Inside LaserWriter

Introduction

The purpose of Inside LaserWriter is to give you the information that you need to develop your own applications for Macintosh and other personal computers that take advantage of the unique features of the Apple LaserWriter laser printer. A complete version of Inside LaserWriter is under development at Apple, but it is not available at this time. However, the following Appendices from Inside LaserWriter are available and are included in this document for your use today. When the final version of Inside LaserWriter becomes available, you will be sent an update that contains the sections of Inside LaserWriter that are currently missing.

Appendix A - The Postscript Language Manual

Appendix B - The Postscript Cookbook

Appendix C - The Adobe Font Manual

Appendix D - The Advanced Users Supplement

Appendix E - The Apple Talk Printer Access Protocol

Appendix F - Programming and Debugging aids

Appendix G - Example of the things that you can do with LaserWriter

Appendix H - Using the Macintosh Print Manager

Appendix I - Using MacTerminal to talk directly to the Postscript computer in LaserWriter.

Appendix J - Postscript File Structuring Conventions

Proposed format for the future version of Inside LaserWriter

When Inside LaserWriter is completed, it is anticipated that it will contain the following information:

Chapter 1 - How to develop Macintosh applications that will print successfully on the LaserWriter by using the standard Macintosh Print Manager programs.

Chapter 2 - How to develop Macintosh applications that will print successfully on the LaserWriter by using the standard Macintosh Print Manager programs in conjunction with some limited Postscript commands to do things that are not supported in the Print Manager.

Chapter 3 - How to develop Macintosh or other PC applications that will print successfully on the LaserWriter by issuing Postscript commands directly through AppleTalk without using any of the standard Macintosh Print Manager programs.

Chapter 4 - How to develop Macintosh or other PC applications that will print successfully on the LaserWriter by issuing Postscript commands directly through the RS 422 serial connection without using any of the standard Macintosh Print Manager programs.

In addition, the following appendices are expected to be included:

Appendix A - the Postscript Language Manual

Appendix B - the Postscript Cookbook

Appendix C - the Adobe Fonts Manual

Appendix D - the Advanced Users Supplement

Appendix E - the Apple Talk Printer Access Protocol

Appendix F - Programming and Debugging aids

Appendix G - Example of the things that you can do with LaserWriter

Appendix H - Using the Macintosh Print Manager

Appendix I - Using MacTerminal to talk directly to the Postscript computer in LaserWriter.

Appendix J - Postscript File Structuring Conventions

Appendix K - QuickDraw to Postscript Comments

Appendix L - Source of the Apple Header

Appendix M - Syntax of the QuickDraw Translater

Appendix A

The Postscript Language Manual

POSTSCRIPT™ Language Manual

First Edition, Revised
September 1984

Adobe Systems Incorporated

Adobe Systems Incorporated
1870 Embarcadero Road, Suite 100
Palo Alto, California 94303

POSTSCRIPT™ Language Manual
First Edition, Revised
27 September 1984
Copyright © 1984 Adobe Systems, Inc.
All Rights Reserved

POSTSCRIPT is a trademark of Adobe Systems, Inc.

Times and Helvetica® are trademarks of Allied Corporation.
ITC Friz Quadrata, ITC Souvenir, and ITC Galliard
are trademarks of International Typeface Corporation.

Scribe is a registered trademark of UNILOGIC, Ltd.

The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems, Inc. Adobe Systems assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

Preface

POSTSCRIPT is a simple interpretive programming language with powerful graphics primitives. The primary application of POSTSCRIPT is to describe the appearance of text, images, and graphic material on printed pages. Source code in this language communicates a description of how a page looks to a POSTSCRIPT interpreter. This interpreter converts the source code into a device specific format required by any raster output device. Normally, POSTSCRIPT source code is generated by word processing programs, computer aided design programs, and other composition programs. Programmers write POSTSCRIPT source code directly only when setting up applications. In this unconventional use of a programming language, POSTSCRIPT defines a standard, extensible, flexible print file format, which is the interface between document composition applications and raster printing devices.

Most print protocol formats used today are extensions of line printer protocols or terminal protocols. These formats are, by their basic structure, limited in their capabilities. POSTSCRIPT has been designed with general graphics capabilities in mind. POSTSCRIPT treats standard text and more complicated character fonts as special cases of its graphics facilities. In this environment, graphics and text are not implemented as separate packages, but as a unified system. This approach leads to a clean design that allows users freedom and flexibility in creating applications.

