disk unless the disk is erased or damaged. Once downloaded, these fonts are always available as long as the disk is available, even if the disk and the printer have been turned off or the printer has been reset.
RAM resident fonts	These fonts are stored in the random-access memory of the LaserWriter, just like the Laser Prep dictionary and the document program. Although these fonts don't go away at the end of a job, they are lost when the printer is turned off or reset.

As you can see from Table 6-1, the name "permanent" is used rather loosely. Only the first set of fonts, those built into the LaserWriter, are truly permanent in the normal sense of that word. The fonts stored in files on the hard disk are somewhat less permanent because the disk can be erased or lost. In that case, the fonts would need to be loaded to the disk again. The final case, where the fonts are stored in random-access memory just like a program, is the least permanent of these

options. Fonts stored here are more permanent than transient fonts because these fonts will persist for a series of jobs; however, as soon as the printer is reset, these fonts are gone as well.

Transient fonts are loaded by the LaserWriter driver as they are requested in the document, as we discussed earlier. However, the other types of fonts have other types of loading that go with them. The ROM resident fonts, obviously, do not ever have to be downloaded because they are built into the printer. The other types of fonts, however, do have to be downloaded. This is usually done by using a separate program that takes a font file and loads it onto the printer. The Apple LaserWriter Font Utility is one such program. When you start this program, it checks the printer to see if an external disk is attached; if one is, you are offered the option of loading the font to the disk or to memory. Whichever you choose, the font is then sent to that location.

To be recognized as a font, the font file must have certain characteristics. To begin with, the file must be a proper PostScript font and do all the things that are required of a well-formed font dictionary in PostScript. Assuming that, the file must have a file type of LWFN. The PostScript font definition is provided by the use of 'POST' resources, following the same pattern that you already know. Using that method, the font can be stored in the resource fork, using either 'POST' type 1, for standard ASCII text, or 'POST' type 2, for binary data; alternatively, it can be stored in the data fork of the file, using the 'POST' type 4 resource to direct the font to be loaded from there. In other words, a PostScript font, from the Macintosh side, looks remarkably like the PostScript code that you have been using for your previous exercises. Of course, the PostScript code itself is quite different.

▶ Font Management

The LaserWriter driver automatically handles most font management requirements for you. In fact, the driver is so good at this that, in general, you want to leave font management to it whenever possible. However, if you are controlling the communications link with the printer yourself, or if you need to directly manage fonts in the printer for some other reason, then you have to be concerned with two issues. First, you need to be able to discover which fonts are in the printer at any given time. This may take either one of two forms: a general inquiry of all the fonts that are currently loaded or a specific inquiry about an individual font. Second, you need to manage the memory resources that the font uses. In doing that, you need to be aware of two issues: how much memory you currently have and how much memory each font requires.

▶ Font inquiry

When you inquire about fonts that are already installed on the printer, you may either want a complete list of all the fonts on the system, or you may only want to know if one or two specific fonts are present. Each of these requests is handled a bit differently.

The request for a specific font is, at first glance, the easiest of these to handle. In essence, you want to perform the same function that a **findfont** operator would perform if you requested the font. The only problem is that a **findfont** does not tell you whether it finds the font. Instead, typical processing substitutes some standard, default font (usually Courier) for the requested font and returns a status message to that effect. Since you want to determine whether the font is loaded or not, you must send PostScript code that reproduces the process that **findfont** uses. If the printer only has internal fonts, the process is simply to look at the **Font Directory** to discover if the font is listed there. You can use any one of the standard PostScript dictionary operators, such as **known**, to determine this. However, if the printer has an external hard disk for storage, the process is somewhat more complex. In that case, if the font is not located in the **Font Directory**, you must search on the disk to see if it is there because fonts that are on the disk will be accessed and loaded by **findfont**. The code shown in Listing 6-1 handles both of these tasks.

Listing 6-1. Searching for a specific font

```
save
4 dict begin
/sv exch def
    /str (fonts/                                    ) def
    /st2 128 string def
    {
      count 0 gt
      {
        dup st2 cvs (/) print print (:) print
        dup FontDirectory exch known
            {pop (Yes)}
            {
                /filenameforall where
                {   pop
                    str exch
                    st2 cvs
```

```
                        dup length /len exch def
                        6 exch putinterval str 0 len 6 add
                        getinterval mark exch
                        { } st2 filenameforall
                        counttomark 0 gt
                            {cleartomark (Yes)}
                            {cleartomark (No)}
                        ifelse
                    }
                    {pop (No) }
                    ifelse
                }
                ifelse
            print flush
        }
        { exit }
        ifelse
    } bind loop
    (*) print flush
sv
end     %temp
restore
```

The code in Listing 6-1 here provides a response that conforms to the format required for printer description files. This routine expects a list of font names that are being tested, and it provides a response in the format

```
/fontname:Yes or /fontname:No
```

to show whether the font is present or not. This response is both easy to understand and difficult to misinterpret. Some font inquires simply send back an integer, or a Y or N response. This works well if you are only looking for a single font and if you are the sole workstation connected to the printer. For lists of fonts, however, particularly on a large network, there is always the possibility that a reply will be delayed or provided out of the original request sequence. As you can see, with this type of response, you need not be concerned about these issues because each response tells you both the font requested and the status. Notice also that the query is surrounded by a **save**, **restore** pair, so that no memory resources are used even if the request is embedded into another job. The replies are terminated by an asterisk character, 0x2A, which provides a positive indication that the request is finished processing.

Listing all the fonts on the system is actually easier, at least in terms of the amount of code required to do it, as shown in Listing 6-2. This

code simply returns the names of all the fonts on the system, one after another. Each font name begins with a /, which can be used to divide the list into individual names if required (as it normally would be). The == operator usually inserts a line feed, 0x0A, after each font name, but you should not rely too much on these characters, since they may be lost or removed in transmission; the slash (/), on the other hand, will always be in the returned string since your code has deliberately inserted it.

Listing 6-2. Listing all fonts on the system

```
save
    FontDirectory { pop == } forall flush
    /filenameforall where
    {
        /scratch 100 string def
        pop
        (fonts/*)
        { dup length 5 sub 5 exch getinterval == }
        scratch
        filenameforall
    }
    if
    (*) print flush
restore
```

These listings include code to test for and search an external disk file system. If you are interested in this process, the *Apple LaserWriter Reference* manual provides an excellent discussion of disk and file processing. Chapter 7 of this book also contains further examples of how to work with external files.

▶ Memory management

Virtual memory is probably the single most valuable resource in your LaserWriter. This is true for several reasons, but probably no reason is more compelling than the simple fact that running out of memory has particularly bad consequences. When your document exceeds the memory capacity of the printer, one of three things—none of them good—will occur. The best (believe it or not) is that you will get an error, **VMerror**, that tells you that you have exceeded memory and flushes your job. The next best (or least bad, take your pick) is that the

printer will simply not print your job and will return to the idle state after some time—essentially after it times out the job or flushes it internally. This is worse than the previous occurrence because you don't get an error message and have to guess about what went wrong. The last (and worst) is that the printer will reset, flushing not only your job but also any prep files, such as Laser Prep, and any fonts that you may have downloaded to the printer memory. On top of that, you still don't get an error message!

The most common cause of exhausting virtual memory is using too many fonts in your document. Since fonts are implemented as PostScript dictionaries, they require substantial amounts of room inside your printer, typically ranging from 15K through 50K bytes, with the average being about 30K. Many printers, therefore, have a significant limit on the number of downloaded fonts that you can use in a single document. Note that this is on *downloaded* fonts only; the fonts that are built into the ROM naturally do not require any additional space. This limitation also is much reduced, although not eliminated, when the fonts are on disk because then the entire font need not be present in the system at all times.

Discrete management of individual fonts is not very easy on the LaserWriter since PostScript only allows you to recover virtual memory by means of the **save** and **restore** processing. This essentially limits you to a strict Last-In, First-Out policy on adding and removing fonts, since there is no way to remove an early font and leave one that was loaded afterward. However, since each page in a multipage document should be enclosed in a **save**, **restore** pair, you can reload fonts for each page instead of loading them once for the entire job. In that case, you can use many more fonts than would be available otherwise.

Note ▸

This limitation is no longer true in PostScript Level 2. Level 2 provides a full memory recovery mechanism, usually referred to by the colorful name of "garbage collection," that allows you to remove individual fonts (and other objects) from memory once they have been used and are no longer required. Although this does not eliminate the requirement for memory management with fonts, it does give you the tools to do a much better and more efficient job.

▶ Spooler requirements

There are many good reasons to have spooling or background printing when you are writing to any printer. It increases the productivity of the users and also helps keep the printer fully utilized if it is on a network. However, developing or working with a spooler requires certain precautions.

A true spooler is a separate software program that manages all communication and processing for one or more printing devices. Background printing is very similar in effect, but does not have the same requirements for processing because it works on the same computer as your software and therefore can intercept the printing process before any communications links are set up. A spooling package that runs on a remote device, however, must insert itself into the AppleTalk network to intercept, process, and forward printing information from the LaserWriter driver to the printer itself. A spooler, therefore, must be careful to maintain all the responses and other communications packets that are expected by both the device and the driver.

In that context, a spooler must be prepared to handle three main issues:

- Determining the status of the Laser Prep dictionary
- Obtaining and maintaining the list of current fonts
- Finding and downloading the fonts required for each document, if these are not provided in the document itself

Before each print job, the LaserWriter driver sends two queries to the printer: one to determine the current status and version of the Laser Prep file that is available on the printer, and one to determine the current set of fonts. Therefore, the spooler software must be aware of and be able to respond to these requests.

One of the main issues in managing fonts, as you have already read, is in determining when a font needs to be coordinated. All fonts used in a QuickDraw document need to be coordinated, and this task is generally handled by the driver. For a spooler, this task can be quite onerous. There are two distinct and somewhat different approaches to coordinating fonts.

One approach is for the spooler to undertake all the coordination and font downloading tasks itself. In this mode, the spooler always returns a list of coordinated fonts when the driver sends a query. With that response, the driver never coordinates a font unless that font is not on the font list and is therefore included in the document itself. In that case, the driver provides both the font definition and the coordination function.

The second approach is for the spooler to insert itself into the AppleTalk network in place of the printer that it manages, by impersonating the printer. To do this, it must change the AppleTalk name of the printer, remove that name from the list of available printers, and then assume the original device name. In this way, all requests that would have been directed to the printer are now directed to the spooler, which alone communicates with the actual device. This approach is probably the best choice because it is both easier for the spooler and more secure from network and system changes. In this mode, the spooler becomes transparent to the font coordination and downloading process, but it handles the appropriate queries by either answering itself, if it can, or passing the query on to the printer and then forwarding the reply.

Perhaps a simple example can make this a bit clearer. In this approach, the spooler waits for the first font query, for example. When it receives the query, it has no information itself, so it passes the query on to the printer. When the printer replies, the spooler both forwards the response to the original questioner and saves the font list itself. When the workstation that originated the query sends fonts, the spooler updates its list and forwards the fonts to the printer. In this way, if the fonts required coordination, that coordination will be done by the workstation driver. The only thing that the spooler needs to do is to keep track of the fonts on the printer and their status. It can augment its current list with periodic queries to the printer to get a current list of fonts—remember that the font name determines whether it is coordinated or not. From this point on, the spooler returns its internal list of fonts to any font queries, thus speeding up the processing.

Note ▸

The spooler could also appear on the network as a separate device. In that case, it must conform to the Chooser device requirements so that it can be seen and used.

The only danger in the second approach is that the spooler font list might not accurately reflect the actual set of fonts on the printer. This is least likely to happen if the spooler has deleted the printer from the network, so that the printer cannot receive any communications unless they come through the spooler. If that is not possible, the network administrator should ensure that there is no way for additional fonts to be loaded onto the printer outside of the server loop.

▶ Conclusion

This chapter has been primarily a review of the basics of font processing, both on the Macintosh and on the LaserWriter. The chapter began with a general discussion of fonts and font types, and drew your attention to the differences between PostScript fonts and Macintosh fonts—a difference that is rapidly shrinking. Then you looked at the essential matters of how to process fonts. This is primarily concerned with how fonts are named, how to find the font's name, and how to relate names from one environment to those in the other. All this leads quite naturally into the final section of the chapter, which discusses how to manage fonts on the LaserWriter. This requires understanding not only what resources fonts use, but also how you can identify which fonts are now loaded into the printer and what the requirements are in the Macintosh environment for accurate font information. Since such information can best be obtained directly from the printer itself, this leads quite naturally into the next and final chapter of this book, which tells you how to communicate directly with the LaserWriter.

7 ▶ Advanced LaserWriter Operations

▶ Chapter Overview

You have come quite a long way from the start of the book. You have learned about and, more importantly, practiced writing code to access the LaserWriter in a variety of ways. As you have seen, you can use the facilities of the Macintosh Toolbox and the operating system to get quite a bit of utility out of the LaserWriter if you understand exactly what is going on during the processing and how to intervene correctly into that process.

I think that all programmers must be naturally curious, and from that curiosity arises the desire to press on into the next level (whatever that may be) of control of the device. If this were not so, no one would ever write in assembly language, write device drivers, or hand-code graphics in PostScript. In programming the LaserWriter, you have still left one area to the management of the system software: communicating directly with the LaserWriter. So far, all of this has been left in the capable hands of the LaserWriter driver, which has processed all the actual communications that needed to be done. In this chapter, you remove the driver from the process and manage the communications yourself.

Like many things in life, taking control of this process has better and worse consequences for you. On the worse side, it involves a lot of code that you must both create and maintain, and it also requires a lot more attention and management of the printer. On the plus side, however, it allows you to carry on a dialog with the printer and to make full use of the intelligence that is an integral part of every PostScript device. Of course, none of that would be possible if you were not prepared to write the PostScript code required to provide the other half of the

dialog from the printer side. This is why you have been working through all the material up to this point: so that you would be ready to carry on both sides of this dialog.

This chapter, then, concentrates on how you can communicate directly with the LaserWriter. Moreover, there are two primary communications links that you have with the LaserWriter: serial communication and AppleTalk communication. Of these, the serial communication option is both less common and, perhaps paradoxically, better documented. For that reason, in this chapter you will work primarily with the AppleTalk link.

▶ Communication Processing

The LaserWriter is a computer in itself, not simply a mechanical peripheral. It has both the intelligence and the requirement to participate in a dialog with other devices in order to function properly. To do that, the LaserWriter has two methods of communications that it supports: a standard serial interface and the AppleTalk interface. Of these two, the AppleTalk interface allows the LaserWriter to be a full participant in the AppleTalk network, and so allows many other devices to share the LaserWriter. Naturally this is the most common connection method used for a LaserWriter in most circumstances. The serial communication channel, on the other hand, is a connection to a single processor and does not allow direct sharing of the device. The serial connection is most typically used either when the LaserWriter is connected to a device that does not support the AppleTalk protocol or when you want to have an interactive session with the PostScript interpreter. In either case, the printer can no longer be shared with other users.

Many other types of laser printers provide one additional communications link: a parallel printer connection. Rightly or wrongly, the LaserWriter does not support this. Although there may be nontechnical reasons to provide such a channel, from a technical viewpoint it is neither required nor desirable. The parallel port has two putative benefits and one major drawback. The benefits are supposed to be speed and ease of connection. The drawback, however, is quite large and, from a user's viewpoint, a most serious one. Since the parallel channel is generally unidirectional, there is no method of obtaining error or status information from the printer. Since the LaserWriter, like all PostScript printers, has a powerful processor, it can provide useful, and sometimes vital, information to the user about the current job and job requirements. This can range, as you have seen, from simple error reporting to more complex dialogs about what the current fonts are, for example. All this is lost when the printer is connected by a parallel channel.

The supposed benefits also prove to be rather illusory on closer examination. The LaserWriter supports communications over a serial channel up to 57,600 baud; most host computers cannot transmit data that fast. At any speed above 19,200 baud, in my experience, the transmission of data on the serial channel is comparable to or exceeds that on the parallel port, and it preserves the ability to get information back from the printer. As to the ease of connection, there is certainly something to that because handling and setting up the RS-232 (or RS-422 if that is available) port can be quite intimidating for the untutored. Once set up, however, this link requires no more or less care than a parallel connection, and the setup itself can easily be handled by software. For these reasons, no serious technician should be satisfied with a parallel connection to any PostScript device. It is, in my opinion, a benefit, not a drawback, that such a half-brained (since you are sacrificing half of the intelligence of your PostScript processor) approach is not available on the LaserWriter.

▶ AppleTalk functions and processing

AppleTalk is the communications protocol that is used between the LaserWriter printer and the Macintosh computer. Although a real-world network may consist of many devices, including both multiple printers and multiple workstations, for purposes of discussion you only need to consider a network of two: one LaserWriter and one Macintosh. This simple network is not so unrealistic as it might appear, for two reasons. First, many real networks consist of no more than this configuration. Second, even when there are more devices, the effective communication is between two devices, just like this.

Each device on a network is known as a *node*. A node can either be a *workstation node*, such as the Macintosh, or a *server node*, such as the LaserWriter. The software that runs inside each node opens up a *socket*, which is the connection to the network by which the node sends and receives information. The software process that starts and controls the socket is called the socket's *client* and is said to *own* the socket. The socket client must provide code to monitor and respond to socket requests, called the *socket listener*. A socket client is also known as a *network-visible entity (NVE)* because it can be addressed on the network. Notice that a node is not, by itself, an NVE since it is the socket client that provides the network services; moreover, a node might contain more than one client process.

The overall requirements of the AppleTalk network are quite complex, and you can read about them in the *Inside AppleTalk* manual if

you need or want that information. For communications with the LaserWriter, only two parts of AppleTalk are used: the Name Binding Protocol and the Printer Access Protocol, abbreviated PAP. All of this was covered in some detail in Chapter 1. This is the structure that you will be working with for most of this chapter, and, as such, it is described in much more detail in the next section.

▶ Serial communications

The LaserWriter also supports serial communications through the 25-pin RS-232 port. You connect this just like any serial port. The owner's guide for the printer model(s) that you have should give full information about issues such as pin setup, communications parameters, and so on. Each model of the LaserWriter has its own method of setting these—usually either by setting dip switches or by setting a rotary switch on the device. These parameters can also be set by PostScript commands sent to the printer. When that is done, the choices made by the software will override the switches and will persist even after the job is completed. If you set the parameters in that way, any change in the switch settings resets them to the values determined by the switch setting and the software selections are reset. This can be a great boon if you have made some incorrect change and need to get back to a known state.

If you are using serial communications, you must set all the communications parameters. Generally this is easily done from software in the host computer. The single option that is often not easily set is the handshake that needs to be coordinated between the printer and the host. The handshake is essentially a method for one party in the communi-cations link to notify the second party that the second party must either stop sending data or that it may start sending data. This is required so that the data can be processed correctly at the receiving end.

There are two standard handshake methods: hardware and software. The LaserWriter supports either one of these methods. The hardware method, usually referred to as DTR/DSR, uses particular pins on the connecting wire to signal between the parties. On a standard RS-232 cable, this is done using pin 6 (DSR) and pin 20 (DTR). To use this method, you must be sure to connect these two pins on each end of the cable. The software method, usually called XON/XOFF, uses special characters to perform the same signalling function: 0x11 for XON and 0x13 for XOFF. Both methods require agreement between the two parties. You must determine which of these methods your host uses and then set the LaserWriter to use the same one. The LaserWriter default is to use the XON/XOFF method, and most software systems support this type of

handshake. If your software does not, you must reset the LaserWriter to use DTR/DSR and ensure that the cable is correctly connected to both ends. Complete instructions for setting these parameters are contained in your owner's guide, and are also covered in the *Apple LaserWriter Reference* manual, described in the Bibliography.

▶ Emulation modes

Emulation modes are related to, although distinct from, specific communications options. The LaserWriter printers all support some form of emulation, which allows them to behave like another printer. The original LaserWriter, the LaserWriter Plus, and the LaserWriter IINT can emulate the popular Diablo 630 daisy-wheel printer. The LaserWriter IINTX and the Personal LaserWriter NT also provide emulation for the Hewlett-Packard LaserJet+, a popular non-PostScript laser printer. These modes are primarily useful with software or systems that do not support PostScript directly, but that do support one or another of these printers.

All emulation is conducted through the 25-pin serial connection. The emulation mode can be selected by settings of the control switches of the LaserWriter or by software commands. As is true for all the communications parameters, these options are fully described in the references given earlier.

If you are interested in how these emulations work, an excellent discussion of printer emulation in general appears in *PostScript Language Program Design*. However, since the focus here is on programming the LaserWriter from the Macintosh, these modes are not discussed any further here. Instead, the remainder of the chapter concentrates on AppleTalk communications exclusively.

▶ Printer Access Protocol

The Printer Access Protocol (PAP) sets up, maintains, and closes connections between workstations, such as the Macintosh computer, and printers, such as the LaserWriter. Although it is called the *Printer* Access Protocol, there is nothing in it that specifically relates to printing; it is named that way simply because it was designed to provide a link between workstations and printers, or print servers. PAP calls can be used for other types of server communication as well, if required. PAP is a *session-level* protocol, which means that it functions on the fifth layer of the standard network framework. For our purposes, this simply means that it uses several other AppleTalk layers to carry out its requirements. A complete description of layers and of how PAP uses the other AppleTalk layers is contained in the *Inside AppleTalk* manual.

PAP performs four basic functions. Upon request, it sets up a connection between two clients. It sends and receives data. It monitors the connection to ensure that it remains in working condition, and it closes the connection when the data transfer is complete. PAP, in itself, can process multiple connections at any given time. However, the LaserWriter itself only processes a single job at a time, so that on the network only one connection will be open at a time between a specific LaserWriter NVE and a specific Macintosh NVE.