This document is written for people interested in interfacing existing application programs to POSTSCRIPT, in creating new application programs to generate POSTSCRIPT files, or in creating applications in the POSTSCRIPT language itself. Upon first reading, you may think that POSTSCRIPT is an ''overkill'' design. The POSTSCRIPT language is both general and powerful. Even though the more powerful facilities of the language are rarely used, the language includes them for completeness. It has been our experience that limiting a language design and restricting its scope only leads its users into ''work-arounds'' or ''arcane hackery'' as they reach for missing features.

The User's Manual begins with a ''Basic Overview'' of POSTSCRIPT, followed by a more detailed ''Reference Section.'' The casual style of the former is not intended to define POSTSCRIPT's capabilities in an exhaustive way, but instead is meant to give the user the ''flavor'' of POSTSCRIPT. Discussion in this chapter focuses largely on basic language features. The second chapter provides a complete description and precise semantics for all of POSTSCRIPT's built-in operators, with an in-depth discussion of the graphics and printing operators.

While reading the first part of the document, you are encouraged to look ahead to the reference section for additional detail or to find answers to questions you may have.

Table of Contents

1. Introduction 1
2. Basic Overview 5
   - 2.1. Basic Ideas and Motivation 6
   - 2.2. Raster Printer File Formats 8
   - 2.3. The POSTSCRIPT Imaging Model 9
   - 2.4. Coordinate Systems and Device Independence 11
   - 2.5. POSTSCRIPT Syntax 13
   - 2.6. The POSTSCRIPT Interpreter 16
     - 2.6.1. Basic Operation 16
     - 2.6.2. The Assignment Operators 17
     - 2.6.3. Control Operators 19
   - 2.7. Graphics Operators 21
     - 2.7.1. The Graphics State 21
     - 2.7.2. Geometric shapes 22
     - 2.7.3. Transformations 23
     - 2.7.4. Character Shapes (fonts) 24
   - 2.8. Summary 27
3. Reference Section 33
   - 3.1. Data Structures and Types 34
     - 3.1.1. POSTSCRIPT Stacks 34
     - 3.1.2. POSTSCRIPT Objects 35
     - 3.1.3. Composite Objects 38
   - 3.2. Argument and Error Handling 40
   - 3.3. Immediate and Delayed Execution 41
   - 3.4. POSTSCRIPT Operators 44
     - 3.4.1. Stack Operators 45
     - 3.4.2. Arithmetic and Math Operators 49
     - 3.4.3. Polymorphic Operators 56
     - 3.4.4. Array Operators 62
     - 3.4.5. Dictionary Operators 64
     - 3.4.6. String Operators 69
     - 3.4.7. Relational, Boolean, and Bitwise Operators 72
     - 3.4.8. Control Operators 77
     - 3.4.9. Type, Conversion, and Property Operators 82
     - 3.4.10. File Operators 87
     - 3.4.11. Virtual Memory Operators 94
     - 3.4.12. Miscellaneous Operators and Functions 96
   - 3.5. Graphics Operators 98
     - 3.5.1. Graphics State Operators 101
     - 3.5.2. Coordinate System and Matrix Operators 103
     - 3.5.3. Character and Font Operators 112

3.5.4. Path Construction Operators 117
3.5.5. Graphics Output Operators 125
3.5.6. Device Setup Operators 142
3.5.7. Character Cache Management Operators 144
3.6. Error Operators 147
A. Font Machinery 153
B. Implementation Limits 163
Revision History 167
Operator Index 169
Index 175

1

INTRODUCTION

POSTSCRIPT is a language for describing the appearance of pages in documents. The language specifies character shapes (fonts), their placement, and their orientation. It describes graphic shapes (lines and areas). It also specifies the position, scale, and orientation of scanned images. Although the language is general and flexible, this document emphasizes its use for the printing application. In particular this manual describes how to position text on a page, how to generate graphic objects, and how to manipulate and place scanned images.

In a typical application a POSTSCRIPT source consists of two different parts. The first part is the *prologue*. The prologue is application specific and is written once by a programmer; it becomes the front section for every POSTSCRIPT source document the application produces. Each prologue mostly contains definitions that match the output functions of the application to the capabilities that POSTSCRIPT supports.