At this point, a little terminology is in order. Dealing with communications, whether between the printer and a Macintosh or between a central computer and a remote site, always raises the question of who is in charge; you may have recognized this problem when you attempt to "upload" or "download" information between two Macintosh systems. Here, the LaserWriter is providing printing services to the network, while the Macintosh is using those services. In this case, the LaserWriter is called the *server* and becomes an NVE by opening its socket and registering its name on the network. Every socket that uses PAP calls is known as a *PAP client*. The LaserWriter therefore becomes the *server PAP client*, while the Macintosh—or, more accurately, the code within the Macintosh system that is using the printing services—is known as the *workstation PAP client*. When the LaserWriter driver is managing the connection to the LaserWriter, it is the workstation PAP client; now that your application will be performing these tasks, it becomes the workstation PAP client.

The tasks performed by the two clients are similar but not identical; this means that PAP is not a symmetric protocol. The server has calls to support functions that are not used at the workstation, and vice versa. In particular, the server has calls to register its name on the network and to remove or change its name if required. Typically, a LaserWriter print job follows the cycle shown in Figure 7-1.

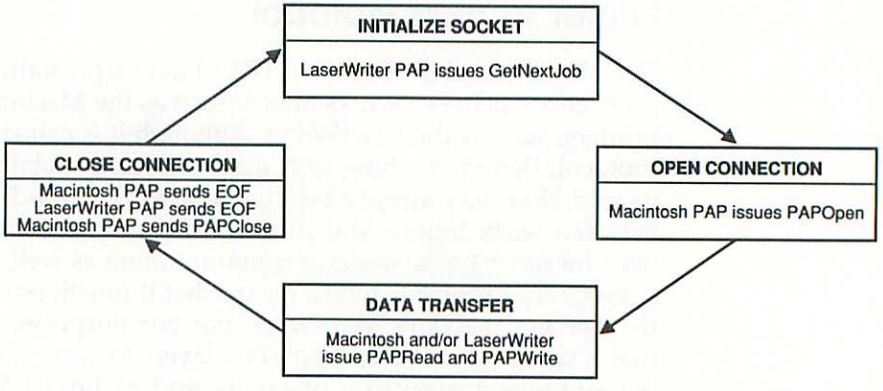

Figure 7-1. Typical PAP print job cycle

Once the server has successfully opened its PAP socket, it announces its name on the network and waits for a job addressed to it. At this point, the workstation can request a connection and begin a data transfer. When the transfer is complete, the workstation terminates the transfer and both ends close the connection. When the server is finished processing the job, it then returns to the ready state, waiting for the next job. A workstation requests a connection with a specific server by issuing a PAPOpen call for that server. If there are multiple requests for the next job, the server will *arbitrate* the requests and take the one that has been pending the longest. Once the request is selected, the connection is opened between the two clients and the process begins again.

After a connection is made, the data transfer begins. PAP performs two functions during this process. First, it uses the lower layers of the AppleTalk protocol to transfer the actual data. Second, it monitors the communications link to detect and correct half-open connections.

A *half-open connection* is one that occurs when one of the clients goes down or terminates processing without informing the other client. PAP must detect these conditions and close the remaining client in an orderly manner. For this purpose, PAP maintains a connection timer at each client, which is normally set for 2 minutes. Any packet of information received at a client resets the timer in that client. If the timer expires without being reset, PAP assumes that the other client has become inactive and closes the channel. For this reason, PAP clients send periodic *tickler* packets when no other work is being processed. This also explains why the system must ensure that an application is given enough processing time to maintain the connection; otherwise, the server PAP client automatically terminates the connection.

PAP uses a *read-driven* model for data transfer. This means that, before data can be sent over an established connection, the other end of the connection must send a transaction request indicating that it is ready to receive data. The PAP client at either end can issue such a request by executing a PAPRead call. Once the PAPRead is issued, the other client can then send data using PAPWrite calls.

Normally, data transfer continues in this fashion until the workstation PAP client is finished sending data and notifies the server client that the transfer is complete by sending an EOF indication. At this point, the server returns an EOF to confirm that the reception is complete, and the workstation PAP client closes the connection with a PAPClose and terminates the link with a PAPUnload. The termination process cancels any outstanding connection requests, including any tickler packets that might still be pending.

If at any point the server PAP client determines that the connection is half-open, then the server client will terminate the connection unilaterally by issuing a PAPClose itself.

The workstation PAP client can also inquire about the current status of a given server as long as the server client is active on the network; that is, if the server has an active NVE. This *status check* does not require that a connection be established before being issued. To do this, the server PAP client provides a status string to its PAP routines when it opens and updates that string periodically as it is processing. The workstation PAP client can then issue a PAPStatus call for the server. When the server PAP receives that call, it returns the status string that it has already stored. Notice that the server does not actually process the request directly.

▶ PAP call definitions

To use these routines, you need to know what each one does and how to call it. The definitions here present only the workstation calls; the full definitions of both workstation and server calls (and much additional information) are provided in *Inside AppleTalk* in the chapter on the Printer Access Protocol.

PAPOpen

The workstation PAP client issues a PAPOpen call to initiate a connection with a specified server. The call requires the following parameters.

- Server specification by either the network name of the server or the server's session listening socket address.
- Flow quantum, which is the number of 512-byte buffers that are allocated for read data; the LaserWriter uses a flow quantum of 8.
- Buffer address where the open status string is to be returned.

The call returns the following parameters.

- The connection reference number, also called the connection id. This is a unique number assigned by the workstation PAP routines to identify this connection.
- A result code, which may have the following values:

0	no error
0xFFFF	printer busy

If the result code indicates that the server is busy, the workstation PAP client will continually retry the connection at approximately 5-second intervals. It is up to the application to terminate the retry process if it has not been successful within some specified time; the PAP routines will retry indefinitely.

PAPRead

Either client may issue a PAPRead when it is ready to receive data from the other client on the connection reference number. Notice that a PAPRead does not actually retrieve data; it simply sets the client up to receive data when the other client decides to send. The call requires the following parameters.

- The connection reference number.
- The address of the buffer to receive the data. This buffer must not be less than the size implied by the flow quantum that was set in the PAPOpen.

The call returns the following parameters.

- The size of the data read.
- The result code for the read process.
- An indication if this is an EOF.

PAPWrite

Either client may issue a PAPWrite to send data to the other client on the connection specified by the connection reference number. The call requires the following parameters.

- The connection reference number.
- The address of the buffer where the data to be sent is stored.
- The size of the data to be sent. This size must not be larger than the size implied by the flow quantum that was set in the PAPOpen; if it is, the call returns with an error.
- An indication if this is an EOF.

The call returns the following parameter.

- The result code for the write process.

When you issue the PAPWrite call, no data is necessarily transferred at that time. If the server client has not issued a PAPRead, then the data cannot be transmitted and the workstation client queues the data until the server issues the next PAPRead.

PAPClose

Either client may issue a PAPClose to terminate processing on the connection indicated by the connection reference number. Normally this is done by the workstation client when it is done sending data to the server. The call requires the following parameter.

- The connection reference number.

The call returns the following parameter.

- The result code for the close process.

PAPUnload

The PAPUnload call is not a documented PAP call, in that it is not listed in *Inside AppleTalk*. Nevertheless, it is required to finish tearing down the PAP communication process. At this point, the connection has been terminated, so there is no longer any specific connection reference and there are no parameters for the call. The call does return a result code, which indicates an error if one occurs.

PAPStatus

The workstation PAP server issues the PAPStatus call to determine the current status of the given server. This call can be used at any time, whether a connection to the server is open or not. Upon completion, the call returns a Pascal string that contains the server status response. The call requires the following parameters.

- Server specification by either the network name of the server or the server's session listening socket address.
- Buffer address where the status string is to be returned.

The call returns the following parameter.

- The result code for the status query.

Each of the three calls, PAPOpen, PAPRead, and PAPWrite, also has a status variable associated with it. This status variable has the following values.

Value	*Meaning*
<0	An error has occurred in the process.
0	Process is terminated without an error.
>0	Process is still working.

This status variable is used to determine the current state of the process in your code, as you will see in the glue routines in the next section.

▶ PAP glue routines

The PAP calls that you have just read about are not part of the Macintosh Toolbox. Instead, the PAP code is a part of the device driver code resources that are maintained as a 'PDEF' resource in the driver itself. The PAP code in Apple drivers is in 'PDEF' number 10. Since the code is in a code resource, you must access that code through assembly language "glue" routines, similar to the ones used for linking to the Printing Manager itself.

Note ▸

Obviously, when you are accessing system resources of any type, each new release of the system may move those resources. For most of the code in this book, however, the resources that you have been relying on are well documented and, in some cases, are publicly defined. In this case, however, you are using a relatively undocumented resource, and one that Apple defines as being simply a convention, rather than a requirement for drivers. Therefore, when using this technique, it is important that you check that the 'PDEF' code actually exists. In the code here, this is done with the Think C standard CheckResource function. In a commercial application, this would be a rather rude way to inform the user that you have lost the location of the PAP code; a better method would be an alert dialog that requests the user to locate a working copy of the LaserWriter driver.

These glue routines do not actually do much processing themselves. They simply take the parameters that are passed to them, place the required information onto the stack, and then branch to the location in the PAP code where the desired function is executed. The parameters in the code here are maintained as global variables rather than being passed as calling parameters; this is fairly language independent and also avoids the problems of parameter sequence and retrieval. This glue code is presented in Listing 7-2, and the associated header information is shown in Listing 7-1. The header consists of only the typedef for the status return and the prototypes of the glue functions.

Listing 7-1. Header for PAP glue routines, PAPGlue.h

```
#define _H_PAPGlue

/*********   program defined structures  *********/
typedef struct
{
    long int    systemstuff;
    char        statusstr[256];
 } papStatusRec;


/* PAP routines          */
OSErr LW_PapOpen (void);
OSErr LW_PapRead (void);
OSErr LW_PapStatus (void);
OSErr LW_PapWrite (void);
OSErr LW_PapClose (void);
OSErr LW_PapUnload (void);
```

Listing 7-2. PAP Glue routines, PAPGlue.c

```
/*****************************************************************
 * PAPGlue.c
 *
 *                        The Printer Access Protocol interface
 *
 *
 *    Programming the LaserWriter - Addison-Wesley
```

Listing 7-2. PAP Glue routines, PAPGlue.c (continued)

```
 *    by David Holzgang
 *    © Copyright 1990 David Holzgang All Rights Reserved
 *
*************************************************************************/

#include <stdio.h>
#include "PAPGlue.h"

/************************  Globals   ***************************/
int               gConnectId;
int               gFlowQuantum;
char              *gLaserName;
long unsigned     gLaserNode;
papStatusRec      gLaserStatus;
int               gOpenState;
long              gPapAddress;
char              gReadBuffer[4096];
int               gReadEOF;
int               gReadSize;
int               gReadState;
char              *gWriteBuffer;
int               gWriteEOF;
int               gWriteSize;
int               gWriteState;


/*************************************************************************
 *  PAP glue routines
 *
 *    Routines start with a SUB command that makes room for the
 *    result on the stack.  Then they push the appropriate PAP globals
 *    onto the stack and jump to the routine's entry point, which is at
 *    some small offset from the beginning of PAP.  When they return
 *    from the call, they pop the result off the stack and put it into
 *    retval, which is returned.
 ***/
```

Listing 7-2. PAP Glue routines, PAPGlue.c (continued)

```
/* PAP routines */
LW_PapOpen()
{
      int      retval;

      asm
      {
          SUBQ.L   #2,A7
          PEA      gConnectId;
          MOVE.L   gLaserName,-(A7)
          MOVE.W   gFlowQuantum,-(A7)
          PEA      gLaserStatus
          PEA      gOpenState
          MOVE.L   gPapAddress,A0
          JSR      (A0)
          MOVE.W   (A7)+,retval
      }
      return( retval );
}

LW_PapRead()
{
      int      retval;

      asm
      {
          SUBQ.L   #2,A7
          MOVE.W   gConnectId,-(A7)
          PEA      gReadBuffer
          PEA      gReadSize
          PEA      gReadEOF
          PEA      gReadState
          MOVE.L   gPapAddress,A0
          JSR      4(A0)
```

Listing 7-2. PAP Glue routines, PAPGlue.c (continued)

```
            MOVE.W   (A7)+,retval
        }
        return( retval);
}

LW_PapWrite()
{
        int      retval;

        asm
        {
            SUBQ.L   #2,A7
            MOVE.W   gConnectId,-(A7)
            MOVE.L   gWriteBuffer,-(A7)
            MOVE.W   gWriteSize,-(A7)
            MOVE.W   gWriteEOF,-(A7)
            PEA      gWriteState
            MOVE.L   gPapAddress,A0
            JSR      8(A0)
            MOVE.W   (A7)+,retval
        }
        return( retval );
}

LW_PapStatus()
{
        int      retval;

        asm
        {
            SUBQ.L   #2,A7
            MOVE.L   gLaserName,-(A7)
            PEA      gLaserStatus
            PEA      gLaserNode
            MOVE.L   gPapAddress,A0
            JSR      12(A0)
            MOVE.W   (A7)+,retval
        }
        return( retval );
}
```

Listing 7-2. PAP Glue routines, PAPGlue.c (continued)

```
LW_PapClose()
{
      int      retval;

      asm
      {
          SUBQ.L   #2,A7;
          MOVE.W   gConnectId,-(A7)
          MOVE.L   gPapAddress,A0
          JSR      16(A0)
          MOVE.W   (A7)+,retval
      }
      return( retval );
}

LW_PapUnload()
{
      int      retval;

      asm
      {
          SUBQ.L   #2,A7
          MOVE.L   gPapAddress,A0
          JSR      20(A0)
          MOVE.W   (A7)+,retval
      }
      return( retval );
}
```

The code routines all start with a SUB command that makes room for the result on the stack. Then they place the required parameters, as defined in the calls and provided in the global values, onto the stack. Finally, they branch to the offset in the PAP routine that defines the entry point for the given call. When they return, they pop the result code off of the stack and return it.

▶ Using PAP Calls

With all that theory, it's time to have a little practice. This section introduces you to the use of the PAP calls in your exercise code and shows you how to display the status of the LaserWriter.

To begin with, add PAPGlue.c to your project file. I placed mine in the second project segment, where the TBUtilities.c and GlobalVar.c modules are. Compile the glue routines. This gives you the basic routines for accessing and using the PAP protocols.

Next you have to write some code to use these routines. For the first example, you simply add the code to get the current device status and display it. This query is initiated from a Printer Status menu item that is placed in the Special menu.

You can begin by getting some housekeeping out of the way. First, you need to add the Printer Status item to the Special menu; and, if you have not added a Special menu, you should also do that now. This is quite straightforward; use the techniques discussed in the *Think C User's Manual*. Make the Printer Status item menu selection 1048 if you want to use the exact code shown here; otherwise, you must change the cmdPrinterStatus constant in CSimpleLWDoc.h to reflect the number that you have chosen.

Next you must modify two sections in CSimpleLWDoc.c: the **UpdateMenus** method and the **DoCommand** method. The changes required in these methods are shown in Listing 7-3.

Listing 7-3. Revisions to methods in CSimpleLWDoc.c

```
/**********************************************************************
 * DoCommand
 *
 *  This is the heart of your document.
 *  In this method, you handle all the commands your document
 *  deals with.
 *
 *  Be sure to call the default method to handle the standard
 *  document commands: cmdClose, cmdSave, cmdSaveAs, cmdRevert,
 *  cmdPageSetup, cmdPrint, and cmdUndo. To change the way these
 *  commands are handled, override the appropriate methods instead
 *  of handling them here.
 *
 ***/

void WSimpleLWDoc::DoCommand(long theCommand)
```

Listing 7-3. Revisions to methods in CSimpleLWDoc.c (continued)

```
{
      switch (theCommand) {
          /* your document commands here */
          case cmdPtrStatus:
              myPrinter->GetDeviceStatus();
              break;

          default:      inherited::DoCommand(theCommand);
                        break;
      }
}


/**********************************************************************
 * UpdateMenus
 *
 *    Perform menu management tasks
 *    In this case, enable Special Menu items as appropriate
 *
**********************************************************************/

void  WSimpleLWDoc::UpdateMenus()
{
      inherited::UpdateMenus();

      if (isLaserWriter)
      {
          gBartender->EnableCmd(cmdPtrStatus);
      }
}
```

In the **UpdateMenus** method, you simply have to enable the menu command, which is represented by the global variable, cmdPtrStatus. In **DoCommand**, you have to add a new case for the cmdPtrStatus, which only sends a message **GetDeviceStatus** to the printer object, CLaser, that you established when the document was initialized. You also have to update the header file CSimpleLWDoc.h as shown in Listing 7-4.

Listing 7-4. Revisions to CSimpleLWDoc.h

```
/************************************************************************
 * CSimpleLWDoc.h
 *
 *  Document class header for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
************************************************************************/

#define   _H_CSimpleLWDoc          /* Include this file only once */

#include <Global.h>
#include <Commands.h>
#include <CApplication.h>
#include <CBartender.h>
#include <CDataFile.h>
#include <CDesktop.h>
#include <CDecorator.h>
#include <CDesktop.h>
#include <CDocument.h>
#include <CError.h>
#include <CPanorama.h>
#include <CScrollPane.h>
#include <TBUtilities.h>

#define BASE_RES_ID   400

#define BASE_WINDOW   BASE_RES_ID /* Resource ID for WIND template */
#define BASE_PANE     BASE_RES_ID /* Resource ID for ScPn template */

#define cmdPtrStatus      1048    /* Printer Status in Special Menu*/
```

Listing 7-4. Revisions to CSimpleLWDoc.h (continued)

```
struct CSimpleLWDoc : CDocument {
                        /** Instance Variables **/
      struct CLaser    *myPrinter;
      Boolean          isLaserWriter;
      Handle           lwFileDataH;

                                /** Class Methods **/
                                /** Construction/
                                Destruction **/
      void         ISimpleLWDoc(CBureaucrat *aSupervisor, Boolean
                   printable);
      void         Dispose();

                                    /** Action/
                                    Deactivation **/
      void         DoCommand(long theCommand);

      void         Activate(void);
      void         Deactivate(void);

      void         UpdateMenus(void);

      void         NewFile(void);
      void         OpenFile(SFReply *macSFReply);
      void         BuildWindow(Handle theData);

      Boolean      TestDraftStatus(void);

                                    /** Filing **/
      Boolean      DoSave(void);
      Boolean      DoSaveAs(SFReply *macSFReply);
      void         DoRevert(void);
};
```

This sets the stage for the real work, which, as usual, occurs in the CLaser.c module. In this module you need to add a new method, **GetDeviceStatus**, which is shown in Listing 7-5.

Listing 7-5. New method, **GetDeviceStatus**, in CLaser.c module

```
/**********************************************************************
 GetDeviceStatus
 *
 *      Returns current status of chosen Device
 *          and displays it in a dialog window.
 *
 *
 ********/

void  CLaser::GetDeviceStatus( void )
{
      DialogPtr    feedbackDialog;
      Str255       messageStr;
      short        itemHit;
      Boolean      dialogDone       = FALSE;

      SetCursor(*gWatchCursor);
      Cur_Prntr();

      feedbackDialog = GetNewDialog(STATUS_DITL, NIL_POINTER,
          MOVE_TO_FRONT);
      ShowWindow(feedbackDialog);

      /* put up name of printer to query  */
      ParamText(gLaserName, NIL_STRING, NIL_STRING, NIL_STRING);
      DrawDialog(feedbackDialog);

      LW_PapStatus();
      if (gLaserStatus.statusstr[0]<1)
      {
          GetIndString(messageStr, STR_STATUS, NO_RESP);
          ParamText(gLaserName, messageStr, NIL_STRING, NIL_STRING);
          SysBeep(10);
          DrawDialog(feedbackDialog);
      }
      else
      {
          ParamText(gLaserName, gLaserStatus.statusstr, NIL_STRING,
          NIL_STRING);
          DrawDialog(feedbackDialog);
      }
```

Listing 7-5. New method, **GetDeviceStatus**, in CLaser.c module (continued)

```
    SetCursor(&arrow);
    while ( dialogDone==FALSE )
    {
        ModalDialog(NIL_POINTER, &itemHit);
        switch ( itemHit )
        {
            case  QUERY_OK :
                    DisposDialog(feedbackDialog);
                    dialogDone=TRUE;
                    break;
        }
    }

    return;

}
```

This new method requires no information and returns none. It gets the current status of the device and displays it in a simple status dialog box, which you will define. The method begins by setting the cursor to the watch cursor, and then uses the familiar **Cur_Prntr** method to identify the current printing device. Notice that you need to do this just before you do the status inquiry so that you have the most recent and valid information regarding the current device. To set up the connection for the PAP handling, you must add the code shown in Listing 7-6 to **Cur_Prntr**.

Listing 7-6. Revised code for **Cur_Prntr**

```
laserNameH = GetResource( 'PAPA', -8192 );
    CheckResource( laserNameH );
HLock( laserNameH );
itsLaserName = *laserNameH;
gLaserName = *laserNameH;
HUnlock( laserNameH );
DetachResource( laserNameH );

papDriverH = GetResource( 'PDEF', 10 );
    CheckResource( papDriverH );
HLock( papDriverH );
gAddress = (long)*papDriverH;
DetachResource( papDriverH );

CloseResFile( filNum );
```

This revised code now sets some global variables, which are shared with the PAP glue routines. These are used by the glue routines as parameters to the PAP calls. There are two changes in this code from the previous code. First, you have to set the global variable gLaserName to the name of the printer. Second, you establish the address of the PAP routines in 'PDEF' 10, as described earlier, as the global variable gPapAddress.

Once the current values of the required global variables are set, you can continue with the code in **GetDeviceStatus**. The next task is to display a dialog window with space for two variable strings: the name of the LaserWriter that you are querying and the current status. Then you show the dialog window, place the initial string into it, and draw it.

Here is the highlight of the operation: now you call PAPStatus. This returns the status value in the global variable gLaserStatus, which has the format that you defined in Listing 7-1 as a papStatusRec. Out of this, you test the return string for a valid length. If the string has no valid length, then you did not get a satisfactory status response. The dialog is then set to display a standard message, derived from a 'STR#' resource, which informs the user that there was no response. You beep the system and redraw the dialog with the new message. If there is a valid response, then you display that response and draw the dialog.