For example, if an application generates many instances of a given symbol, then a definition of a subroutine to generate the symbol belongs in the prologue. When this is done, the application program may insert calls to that subroutine to place instances of the symbol on the page.

The second part of a POSTSCRIPT source is the *script*. The script is output by the application program, and is very stylized, repetitive and simple. It normally consists of operands (numbers and strings) followed by names of either POSTSCRIPT operators or predefined subroutines. The script uses more general facilities of the POSTSCRIPT language only rarely.

There is nothing in the POSTSCRIPT language that formally distinguishes the prologue from the script, but we make the distinction in this document because it is useful for talking about how POSTSCRIPT is typically used.

The syntax of POSTSCRIPT is simple, and it uses only the printable subset of the ASCII character set (plus the *newline* marker). POSTSCRIPT uses no control characters other than the newline marker. We chose this character subset to maximize machine independence rather than information density. Restricting ourselves to this representation keeps POSTSCRIPT files "human readable", and it simplifies storage and communication of these files among many different computer systems.

The semantics of the POSTSCRIPT language is as simple as its syntax. POSTSCRIPT models a stack machine: that is, POSTSCRIPT accepts operands, which it pushes on an operand stack, and it accepts operators, which operate on those operands. Within this machine model, POSTSCRIPT implements the features found in most modern computer languages. The language supports arrays, mixed mode arithmetic, control structures, subroutines, symbol tables, and an extensive set of built-in operators for handling text, graphics, and images.

We have accomplished several goals in POSTSCRIPT's design:

1. The language and imaging model are both host machine and graphics device independent. These properties allow POSTSCRIPT sources to act as a standard file format for describing images. Therefore, POSTSCRIPT sources can be used on a variety of combinations of host computers and raster output devices. Display devices can range from one bit per pixel displays to full color displays. Printing devices can range from low-resolution matrix printers to high resolution laser printers.

2. A program that generates a POSTSCRIPT source file need not be complicated or maintain a large amount of state information. A program can stream POSTSCRIPT source incrementally to a file. This attribute of the language allows even small machines to generate complex POSTSCRIPT sources.

3. Each application maintains its own view of how text and graphics are to be generated. The extensive programmability of POSTSCRIPT caters to the application without forcing it into a POSTSCRIPT mold. A generating program is able to extend the language so that the file generated relates directly to the application and is therefore more natural, readable, and compact.

2

BASIC OVERVIEW

There are two complementary approaches toward describing the POSTSCRIPT language. On one hand, POSTSCRIPT is a programming language with powerful built-in graphics functions. On the other, POSTSCRIPT is a printed page description language that includes general purpose programming language features. Either of these views could serve as a basis for describing POSTSCRIPT, but either one taken alone does not tell the entire story. Both views are equally valid and they interact to provide a complete model for understanding POSTSCRIPT.

This overview will use examples from POSTSCRIPT used as a printed page description language. Bear in mind, however, that with a change of application-specific operators, the POSTSCRIPT language framework could serve equally well for many other application areas.

Basic Ideas and Motivation

Let's assume that some computer application program needs to print pages on a raster printer (a not too unreasonable assumption). Immediately, two different design choices are apparent: it can operate the printer(s) directly, or it could write a description of the pages in such a way that a separate process can print the pages from those descriptions.

The first choice does not generalize well. The application must fill itself with printer device specific details which clutter the application. If a different printer must be used, a large amount of additional printing program logic must be embedded in the application program. Furthermore, other programs cannot take advantage of this application program's special printing capabilities. They must include this code themselves to be able to perform the same functions. This choice is appropriate only if no alternative is available.

With the second choice, we enter the realm of print file formats. These formats have been around since computers used printers. Once again the design choices can be divided into two classes. The print file format can be either static or dynamic.

A static format provides some fixed set of operations (sometimes called control codes) together with a syntax regarding the placement of the operations and the arguments they must take. Some line printer formats are classic examples of static print file formats. The first character of a record is the control code; it determines paper motion: none, next line, next page, etc. The rest of the record is the actual character data to be printed on the chosen line.

A dynamic format allows considerably more flexibility than a static format. The operator set may be extensible, and the exact meaning of an operator may not be known until it is actually encountered. A static format might offer an operation that repeats a fixed number of times, while a dynamic format might allow a loop with an index variable.

This classification of print file formats into static and dynamic formats is, admittedly, an oversimplification. Some formats have elements of both styles, as in a format that is mostly static but which allows macro expansion or limited use of variables. Even though some existing print file formats do fall into this gray area, this static/dynamic distinction can be useful when comparing the capabilities of different formats.

A print format that is primarily static, but which purports to cover a lot of graphic and text operations, tends to grow operators wildly. A dynamic format that allows primitive operations to be combined according to the wishes of the user writing the print file will always be superior to a static format that tried to anticipate all possible needs. As we will see in later sections, for very complicated page layouts, there may be information that the printed page description writer cannot know until the page is actually imaged on its specific printer. Dynamic formats that allow reading of crucial information and using this information in calculations will clearly be able to specify more sophisticated images.

POSTSCRIPT goes all the way over to the dynamic side of this classification. POSTSCRIPT includes many graphic operations, and it allows them to be combined in any possible manner. It not only has variables, but it allows arbitrary computations in the process of interpreting the page description. It has a rich set of programming language control structures for combining its elementary elements. Also, while some print file formats may appear to have these capabilities only through contorted, unintended uses of some of their features, POSTSCRIPT has provided the complete set of dynamic features by design, making their use natural and efficient.

Thus we have POSTSCRIPT, a dynamic print format whose page descriptions are actually programs to be run by an interpreter. POSTSCRIPT programs can degenerate into a form that resembles a static format: a sequence of argument, operator, argument, operator, and so on. In fact, many POSTSCRIPT programs will have this boring repetitive nature, having been generated by an application program that knew exactly what it needed. However, when the need arises, the power is there to be applied by the knowledgeable application designer.

Raster Printer File Formats

For any print format, several questions should be asked:

- Is it complete? (Can it describe any printed page?)
- Is it easy to generate?
- Is it easy to interpret?
- Is it easy for a person to understand it?
- Is it valid for more than one printer?
- Is it easy to transmit?
- Is it compact?
- Can pieces of a description already built be used to compose more complicated pages?
- Can it emulate other formats?

The answers to these questions and the relative importance one attaches to each one will vary from application to application. However, medium and high resolution raster printers add new capabilities to computer generated pages that complicate the questions somewhat.

A raster printer produces its image by writing small dots onto a page. The size and positioning of these dots is expressed as the printer's resolution, in terms of how many dots fit in one inch. High resolution refers to many dots per inch, say 1000 or more. Medium resolution refers to somewhere between 300 to 600 dots per inch. All raster printers form letter shapes by writing a pattern of these dots for each letter. At medium resolution and above, this technology enables these printers to form almost any typeface in any size and at any rotation.

However, many print file formats are incapable of describing pages that use these capabilities of raster printers. Print file formats that are a holdover from the days of impact printing have no notion of scalable letters and sophisticated graphics. Some print formats that do address these opportunities are crippled by their separation of text and graphics, treating text in such a way that the flexibility inherent in the printer is lost.

To fully evaluate a print file format in terms of the above questions, one should take into account not only the surface form of the language (which will answer some of the questions) but also the imaging model of the language. An imaging model describes how a picture is made, what operations are allowed, how the operations may be combined, etc.

The POSTSCRIPT Imaging Model

The POSTSCRIPT imaging model is a simple and unified view of two-dimensional graphics borrowed from the graphic arts industry.[1] An image is built up by placing ink on a page in selected areas. The ink may be in the form of letter shapes, general filled shapes, lines, or halftone representations of photographs. The ink itself may be in color or in black, white, or any shade of gray. Any of these elements may be cropped to within any shape as they are placed onto the page. Once a page has been built up to the desired form, it may be printed on an output device.

POSTSCRIPT maintains an implicit *current page* that accumulates the marks made by the POSTSCRIPT *imaging operators*. When a program begins, the current page is completely white. As each imaging operator executes, it places marks on the current page. Each new mark completely obscures any marks that it may overlay. This method is known as a *painting model*; no matter what color a mark has – white, black, gray or color – it is put onto the current page as if it were applied with opaque paint. When the **showpage** or **copypage** operators are executed, the current page is printed on the output device (**showpage** clears the current page after printing; **copypage** leaves the current page unchanged.)