In either case, you restore the cursor to the standard arrow and begin the ModalDialog test for a response. In this case, the only possible response is to hit the GotIt button. This is primarily designed as a convenience for the user so that the message doesn't disappear so fast that he or she doesn't have time to see it—one of the most frustrating occurrences, in my experience. When the user clicks on the box, you terminate processing and release the dialog.

Before you leave CLaser.c, you need to add the required global variables and defined constants to the beginning of the file. These are shown in Listing 7-7. You must also remember to update the header file CLaser.h, as shown in Listing 7-8.

Listing 7-7. Global variables and constants for CLaser.c

```
#include "CError.h"
#include "CPrinter.h"
#include "CLaser.h"
#include "PAPGlue.h"

#define CURRENT_VERSION     1
#define LAS_NOT_AVAL        415
#define DRAFT_DITL          401
#define STATUS_DITL         405
#define STR_STATUS          400
#define NO_RESP             1
#define MOVE_TO_FRONT       -1L
#define NIL_STRING          "\p"
#define NIL_POINTER         0L
#define QUERY_OK            1

extern  CError          *gError;        /* Error handler */
extern  CursHandle      gWatchCursor;   /* Watch cursor
                                           for waiting  */
extern  char            *gLaserName;
extern  long            gPapAddress;
extern  papStatusRec    gLaserStatus;

static  Boolean         gMarkDraft;   /* Draft Box checked
                                         (or not)    */
static  TPPrDlg         gPrtJobDialog; /* Job Dialog
                                          pointer     */
static  int             prFirstItem;  /* Box item number */
static  ProcPtr         prPItemProc;  /* Default Job
                                         Dialog Item handler*/
```

Listing 7-8. Updated header file CLaser.h

```
/**********************************************************************
 * CLaser.h
 *
 *        Interface for the LaserWriter Printer Class
 *
 *    Programming the LaserWriter - Addison-Wesley
 *    by David Holzgang
 *    © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#define _H_CLaser

#include "CPrinter.h"                /* Interface for its superclass */

struct CLaser : CPrinter {           /* Class Declaration            */
                                     /** Instance Variables **/
      char                *itsLaserName;
      char                gUserName[32];

                                     /** Instance Methods    **/
                                         /** Contruction/Destruction **/
      Boolean     ILaser(CDocument *aDocument, THPrint aMacTPrint);

                                         /** Accessing   **/
      void        GetDeviceInfo(short *devNum);
      void        GetDeviceStatus( void );
      Boolean     GetDraftStatus( void );

                                         /** Printing **/
      void        DoPrint(void);

                                         /** Special **/
      void        Cur_Prntr( void );


      Boolean     MyJobDialog(THPrint aMacTPrint);

};
```

Listing 7-8. Updated header file CLaser.h (continued)

```
/*    these are aux fcns that are NOT methods                    */
      pascal  TPPrDlg     MyJobDlgInit (THPrint aMacTPrint);
      pascal  void        MyJobItems(TPPrDlg theDialog, int itemNo);
              int         AppendDITL(DialogPtr theDialog, int
                              theDITL);
```

As usual, this requires some additional resources in the resource file: an additional string, a new dialog box display, and some new items to go in the box. Listing 7-9 shows you the RMaker file DLOG.make.r in the usual way.

Listing 7-9. DLOG.make.r file for creating new resources

```
!SimpleLW.π.rsrc

* added resource declarations follow ...


* place next resource
TYPE DLOG
Status,405      ;; New Resource #405
Status            ;; dialog name
60 98 182 415     ;; box rect
Invisible NoGoAway
1
0
405

* place next resource
TYPE DITL
Status Display,405      ;; New Resource #405
3                           ;; three items in list

button
93 128 113 188
Got It
```

Listing 7-9. DLOG.make.r file for creating new resources (continued)

```
staticText
10 25 29 282
Printer: "^0"

staticText
33 25 87 282
^1

* place next resource
TYPE STR#
Comm Status,400          ;; New Resource #400
1
Unable to communicate with printer!
```

That really wasn't too hard, as you see. Now you can compile and execute the project. The display should look something like Figure 7-2.

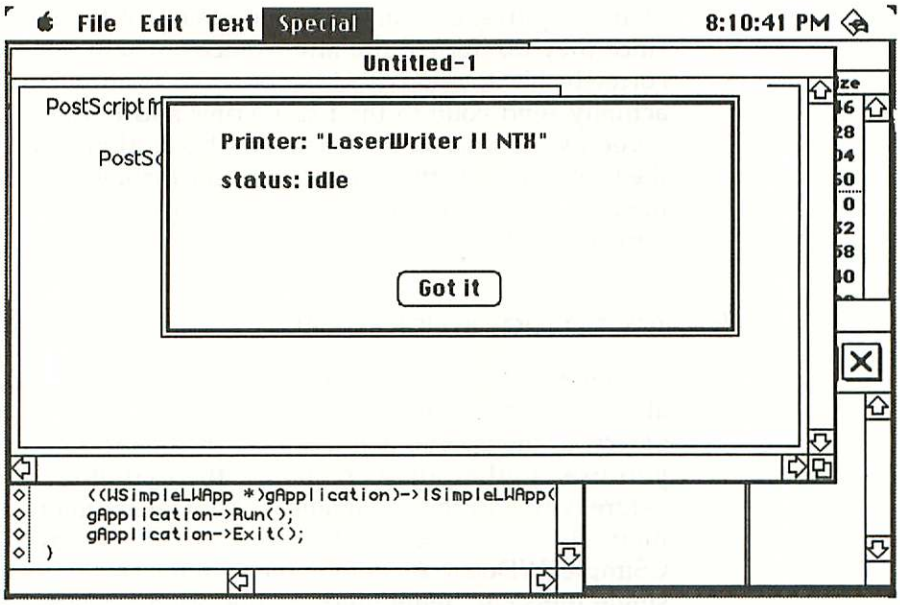

Figure 7-2. Status display for a LaserWriter

You may still be wondering why the command processing is placed into the document code rather than into the application. At first glance, it might seem that you should be able to test the status without a document. Well, you can, but it is much more complicated in this environment. To test the status, you must identify the current device and get the PAP routines' address from the device driver. To do that, you must have a print record. Now, there is nothing to prevent you from creating a print record without any associated document; however, if you think about it for a moment, you will see that this scenario is somewhat unlikely. Generally, if you want to print, you want to print something; and, in the object-oriented world of Think C, that means having a document. Therefore, the easiest way to implement the status inquiry is to open a document, with all that it implies. This is really quite reasonable, since most applications gray out the Print command, and hence will not have a print record, when no document is available. As I said, you can work around this if you want; I leave it to you as an exercise.

▶ Obtaining Device Information

Status inquires, of course, are the easiest of the PAP commands to use since they do not require any connection setup or monitoring to work correctly. Next, you extend the previous example in a simple fashion to actually send code to the LaserWriter and retrieve a response. You do several variations on that theme in this section as you perform many of the standard tests that are required for managing communications with the LaserWriter: testing for device features, testing for the presence of a font, and testing for a Laser Prep version.

▶ Testing feature information

The next example adds new functionality to the modules that you have already created. You begin by adding another new menu item, Printer Query, to the Special menu. I gave this menu selection number 1050; if you use another number, change the definition of cmdPtrQuery. As before, you add this command to the **DoCommand** and **UpdateMenus** methods in CSimpleLWDoc.c and add the cmdPtrQuery constant to CSimpleLWDoc.h. As before, the case for **DoCommand** simply sends a single message, **SendQuery**, to the printer object. I will not reproduce that code here, as you are, by now, so familiar with this process that I don't think these listings serve any real purpose.

You can move on, therefore, to the method that you have identified, **SendQuery**. As before, this goes into CLaser.c, which is becoming a real workhorse in this process. Listing 7-10 shows you the code.

Listing 7-10. Code for **SendQuery** method

```
/*****************************************************
 *  SendQuery
 *
 *   this routine will send whatever is in the resource
 *   'POST' 400 to the LaserWriter
 **
 *****/
void    CLaser::SendQuery (void)
{
    Handle  resHand;
    long    resSiz;
    OSErr   osErr;

    gFeedbackHandle = NewHandle (0L);
    if ( gFeedbackHandle == 0 || MemError() )
        gError->CheckOSError( CANT_CREATE_HAND );
    gFileHandleSize = 0;

    resHand = GetResource ('POST', QUERY_POST);
        CheckResource( resHand );
    HLock(resHand);
    resSiz = SizeResource(resHand);

    SetCursor(*gWatchCursor);
    Cur_Prntr();

    if ( Open_Com_Channel() )
        gError->PostAlert(STR_STATUS, NO_RESP);

    osErr = Send_To_Printer(*resHand, (short)resSiz);
    if (osErr != noERROR)
        End_PAP_Comm();
    else
        Close_Com_Channel();
```

Listing 7-10. Code for **SendQuery** method (continued)

```
    SetCursor(&arrow);

    HUnlock (resHand);
    ReleaseResource (resHand);

    if (gFileHandleSize)
        HandleFeedback();
    DisposHandle (gFeedbackHandle);
    return;
}
```

This code begins by creating a new handle of zero length as a storage point for any reply that comes back from the server. Then you load the 'POST' resource that contains your query code into the application and lock it up. Notice here that you are *not* using a 'POST' resource number in the way you did in earlier examples; those 'POST' resources have a special numbering scheme (they begin at 501) and they follow a certain fixed format. Here, your 'POST' resource is numbered 400 and is simply an ASCII text file containing the required PostScript code. You could just as easily used a 'STR ' resource if you had wanted, or you could have used your own format. However, as was discussed earlier, 'POST' resources are conventionally used to hold PostScript code in many circumstances, and this is a good example of following that convention.

Once you have the PostScript code, you then open the communications channel to the LaserWriter using the new method, **Open_Com_Channel**. This is the first of a series of methods that you will explore in a moment. These methods provide the link from your C routines or methods into the PAP glue routines. For now, all you need to note is that the routine returns a value that indicates whether or not an error has occurred. If one has occurred, the global error handler, gError, is invoked to post an alert box with a special message string. If no error has occurred, then the resource information is sent to the printer by using the new method, **Send_To_Printer**. If this returns an error, the communications channel is immediately closed by calling the **End_PAP_Comm** method; if there is no error, you take the more normal route and call **Close_Com_Channel**, which matches the original **Open_Com_Channel**, and which itself uses **End_PAP_Comm** as part of the processing. Finally, you reset the cursor to the standard arrow and unlock and release the 'POST' resource. However, you aren't done yet. You still must retrieve the response from the LaserWriter. You do that

by first testing the global gFileHandleSize for a nonzero value. If it is not zero, then there has been a response, so you call the auxiliary method, **HandleFeedback**, to display the result. Finally, you release the feedback handle and exit the routine.

The **HandleFeedback** method is very similar to the previous **GetDeviceStatus** method in its operation. In fact, it is so similar that you could easily combine much of the code into a new auxiliary method that could be used from both **GetDeviceStatus** and **SendQuery**. I didn't do that here for clarity of presentation and ease of understanding; generally, I wouldn't do it anyway unless I had a real storage problem. Nevertheless, for analytical purposes, you can see from Listing 7-11 just how much these have in common.

Listing 7-11. Listing of **HandleFeedback** method

```
/*****************************************************
 *  HandleFeedback
 *
 * this function takes whatever is inside the feedback
 * handle and places it into a modal dialog
 *
 ***/
void    CLaser::HandleFeedback (void)
{
    DialogPtr   feedbackDialog;
    char        *feedBack;
    short       itemHit;
    Boolean     dialogDone      = FALSE;

    HLock (gFeedbackHandle);
    feedBack=*gFeedbackHandle;

    SetCursor(*gWatchCursor);
    Cur_Prntr();

    feedbackDialog=GetNewDialog (STATUS_DITL,
    NIL_POINTER, MOVE_TO_FRONT);
    ShowWindow (feedbackDialog);

    /* put up name of printer to query  */
    ParamText (gLaserName, NIL_STRING, NIL_STRING,
    NIL_STRING);
    DrawDialog (feedbackDialog);
```

Listing 7-11. Listing of **HandleFeedback** method (continued)

```
        /**  we have to convert the feedBack into a Pascal
        string **/
        CtoPstr ((char*)feedBack);
        ParamText (gLaserName, feedBack, NIL_STRING,
            NIL_STRING);
        DrawDialog (feedbackDialog);

        SetCursor (&arrow);
        while (dialogDone == FALSE)
        {
            ModalDialog (NIL_POINTER, &itemHit);
            switch ( itemHit )
            {
                case  QUERY_OK :
                        DisposDialog (feedbackDialog);
                        dialogDone=TRUE;
                        break;
            }
        }
        HUnlock (gFeedbackHandle);
}
```

Both methods use the same dialog box and the same items. **HandleFeedback** begins by locking the feedback handle and then setting it to a local variable. After that, the code is essentially identical, except that the feedback, since it comes directly from your PostScript code, is not in Pascal format, but has been converted into C format internally. Now you have to restructure it into Pascal format for display purposes. The routine ends by unlocking the handle to free up the storage.

You also need to update the global information at the beginning of CLaser.c There are quite a few new items, many of which occur in the auxiliary routines that you will look at in a moment. For now, Listing 7-12 simply shows you the required global variables and constants. You also, of course, need to update the header file with your new routines. This is shown in Listing 7-13.

Listing 7-12. Global variables and constants in CLaser.c

```
#include "CError.h"
#include "CPrinter.h"
#include "CLaser.h"
```

Listing 7-12. Global variables and constants in CLaser.c (continued)

```
#include "PAPGlue.h"
#include "TBUtilities.h"

#define CURRENT_VERSION      1
#define CANT_CREATE_HAND     404
#define CANT_OPEN_CHANNEL    405
#define LAS_NOT_AVAL         415
#define DRAFT_DITL           401
#define STATUS_DITL          405
#define STR_STATUS           400
#define QUERY_POST           400
#define NO_RESP              1
#define MOVE_TO_FRONT        -1L
#define NIL_STRING           "\p"
#define NIL_POINTER          0L
#define QUERY_OK             1
#define ERROR                1
#define noERROR              0

extern  CError      *gError;        /* Error handler */
extern  CursHandle  gWatchCursor;   /* Watch cursor for waiting    */

/*  PAP communications global variables        */
extern  char            *gLaserName;
extern  int             gConnectId;
extern  int             gFlowQuantum;
extern  papStatusRec    gLaserStatus;
extern  int             gOpenState;
extern  long            gPapAddress;
extern  char            gReadBuffer[4096];
extern  int             gReadEOF;
extern  int             gReadSize;
extern  int             gReadState;
extern  char            *gWriteBuffer;
extern  int             gWriteEOF;
extern  int             gWriteSize;
extern  int             gWriteState;
```

Listing 7-12. Global variables and constants in CLaser.c (continued)

```
static  Handle      gFeedbackHandle;
static  long        gFileHandleSize;
static  Boolean     gMarkDraft;      /* Draft Box checked (or not) */
static  TPPrDlg     gPrtJobDialog;   /* Job Dialog pointer         */
static  int         prFirstItem;     /* Box item number            */
static  ProcPtr     prPItemProc;     /* Default Job Dialog Item
                                        handler  */
```

Listing 7-13. New version of CLaser.h

```
/**********************************************************************
 * CLaser.h
 *
 *        Interface for the LaserWriter Printer Class
 *
 *    Programming the LaserWriter - Addison-Wesley
 *    by David Holzgang
 *    © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#define _H_CLaser

#include "CPrinter.h"          /* Interface for its superclass     */

struct CLaser : CPrinter {     /* Class Declaration                */
                                  /** Instance Variables **/
      char                *itsLaserName;
      char                gUserName[32];

                                  /** Instance Methods    **/
                                    /** Contruction/Destruction **/
      Boolean      ILaser(CDocument *aDocument, THPrint aMacTPrint);
```

Listing 7-13. New version of CLaser.h (continued)

```
                                        /** Accessing    **/
        void            GetDeviceInfo(short *devNum);
        void            GetDeviceStatus( void );
        Boolean         GetDraftStatus( void );
        void            SendQuery ( void );
        void            HandleFeedback ( void );

                                        /** PAP Communications
                                            Interface **/
        short           Open_Com_Channel( void );
        short           Send_To_Printer(char *pgBuf, short pgBufSize);
        short           Get_Response( void );
        short           Close_Com_Channel( void );
        void            End_PAP_Comm( void );

                                        /** Printing **/
        void            DoPrint(void);

                                        /** Special **/
        void            Cur_Prntr( void );
        Boolean         MyJobDialog(THPrint aMacTPrint);

};

/*      these are aux fcns that are NOT methods                 */
        pascal  TPPrDlg     MyJobDlgInit (THPrint aMacTPrint);
        pascal  void        MyJobItems(TPPrDlg theDialog, int itemNo);
                int         AppendDITL(DialogPtr theDialog, int theDITL);
```

Auxiliary methods

This code uses five auxiliary methods that essentially provide the logical link from the requirements of a program to the glue routine interface. The code for all these routines is shown in Listing 7-14.

Listing 7-14. Auxiliary PAP methods

```
/*****************************************************************
*  Open_Com_Channel
*
*                  opens the AppleTalk link with the LaserWriter
***/
short      CLaser::Open_Com_Channel(void)
{
       long     Current_Time;
       long     Start_Opening;
       char     *buff;

       gFlowQuantum = 8;
       gConnectId = 0;

       if ( LW_PapOpen() )
              return (ERROR);

       Start_Opening = TickCount();
       while ( gOpenState > 0 )
       {
           Current_Time = TickCount();
           if ( (Current_Time - Start_Opening) > 1800 )
                  return (ERROR);
       }

       if ( gOpenState < 0 )
              return (ERROR);

       gReadState = 0;
       gWriteState = 0;
       gReadEOF = 0;
       gWriteEOF = 0;
       gReadSize = 0;
       gWriteSize = 0;
       return (noERROR);
}


/*****************************************************************
*  Send_To_Printer
*
*         takes care of all the comm between the program and the printer
*         when we send the temp file down
```

Listing 7-14. Auxiliary PAP methods (continued)

```
***/
short     CLaser::Send_To_Printer( char *pgBuf, short pgBufSize )
{
      register char   *buffPtr;

      buffPtr = pgBuf;
      while (pgBufSize > 0)
      {
          if (pgBufSize > 512)
              gWriteSize = 512;
          else
              gWriteSize = pgBufSize;

          gWriteBuffer = buffPtr;
          if ( Get_Response() != noERROR )
              return (ERROR);

          if ( LW_PapWrite() )
              return (ERROR);

          while ( gWriteState > 0 )
          {
              if ( Get_Response() != noERROR )
                  return (ERROR);
          }

          if ( gWriteState < 0 )
              return (ERROR);

          pgBufSize -= 512;
          buffPtr += 512;
      }
      return (noERROR);
}

/*****************************************************************
*  Get_Response
*
*         retrieves any messages over the AppleTalk channel and
*         stores them in the feedback handle
```

Listing 7-14. Auxiliary PAP methods (continued)

```
***/
short      CLaser::Get_Response(void)
{
      char     *feedBackPtr;
      char     *readPtr;
      long     oldSize;
      short    loop          = 0;
      Handle   ItemsHandle;

      if (gReadState <= 0)
      {
          if (gReadState < 0)
              return (ERROR);

          if (gReadSize > 0)
          {
              readPtr = &gReadBuffer[gReadSize-1];
              if (*readPtr == 10)
                  *readPtr = 13;
              *(++readPtr) = '\0';
              for (loop = 0; loop <= gReadSize; loop++)
              {
                  if ( gReadBuffer[loop] == 10 )
                          gReadBuffer[loop] = 13;
              }

              oldSize = gFileHandleSize;
              gFileHandleSize += gReadSize;
              SetHandleSize (gFeedbackHandle, gFileHandleSize);
              if ( MemError() == noErr )
              {
                  readPtr = &gReadBuffer[0];
                  feedBackPtr = (char *)*gFeedbackHandle + oldSize;
                  for (loop = 0; loop <= gReadSize; loop++)
                      *feedBackPtr ++= *readPtr++;
              }
          }
          if ( LW_PapRead() )
              return (ERROR);
      }

      return (noERROR);
}
```

Listing 7-14. Auxiliary PAP methods (continued)

```
/*****************************************************************
*  Close_Com_Channel
*
*         will retrieve any last messages and will end communication
*         between program and the AppleTalk connection
***/
short      CLaser::Close_Com_Channel(void)
{
      Get_Response();

      gWriteEOF = 1;
      gWriteSize = 0;

      if ( LW_PapWrite() )
      {
          End_PAP_Comm();
          return (ERROR);
      }
      while (gReadEOF == 0)
      {
          Get_Response();
      }
      End_PAP_Comm();
      return (noERROR);
}


/*****************************************************************
*  End_PAP_Comm
*
*         calls glue routines to close and conclude the AppleTalk
*         connection
***/
void  CLaser::End_PAP_Comm(void)
{
      LW_PapClose();
      LW_PapUnload();
}
```

Let's look at these routines in order. The first one, naturally enough, is **Open_Com_Channel**. This routine opens the PAP communications link. It sets the flow quantum, initializes the connect reference number, and then calls **LW_PAPOpen**. If PAPOpen does not return an error, the processing continues with a loop that tests the global gOpenState to see if the link is open. When the link opens, gOpenState goes to zero (or to a negative value if there was an error). The routine waits 30 seconds for a response; as you read above, PAPOpen retries the connection repeatedly for an indefinite period. If there has been no success within that time, the routine terminates with an error indication. If there is a negative return, an error indication is also returned. If the connection was opened successfully, however, then the routine sets all the global values to their defaults and returns. Note that the all-important connection reference number is automatically stored in the appropriate global variable, gConnectId, by the **LW_PAPOpen** routine.

The next auxiliary is **Send_To_Printer**. This routine sends the data presented to it as parameters to the LaserWriter. The data is sent by a *while* loop that sends the buffer in 512-byte increments. You could use larger buffers, up to the 4096 maximum, but 512 bytes is very convenient if you are providing some indication of processing to a user, for instance, since it allows frequent updates on the status of the transmission. The loop is quite simple in itself. It sets gWriteSize to the desired length, checks for a response, writes the data, checks again for a response and for errors in the write process, bumps the two pointers, and then starts again. If the loop finishes the transmission without posting an error, the write has been successful and the routine returns with no error.