The *imaging operators* (those that place marks on the current page) are the **fill**, **stroke**, **image**, and **show** operators. **fill** marks an area on the current page; **stroke** marks lines on the current page; **image** paints a halftone gray-scale scanned image onto the current page; and the **show** operators paint character shapes onto the current page. Each of these operators requires several arguments, some explicit and some implicit.

Chief among the implicit arguments is the *current path* (used by **fill**, **stroke** and **show**.) This object describes a sequence of connected and disconnected points, lines and curves that together describe shapes and their positions. The current path is built up through the sequential application of the *path operators*, each of which modifies the current path in some way (mostly by appending one new element to the current path.) Other implicit arguments to the imaging operators include the current color, current line thickness, current font (typeface-size-rotation combination), etc. Each implicit argument has its own corresponding set and examine operators; each

[1] A detailed, technical description of a similar imaging model has appeared in a paper by John Warnock and Douglas Wyatt, titled "A Device Independent Graphics Imaging Model for Use with Raster Devices," in the July 1982 *Computer Graphics* Volume 16, number 3, pp. 313-320. The description given here is in terms that a POSTSCRIPT programmer should understand before using POSTSCRIPT to prepare printed pages.

may be set to a new value at any time. The values held in each of the implicit arguments at the time that an imaging operator is executed will affect the behavior of that operator.

The *path operators*, which include **newpath, moveto, lineto, curveto, arc, closepath** and others all modify the current path as they are executed. None of these operators affects the current page directly; that is left to the imaging operators. The path operators build up a shape comprised of connected and/or unconnected points, straight line segments and curves. These shapes are unrestricted – they may be convex, concave, or even mutually and self intersecting. The positions of these elements in a path are specified by real numbers; the resolution of the output device in no way constrains the definition of a path.

POSTSCRIPT programs that make printed pages will contain many instances of the following pattern: build a path using path operators; set any implicit arguments (if their values need to change); perform an imaging operation.

There is one additional implicit element in POSTSCRIPT's imaging model that modifies the foregoing description. POSTSCRIPT maintains a *current clipping path* that outlines the area of the current page that may be imaged upon. Initially, this clipping path outlines the entire imageable area of the current page; parts of the image description which lie off of the page (outside the clipping path) are discarded. By using the **clip** operator, a POSTSCRIPT program can shrink the current clipping path to any shape desired. It is quite normal for an imaging operator to attempt to place marks outside of the current clipping path. Those marks within the clipping area will make it onto the current page; those marks that fall outside the clipping area will not affect the current page.

Coordinate Systems and Device Independence

The arguments to path operators describe points on the page (or outside of the page) by means of coordinates, i.e., a pair of real numbers x and y that locate a point within a Cartesian coordinate system superimposed on the current page. POSTSCRIPT defines a standard, default coordinate system that POSTSCRIPT programs may depend on for locating any point on the page.

Output devices vary greatly in the built-in coordinate systems that they use to address actual device points within their imageable area. We call this coordinate space, idiosyncratic to each output device, *device space.* Device space origins can be anywhere on the output page; the paper moves through different printers in different directions; and some devices even have different resolutions in different directions. Coordinates specified in a POSTSCRIPT program, however, are device independent since they refer to locations within an ideal coordinate space that always bears the same relationship to the current page regardless of the output device on which printing will be done. We call this coordinate system *user space,* as it is the coordinate system that programs use to specify points. The POSTSCRIPT interpreter automatically transforms points specified in user space into the device space of the specified device. For the most part, this transformation is hidden from the POSTSCRIPT program; a program needs to consider device space only rarely for certain special effects. This independence of user space from device space is a major contributor to the device independent nature of POSTSCRIPT page descriptions.

To specify a coordinate system with respect to the current page, we must know the location of the origin, the orientation of the x and y axes and the lengths of the units along each axis. Initially, the user space origin is located at the lower left corner of the output paper, with the positive x axis extending horizontally to the right and the positive y axis extending vertically upward (as in standard mathematical practice.) The length of a unit along the x axis and along the y axis is 1/72 of an inch. We call this coordinate system *default user space.*

Although the choices made for default user space are arbitrary, the important point is that they provide a consistent, dependable starting point for POSTSCRIPT programs regardless of the output device being used. The POSTSCRIPT program may then modify its user space into one more suitable for its needs (if necessary) by applying *coordinate transformation operators.* The features of default user space were chosen for their math-

ematical simplicity and convenience. The location and orientation of the axes follows mathematical tradition and gives positive coordinates to points on the current page. The unit size, 1/72 of an inch, is very close to the size of a printer's point (a printer's point is actually 1/72.27 of an inch) which is a standard measuring unit used in the printing industry. Note that the coordinates used in POSTSCRIPT programs may be decimal numbers containing fractional parts, so that the initial choice of unit size does not constrain points to any arbitrary grid.