The next auxiliary function, **Get_Response**, gets a response from the server. The important thing to remember here is that PAP is read-driven. Thus the PAPRead call only allows the workstation client to receive data; it doesn't bring it in. The first time through, therefore, this routine merely issues the **LW_PAPRead** call and exits.

The routine begins by testing the gReadState variable. If it is less than or equal to zero, you can proceed; otherwise, you will exit without doing anything because read processing is continuing. When you can proceed, you first test for an error; if there was one, you exit immediately. Otherwise, the read processing is complete and you are ready to retrieve the data. The retrieval process also does some data manipulation as part of its processing. It converts the ASCII data that is returned from the LaserWriter into a standard C string, and it also converts the carriage return characters, 0x0A (10), into linefeed characters, 0x0D (13). This is necessary so that the display presentation will

look the way you expect it to look; carriage returns are not supported on the screen and they show up as the missing character box, whereas linefeeds are understood. Then you convert the returned data into a handle for easier processing and end the retrieval process. The last step is to enable the cycle again by issuing a new **LW_PAPRead** call.

The last two routines work together to close out the communications link with the server. **Close_Com_Channel** is the standard closing routine. It first gets any previous message from the server. Then it sets the gWriteEOF variable to indicate an EOF and writes it to the server. If there is an error at this point, the channel is forced closed by calling **End_PAP_Comm** and an error return is made. However, if the write completes successfully, the close processing goes into a loop, waiting to receive the matching EOF indication from the server client. Until that comes, the routine continues to wait for a response. Once the EOF is returned, the workstation client is directed to close the connection by calling **End_PAP_Comm**. The **End_PAP_Comm** routine simply forces the termination of the communications by issuing an **LW_PAPClose** followed immediately by an **LW_PAPUnload**.

This completes the auxiliary routines that you need to access the PAP glue. Now all that stands between you and your LaserWriter is the PostScript code and a 'POST' resource.

PostScript code resources

The easiest way to build a 'POST' resource such as the one you need here is to build it directly with ResEdit. In the previous examples, you have built all the resources, including some 'POST' resources, using RMaker files. However, as you may remember from the last time you built a 'POST' resource, you have to insert hexadecimal constants repeatedly into the file in order to get it to load and run correctly. This is both cumbersome and annoying. It is much easier to create a new 'POST' resource from ResEdit and fill it with the code that you want to use. This is the approach that you will use here.

To begin, however, there is the small matter of the PostScript code that you should use for the query. As your first test, you can use the code shown in Listing 7-15, which tests for a particular feature on the LaserWriter.

Listing 7-15. PostScript code to test **pagestackorder**

```
save
    statusdict begin
    pagestackorder
    == flush
    end
restore
```

This is quite short and straightforward; you execute the **pagestackorder** operator, which is located in the **statusdict**. This returns **true** if the pages of the printer stack in correct order (1 through *n*) as they do in the LaserWriter II, or **false** if they stack in reverse order (*n* through 1) as they do in the original LaserWriter. This is a useful item of information for an application, which can make the user's life easier by printing pages in reverse order if desired, thus making them collate correctly when **pagestackorder** is **false**. In fact, many applications do this, but they usually implement it through a check box on the job dialog. By testing this, you could decide to check the box or not as a default, for example.

Create this code in the Think C editor, and then select all the code and copy it. Next, launch ResEdit and open the file SimpleLW.π.rsrc in your current directory. Then create a new resource of the type 'POST', as shown in Figure 7-3. When you get the new 'POST' resource, open it as New and you will get the general hexadecimal editor view, shown in Figure 7-4.

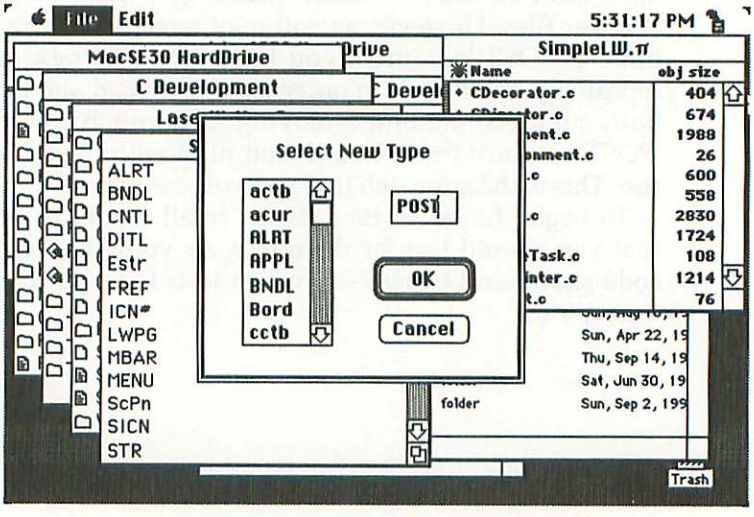

Figure 7-3. Creating a new 'POST' resource

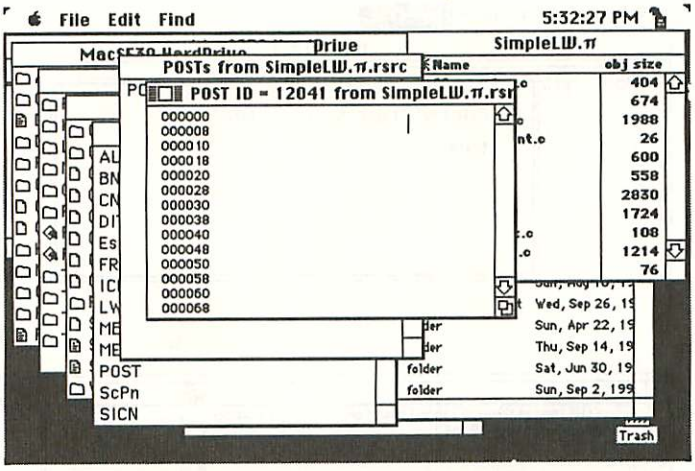

Figure 7-4. Hexadecimal editor in ResEdit

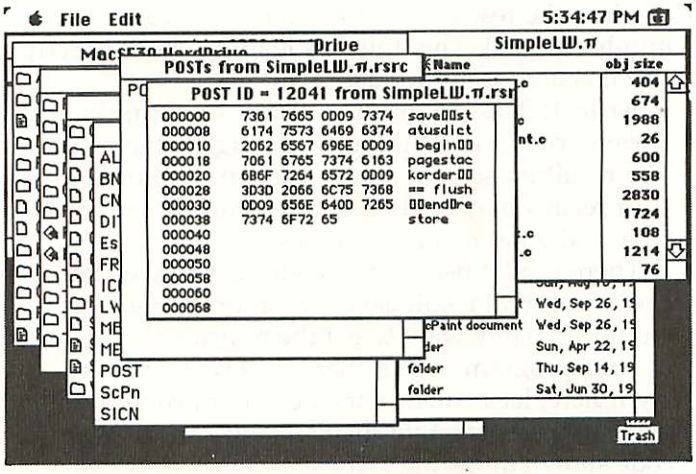

Figure 7-5. 'POST' resource loaded with PostScript code

Be sure to position your text insertion cursor (the I cursor, not the arrow) in the right, or text, side of the resource. Now select Paste from the Edit menu. This loads the copy of the PostScript text into the resource as you see in Figure 7-5.

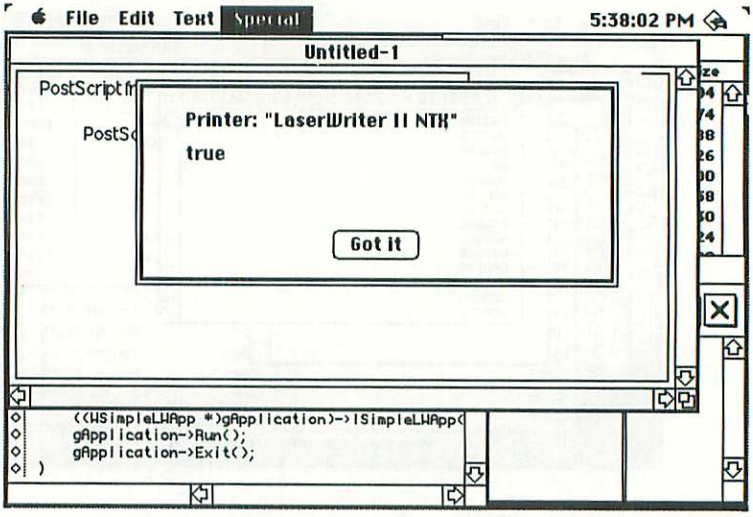

Figure 7-6. Result from **pagestackorder** inquiry

Close the resource and use the Get Info command to set the resource number to 400. This finishes loading the PostScript into the resource, and it is now ready for use.

At last! The program is ready to compile and run. As is almost always true (if everything works right), the result is a bit anticlimactic. The resulting screen, on my system, is shown in Figure 7-6. Note that your results may be different, depending on the type of printer that you have and what name you are using for it.

There is a lot behind this code, as you well know by this point. However, the actual result isn't very informative. This format is the way you would probably want to get the return string if you were going to use it in your program, rather than display it. However, since you're displaying it here, let's enhance the PostScript code slightly to get an improved and more informative display. You can do that quite easily using the code shown in Listing 7-16.

Listing 7-16. Revised code for showing **pagestackorder**

```
save
    statusdict begin
    pagestackorder
    {(Pages come out in correct order.)}
    {(Pages come out in reverse order.)}
    ifelse
    print flush
    end
restore
```

This code uses the **pagestackorder** Boolean return to determine which of two strings to display. Note that you use the **print** operator this time for the display, since you don't require—indeed don't want—the conversion provided by the == operator. Next, copy and paste this text into the 'POST' resource in place of the previous code using ResEdit. When you run again, you get the more intelligible result shown in Figure 7-7.

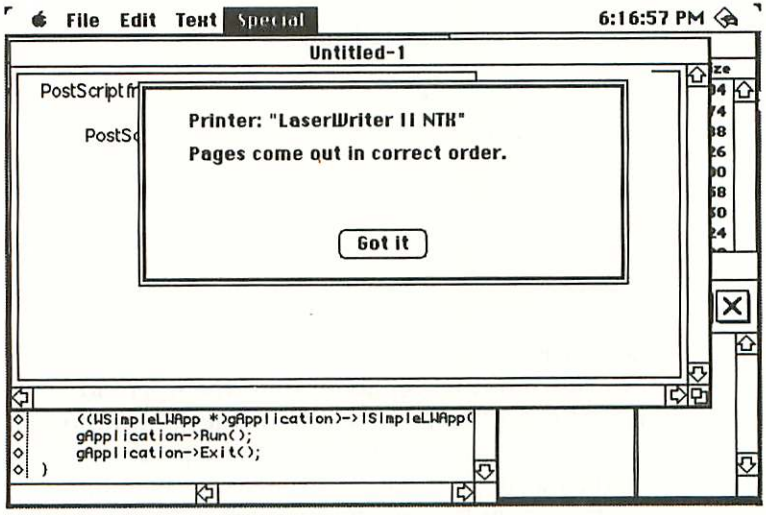

Figure 7-7. Result from revised PostScript code

Another typical requirement is to ask for a specific font. You read about this type of query in Chapter 6, and you even saw some code for it. Now you can use that code to determine an actual response. You can use the code shown in Listing 6-1, and you should, if you have a hard disk attached to your LaserWriter. However, if you don't, then you can use the simpler code shown in Listing 7-17.

Listing 7-17. PostScript code to test for a font or fonts

```
save
2 dict begin
    /sv exch def
    /st2 128 string def
    {
      count 0 gt
          {
            dup st2 cvs (/) print print (:) print
            dup FontDirectory exch known
                {pop (Yes)}
                {pop (No) }
            ifelse
            print flush
          }
          { exit }
      ifelse
    } bind loop
    (*) print flush
sv
end %temp dict
restore
```

Recall how this works: You place the name or names of fonts that you want information about onto the operand stack and then execute this code. It returns the name of the font, followed by a Yes or a No response to show whether the font is present or not; the list of fonts is ended by a single * character for ease of parsing.

In the example here, you will not be parsing the reply, only displaying it. Nevertheless, you still have the problem of how to place the font name or names that you want to request onto the operand stack. The obvious answer is to send the names down from your application just before the inquiry. To place the name in the resource would be ridiculous; you will want to change them. In a real application, you

might even ask the user to enter the name in a dialog box, or something of that nature. Here, you simply insert the names directly from strings in the application. With what you know now, this is a piece of cake. The required changes to **SendQuery** are shown in Listing 7-18.

Listing 7-18. Changes to **SendQuery** to ask about fonts

```
/*****************************************************
 *  SendQuery
 *
 *   this routine sends a list of font names
 *   and whatever is in the resource 'POST' 400
 *   to the LaserWriter
 **
 *****/
void    CLaser::SendQuery (void)
{
    Handle  resHand;
    long    resSiz;
    OSErr   osErr;
    char    *fontString;
    int     fontSiz;
    char    *fontP;

    gFeedbackHandle = NewHandle (0L);
    if ( gFeedbackHandle == 0 || MemError() )
        gError->CheckOSError( CANT_CREATE_HAND );
    gFileHandleSize = 0;

    resHand = GetResource ('POST', QUERY_POST);
        CheckResource( resHand );
    HLock(resHand);
    resSiz = SizeResource(resHand);

    SetCursor(*gWatchCursor);
    Cur_Prntr();

    if ( Open_Com_Channel() )
        gError->PostAlert(STR_STATUS, NO_RESP);

    fontString = "\p/Times-Roman /Stempel-Schneider\n";
    fontP = fontString+1;
```

Listing 7-18. Changes to **SendQuery** to ask about fonts (continued)

```
        fontSiz = fontString[0];
        osErr = Send_To_Printer(fontP, fontSiz);
        if (osErr != noERROR)
            End_PAP_Comm();

        osErr = Send_To_Printer(*resHand, (short)resSiz);
        if (osErr != noERROR)
            End_PAP_Comm();
        else
            Close_Com_Channel();

        SetCursor(&arrow);

        HUnlock (resHand);
        ReleaseResource (resHand);

        if (gFileHandleSize)
            HandleFeedback();
        DisposHandle (gFeedbackHandle);
        return;
}
```

This is quite simple; you just take the font names and send them using **Send_To_Printer** before you send the resource file PostScript. The only thing of special note here is the \n as a terminator for the string. This is required to force a new line between the font names and the first character of the resource PostScript. This may not be necessary in some cases, but it never hurts because PostScript ignores it if it is unneeded; however, if it is required and is not present, you will get a PostScript error.

Insert the new inquiry into the 'POST' resource in place of the old one, recompile the CLaser.c module, and run. Depending on your choice of fonts and what you have loaded on your LaserWriter, you should see something like Figure 7-8.

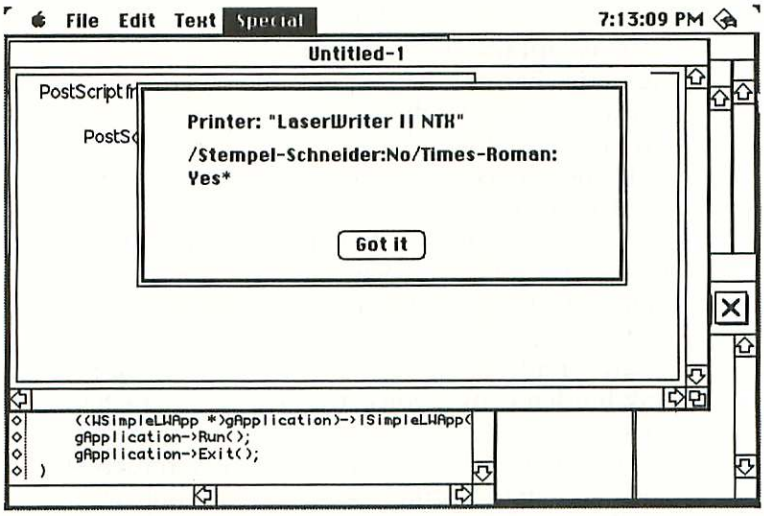

Figure 7-8. Response to the font query

▶ Testing the Laser Prep version

Testing for the Laser Prep version is much like the previous tests that you did in the last examples. It basically consists of sending the correct PostScript code to the LaserWriter and retrieving the response. The only catch in this process is that you have to understand some things about the **md** dictionary in addition to the PostScript code before you can get a valid response.

Since **md** is essentially an internal Apple dictionary, you naturally should not expect it to contain any specific set of functions unless you verify that it is a version that you know has what you want. The only guaranteed method for doing this is to check the **av** variable in the dictionary. This is the *only* item in the dictionary that Apple guarantees will remain always defined and will always have the same information: the current dictionary version stored as a number—an integer number, in all versions to this point. Technical Note #152 gives you a description of how this works and some brief code on testing the version number. However, the code there is a bit limited; it does not provide for a range of values, and it returns a single numeric character.

Although Laser Prep changes with time, many versions have the same functions in them; the **psb**, **pse** functions that you read about earlier are good examples of functions that have not changed over time. If you were relying on these functions, for example, you might easily

decide that any Laser Prep file within a rather broad range of versions was acceptable for your processing purposes. Returning a single numeric character has some value when you are testing the return and taking some action based on it. You can then use a *switch* statement to provide branching.

If you are using that technique, however, it will be easier for you to return an integer instead of an integer string. Thus you should either use the = operator on the string, instead of the **print** operator shown below, or (my preference) use the == operator and insert actual integer values, instead of integer strings, as the return. When you are working in two languages, such as PostScript and C, I think that it is much clearer if all the interaction takes place in an identical fashion, without any hidden conversions. That makes the matching coding much easier to follow.

Here you will, once again, simply return strings for display. The code shown in Listing 7-19 gives you three clear, unambiguous returns for display.

Listing 7-19. PostScript code to test for range of Laser Prep versions

```
/md where
{
    /md get /av get
    cvi dup
    65 ge exch
    70 le
    and
    {
        (Good version)
    }
    {
        (Not correct version)
    }
    ifelse
}
{
    (Laser Prep not loaded yet)
}
ifelse
print flush
```

Clearly, you should replace the version numbers given here (65 and 70) with the range of versions that you want to test. The versions are inclusive, so that both version 65 and version 70 are acceptable in this case. To test this, simply replace the 'POST' 400 resource in your current file with this code and remove the additional lines in **SendQuery** that insert the font names. Then recompile and run; the result should look something like Figure 7-9. Of course, the response will vary depending on whether you have actually loaded Laser Prep yet (I haven't, as you see), and whether the loaded version matches the test in your code.

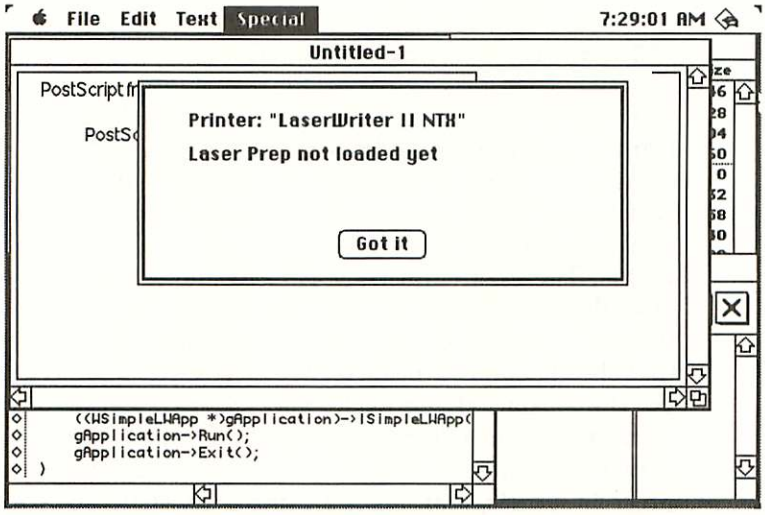

Figure 7-9. Response to Laser Prep query

▶ Testing for a spooler

There is one last issue that you should know about for this type of processing. If you are going to carry on a dialog with the LaserWriter, as you have been here, you need to be able to assure yourself that the device at the other end is, indeed, a LaserWriter printer and not a spooler that has replaced the printer on the network as described and discussed previously.

You can do that by sending a packet of simple code that depends on having a PostScript interpreter receive and process it. Spoolers are not full PostScript devices, after all, and therefore will only respond to a limited number of queries. By choosing both the format and content of the query, you can pretty fairly determine whether the server client that is responding to you is a real LaserWriter or a spooler in LaserWriter's clothing. Sample code for this is shown in Listing 7-20.

Listing 7-20. Code to test for a spooler

```
%!PS-Adobe-2.0 Query
%%Title: Query to determine if we are talking to a
%%Spooler
%%Creator: LaserTalk
%%For: David Holzgang
%%?BeginPrinterQuery
    statusdict begin
    product == flush
    end
%%?EndPrinterQuery: Spooler
```

The code in Listing 7-20 returns the product name if the device is a printer, and the string 'Spooler' if it is a LaserShare spooler or another compatible spooler. A similar process is defined in Technical Note #133, which returns a single number instead of a string from the LaserWriter.

This code deserves a closer look, so that you understand why it works. The PostScript code is clear enough; you simply place a string onto the operand stack and then send it back to the device. If you get that string (or integer, or whatever) back, you can reasonably assume that the device is a LaserWriter or compatible device. But how do you get the second string back? Well, that depends on the spooler understanding the structure of a standard query, as defined in the manual *Structuring Conventions, Version 2.0*, by Adobe Systems. Notice, in the preceding code, that this is a completely structured query: it begins with the header, has the required comments, and so on. This is essential if you are going to receive the response that you expect from the spooler. Remember that the spooler does not know PostScript; in fact, you are relying on that to discriminate between the LaserWriter and the spooler. However, the spooler should know the structuring conventions, and when it receives this query, it should respond with the default return value given in the **%%?EndSpoolerQuery** comment.