For its convenience, a POSTSCRIPT program may move the user coordinate system with respect to the current page and even change the size of the *x* and *y* units independently. It accomplishes this with the coordinate transformation operations **translate**, **rotate** and **scale**. **translate** moves the user space origin to a new position with respect to the current page while leaving the orientation of the axes and the unit sizes unchanged. **rotate** turns the user space axes about the current user space origin, leaving the unit lengths unchanged in their current directions. **scale** modifies the unit lengths independently along the current *x* and *y* axes, leaving the origin location and the orientation of the axes unchanged. (For very sophisticated users, POSTSCRIPT actually allows any linear transformation to be specified from user space to device space by means of the **setmatrix** operator.) Thus, what may appear to be absolute coordinates in a POSTSCRIPT program are actually quite changeable with respect to the current page, since they are described in a coordinate system that may slide around and shrink or expand.

POSTSCRIPT Syntax

A POSTSCRIPT program is written so as to be readable by people. All program text and data are written in the printable subset of the ASCII character set (plus the carriage return character.) This written form promotes ease of communication between the producer of a POSTSCRIPT program and the machine on which the POSTSCRIPT interpreter resides, since any communication network should at least be able to transmit characters, in addition to making the programs easily read.

There are five distinct syntactic constructs in POSTSCRIPT. These are:

- numbers (reals and integers)
- strings
- names
- procedure bodies and arrays
- comments

POSTSCRIPT treats spaces, tabs, and newlines equivalently: they serve to separate (or *delimit*) other syntactic constructs such as names and numbers from each other. Any number of these characters appearing in a row are treated in the same manner as if there were just one. The characters "(", ")", "<", ">", "[", "]", "{", "}", "/", and "%" are special: they serve to delimit syntactic entities such as strings, procedure bodies, and comments. Any of these characters terminates the entity preceding it, and is not included in it.

Numbers in POSTSCRIPT include signed integers, such as

```
123 -98 43445 0 +17
```

reals, such as

```
-.002 34.5 -3.62 123.6e10 1E-5 -1. 0.0
```

and radix (integer) numbers, such as

```
8#1777 16#FFFE 2#1000
```

These take the form *base#number* where *base* is in the range *2* through *36*. The *number* is then represented in this base with digits ranging from *0* through *base-1*. Digits greater than *9* are represented by the letters *A* (or *a*) through *Z* (or *z*). Note that, although the machine representation of the number may be negative, these numbers should be considered as unsigned (positive) integers. This notation is intended for specifying character codes (when needed), and bit patterns for bitwise operations.

A string in POSTSCRIPT is delimited by balanced parentheses. This notation is POSTSCRIPT's way of "quoting" a string body. The following are examples of valid strings.

```
(This is a string)
(Strings may contain newlines
and such.)
(Strings may contain special characters:*-&^% and
balanced parentheses ()(and so on))
(The following string is an "empty" string.)
()
(It has 0 (zero) length.)
```

To insert unbalanced parentheses into a string the backslash character is used. "\" is an escape character instructs the scanner to insert the next character into the string. For example:

```
(\(\\)
```

represents the string "(\". Special non-graphic characters can be represented in strings by using the backslash escape character. Certain characters following it have special meaning:

\n	LF	linefeed (newline)
\r	CR	carriage return
\t	HT	horizontal tab
\b	BS	backspace
\f	FF	form feed
\\	\	backslash
\(	(	left parenthesis
\)	)	right parenthesis
ddd	*ddd*	octal byte
newline		no character - both are ignored

The *ddd* form may be used to include any octal character constant into a string. One, two, or three octal digits may be specified (with high-order overflow ignored). If the character following the backslash is not one of the above, the backslash is ignored. The *newline* form is used to break a string into a number of lines, but not have the newlines be part of the string.

```
(These\
 two strings \
are the same.)
(These two strings are the same.)

(This string has a newline in it.
)
(So does this one.\n)
```

A string may also be defined in hexadecimal (base 16) notation by delimiting a sequence of hex characters (the digits *0* through *9* and the letters *a* through *f* and *A* through *F*) with "<" and ">". Each pair of hex digits defines one character of the string. Spaces, tabs, and newlines are ignored. For example, "<901fa3>" is a 3-character string containing the characters whose hex codes are 90, 1f, and a3.