Unfortunately, some spooler versions that do not respond correctly to this type of query do exist. When you perform this test, therefore, you should include a timing loop to wait only a certain time for the response. If the spooler does not understand the query format, you will receive no answer within some reasonable time. In that case, you should assume that the server device is not a LaserWriter.

▶ File Handling

The LaserWriter IINTX provides for an external hard disk drive that can be used as part of the storage facilities on the LaserWriter. Typically, the drive is used for font storage and font caching, but it can also be accessed from a PostScript program, using standard PostScript operators. In fact, PostScript provides a standard set of file operators that you can use for reading and writing file streams of all types, including the standard input and output files as well as files on external devices, such as the disk.

▶ File structure

The PostScript language provides a full set of operators for creating and using file streams. All PostScript devices use standard file streams for input and output that are defined under all circumstances, even if the device does not have any external file storage. In other words, these files are defined on all the LaserWriter devices, from the original LaserWriter to the IINTX, even if there is no hard disk attached. These files can be summarized as follows:

File name	*Description*
(%stdin)	The standard input file stream; that is, the file that provides the PostScript language input to the interpreter. This file can also be used to provide image data and other input to the PostScript application. This file usually represents the input communication channel that is being used by the interpreter; in the examples here, that means the AppleTalk communications channel.
(%stdout)	The standard output file stream; that is, the file that receives data that is sent by the application unless some other file is explicitly invoked. This file is the reverse channel where the data that is transmitted by the application appears; generally, it is the same communication channel as the input channel.
(%stderr)	The standard error output file stream; that is, the file where the error information is sent by the interpreter. This is generally identical to the standard output file, but you can reassign it to another communication channel or another device if you want.

These are generic names for these standard files, and when you use them you do not need to worry about the actual communications channel that they represent; in other words, if you reference the *%stdin* file, it may represent the AppleTalk communications, or the serial communications channel; in either case, your application will work in practically the same way. (I say "practically" because there are some small differences in how the two channels handle some character codes, primarily line endings.)

These basic PostScript files represent the input and output data streams that come to the printer. These files are present in all PostScript printers or other output devices, since they are the mechanism whereby the interpreter itself retrieves and sends information. The PostScript language provides a complete set of operators for creating, accessing, and controlling these file streams as part of the standard language operators. If you wish to learn more about how to use these standard files, there is a good, if somewhat terse, discussion of them in the book *PostScript Language Program Design* (Addison-Wesley, 1988). In addition, PostScript language operators are provided for creating and managing files on an external device, such as a disk, if one is attached to the printer. You should notice that the operators that control such devices are only defined on printers that may have one. In other words, in the LaserWriter family, the PostScript operators that allow you to access and use the disk file system are only defined on the IINTX. This fact is used as part of most PostScript applications as a method to test whether the printer supports external devices, as you will see in the following code examples. Definition and use of these operators is discussed in some detail in the *Apple LaserWriter Reference* manual. In this section of the chapter you will look at how to use these operators to retrieve and use files on external devices.

The PostScript language file system supports files on a variety of external devices, which depend on what is supported by the specific device. For the LaserWriter family, this usually means an external hard disk, which is what you will work with here; however, if the printer supports other devices, such as a font cartridge for example, the same or similar procedures can be used to create and manipulate files on these devices as well. Device names are quite rigid in the PostScript language. If you want to determine what device names are in use on your printer, you can use the code in Listing 7-21 to list the names of the devices on your system.

Listing 7-21. Code to list device names

```
clear
/scratch 100 string def
(%*) {0 6 getinterval} scratch filenameforall
count array astore
0 get =
```

This code generates an array with the names of all the devices on the file system. There is one entry in the array for each file, so that there will be many duplicate device names; the last line returns the first device name over the communication channel. Notice that this code is somewhat unfriendly, because it clears the operand stack before executing to ensure that it can create the desired array. If there is more than one device, you should save the generated array and examine it for all the device names. That task requires quite a bit more code and is beyond the scope of what you want to accomplish here.

The PostScript language is not provided with full operating system features for handling files, nor does it require them, because it is designed to be used in a single device with a well-defined task. By the same token, there is no operating system that is implemented to provide system services to a program. Therefore, some of the file features that you may have come to expect are not present in PostScript file handling. In particular, PostScript uses a simple, flat file storage structure on external devices such as the disk. There are no directories and no real file hierarchies within the PostScript disk file system. Since there is no operating system, there is also no file management; issues such as record or file locking, file backup and maintenance, and so on must be done by the application if they are done at all.

However, PostScript does use what might be called a "pseudo-directory" system for its disk files. What this means is that the names of the files are constructed to have a hierarchical format that imitates a directory naming structure. That is, the file names consist of two segments, divided by a slash. The first part of the name is used as a directory name, and the second part is the file name. Although these are not true directories, this convention allows you to search the disk for files in a standard manner, since the file system implements standard UNIX and DOS wildcard characters * and ? for file name searches. For example, if you have loaded the Stone Serif font on the disk it will be stored on the disk under the name font/StoneSerif. This allows you to look up all font names, for example, by searching for all files with the name string (font/*), where the * is the standard wildcard character for matching any substring. It is strongly recommended that you follow this naming system for your own files as well.

The conventions that are normally used for PostScript files on the disk can be summarized as follows:

- File names are case sensitive. They cannot be longer than 100 characters, and they must not begin with the character %, which is reserved for device names and for the standard input and output file streams.
- All system file names are of the form *directory/filename*, as described previously.
- Do not create file names that begin with the prefixes *Sys/*, *FC/*, and *DB/*. These prefixes are reserved for use by the interpreter for special files.

There is one exception to this last rule. You may decide to create or modify a special system file named *Sys/Start*. The creation and use of this file is particularly described in the *Apple LaserWriter Reference* manual.

▶ Disk operations

The hard disk that can be attached to the LaserWriter IINTX is used for three purposes:

- Virtual memory for display-list buffers when the interpreter is creating complex pages.
- Non-volatile storage for font characters. This is essentially an extension of the standard PostScript font cache to the disk, with the added benefit that the font cache is not wiped out when you reset the printer.
- Storage of user files. These will most often be fonts, but may be any files that you want.

The disk must be initialized before it can be used; there is a standard PostScript operator to provide this function. The amount of space that is available for each of these purposes is determined by the partitioning of the disk. The disk is partitioned into two segments: one is for the user files, and the other is used by the system for font caching and display-list storage. You can determine the percentage of the disk for each function; the standard division is 20% for user files and 80% for system use.

The PostScript language operators that are used to initialize, control, and access the disk are defined in the *PostScript Language Reference Manual*, Second Edition, although not in the First Edition, unfortu-

nately. They are also provided in the *Language Supplement* that accompanies any printer or other output device that supports an external file system. For the LaserWriter, these operators are defined in the *Apple LaserWriter Reference* manual. They are also defined, along with many other common device-specific operators, in the *PostScript Programmer's Reference Guide.*

If you have a hard disk attached to your LaserWriter, Version 2.0 of the LaserWriter Font Utility will determine if a hard disk is attached and allows you to initialize the disk and load fonts to it. The Font Utility uses the standard defaults as far as using the disk goes, but it allows you to use the disk as an additional storage device without knowing anything about PostScript file operators and so on.

▶ Using the disk

As an example of how you might work with the disk, following is a short code section that lists the fonts on a system. Listing 7-22 is the same code that you saw earlier as Listing 6-2.

Listing 7-22. Listing all fonts on the system

```
save
    FontDirectory { pop == } forall flush
    /filenameforall where
    {
        /scratch 100 string def
        pop
        (fonts/*)
        { dup length 5 sub 5 exch getinterval == }
        scratch
        filenameforall
    }
    if
    (*) print flush
restore
```

You already know what this code does from the previous discussion; this time, let's discuss how it accomplishes its task. The code begins by simply loading the **FontDirectory** onto the operand stack and listing all the fonts that have been registered there. The list that this produces will be adequate only if there is no external disk attached to the system, so the code then continues by testing for the **filenameforall** operator. If this operator is present, then a file system and some external device are

attached to the printer. This device need not be a disk; it might be a cartridge or some other external device. Since all external devices are handled in a similar manner in PostScript, it is not necessary to make a specific test for a variety of devices, such as a separate disk and cartridge and so on. Using this operator retrieves all of the fonts on all of the devices.

If **filenameforall** exists, then you begin to list any fonts on the system. Since **where** returns **true** and the dictionary that contains the operator, you first **pop** the dictionary from the stack since you don't require it here. Next you set up for the **filenameforall** operator. Here is the definition of this operator in the standard format.

pattern proc scratch **filenameforall** — —

searches through the file directory for all file names that match *pattern*. When it finds a file that matches *pattern*, it executes the procedure *proc* for that file name. When *proc* is executed, the matching file name is on the operand stack in the scratch string, *scratch*. The *pattern* string may contain two types of wildcard characters, to allow searches for multiple files in a single access: the * character matches any substring, and the ? character matches any single character. The *scratch* string must be large enough to contain the file name that is returned; if it is not, you will get an error. At present, the largest possible file name is 100 characters. The order in which files are returned is not defined, and any files that are created or destroyed by *proc* may be listed by the operator or not. The returned file names do not include the device name unless the *pattern* begins with a device name or some substring of the device name. Thus, for example, the string (*) returns all file names without the device name, whereas the string (%*) returns file names with device names.

In the code segment here, the pattern is (fonts/*). This uses the wildcard character * to find all file names that begin with *fonts/* as a directory. Font downloaders, such as Apple LaserWriter Font Utility, place all the fonts that you load onto a hard disk into this directory. The scratch string that is used is made exactly 100 characters long, so that it will be long enough for the longest possible file name.

The procedure being used may look a little complex, but actually it is quite simple. Remember that the procedure is called with the name of the file on the operand stack; for example, the stack might contain the string

```
fonts/StempelSchneidler
```

Now, when you examine this, you immediately see that it would be useful to eliminate the substring "fonts" from this name before returning the string. This accomplishes two purposes: First, it reduces the number of characters that have to be transmitted and, second, it makes the font names from the disk comparable to the font names that were generated from the **FontDirectory**. The code here, therefore, strips off the first five characters of the returned name using a **getinterval** command, and then returns the resulting name.

▶ Using a PPD file

I mentioned above that much of this type of code—to determine fonts and so on—is somewhat dependent upon the device that you are using. For our purposes, this isn't much of an issue because, by definition, you are working on and writing code for an Apple LaserWriter. However, many other devices can be attached to the Macintosh that look, from a software point of view, just like a LaserWriter, but that have different device characteristics. The obvious question then arises: How can you write code that works on all these devices?

Being very aware of this problem, Adobe Systems has created a type of file that can be read by an application to provide many types of device-dependent information. These files are called Adobe PostScript Printer Description files, or PPD files for short. The PPD file contains a lot of device-specific information in a format that can be parsed and used by an application, including specific PostScript code to perform tasks like testing for a specific font or fonts or listing all the fonts in the system. Since this is not a book on PostScript programming, we will not discuss the content or format of PPD files here, except to note that the code used in this chapter to list specific fonts or all fonts on a device is quite typical of the code that you might find in a PPD file.

If you are going to write a commercial application that requires this type of dialog with a PostScript device, I recommend that you become familiar (if you are not already) with PPD files; the complete description of the PPD file is available from Adobe Systems.

▶ Conclusion

This chapter covers the process of direct communication with the LaserWriter, using the Printer Access Protocol. This is the final level of control that you can have, and it places you in complete command of the entire process. The chapter has shown you how to manage and use the PAP protocol from your program to get the results that you want.

Now you can see and understand how this book has been structured. You began with the basic, standard Macintosh Printing Manager processing. You expanded that processing to include sending PostScript commands to the LaserWriter, using the LaserWriter driver as a willing accomplice in the process. With this, you explored how to do some PostScript programming. Unfortunately there was no room here for a substantial amount of PostScript, but you had the chance, at least, to discover some of the more useful and common principles of the language. Next you explored in more detail how to work with the LaserWriter driver and the Laser Prep file. At each level, you are coming closer to the heart of the process. After a quick look at the issues in font processing, you came, finally, to this chapter on direct communication with the LaserWriter.

Each of these pieces was needed in order to reach this goal, which has always been our target. To work outside the LaserWriter driver, you must first know what is inside it and how to exploit that. To make use of the power of direct communications, you must know at least some PostScript so that you can generate the responses that you want from the interpreter. You have covered all of these topics in this book.

As we discussed in the Introduction, direct communication with the LaserWriter is neither desirable nor required for every application. When it is, however, the tools and techniques are available to help you get the most out of the device. Finally, more knowledge of the mechanisms that make the Macintosh such an outstanding platform for high-quality printing should make you a better developer or programmer and, even if you never use these techniques again, they will give you a deeper appreciation of the processing that is involved in this task.

Appendix
Complete Example Code

```
/*****************************************************************
 * SimpleLW.c
 *
 *  A starter file for writing programs using the LaserWriter...
 *  uses the THINK Class Library
 *
*****************************************************************/
#include "CSimpleLWApp.h"

extern  CApplication    *gApplication;

void main()

{
    gApplication = new(CSimpleLWApp);
    ((CSimpleLWApp *)gApplication)->ISimpleLWApp();
    gApplication->Run();
    gApplication->Exit();
}
```

```
/**********************************************************************
 * CSimpleLWApp.h
 *
 *  Application class header for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#define     _H_CSimpleLWApp     /* Include this file only once */

#include <Global.h>
#include <Commands.h>
#include <CApplication.h>
#include <CBartender.h>

#define ABOUT_ALERT         400
#define NIL_POINTER         0L

#define NOT_A_NORMAL_MENU   -1

struct CSimpleLWApp : CApplication {

    /* instance variables */
    MenuHandle      theFontMenu;
    MenuHandle      theSizeMenu;
    MenuHandle      theStyleMenu;

    /* method definitions */
    void    ISimpleLWApp(void);
    void    SetUpFileParameters(void);
    void    SetUpMenus(void);
    void    UpdateMenus(void);

    void    DoCommand(long theCommand);

    void    CreateDocument(void);
    void    OpenDocument(SFReply *macSFReply);

    void    Exit(void);
};
```

```
/************************************************************************
 * CSimpleLWApp.c
 *
 *  Application methods for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
************************************************************************/

#include "CSimpleLWApp.h"
#include "CSimpleLWDoc.h"

extern  OSType  gSignature;
extern  CBartender *gBartender;


#define MENUtext    4       /* The text menu is not one of the
                               reserved menus */
#define MENUstyle   12      /* The style menu is not one of the
                               reserved menus */

/*  set up item numbers for our menu selections */
#define FONT_ITEM   1024
#define SIZE_ITEM   1025
#define STYLE_ITEM  1026
#define PRNTR_ITEM  1048


/************************************************************************
 * ISimpleLWApp
 *
 *  Initialize the application. Your initialization method should
 *  at least call the inherited method. If your application class
 *  defines its own instance variables or global variables, this
 *  is a good place to initialize them.
 *
 ***/

void CSimpleLWApp::ISimpleLWApp(void)
```

```
{
    CApplication::IApplication(4, 20480L, 2048L);
}

/**********************************************************************
 * SetUpFileParameters
 *
 *  In this routine, you specify the kinds of files your
 *  application opens.
 *
 *
 ***/

void CSimpleLWApp::SetUpFileParameters(void)

{
    inherited::SetUpFileParameters();    /* Be sure to call the
                                            default method */

        /**
         ** sfNumTypes is the number of file types
         ** your application knows about.
         ** sfFileTypes[] is an array of file types.
         ** You can define up to 4 file types in
         ** sfFileTypes[].
         **
         **/

    sfNumTypes = 2;
    sfFileTypes[0] = 'TEXT';
    sfFileTypes[1] = 'EPSF';

        /**
         ** Although it's not an instance variable,
         ** this method is a good place to set the
         ** gSignature global variable. Set this global
         ** to your application's signature. You'll use it
         ** to create a file (see CFile::CreateNew()).
         **
         **/

    gSignature = 'LWPG';
}
```

```
/**********************************************************************
 * SetUpMenus
 *
 *      Set up menus which must be created at run time, such as a
 *      Font menu. You can eliminate this method if your application
 *      does not have any such menus.
 ***/

void    CSimpleLWApp::SetUpMenus()
{
    inherited::SetUpMenus();     /* Superclass takes care of adding */
                                 /* menus specified in a MBAR id=1  */
                                 /* resource                        */

        /**
         ** Add the hierarchical menus in the resource file to the
         ** Text menu. Remember, MENUfont, MENUsize and MENUstyle are
         ** reserved font numbers.
         **
         **/
    theFontMenu = GetMenu( MENUfont );
    theSizeMenu = GetMenu( MENUsize );
    theStyleMenu = GetMenu ( MENUstyle);

    InsertMenu( theFontMenu, NOT_A_NORMAL_MENU);

        /**
         ** Add the fonts in the  system to the FONT menu
         **
         **/
    AddResMenu(GetMHandle(MENUfont), 'FONT');

    InsertMenu( theSizeMenu, NOT_A_NORMAL_MENU);
    InsertMenu( theStyleMenu, NOT_A_NORMAL_MENU);

        /**
         ** The UpdateMenus() method sets up the dimming
         ** for menu items. By default, the bartender dims
         ** all the menus, and each bureaucrat is reponsible
         ** for turning on the items that correspond to the commands
         ** it can handle.
```

```
        **
        ** Set up the options here. The edit pane's UpdateMenus()
        ** method takes care of doing the work.
        **
        ** To start with, you want all the items on the Text Menu
        ** enabled. (Later change this when working on non-text files.)
        **
        **/
    gBartender->SetDimOption(MENUtext, dimNONE);

        /**
        ** For Font and Size menus, you want all the items to
        ** be enabled all the time. In other words, you don't
        ** want the bartender to ever dim any of the items
        ** in these two menus.
        **
        **/
    gBartender->SetDimOption(MENUfont, dimNONE);
    gBartender->SetDimOption(MENUsize, dimNONE);
    gBartender->SetDimOption(MENUstyle, dimNONE);

        /**
        ** For Font and Size menus, one of the items
        ** is always checked. Setting the unchecking option
        ** to TRUE lets the bartender know that it should
        ** uncheck all the menu items because an UpdateMenus()
        ** method will check the right items.
        ** For the Style menu, uncheck all the items and
        ** let the edit pane's UpdateMenus() method check the
        ** appropriate ones.
        **
        **/
    gBartender->SetUnchecking(MENUfont, TRUE);
    gBartender->SetUnchecking(MENUsize, TRUE);
    gBartender->SetUnchecking(MENUstyle, TRUE);
}

/*****************************************************************
 * UpdateMenus {OVERRIDE}
 *
 *      Perform menu management tasks
 ***/
```

```
void    CSimpleLWApp::UpdateMenus()
{
    inherited::UpdateMenus();    /* Enable standard commands        */

    /* &&& Enable the commands handled by your Application class  */
}

/**********************************************************************
 * DoCommand
 *
 *  Your application will probably handle its own commands.
 *  Remember, the command numbers from 1-1023 are reserved.
 *  The file Commands.h contains all the reserved commands.
 *
 *  Be sure to call the default method, so you can get
 *  the default behvior for standard commands.
 *
 ***/
void CSimpleLWApp::DoCommand(long theCommand)

{
    switch (theCommand) {

        /* Your commands go here */

        case cmdAbout:
            NoteAlert( ABOUT_ALERT, NIL_POINTER );
            break;

        default:    inherited::DoCommand(theCommand);
                    break;
    }
}

/**********************************************************************
 * CreateDocument
 *
 *  The user chose New from the File menu.
 *  In this method, you need to create a document and send it
 *  a NewFile() message.
 *
 ***/
```

```
void CSimpleLWApp::CreateDocument()

{
    CSimpleLWDoc    *theDocument;

    theDocument = new(CSimpleLWDoc);

        /**
         ** Send your document an initialization
         ** message. The first argument is the
         ** supervisor (the application). The second
         ** argument is TRUE if the document is printable.
         **
         **/

    theDocument->ISimpleLWDoc(this, TRUE);

        /**
         ** Send the document a NewFile() message.
         ** The document will open a window, and
         ** set up the heart of the application.
         **
         **/
    theDocument->NewFile();
}

/**********************************************************************
 * OpenDocument
 *
 *  The user chose Open... from the File menu.
 *  In this method you need to create a document
 *  and send it an OpenFile() message.
 *
 *  The macSFReply is a good SFReply record that contains
 *  the name and vRefNum of the file the user chose to
 *  open.
 *
 ***/

void CSimpleLWApp::OpenDocument(SFReply *macSFReply)
```

```
{
    CSimpleLWDoc    *theDocument;

    theDocument = new(CSimpleLWDoc);

        /**
         ** Send your document an initialization
         ** message. The first argument is the
         ** supervisor (the application). The second
         ** argument is TRUE if the document is printable.
         **
         **/

    theDocument->ISimpleLWDoc(this, TRUE);

        /**
         ** Send the document an OpenFile() message.
         ** The document will open a window, open
         ** the file specified in the macSFReply record,
         ** and display it in its window.
         **
         **/
    theDocument->OpenFile(macSFReply);
}

/**********************************************************************
 * Exit
 *
 *  Chances are you won't need this method.
 *  This is the last chance your application gets to clean up
 *  things like temporary files.
 *
 ***/

void CSimpleLWApp::Exit()

{
    /* your exit code here */
}
```

```
/**********************************************************************
 * CSimpleLWDoc.h
 *
 *  Document class header for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#define _H_CSimpleLWDoc          /* Include this file only once */

#include <Global.h>
#include <Commands.h>
#include <CApplication.h>
#include <CBartender.h>
#include <CDataFile.h>
#include <CDesktop.h>
#include <CDecorator.h>
#include <CDesktop.h>
#include <CDocument.h>
#include <CError.h>
#include <CPanorama.h>
#include <CScrollPane.h>
#include <TBUtilities.h>

#define BASE_RES_ID          400

#define BASE_WINDOW          BASE_RES_ID      /* Resource ID for WIND
                                                 template */
#define BASE_PANE            BASE_RES_ID      /* Resource ID for ScPn
                                                 template */

struct CSimpleLWDoc : CDocument {

                                    /** Construction/Destruction **/
    void        ISimpleLWDoc(CBureaucrat *aSupervisor, Boolean
printable);
    void        Dispose();

    void        DoCommand(long theCommand);
```

```
    void        Activate(void);
    void        Deactivate(void);

    void        NewFile(void);
    void        OpenFile(SFReply *macSFReply);
    void        BuildWindow(Handle theData);

                                    /** Filing **/
    Boolean     DoSave(void);
    Boolean     DoSaveAs(SFReply *macSFReply);
    void        DoRevert(void);
};
```

```
/**********************************************************************
 * CSimpleLWDoc.c
 *
 *  Document methods for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#include "CSimpleLWDoc.h"
#include "CSimpleLWPane.h"


extern  CApplication *gApplication; /* The application */
extern  CBartender  *gBartender;    /* The menu handling object */
extern  CDecorator  *gDecorator;    /* Window dressing object   */
extern  CDesktop *gDesktop;         /* The enclosure for all windows */
extern  CBureaucrat *gGopher;       /* The current boss in the chain
                                       of command */
extern  OSType      gSignature;     /* The application's signature */
extern  CError      *gError;        /* The global error handler */

/**********************************************************************
 * ISimpleLWDoc
 *
 *  This is the document's initialization method.
 *  If your document has its own instance variables, initialize
 *  them here.
 *  The least you need to do is invoke the default method.
 *
 ***/

void CSimpleLWDoc::ISimpleLWDoc(CBureaucrat *aSupervisor, Boolean
printable)

{
    CDocument::IDocument(aSupervisor, printable);
}

/**********************************************************************
```

```
 * Dispose
 *
 *  This is your document's destruction method.
 *  If you allocated memory in your initialization method
 *  or opened temporary files, this is the place to release them.
 *
 *  Be sure to call the default method!
 *
 ***/

void CSimpleLWDoc::Dispose()

{
    inherited::Dispose();
}

/*****************************************************************
 * DoCommand
 *
 *  This is the heart of your document.
 *  In this method, you handle all the commands your document
 *  deals with.
 *
 *  Be sure to call the default method to handle the standard
 *  document commands: cmdClose, cmdSave, cmdSaveAs, cmdRevert,
 *  cmdPageSetup, cmdPrint, and cmdUndo. To change the way these
 *  commands are handled, override the appropriate methods instead
 *  of handling them here.
 *
 ***/

void CSimpleLWDoc::DoCommand(long theCommand)

{
    switch (theCommand) {

        /* your document commands here */

        default:    inherited::DoCommand(theCommand);
                    break;
    }
}
/*****************************************************************
```

```
 * Activate
 *
 *  Your document is becoming active-- the front window.
 *
 *  In this method you can enable menu commands that apply only when
 *  your document is active.
 *
 *  Be sure to call the default method to get the default behavior.
 *  The default method enables these commands: cmdClose, cmdSaveAs,
 *  cmdSave, cmdRevert, cmdPageSetup, cmdPrint, cmdUndo.
 *
 ***/

void CSimpleLWDoc::Activate(void)

{
    inherited::Activate();
}

/*****************************************************************
 * Deactivate
 *
 *  Your document is becoming inactive-- another window is in front
 *
 *  In this method you can disable menu commands that don't apply
 *  when your document isn't active.
 *
 *  Be sure to call the default method to get the default behavior.
 *  The default method disables these commands: cmdClose, cmdSaveAs,
 *  cmdSave, cmdRevert, cmdPageSetup, cmdPrint, cmdUndo.
 *
 ***/

void CSimpleLWDoc::Deactivate(void)

{
    inherited::Deactivate();
}
```

```
/*****************************************************************
 * NewFile
 *
 *  When the user chooses New from the File menu, the CreateDocument()
 *  method in your Application class will send a newly created document
 *  this message. This method needs to create a new window, ready to
 *  work on a new document.
 *
 *  Since this method and the OpenFile() method share the code for
 *  creating the window, you should use an auxiliary window-building
 *  method.
 *
 ***/
void CSimpleLWDoc::NewFile(void)

{
    Str255      wTitle;         /* Window title string          */
    short       wCount;         /* Index number of new window   */
    Str63       wNumber;        /* Index number as a string     */
        /**
         ** BuildWindow() is the method that
         ** does the work of creating a window.
         ** It's parameter should be the data that
         ** you want to display in the window.
         ** Since this is a new window, there's nothing
         ** to display.
         **
         **/

    BuildWindow(NULL);

        /**
         ** Send the window a Select() message to make
         ** it the active window.
         **/

    itsWindow->GetTitle(wTitle);  /* Append an index number to the */
    wCount = gDecorator->GetWCount();   /*   default name of the
                                            window        */
    NumToString((long)wCount, wNumber);
    ConcatPStrings(wTitle, (StringPtr) "\p-");
    ConcatPStrings(wTitle, wNumber);
    itsWindow->SetTitle(wTitle);
```

```
    itsWindow->Select();
}

/***
 * OpenFile
 *
 *  When the user chooses Open… from the File menu, the
 *  OpenDocument() method in your Application class will
 *  let the user choose a file and then send a newly created
 *  document this message. The information about the file is
 *  in the SFReply record.
 *
 *  In this method, you need to open the file and display its contents
 *  in a window. This method uses the auxiliary window-building method.
 *
 ***/

void CSimpleLWDoc::OpenFile(SFReply *macSFReply)

{
    CDataFile    *theFile;
    Handle       theData;
    Str63        theName;
    OSErr        theError;

        /**
         ** Create a file and send it a SFSpecify()
         ** message to set up the name, volume, and
         ** directory.
         **
         **/

    theFile = new(CDataFile);
    theFile->IDataFile();
    theFile->SFSpecify(macSFReply);

        /**
         ** Be sure to set the instance variable
         ** so other methods can use the file if they
         ** need to. This is especially important if
         ** you leave the file open in this method.
         ** If you close the file after reading it, you
```

```
     ** should be sure to set itsFile to NULL.
     **
     **/

itsFile = theFile;

    /**
     ** Send the file an Open() message to
     ** open it. You can use the ReadSome() or
     ** ReadAll() methods to get the contents of the file.
     **
     **/

theError = theFile->Open(fsRdWrPerm);

    /**
     ** Check to see if we were able to open
     ** the file. Send the error handler
     ** a CheckOSError() message. If there was
     ** an error, CheckOSError returns false
     ** and reports the error in an alert.
     ** The default error message displays the
     ** error number.
     ** You can use Estr resources to customize
     ** the error message.
     **
     ** Note that we send ourselves a Dispose()
     ** message. Since we're not going to open,
     ** we should get rid of the object.
     **/

if (!gError->CheckOSError(theError)) {
    Dispose();
    return;
}

    /**
     ** Make sure that the memory request to read
     ** the data from the file doesn't use up any
     ** of our rainy day fund and that the GrowMemory()
     ** method (in the application) knows that it's OK
     ** if we couldn't get enough memory.
     **
     **/
```

```
gApplication->RequestMemory(FALSE, FALSE);
theFile->ReadAll(&theData);    /* ReadAll() creates the handle */


    /**
     ** If there isn't enough memory to open,
     ** post the error (should be -108)
     ** and get rid of ourselves.
     **
     **/

if (theData == NULL || GetHandleSize(theData) > 10240L) {
    ParamText("\pThe file is too big.", "", "", "");
    PositionDialog('ALRT', 128);
    InitCursor();
    Alert(128, NULL);

    if (theData != NULL)
        DisposHandle(theData);
    Dispose();              /* Don't leave this object around */
    return;
}


BuildWindow(theData);

    /**
     ** In your application, you'll probably store
     ** the data in some form as an instance variable
     ** in your document class. For this example, there's
     ** no need to save it, so we'll get rid of it.
     **
     **/
DisposHandle(theData);

    /**
     ** In this implementation, we leave the file
     ** open. You might want to close it after
     ** you've read in all the data.
     **
     **/
```

```
    itsFile->GetName(theName);
    itsWindow->SetTitle(theName);
    itsWindow->Select();/* Don't forget to make the window active  */

}

/**********************************************************************
 * BuildWindow
 *
 *  This is the auxiliary window-building method that the
 *  NewFile() and OpenFile() methods use to create a window.
 *
 *  In this implementation, the argument is the data to display.
 *
 ***/

void CSimpleLWDoc::BuildWindow (Handle theData)

{
    CScrollPane     *theScrollPane;
    CSimpleLWPane   *theMainPane;
    Rect            sizeRect;

        /**
         ** First create the window and initialize
         ** it. The first argument is the resource ID
         ** of the window. The second argument specifies
         ** whether the window is a floating window.
         ** The third argument is the window's enclosure; it
         ** should always be gDesktop. The last argument is
         ** the window's supervisor in the Chain of Command;
         ** it should always be the Document object.
         **
         **/

    itsWindow = new(CWindow);
    itsWindow->IWindow(BASE_WINDOW, FALSE, gDesktop, this);

        /**
         ** After you create the window, you can use the
         ** SetSizeRect() message to set the window's maximum
         ** and minimum size. Be sure to set the max & min
         ** BEFORE you send a PlaceNewWindow() message to the
```

```
      ** decorator.
      **
      ** The default minimum is 100 by 100 pixels. The
      ** default maximum is the bounds of GrayRgn() (The
      ** entire display area on all screens.)
      **
      **/
SetRect(&sizeRect, 100, 100, 600, 500);
itsWindow->SetSizeRect(&sizeRect);

theScrollPane = new(CScrollPane);

    /**
     ** You can initialize a scroll pane two ways:
     **     1. You can specify all the values
     **        right in your code, like this.
theScrollPane->IScrollPane(itsWindow, this, 10, 10, 0, 0,
                           sizELASTIC, sizELASTIC,
                           TRUE, TRUE, TRUE);
     **     2. You can create a ScPn resource and
     **        initialize the pane from the information
     **        in the resource as we do here:
     **
     **/

theScrollPane->IViewRes('ScPn', BASE_PANE, itsWindow, this);

    /**
     ** The FitToEnclFrame() method makes the
     ** scroll pane be as large as its enclosure.
     ** In this case, the enclosure is the window,
     ** so the scroll pane will take up the entire
     ** window.
     **
     **/

theScrollPane->FitToEnclFrame(TRUE, TRUE);

    /**
     ** itsMainPane is the document's focus
     ** of attention. Some of the standard
     ** classes (particularly CPrinter) rely
     ** on itsMainPane pointing to the main
```

```
      ** pane of your window.
      **
      ** itsGopher specifies which object
      ** should become the gopher. By default
      ** the document becomes the gopher. It's
      ** likely that your main pane handles commands
      ** so you'll almost always want to set itsGopher
      ** to point to the same object as itsMainPane.
      **
      ** Note that the main pane is the
      ** panorama in the scroll pane and not
      ** the scroll pane itself.
      **
      **/

itsMainPane = new(CSimpleLWPane);
itsGopher = itsMainPane;

    /** The FitToEnclosure() method makes the pane
      ** fit inside the enclosure. The inside (or
      ** interior) of a scroll pane is defined as
      ** the area inside the scroll bars.
      **/

((CSimpleLWPane*)itsMainPane)->ISimpleLWPane(theScrollPane, this);

    /**
      ** Send the scroll pane an InstallPanorama()
      ** to associate our pane with the scroll pane.
      **
      **/

theScrollPane->InstallPanorama((CPanorama*)itsMainPane);

if (theData)
    /* do something here if this window contains data */;


    /**
      ** Let Decorator place window on screen
      ** ...mostly useful for multiple windows
      **
      **/
```

```
    gDecorator->PlaceNewWindow(itsWindow);
}

/**********************************************************************
 * DoSave
 *
 *  This method handles what happens when the user chooses Save from
 *  the File menu. This method should return TRUE if the file save
 *  was successful. If there is no file associated with the document,
 *  you should send a DoSaveFileAs() message.
 *
 ***/

Boolean CSimpleLWDoc::DoSave(void)

{
        /**
         ** If you closed your file in your NewFile() method,
         ** you'll need a different way than this to determine
         ** if there's a file associated with your document.
         **
         **/

    if (itsFile == NULL)
        return(DoSaveFileAs());
    else {

        /**
         ** In your application, this is where you'd
         ** write out your file. if you left it open,
         ** send the WriteSome() or WriteAll() mesages
         ** to itsFile.
         **
         **/

        dirty = FALSE;          /* Document is no longer dirty   */
        gBartender->DisableCmd(cmdSave);
        return(TRUE);           /* Save was successful           */
    }
}
```

```
/**********************************************************************
 * DoSaveAs
 *
 *  This method handles what happens when the user chooses Save As…
 *  from File menu. The default DoCommand() method for documents
 *  sends a DoSaveFileAs()message which displays a standard put file
 *  dialog and sends this message. The SFReply record contains all
 *  the information about the file you're about to create.
 *
 ***/

Boolean CSimpleLWDoc::DoSaveAs(SFReply *macSFReply)

{
        /**
         ** If there's a file associated with this document
         ** already, close it. The Dispose() method for files
         ** sends a Close() message to the file before releasing
         ** its memory.
         **
         **/

    if (itsFile != NULL)
        itsFile->Dispose();


        /**
         ** Create a new file, and then save it normally.
         **
         **/

    itsFile = new(CDataFile);
    ((CDataFile *)itsFile)->IDataFile();
    itsFile->SFSpecify(macSFReply);
    itsFile->CreateNew(gSignature, 'TEXT'); /* NB: this must be
                                               changed */
    itsFile->Open(fsRdWrPerm);

    itsWindow->SetTitle(macSFReply->fName);

    return( DoSave() );
}
```

```
/*****************************************************************
 * DoRevert
 *
 *  If your application supports the Revert command, this method
 *  should close the current file (without writing anything out)
 *  and read the last saved version of the file.
 *
 ***/

void CSimpleLWDoc::DoRevert(void)

{
}
```

```
/**********************************************************************
 * CSimpleLWPane.h
 *
 *  Pane class header for LaserWriter programming application.
 *
 *  Programming the LaserWriter - Addison-Wesley
 *  by David Holzgang
 *  © Copyright 1990 David Holzgang All Rights Reserved
 *
 **********************************************************************/

#define _H_CSimpleLWPane/* Include this file only once */

#include <Commands.h>
#include <CBartender.h>
#include <CDocument.h>
#include <CEditText.h>
#include <CPicture.h>

struct CSimpleLWPane : CPicture {

                /** Instance Variables  **/

                /** Contruction/Destruction **/
    void    ISimpleLWPane(CView *anEnclosure, CBureaucrat *aSupervisor);

                /** Draw **/
    void    Draw(Rect *theArea);

                /** Mouse and Keystrokes **/
    void    DoClick(Point hitPt, short modifierKeys, long when);
    Boolean     HitSamePart(Point pointA, Point pointB);

                /** Cursor **/
    void    AdjustCursor(Point where, RgnHandle mouseRgn);

                /** Scrolling **/
    void    ScrollToSelection(void);
};
```

```
/**********************************************************************
 * CSimpleLWPane.c
 *
 * Pane methods for LaserWriter programming application.
 *
 * Programming the LaserWriter - Addison-Wesley
 * by David Holzgang
 * © Copyright 1990 David Holzgang All Rights Reserved
 *
**********************************************************************/

#include "CSimpleLWPane.h"

/*** Global Variables ***/

/*** Class Constants ***/

/**********************************************************************
 * ISimpleLWPane
 *
 * Initialize a LWPane object.
 *
 ***/

void CSimpleLWPane::ISimpleLWPane(
                                CView           *anEnclosure,
                                CBureaucrat     *aSupervisor
                                )
{
    Rect        margin;

    CPicture::IPicture(anEnclosure, aSupervisor, 1, 1, 0, 0,
                    sizELASTIC, sizELASTIC);
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the pane a little margin.
         ** Each element of the margin rectangle
         ** specifies by how much to change that
         ** edge. Positive values are down and to
         ** right, negative values are up and to
         ** the left.
         **
         **/
```

```
    SetRect(&margin, 2, 2, -2, -2);
    ChangeSize(&margin, FALSE);
}


/***
 * Draw {BASIC}
 *
 * Draw the contents of the Pane. The area parameter, specified in
 * the pane's Frame coordinates, indicates the portion of the Pane
 * which needs to be drawn.
 ***/

void    CSimpleLWPane::Draw(
        Rect            *area)
{

    MoveTo(frame.left, frame.top); /*XXX Draws an X from corner*/
    LineTo(frame.right, frame.bottom);  /*to corner in the Pane*/
    MoveTo(frame.right, frame.top);
    LineTo(frame.left, frame.bottom);

}


/***
 * DoClick
 *
 * The mouse went down in the pane.
 * In this method you do whatever is appropriate for your
 * application. HitPt is given in frame coordinates. The other
 * parameters, modiferKeys and when, are taken from the event
 * record.
 *
 * If you want to implement mouse tracking, this is the method
 * to do it in. You need to create a subclass of CMouseTask and
 * pass it in a TrackMouse() message to the pane.
 *
 ***/

void CSimpleLWPane::DoClick(Point hitPt, short modifierKeys, long when)
```

```
{
    /* what happens when the mouse goes down */
}


/***
 * HitSamePart
 *
 * Test whether pointA and pointB are in the same part.
 * "The same part" means different things for different
 * applications.In the default method, "the same part" means
 * "in the same pane." If you want a different behavior,
 * override this method. For instance, two points might be
 * in the same part if they're within n pixels from each
 * other.
 *
 * PointA and pointB are both in frame coordinates.
 *
 ***/

Boolean  CSimpleLWPane::HitSamePart(Point pointA, Point pointB)

{
    return inherited::HitSamePart(pointA, pointB);
}


/***
 * AdjustCursor
 *
 * If you want the cursor to have a different shape in your pane,
 * do it in this method. If you want a different cursor for
 * different parts of the same pane, you'll need to change the
 * mouseRgn like this:
 *  1. Create a region for the "special area" of your pane.
 *  2. Convert this region to global coordinates
 *  3. Set the mouseRgn to the intersection of this region
 *     and the original mouseRgn: SectRgn(mouseRgn, myRgn,
 *     mouseRgn);
 *
 * The default method just sets the cursor to the arrow. If
 * this is fine for you, don't override this method.
 *
 ***/
```

```
void CSimpleLWPane::AdjustCursor(Point where, RgnHandle mouseRgn)

{
    inherited::AdjustCursor(where, mouseRgn);
}


/***
 * ScrollToSelection
 *
 * If your pane is based on a Panorama (as this example is),
 * you might want to define what it means to have a selection
 * and what it means to scroll to that selection.
 *
 ***/

void CSimpleLWPane::ScrollToSelection(void)

{
    /* scroll to the selection */
}
```

Bibliography

This is a list of books and other resources about both the Macintosh and PostScript. Each list is broken down into two sections: The first provides standard reference materials about the subject, and the second provides useful or interesting extensions on the basic information. In my opinion, you probably cannot continue much beyond this book without most of the standard reference materials. I have them all and referred to them throughout the process of writing this book. The extensions, on the other hand, are just my personal choices of books that I use or have used to find solutions—in many cases, particularly on the Macintosh, out of a very large number of possibilities. Inclusion in this list implies nothing except my personal endorsement, and exclusion implies nothing at all.

References about Macintosh Programming

Inside Macintosh, Volumes I to V, (Addison-Wesley). As is always true when programming the Macintosh, all of the volumes are useful at one point or another; you really can't program the Macintosh without a full set of these. For System 7.0, of course, you must add the new Volume VI.

Macintosh Technical Notes. Obviously, not all the Technical Notes are required for programming the LaserWriter, but the complete list of those that deal with the LaserWriter in one way or another is fairly extensive—too extensive to list here and likely to be out of date by the time you read this, anyway. Use the listings under "LaserWriter" and "Printing Manager" in the latest Index of the Technical Notes for a

complete list. The most important ones are Technical Notes #72 and #91, which give the basics of LaserWriter programming; Technical Note #95, which covers adding information to the Printing Manager dialog boxes; and Technical Note #192, which updates some of the previous information about the LaserWriter to correspond with the LaserWriter version 5.0 (and later) device drivers.

Apple LaserWriter Reference, Addison-Wesley, 1988. A book about how the LaserWriter works. This includes all the PostScript operators that are specific to the LaserWriter family of devices, including all the file operators for the IINTX; an important resource if you are going to program the LaserWriter.

Think C 4.0 Manual, Symantec. This manual provides the complete definitions and relations of all the basic classes and methods in Think C 4.0; in particular, it documents the CPrinter class, which you use as a basis for the LaserWriter programming methods in this book.

▶ Macintosh Programming Techniques

Knaster, Scott. *Macintosh Programming Secrets,* Addison-Wesley, 1988. This is, in my opinion, the best single reference on advanced Macintosh programming techniques and maybe the only one written in English (as opposed to "techno-speak").

Mark, Dave. *Macintosh Programming Primer,* Volume 2, Addison-Wesley, 1990. This continues and extends the previous excellent volume to include more complex tasks and Think C 4.0 object-oriented materials; it's a good introduction to Think C 4.0, including the object-oriented extensions if you are not familiar with these.

▶ References about PostScript Programming

Adobe Systems. *PostScript Language Reference Manual, Second Edition,* Addison-Wesley, 1990. The basic reference that defines the PostScript language. The second edition of this valuable reference includes many new operators from Level 2. It is fully revised with improved definitions and descriptions of interpreter processing.

Adobe Systems. *PostScript Language Program Design,* Addison-Wesley, 1988. An advanced manual that gives good advice on how to use PostScript for many common tasks.

Holzgang, David. *PostScript Programmer's Reference Guide,* Scott, Foresman, 1989. A complete language reference with operator definitions and sample programs. Includes color operators and a complete discussion of device-specific operations and operators.