A comment in POSTSCRIPT is preceded by "%" and terminated by a

newline. Outside of a string body, the POSTSCRIPT scanner treats comments as delimiters. The following is a comment:

```
% this is a comment
```

A name in POSTSCRIPT is simply a sequence of non-special characters not contained in a string or comment. That is, any sequence of characters bounded by delimiters and not containing a delimiter is a name, unless it can be interpreted as a number in which case it is a number. All printing characters except the special ones can appear in names, including punctuation characters. The following are examples of valid POSTSCRIPT names:

```
abc Offset $$ 23A 13-456 *&$ $MyDict myProc @pattern
```

The forward slash "/" is used to specify a *literal* name. The slash is not part of the name itself, but is a prefix which indicates that the following name is a literal. Hence, the slash character may not be a part of any syntactic name in POSTSCRIPT.

A procedure body begins with a "{" and ends with a balancing "}". An array begins with a"[" and ends with a balancing "]". Numbers, strings, names, comments, and other procedure bodies or arrays may occur between the delimiting braces. An example of a valid procedure body is:

```
{add 2 div}
```

and an example of a valid array is:

```
[23 45.2 (a string) /aName [(abc) 16#7E] {dup mul}]
```

Note that POSTSCRIPT arrays need not be homogeneous.

The POSTSCRIPT Interpreter

2.6.1. Basic Operation

The POSTSCRIPT interpreter is the process that executes the POSTSCRIPT language according to the rules listed below. These rules tell us the order in which operations are carried out, and how the pieces of a POSTSCRIPT program fit together to produce the desired results. In this section, we shift our emphasis from POSTSCRIPT the page description language to POSTSCRIPT the programming language so as to show the operation of the interpreter in as simple a manner as possible. We will return to POSTSCRIPT the page description language in the next section.

The POSTSCRIPT interpreter manipulates entities called POSTSCRIPT *objects*. Each of the syntactic types discussed in the previous section (except comments) corresponds to its own kind of POSTSCRIPT object. There are also several kinds of POSTSCRIPT object that have no direct syntactic representation (such as *dictionary objects* and *file objects*) that are created through their own *creation operators*.

The characters in the POSTSCRIPT program, written according to the syntax given in the previous section, are not themselves POSTSCRIPT objects. The syntax rules specify how the POSTSCRIPT interpreter groups and separates these input characters into *tokens*, which the interpreter can then convert into POSTSCRIPT objects, with which it can execute the program. Thus, the interpreter converts a token consisting of digits into a POSTSCRIPT *number object*, a token enclosed by parentheses into a POSTSCRIPT *string object*, a token beginning with a letter into a POSTSCRIPT *name object*, and sequence of tokens surrounded by brackets or braces into a POSTSCRIPT *array object*. The POSTSCRIPT interpreter proceeds as follows: it scans the input stream (ignoring comments) for the next token, converts it into a POSTSCRIPT object, and acts on that object according to its type. If a token is a string, a number, a procedure body, an array, or a literal name (a name with a "/" prefix), then the interpreter converts that token to its corresponding POSTSCRIPT object, and pushes that object onto a stack called the *operand stack*. If the token is an evaluated name (a name with no "/" prefix), then the interpreter looks up that name for its value (more about the details of name lookup later) and takes action depending on whether or not that value is *executable*. If the value is not executable, the interpreter merely pushes the value onto the operand stack. If the value is executable, then the interpreter executes that value immediately, before processing the next input token.

For example, if the input stream contains:

```
123 456 add
```

then the POSTSCRIPT interpreter reads "**123**", pushes the number 123 on the operand stack, reads "**456**", pushes the number 456 on the operand stack, and reads "**add**" as a name. The interpreter then looks up the name **add**, finds that it is associated with an intrinsic POSTSCRIPT operator (which is executable), and executes this operator (which adds two numbers). The **add** operator removes 123 and 456 from the operand stack and pushes their sum, 579, onto the operand stack.