Adobe Systems. *PostScript Language Supplement,* Adobe Systems. Every PostScript device, including the Apple LaserWriter family, has an individual language supplement that is normally supplied with the device, either as a part of the device manual or as a separate manual. These define the PostScript operators that are specific to the given device, using the same format and conventions that are used in the *PostScript Language Reference Manual.* For the LaserWriter family of devices, the same information is provided in the *Apple LaserWriter Reference* manual. A complete selection of device-specific operators is also provided in the *PostScript Programmer's Reference Guide.*

Adobe Systems also publishes a set of Technical Notes that covers a variety of subjects for the PostScript developer. You can get a current list from the Developer Support Group at Adobe. The most valuable of these monographs are

- *Document Structuring Conventions Specification, Version 2.1*
- *Adobe Font Metric Files Specification, Version 3.0*
- *PostScript Printer Description Files Specification, Version 3.0*

In all cases, the versions listed above are the most recent that I had at the time of writing; obviously, they are subject to change. You can get the most recent copies of these documents, along with a full list of the available information, from

Developer Support Group
Adobe Systems, Inc.
1585 Charleston Road
P.O. Box 7900
Mountain View, CA 94039-7900

▶ PostScript Programming Techniques

Smith, Ross. *Learning PostScript: A Visual Approach,* Peachpit Press. An excellent book that guides beginners into PostScript programming by teaching graphic concepts and techniques. It is specifically designed to introduce non-programmers to PostScript. The only drawback is that it is oriented toward the IBM environment; however, you can easily adapt the techniques in Chapter 4 to work with the examples in this book.

Holzgang David. *Understanding PostScript Programming, Second Edition*, Sybex, 1988. A basic tutorial on programming in PostScript. This book assumes that you know some programming language (such as Basic or C). It teaches progressively all of the most important PostScript programming functions and techniques.

Roth, Steve. *Real World PostScript*, Addison-Wesley, 1988. This book provides the best coverage of advanced PostScript issues, such as halftone processing and color processing, that is currently available. It covers a wide range of topics and has some interesting and enlightening examples.

▶ PostScript Utilities

If you are going to do any PostScript programming, or indeed if you run many PostScript files, you will eventually want to improve the error reporting for your LaserWriter or other PostScript device, as we discussed in the main section of the book. *PinPoint Error Reporter* is an inexpensive commercial error reporting package that provides a complete description of the error, where it happened in your program, and the state of the printer at the time of the error. In my biased view (since I wrote it) it is the best commercial package for this function. It is available, along with other PostScript utilities and resources, from

Cheshire Group
321 South Main Street, Suite 36
Sebastopol, CA 95472
(707) 887-7510

In addition, if you are going to do any serious PostScript development, you should consider purchasing *Lasertalk* from Adobe Systems. This provides interactive access to your LaserWriter and also provides a complete PostScript programming environment, including online help and procedural debugging with breakpoints.

Index

⌘ (Command key or Apple key), 30, 281
* (asterisk), 387
%%BeginDocument: (body comment), 216
%%BeginFont: (body comment), 216
%%BeginProcSet: (body comment), 216
%%BeginSetup: (body comment), 216
%%BoundingBox: (header comment), 215, 221
%%CreationDate: (header comment), 214, 222
%%Creator: (header comment), 214, 222
%%DocumentFonts: (header comment), 215, 221
%%EndComments: (header comment), 215
%%EndProlog (body comment), 216
%%For: (header comment), 214
%%Page: (page comment), 211, 217
%%PageFonts: (page comment), 217
%%Pages: (header comment), 214
%%Title (header comment), 214, 222
%%Trailer (page comment), 217
% (percent sign)
  as comment indicator, 57, 210–211
  within strings, 62
%stderr file, 59, 385–386
%stdin file, 59, 385–386
%stdout file, 59, 385–386
; (semicolon), 278
<> (angle brackets), 57, 61
? (question mark), 387
\ (backslash), 60
{} (braces), 57, 63, 133
[] (brackets), 57, 62
() (parentheses), 57, 60, 61
/ (slash)
  as font name indicator, 327
  as name literal delimiter, 57, 68
| (vertical bar), 320

A
Addition in executable arrays, 63
Add operator, 64
Adobe Font Metric Files Specification (Adobe Systems), 316
Adobe Font Metrics (AFM), 316
Adobe Illustrator, 13
Adobe Systems, 427
Adobe Type Manager (ATM), 312
AFM (Adobe Font Metrics), 316
Aldus PageMaker, 13
aload operator, 146, 149
Alphabets, 317
Angle brackets (<>), 57, 61
AppendDITL code routine, 257–260
Apple key (command key), 30, 281
Apple LaserWriter printers. *See* LaserWriter printers.
Apple LaserWriter Reference (Addison-Wesley), 4, 337, 386, 389, 426

AppleTalk, xxi, 9, 14–19, 334
  connecting, 28
  controlling from Chooser dialog box, 21–22
  device name storage in, 22
  functions, 335–336
AppleTalk Network System Overview (Addison-Wesley), 15
AppleTalk Transaction Protocol (ATP), 16 (Fig.)
Arbitration (of job requests), 339
arc operator, 173
arcto operator, 174
Arrays
  delimiters for, 57
  executable, 63
  as PostScript objects, 62–63
Ascenders, 70–71
  POST resources in, 113
Ascending/descending order, 215
ASCII
  for encoding fonts, 320
  for encoding PostScript, 40
  returning, 126, 129
Asterisk (*) wildcard character, 387
atend indicator, 214, 215
ATM (Adobe Type Manager), 312
ATP (AppleTalk Transaction Protocol), 16 (Fig.)

B
Background printing, 21 (Fig.), 22, 79
  avoiding with PicComment, 104, 105, 109, 116
Background spooler printing, 16, 17–18, 329
Backslash (\), 60
\x string operators, 60
Backspace, 60
Backups, 275
banddevice operator, 104, 212
Baseline, 70
Batch mode, 30
Baud rates, 28, 32, 335
BDevLaser constant, 230
%%BeginDocument: (body comment), 216
%%BeginFont: (body comment), 216
%%BeginProcSet: (body comment), 216
%%BeginSetup: (body comment), 216
Binary object/token encoding, 40
Bitmapped fonts, 11–12, 71, 311–312
Bitmaps, 38, 118–119
Body comments, 213, 215–216
Boldface, 69–70
Boolean values, 58, 232–233
Bottlenecks, 14
  saving definitions as files, 25
  transformation of coordinates by, 192
%%BoundingBox: (header comment), 215, 221
Bounding box, 73, 174, 193, 223
Braces ({}), 57, 63, 133
Brackets ([]), 57, 62
BuildWindow method, 84–87
Bullet (octal character code), 60

C
Caching of font characters, 71, 120
Canon copiers, 3
Carriage return, 60. *See also* Newline (return or linefeed) character
Case sensitivity in PostScript
  code, 132
  files, 388
  keywords, 211
Cents sign (octal character code), 60
Character coordinate system, 72
Character mapping, 317
Character set encoding, 317, 320
Character width, 72
CharStrings dictionary, 314, 317
Cheshire Group, 304, 428
Chooser dialog box, 21–22
Circles, 170–174
C language, xvii–xviii, 98
CLaser class, 231, 253–254, 256, 356–357, 364–366
CLaser.h header file code routine, 236–239, 262–263, 269–270, 357–358, 366–372
Class constants, 93, 95
Cleanup procedures in trailer section, 217
Clients (of sockets), 335, 338
clippath operator, 185
Clipping, 183–191
Clipping path, 184, 190
clipRect, 120
Clip regions, 118
Closed paths, 166–174
closepath operator, 169

Code, xviii, xix. *See also Appendix; individual named routines* and Speed
PostScript,
cautions on use of, 13, 157
to list device names, 387
to list fonts, 327, 389–390
need for structural discipline in, 39, 207
operators to avoid in, 104–105, 212
to search for fonts, 325–326
to test for fonts, 378
to test for Laser Prep version, 382–383
to test for spooler, 383–384
QuickDraw, techniques to avoid in, 118–120
Colon (:), 211, 278
Colors, 46, 175, 179
Columns, printing in, 150, 154–157
Command key (Apple key), 30, 281
Comments. *See also* PicComment
body, 213, 215–216
continuation, 212
for defining program structure, 208, 210
in EPS (Encapsulated PostScript), 220–222
header, 213–215
indicator for, 57
operators to avoid in, 212
page, 213, 216–218
in PostScript programs, 61–62, 297
structural, 208
Communications networks. *See also* AppleTalk
serial communications, 28–29, 334–335, 336–337
Composite object types, 58, 145
Conforming documents, 211
Continuation comments, 212
Coordinated fonts, 320
Coordinates, 49–52
fractional, 51
of grafPorts, 9
moving/scaling, 52–55, 191
of output devices, 52
PostScript versus QuickDraw, 78, 101, 104, 135, 192–193
in stringwidth operations, 160
Coordinate transformation matrix (CTM), 52
copypage operator, 104, 212, 292
CPrinter class, 231
%%CreationDate: (header comment), 214, 222
%%Creator: (header comment), 214, 222
CSimpleLWDoc class, 81–83, 349352
CSimpleLWPane class, 87–89, 93–95
CTM (coordinate transformation matrix), 52, 146
Ctrl-D (end-of-transmission character), 30, 31
Curly braces. *See* Braces
Cur_Prntr code routine, 240–242, 354–355
Current color, 175
Current dictionary, 69
Current graphics state, 55–56
currentlinecap operator, 56
currentmatrix operator, 146
Current page, 47
Current path, 47
Current point, 47
Curves, 170–174. *See also* Graphics

D

DashedLine (PicComment), 291, 292
DashedStop (PicComment), 291, 292
Datagram Delivery Protocol (DDP), 16 (Fig.)
Data Terminal Ready (DTR), 29
DDP (Datagram Delivery Protocol), 16 (Fig.)
Debugging
aiding with unnested procedure calls, 257
interactive mode for, 31
with LaserTalk, 32
PostScript routines, 132–133
Defaults, PostScript versus QuickDraw, 193
Default user space, 50
def operator, 133
Derived fonts, 312
Descenders, 70
Descending/ascending order, 215
Destination application, 219
Device independence
of PicComment method, 102, 116–118
of PostScript, 40–41

Device independence *(continued)*
of printing, 20, 26–28
Device control operators, 42
Device identification, 229–231
Device resolution, 44–45
Devices
displaying names of, 386–388
as network nodes, 335
obtaining information about, 229–231, 360–381
Device space, 52
Diablo 630 printer (emulation of), 8, 31, 337
Dialog boxes, 246–249
adding information to, 252–264
altering, 249–251
Diamond shapes, 183–191
Dictionaries, 67–69
defining and loading, 294–300
downloading conditionally, 305
downloading permanently, 306
fonts as, 313
importance of removing, 145
Laser Prep, 79, 289–290
personal, 123, 209
procedures in, 139–145
Dictionary stack, 66
dictstackunderflow error, 69
Display PostScript Programming (Addison-Wesley), 316
Dispose method, 99
DITL (LaserWriter driver resource), 276–277
DITL.make.r code routine, 263
DLOG.make.r code routine, 358–359
%%DocumentFonts: (header comment), 215, 221
Documents as programs, 207
Document Structuring Conventions, 209–210, 218
DoPrint method, 253–254
Dot-matrix printers, 43, 44
Dots per inch (dpi), 44
Double dagger (octal character code), 60
Downloading
dictionaries, 305–306
fonts, 121, 123, 221, 225, 322–324, 328
Laser Prep dictionary, 289–290
permanent and temporary, 122–123, 322–324
of program prologues, 211
dpi (dots per inch), 44
"DRAFT" message
adding dialog box option for, 252–253, 264, 273–275
defining and loading dictionary for, 294–300
Draft print mode, 77–78
DRAM (dynamic, random-access memory), 4, 5, 7
DrawChar, 119
Draw method code, 89–90, 95–98, 270–273
with new dictionary, 295, 297–299
with PicComments, 105–111, 129–131, 297–300
Drivers. *See* Printer drivers, LaserWriter drivers
DTR (Data Terminal Ready), 29
DTR/DSR handshake method, 336–337
Dynamic languages, 39–40

E
Emulation modes, 30–31, 337
for Diablo 630, 8
for HP LaserJet Plus, 8
Encapsulated PostScript. *See* EPS (Encapsulated PostScript)
Encoding
of fonts, 317–318, 320
of PostScript output, 40
Encoding array, 314, 317
End-of-transmission character, 30
%%EndComments: (header comment), 215
EndFormsPrinting (PicComment), 292
%%EndProlog (body comment), 216
Enter key. *See* Newline (return or linefeed) character
EPS (Encapsulated PostScript), 207
code routine for graphic placement, 223–224
file format, 220–222
need for, 219
transformations, 223–225
EPSF keyword, 213
EraseOval, 119
erasepage operator, 104, 212
EraseRect, 119
Erasing, caution on, 119, 193
Error messages
through AppleTalk, 80
in interactive mode, 31
through PREC resource, 301–304

Error messages *(continued)*
VMerror, 327
code –415, 243, 244, 245
code –4096, 245
code –4097, 245
code –4098, 245
code –4099, 245
code –4100, 245
code –4101, 244, 245
code –8130, 117, 245
code –8132, 245
code –8133, 245
code –8149, 246
code –8150, 246
code –8158, 246, 282
code –8159, 246
code –8160, 246
Errors
caution on handling in Printing Manager calls, 78, 79, 116
dictstackunderflow, 69
importance of testing for, 132
PostScript standard output file for (%stderr file), 59, 385–386
sample code for handling, 93, 243–246
undefined, 68
Escape fonts, 293
Estr.Make.r code routine, 244–245
Examples of code. *See the Appendix and individual named routines*
exch operator, 165
Executable arrays, 63
Execution stack, 66
Exiting server loop, 46, 306
ExitServer keyword, 213
exitserver operator, 104, 212

F
Feedback, provision of, 39–40
File-resident fonts, 323
Files, 59
naming, 387–388
standard PostScript, 59, 385–388
File streams, 385–388
fill operator, 47, 49, 179, 190
Film output devices, 44
findfont operator, 314, 325
FOND resource, 312–313, 318
Font cartridges, 386
Font coordination, 320
FontDirectory, 314
Font families, 69
FontID (PostScript object), 59
Font Manager, 8, 10–12
and the FOND resource, 312–313
Font metrics, 70, 314–316
and Adobe Font Metrics (AFM), 316
for the Macintosh, 315
for PostScript, 316
Font operators, 42
Fonts. *See also* Font Manager; Text
bitmapped, 11–12, 71, 311–312
caching, 71, 120
checking for presence of, 79
coordinated, 320
displaying information about, 203–206
downloaded, and virtual memory usage, 121, 328
downloading, 123, 322–324
downloading for EPS files, 221, 225
encoding, 317–318, 320
file-resident, 323
intrinsic versus derived, 312
in LaserWriter printers, 7–8
limiting number of, 120
listing, 325–327, 327, 389–390, 389–391
Macintosh, 312–313
measurements of, 70–71
metrics for, 72–73
naming, 320–321
oblique, 319–320
organization of, 72
outline, 71, 311–312
permanent, 323–324
PostScript, 69–71, 69–73, 313–314, 316, 318
PostScript Escape, 293
PostScript names of, 318–319
RAM-resident, 323
ROM-resident, 323, 324, 328
for screen versus printer, 11, 118
selecting, 318–322
selecting with PostScript code, 101
size menu for, 11 (Fig.)
testing for, 378–381
transient, 323–324
%%For: (header comment), 214
Foreground printing, 16–17, 79
Formfeed indicator, 60

FormsPrinting (PicComment), 292
for operator, 148–149
Fountains, 180–183
Fractional coordinates, 51
framedevice operator, 104, 212
frameRoundRect procedure, 173, 175
frRndRect procedure, 175, 178

G
"Garbage collection," 328
GetDeviceInfo, 239–240
GetDeviceStatus, 353–354
GetDraftStatus, 268–269
Glue routines, 343–348
gMarkDraft code routine, 267–268
grafPort, 9
  drawing pages into, 78
  initializing, 78
  printing, 13
Graphics. *See also* QuickDraw
  and color, 46, 175, 179
  curves, 170–174
  and diamond shapes, 183–191
  for "fountain" effect, 180–183
  and line operations, 158–166, 193–195
  as PostScript objects, 43
  PostScript's capability for, 38, 41
  and rectangles, 166–170
  and shading, 175–180
Graphics operators, 42
Graphics state, 55–56, 161–166
Graphics state stack, 66–67
grestoreall operator, 104, 212
grestore operator, 67, 164, 175, 178
gsave operator, 66, 164, 175, 178

H
Half-open connections, 339
Halftones, 175
HandleFeedback method, 363–364
Handshake methods, 336–337
Hard disks, 323, 386, 388–391
Hardware
  for communications control, 29, 32
  printers, 4–8
Header comments, 213–215, 217–218
Headers, 208, 294–306
Hewlett-Packard LaserJet printers, 3, 8, 31, 229, 337
Hexadecimal string delimiters, 57, 61
Hints (for outline fonts), 312
History of LaserWriter printers, 2–3
Holzgang, David, 426, 428
Horizontal tab indicator, 60
HP LaserJet printers, 3, 8, 31, 229, 337
HSB color model, 175

I
IINT/IINTX printers, 5, 29, 38, 337, 385, 386
IISC printers, xi, 5, 6
image operator, 47
Imagesetter output devices, 44
ImageWriter, 77
index operator, 64
Index values, 58
  for arrays, 63
  for octal character codes, 61
  for stacks, 66
  for strings, 59–60
initclip operator, 104, 212
initgraphics operator, 104, 212
initmatrix operator, 104, 212
Ink-jet printers, 44
Input file stream (%stdin file), 59, 385–386
Inside AppleTalk (Addison-Wesley), 15
Inside Macintosh, 26, 103, 425
Integers, 58
Interactive mode, 31–32, 125
Interpreted languages, 38–39, 207
Interpreter, 38, 210
Intrinsic fonts, 312
Invert grafverb, 118
ISimpleLWDoc method, 232–236
Italics, 69
  versus oblique fonts, 319–320
Item hooks, 251

J
Job dialog, 247, 249–251, 252–264
Job dialog box, 12, 23, 247

K
Kerning and kerning pairs, 315
Keys (in PostScript dictionaries), 68
Keywords (in PostScript structure comments), 211, 213, 218
Knaster, Scott, 103, 426

L
Label (in %%Page: program comment), 217
Larger Print Area option, 122, 190, 248
LaserJet printers (Hewlett-Packard), 3, 8, 31, 229, 337
Laser Prep, xv, xvii, 9, 14, 280–281
  and control variables, 248
  dictionaries, 79, 289–290
  dictionary, 79
  downloading, 289–290
  format of, 282–289
  loading and downloading, 123
  and md dictionary, 79, 280, 284, 305, 381
  as PostScript prologue, 208, 280
  saving output as files, 25, 281–282
  status restoration and, 121
  testing for version of, 381–383
  translations and functions of, 281–285
  versions of, 118, 290, 305
LaserTalk, 32, 428
LaserWriter driver, 9, 14
  bypassing, 333–334
  dialog boxes, 246–249, 264–275
  editing resources of, 275–280
  and font processing, 318, 320, 321–324, 329
  process overview of, xv, xvii, 78–80
  version of, 21, 117–118, 118, 293–294
LaserWriter files, 249
LaserWriter Font Utility, 123, 324, 389
LaserWriter IINT/IINTX printers, 5, 29, 38, 337, 385, 386
LaserWriter IISC printers, xi, 5, 6
LaserWriter II series printers, 5
LaserWriter Plus printers, 4, 337
LaserWriter printers. *See also* AppleTalk; Laser Prep; LaserWriter driver; LaserWriter programming
  advantages of, xii
  cautions on programming for, 13, 26
  communications modes of, 28–29, 334–337
  history of, 2–3
  levels of support for, xii–xiv
  operating modes of, 30–33
  overview of, xi–xii, 4–8
  Personal models, xi, 5–6, 337
LaserWriter programming. *See also* Code
  altering Printing Manager dialogs, 249–251
  identifying LaserWriter printers, 229–246
  job dialog modifications, 252–264
  Laser Prep, 280–281
  Laser Prep file downloading, 289–290
  Laser Prep file formats, 285–289
  Laser Prep translations, 281–285
  LaserWriter driver dialog boxes, 246–249, 264–275
  LaserWriter driver resources, 275–280
  techniques to avoid, 118–120
LaserWriter Reference (Addison-Wesley), 28, 31
LaserWriter SC printers, xi
Learning PostScript: A Visual Approach (Peachpit Press), 427
Left side bearing, 73
Level 2 (PostScript), 180, 328
LIFO (last-in, first-out) stacks, 65
linecap parameter, 56
Linefeed. *See* Newline (return or linefeed) character
linejoin parameter, 56
LineLayoutOff (PicComment), 291, 292
LineLayoutOn (PicComment), 291, 292
Lines. *See* Graphics
lineto operator, 47, 48, 158
Literals (PostScript objects), 43
LocalTalk, 6, 14–15
Loops, 148–149

M
Macintosh. *See also* LaserWriter printers
  advantages of, 26
  required familiarity with, xvii
Macintosh fonts, 312–313
Macintosh Programming Primer (Addison-Wesley), xvii, 426
Macintosh Programming Secrets (Addison-Wesley), 103, 426
Macintosh Technical Notes, 425–426
Mapping, 317
Mark, Dave, xvii, 426
Mark (PostScript object), 59, 62
Marks (for image creation), 46
Mathematical operators, 42

matrix operator, 146
md (Laser Prep dictionary), 79, 280, 284, 305, 381
Memory requirements, 78. *See also* Virtual memory
Messages, 30, 245–246. *See also* Error messages
Modes of operation, 30–32, 32
Monospaced fonts, 70, 314–315
Motorola processors, 4, 5
moveto operator, 47, 48, 158
MultiFinder, 17, 18
  choosing background printing in, 22
MyJobDialog method, 254–255
MyJobDlgInit procedure, 255–256, 265–266
MyJobItems code routine, 261, 266–267

N
Name Binding Protocol (NBP), 16 (Fig.), 336
Name literals, 57, 68
Names (PostScript objects), 43, 59
  as keys, 68
Naming
  files, 387–388
  fonts, 320–321
  PostScript output files, 25
  printers, 22
  workstations, 22
NBP (Name Binding Protocol), 16 (Fig.), 336
Network-visible entities (NVE), 335
Networks. *See also* Workstations
  devices as nodes on, 335
  and serial communications, 28–29, 334–335, 336–337
Newline (return or linefeed) character
  \n as indicator for, 60
  hexadecimal indicator for, 295
  as separator, 57
newpath operator, 47, 185, 190
Nodes, 335
Nominal objects, 58–59
Non-conforming documents, 211
NotXOR transfer mode, 118
nulldevice operator, 104, 212
Null (PostScript object), 59
Number, in %%Page: program comment, 217
Numeric objects, 58
NVE (network-visible entity), 335

O
Objects, test, 1–2
Objects (PostScript), 43, 56, 58–61
  arrays and procedures as, 62–63
  comments as, 61–62
  composite, 145
  notating, 57
Oblique fonts, 319–320
Octal character codes, 60–61
OffSetRect call, 118
Open paths, 166
Operands, 64, 67
  defining versus stacking, 165, 173
Operand stack, 67
Operating modes, 30–33
Operators
  file handling in, 385–388
  PostScript, 41–42
  to avoid using, 104–105
Origin, 49–50
  of character coordinate system, 72
  QuickDraw versus PostScript, 101
Outline fonts, 71, 311–312
Output file stream (%stout file), 59, 385–386

P
Page-description languages, 37–38, 207. *See also* PostScript
%%Page: (page comment), 211, 217
Page comments, 213, 216–218
%%PageFonts: (page comment), 217
Page orientation options, 22–23
%%Pages: (header comment), 214
Pages, 37
  creating, 46–47
  current, 47
  ending, 78
  opening, 78
Page Setup... dialog box, 12, 246–247
  options from, 22–23
pagestackorder code routine, 376–377
Painting, 46–49
PAPClose call, 339, 342
Paper size options, 22–23
PAPGlue.c code routine, 344–348
PAPGlue.h code routine, 344
PAPOpen call, 340–341, 343
PAP (Printer Access Protocol), 15–16, 336, 337–340
  call definitions, 340–343
  glue routines, 343–348

PAP *(continued)*
print job cycle, 338 (Fig.)
using calls of, 348–360
PAPRead call, 339, 341, 343
PAPStatus call, 342–343
PAPUnload call, 339, 342
PAPWrite call, 339, 341–342, 343
Parallel port, 334–335
Parentheses (()), 57, 60, 61
Passwords, 123
pathbbox operator, 185
Paths
clipping, 184
constructing, 47–48, 160
current, 47
open and closed, 166–174
painting, 48–49
Patterns, 119–120, 180
PDEF resources, 343
Pedestal sample application, 80
Pen, hiding, 105
Percent sign (%). *See also %%commands in Symbols section at front of index*
as comment indicator, 57, 210–211
within strings, 62
Percent sign/exclamation (%!), for initial program comments, 211, 213, 221
Percent signs/plus sign (%%+), for continuation comments, 212
Permanent downloading, 122–123, 322–324
Permanent fonts, 323
Personal dictionaries, 123, 209
Personal LaserWriter NT printers, xi, 5, 6, 337
Personal LaserWriter SC printers, xi, 5, 6
PicComment, 92–93
device-independence of, 102, 116–118
examples of, 105–111
usage, 102–105, 290
valid with LaserWriter driver, 291–292
PicPlyClo (PicComment), 291
PICT representation, 220, 221, 223
Picture comments, 9
Pictures
as collections of drawing commands, 9
efficiency of, 13
PinPoint Error Reporter (Cheshire Group), 304, 428
Pitch, 70
Pixels, 44, 193–194
Point, current, 47
Points (printers' measurement), 11, 50, 70
PolyBegin (PicComment), 291
PolyEnd (PicComment), 291
PolyIgnore (PicComment), 291
PolySmooth (PicComment), 291
pop operator, 146
Ports, RS232/422, 336
Post-fix notation, 64
POST resources, 112–114, 126, 128, 293
building from ResEdit, 373–377
font definition with, 324
PostScript0 file, 282–284
PostScript, 8, 10, 36–41. *See also* Code; EPS (Encapsulated PostScript); LaserWriter programming
adding to QuickDraw routines, 92–93, 100–101, 290–294
capturing output as file, 24–26, 281–282
cautions on use of, 13, 157
command formats, 64
device-independence of, 40–41
device management, 43–46
dictionaries, 67–69
fonts, 69–71, 69–73, 313–314, 316, 318
graphics operations, 158–195
importance of, xv–xvi, 3, 35
Level 2, 180, 328
memory management, 121–122
need for structural discipline in, 39, 207
objects, 43, 56–63
operators, 41–42
operators to avoid in, 104–105, 212
page structure, 46–56
printers not supporting, xi, 6
procedures, 133–157
program integration, 218–225
program structure, 207–218
stacks, 65–67
switch setting with, 336
testing, 125–133
text operations, 195–206
using in QuickDraw, 290–306
versus QuickDraw, 50, 192–195
PostScriptBegin (PicComment), 103, 121, 291
PostScriptEnd (PicComment), 103, 291

PostScriptFile (PicComment), 103, 104, 105, 107–108, 117, 291
PostScriptHandle (PicComment), 103, 105, 291
PostScript Language Program Design (Addison-Wesley), 195, 337, 386, 426
PostScript Language Reference Manual (Addison-Wesley), 426, 427
PostScript Language Supplement (Adobe Systems), 427
PostScript Printer Description (PPD) files, 391
PostScript Programmer's Reference Guide (Scott, Foresman), 389, 426, 427
PPD (PostScript Printer Description) files, 391
PrCloseDoc call, 78
PrClosePage call, 78
PrDlgMain routine, 251, 252
Precision Bitmap Alignment, 118–119
PREC (LaserWriter driver resource)
  code to create, 296–297, 301–303
  103 for PostScript headers or dictionary data, 294–297
  109 for redefining printer messages, 277–278, 279–280
Print Default call, 13
Print... dialog box, 12, 23, 247
Printer Access Protocol (PAP). *See* PAP (Printer Access Protocol)
Printer drivers, xv, 12, 14
  naming, 229
Printer fonts, 12, 118
Printer resource files, 12
Printers. *See also* LaserWriter printers; Spooler printing
  Diablo, 8, 31, 337
  dot-matrix, 43, 44
  ensuring correct information for, 77
  identifying, 229–230
  ink-jet, 44
  LaserWriter IINT/IINTX, 5, 29, 38, 337, 385, 386
  LaserWriter IISC, xi, 5, 6
  naming, 22
  Personal LaserWriter, xi, 5, 6, 337
Printing. *See also* Printing Manager
  background spooler, 16, 17–18, 209, 329, 329–330
  device-independence of, 20, 26–28
  foreground, 16–17
  process overview, 20–28
  remote spooler/server, 16, 18–19
  software components of, 8–9
Printing grafPort, 13
Printing Manager, 8, 12–13, 22–23
  calls, 76–78
  caution on bypassing of, 26–28
  and device-independence, 102, 116
  downloading, permanent and temporary, 122–123
  error messages sent by, 245–246
  errors corrected by, 245–246
  intercepting output of, 24–26
  and LaserWriter printers, xiv–xv
  optimizing functions of, 118–120
  Toolbox call for, xi, xii, xiii, xiv
  virtual memory management, 120–122
PrintMonitor, 17–18
PrintPage, 270–271
Print record testing, 231–243, 247–248, 360
Print servers, 15
Print spoolers, 209
PrJobDialog call, 23, 77
PrJobInit routine, 251
Procedures
  conversion to, 14
  creating, 133–157
  defining in dictionaries, 68, 139–145
  delimiters for, 57
  as named in Laser Prep dictionary, 284
  nesting of, 257
  as PostScript objects, 43, 63
Program control operators, 42
  page comments, 216–218
Program integration, 218–220
  EPS file format, 220–222
Programs. *See also* Code; LaserWriter programming; PostScript
  as documents, 207
Program structure, 207
  body comments, 215–216
  components of, 208–209
  conventions of, 209–213
  header comments, 213–215
Prologue, 208–209, 211, 215–216
PrOpen call, 76
PrOpenDoc call, 78
PrOpenPage call, 78
Proportional fonts, 70, 314–315
Protocols, 14, 29
PrStlDialog call, 22
PrStlInit routine, 251

PrStl record, 230
PrValidate call, 13, 76, 248
psb, pse, and psu procedures (from md dictionary), 286–289, 305, 381
PSRes193.make.r, code routine for, 111–112
PSResTest.make.r, code routines for, 126, 134, 136–137
PSText1 code routine, 134
PSText2 code routine, 136–137
PSText3 code routine, 139–141
PSText4 code routine, 142–145
PSText5 code routine, 147–149
PSText6 code routine, 150–154
PSText7 code routine, 158–161
PSText8 code routine, 161–165
PSText9 code routine, 165–166
PSText10 code routine, 166–169
PSText11 code routine, 170–174
PSText12 code routine, 175–179
PSText13 code routine, 180–182
PSText14 code routine, 183–185
PSText15 code routine, 186–191
PSText16 code routine, 195–199
PSText17 code routine, 200–201
PSText18 code routine, 201–203
PSText19 code routine, 203–206
psu definition, 287–288
PtrCtlCall, 293–294
Push-pop stacks, 65

Q
Quark XPress, 13
Query keyword, 213
Question mark (?) wildcard character, 387
QuickDraw, 8, 9. *See also* PicComment
  adding PostScript code to, 92–93, 290–294
  techniques to avoid in, 118–120
  translation of commands, xv, xvii, 14, 78–80, 280
  using PostScript in, 290–306
  versus PostScript, 50, 192–195
quit operator, 104, 212

R
RAM-resident fonts, 323
Raster-output devices, 43
Read-driven data transfer, 339
Real numbers, 58
Real World PostScript (Addison-Wesley), 428
Rectangles, 166–170
Reed, Cartwright, xvii
Reference point (of character coordinate system), 72
Regions, 118
Relative moves, 160
Remote spooler/server printing, 16, 18–19
repeat operator, 146
Resolution (on output devices), 44–45
  screen versus printer, 118–119
Resource fork, 108
ResourcePS (PicComment), 103, 104, 105, 117, 291
Resources
  of LaserWriter driver, 275–280
  managing, 120–122
  PDEF, 343
  POST, 112–114, 126, 128, 293, 324, 373–377
  releasing, 77, 98–99
restore operator, 121, 145
Results, 67
Return key. *See* Newline (return or linefeed) character
RGB color model, 175
Risers, 71
rlineto operator, 164
rmoveto operator, 160, 164
ROM-resident fonts, 323, 324, 328
ROM (read-only memory), 4, 5, 7
RotateBegin (PicComment), 292
RotateCenter (PicComment), 292
RotateEnd (PicComment), 292
rotate operator, 52
Roth, Steve, 428
rPage, 120
RS232 and 422 ports, 336

S
Samples of code. *See individual named routines*
save operator, 121, 145
Save (PostScript object), 59
savestate objects, 121
scale operator, 53–55, 191
Scan conversion, 45, 311–312
Screen fonts, 12, 118
  used in documents, 322
Script, 208–209, 211, 216
SCSI ports, 5, 6
Semicolon (;), 278
SendQuery method, 361–362, 379–380

Serial communications, 28–29, 334–335, 336–337
Server loop, 45–46, 122–123
  exiting, 306
  program execution outside of, 122
  and prologue placement, 209, 280
Server nodes, 335
Server PAP clients, 338
Server versus client, 338
Session-level protocols, 337
setcolortransfer operator, 104, 212
setdevice operator, 104, 212
setgray operator, 119, 178
setlinecap operator, 56
setlinewidth operator, 170
SetLineWidth (PicComment), 291
setmatrix operator, 104, 212
SetOrigin call, 118
setsccbatch operator, 104, 212
setscreen operator, 104, 212
settransfer operator, 104, 212
Shading, 175–180
show operator, 47, 101
showpage operator, 105, 212, 219, 223
Simple object types, 58
Slash (/)
  as font name indicator, 327
  as name literal delimiter, 57, 68
Smith, Ross, 427
Socket listeners, 335
Sockets, 335
Software. *See also* Code; LaserWriter programming; PostScript
  of communications protocols, 29, 32
  handshake method, 336–337
  interpreters as, 38–39
  of LaserWriter printer components, 8–9
  spoolers as, 329
Source application, 219
Spaces
  as handled in PostScript interpreter, 57
  in PostScript names, 68
Speed
  coding techniques for, 119–120
  and font caching, 71
  interpreter and, 39
  of spooler/servers, 116
Spooler, testing for, 383–384
Spooler printing, 16, 17–18, 209, 329, 329–330
  testing for, 383–384
Spooler/servers, 16, 18–19
  speed considerations for, 116
Spool print mode, 77, 120
Stack operators, 42
Stacks (in PostScript), 65–67
Starter sample application, 80
Status checks (for networks), 340
Status control operators, 42
Status requests, 80
  responses to, 279
  using PAP calls, 342–343, 348–360, 360–373
%stderr file, 59, 385–386
%stdin file, 59, 385–386
%stdout file, 59, 385–386
StringBegin (PicComment), 291
StringEnd (PicComment), 291
Strings, 57, 59–61
stringwidth operator, 160
STR (LaserWriter driver resources), 276, 293
stroke operator, 47, 48–49, 160, 169–170
Structural comments, 208
Style dialog, 246, 249–251
Style dialog box, 12, 22–23, 246–247
Style mapping, 319–320
Subpaths, 47
Switches (for communications options), 32
Sys/Start file, 388
System. *See also* Versions
System 6.0.4, 21
System 7.0
  Laser Prep bottlenecks and, 14
  and Laser Prep downloading, 123, 306
  and Laser Prep downsizing, 281
  and TrueType, 12, 310, 312
  viewing output results, 24–26
systemdict, 69

T
Tab character, 57, 60
TCL Demos folder, 80
TeachText, 25, 127
Temporary downloading, 122–123, 322–324

TestDraftStatus method, 273–274
Text. *See also* Fonts
  controlling, 200–203
  displaying, 195–200
  as graphics, 38, 43, 45
TextBegin (PicComment), 291
TextBox, 119
TextCenter (PicComment), 291
TextEnd (PicComment), 291
Text handles, 97
TextIsPostScript (PicComment), 103, 105, 291
Think C 4.0 language, xvii–xviii, 98
Think C 4.0 Manual (Symantec), 426
Tickler packets, 339
%%Title (header comment), 214, 222
Tokenizing, 43
Tokens (PostScript objects), 43, 56
Toolbox calls, xvii
TPrDlg record, 250, 252
TPrJob subrecord, 77
%%Trailer (page comment), 217
Trailers/trailer comments, 208, 213, 214, 217–218
transform operator, 195
Transient fonts, 323–324
translate operator, 52
True/false values, 58
TrueType, 12, 310, 312
Tuning, 45

U
Undefined errors, 68
Understanding PostScript Programming (Sybex), 428
Unlimited Downloadable Fonts option, 121
userdict, 69
User name retrieval, 243
User space, 50–52
Utilities
  LaserTalk, 32, 428
  PinPoint Error Reporter, 304, 428

V
Values (in PostScript dictionaries), 68
Version identifier, 213
Versions. *See also* System 7.0
  of Laser Prep, 118, 290, 305
    testing for, 381–383
  of LaserWriter, 117–118
  of LaserWriter driver, 21, 118, 293–294
  testing for, 242
Vertical bar (|), as coordinated font indicator, 320
Virtual memory
  checking status of, 142–145
  and composite objects, 58
  and fonts, 120, 327–328
  and page size reduction, 122
VMerror, 327

W
wait time setting, 30
White space characters, 57, 61
Wildcard characters, 387
Workstation nodes, 335
Workstation PAP clients, 338
Workstations, 15, 18
  connecting with PAP, 337
  naming, 22

X
X-axis, 49
  of default user space, 50
Xerox Corporation, 3
XON/XOFF handshake method, 336–337
XOR transfer mode, 118

Y
Y-axis, 49
  QuickDraw versus PostScript, 50, 101, 104, 135

Disk to accompany *Programming the LaserWriter*

The complete source code for all the programs and projects listed in *Programming the LaserWriter* by David A. Holzgang is available on one 800K 3 1/2" disk.

Equipment you will need:

Hardware: Macintosh® Plus, Macintosh SE, Macintosh Portable, members of Macintosh II family; LaserWriter or compatible PostScript® printer.

Software: To run: at least System Tools 5.0 (System 4.2/Finder 6.0) and LaserWriter/LaserPrep 5.2; software is compatible with System 7.0.

To compile the source code: Think C 4.0 and the Think C object libraries. A compiled version of the application SimpleLWApp is included.

To order the disk, call, fax, or complete the coupon below. Enclose a check or money order for $20.00 in U.S. funds, or complete credit card information below. California residents add 6% sales tax. Please add $5.00 for orders shipping overseas.

Order from: **Cheshire Group**
321 So. Main St., Suite 36
Sebastopol, CA 95472 USA
Phone: (707) 887-7510 Fax: (707) 887-2595

ORDER FORM

Please send me _____ (quantity) disks to accompany *Programming the LaserWriter* by David A Holzgang. Enclosed is a check or money order for $_____ or charge my credit card as follows:

Type: VISA ☐ ☐ Signature: _____

Number: _____ Expiration date: _____
(be sure to include all numbers)

Name: _____ Title: _____

Company: _____

Address: _____

City: _____ State: _____ Zip: _____

Country: _____

Other Books Available in the Macintosh Inside Out series

- **Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, Dan Shafer
  Here is the information you need to understand and use the power of MacApp, Apple Computer, Inc.'s official development environment for the Macintosh. The book discusses object-oriented concepts, using MPW with MacApp, the MacApp class library, and creating the Macintosh user interface. All examples are in Apple's Object Pascal language.
  576 pages, paperback
  $24.95, book alone, order number 09784
  $34.95, book/disk, order number 55062

- **C++ Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, Dan Shafer
  In this book you will find information on using MacApp with C++, the up-and-coming language for Macintosh development. The book covers object-oriented techniques, MPW, and the MacApp class libraries. All program examples are in C++.
  600 pages, paperback
  $24.95, book alone, order number 57020
  $34.95, book/disk, order number 57021

- **Elements of C++ Macintosh® Programming**
  Dan Weston
  Macintosh programmers will learn just what they need to take the step from C to C++ programming, the future of Macintosh development. The book covers the basics and then teaches how to design practical programs with C++.
  464 pages, paperback
  $22.95, order number 55025

- **Programmer's Guide to MPW®, Volume I**
  Exploring the Macintosh® Programmer's Workshop
  Mark Andrews
  Learn the secrets to unlocking the power of MPW, Apple's official integrated software development system for the Macintosh. The book begins with fundamental skills and concepts and then progresses to more advanced examples culminating in a fully functional application.
  608 pages, paperback
  $26.95, order number 57011

- **ResEdit™ Complete**
  Peter Alley and Carolyn Strange
  This book/disk package contains the actual ResEdit software along with a complete guide to using it. The book shows you how to customize your desktop and then moves on to cover more advanced topics such as creating standard resources, designing templates, and writing your own resource editor.
  576 pages, paperback
  $29.95 book/disk, order number 55075

- **The Complete Book of HyperTalk® 2**
  Dan Shafer
  This hands-on guide covers HyperTalk 2, with its greatly expanded features and capabilities. It offers practical information on commands, operators, and functions as well as detailed explanations of XCMDs, dialog boxes, menus, communications, and stack design. You'll also find plenty of tips and dozens of ready-to-use scripts.
  480 pages, paperback
  $24.95, order number 57082

▶ **Debugging Macintosh® Software with MacsBug®**
Includes MacsBug 6.2
Konstantin Othmer and Jim Straus
This book/disk package is a complete guide to using MacsBug. It includes the actual MacsBug software as well as a hands-on tutorial on using it to debug Macintosh programs. Debugging tips, tricks, and advice appear throughout the book, in addition to numerous examples.
576 pages, paperback
$34.95 book/disk, order number 57049

Order Number	Quantity	Price	Total
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
TOTAL ORDER			_____

Shipping and state sales tax will be added automatically.

Credit card orders only please.

Offer good in USA only. Prices and availability subject to change without notice.

Name _____

Address _____

_____

City/State/Zip _____

Signature (required) _____

___Visa ___MasterCard ___AmEx

Account # _____ Exp. Date _____

Addison-Wesley Publishing Company
Order Department
Route 128
Reading, MA 01867
To order by phone, call (800) 477-2226

Macintosh Inside Out

>$24.95 FPT USA
>$31.95 CANADA

Programming the LaserWriter®

DAVID A. HOLZGANG

Here is the information all Macintosh® programmers have been waiting for! This hands-on guide gives you the inside information on programming Apple® Computer, Inc.'s LaserWriter® and other PostScript®-based and LaserWriter-compatible laser printers, the preferred printers of Macintosh users. Every Macintosh programmer will want to read this practical reference to take advantage of all of the LaserWriter's features and capabilities easily and quickly.

In **Programming the LaserWriter**, PostScript authority David Holzgang takes Macintosh programmers of all levels through the details of accessing the LaserWriter directly and thus bypassing the Apple Printing Manager and its limitations. The book focuses on issues and problems developers face in programming the LaserWriter and offers numerous useful insider's tips, tricks, techniques, and examples. Topics covered include Printing Manager functions, PostScript program construction, and font handling.

You will also learn how to:

- Talk directly to the LaserWriter using AppleTalk® commands
- Test and debug PostScript code on your LaserWriter
- Use advanced LaserWriter features to create enhanced text and graphics
- Build on the book's object-oriented classes and much more.

An appendix consolidates the book's complete example THINK C® 4.0 source code, which is adaptable to other Macintosh programming environments. This thorough coverage of vital LaserWriter concepts and operations makes **Programming the LaserWriter** an essential tool for all Macintosh programmers.

David A. Holzgang is an experienced writer and computer consultant specializing in PostScript and the LaserWriter. He is the author of numerous books including *Display PostScript Programming* (Addison-Wesley, 1990), *PostScript Programmer's Reference Guide*, and *Understanding PostScript Programming*.

52495>
9 780201 570687
ISBN 0-201-57068-8
57068

Cover design by Ronn Campisi
Addison-Wesley Publishing Company, Inc.