The above example models how the POSTSCRIPT interpreter processes all operands and operators. In POSTSCRIPT, there is no explicit expression or statement structure. Instead the POSTSCRIPT interpreter scans its input in a strictly sequential manner. As it encounters each token, it resolves it to an operand, which it pushes onto the operand stack, or to an operation, which it executes. The language is called *postfix*, since operators follow their operands. When an operator executes, it expects its operands to have already been placed on the operand stack, either directly as in the above example or indirectly by the result of execution of preceding operations.

Example 1: The following is a segment of POSTSCRIPT source that evaluates the expression $(a + b) \div (c \times d)$.

```
a b add c d mul div
```

Example 1 shows POSTSCRIPT source that consists of a simple sequence of operands and operators. Note that the operands to the **div** operator were left on the stack by the preceding **add** and **mul** operations. In this example we assume that *a*, *b*, *c*, and *d* have values that are numbers (we will show how this association is made shortly) which are pushed onto the operand stack by the interpreter.

The operation of POSTSCRIPT is dictated solely by the semantics of the operators. By constructing powerful operators, POSTSCRIPT provides most of the facilities found in other programming languages. These facilities are provided so POSTSCRIPT can be extended to meet the needs of the application. In particular, if an application finds it convenient to assign variables, redefine operators, process conditionally, build subroutines, or build shorthand notations for common constructs, then the richness of the POSTSCRIPT operators makes this possible.

2.6.2. The Assignment Operators

Like most programming languages, POSTSCRIPT allows assignment of values to variables. Instead of the infix form (e.g., "abc = 38") used by many languages, POSTSCRIPT accomplishes the same task with a postfix assignment operator. For example:

```
/abc 38 def
```

Here the POSTSCRIPT interpreter scans left to right and pushes the literal name **abc** onto the operand stack, and then it pushes the number 38 onto

the operand stack. The POSTSCRIPT interpreter then looks up the name **def**, which is associated with the built-in define operator, and executes it. This operator associates a *value* (the top element on the stack) with a *key* (the element one below the top of the stack). The define operator then associates the value 38 with the key **abc**. This association now may be used by POSTSCRIPT in future processing. For example the POSTSCRIPT interpreter processes

```
123 abc add
```

in the following way. First, it pushes the number 123 onto the operand stack. Next it encounters the name **abc** and looks it up. Because **abc** is associated with the number 38 (just defined), the interpreter pushes the number 38 onto the operand stack. As in the first example, the interpreter resolves **add** to the add operator, which removes the two operands from the operand stack and pushes their sum, 161, onto the operand stack.

In addition to assigning numbers to names, POSTSCRIPT provides a mechanism for assigning executable procedures to names. As an example, let us suppose we wish to have an operator that averages the top two numbers on the operand stack. The sequence of code that does this is found embedded in the following:

```
123 456 add 2 div
```

The sequence "**add 2 div**" is the POSTSCRIPT version of code that belongs in an "averaging" subroutine. We note here that this subroutine takes its arguments from the operand stack, and returns its results to the operand stack. To define and use this code we write:

```
/ave {add 2 div} def
1024 512 ave
```

then when the POSTSCRIPT interpreter looks up **ave** it will find the procedure body that executes **add 2 div**. This procedure body will remove the numbers 512 and 1024 from the operand stack, and it will push the number 768 onto the stack.

The above definition structure for procedures allows a programmer to reference one procedure from the body of another. It also allows recursive calls of a procedure from itself. For example the following code defines a recursive "factorial" function. This function expects a number n on the stack, and will return $n!$

```
/fact {dup 1 gt {dup 1 sub fact mul} if} def
```

The **def** operator used in the above examples associated numbers and procedures to names. **def** converts string objects to name objects when used as keys. However, **def** can associate any POSTSCRIPT object type with any other object type. Although association between an object and a name is most common, POSTSCRIPT does not restrict the association to this case.

The **def** operator used in the above examples is just one of many operators that perform assignments. These assignment operators rely on POSTSCRIPT *dictionaries*. Dictionaries and the operators that create and

use them provide powerful associative symbol table facilities to the POSTSCRIPT programmer. These facilities are discussed in detail in section 3. For now we will give an overview of what dictionaries are, how they are used, and how they relate to the operation of the POSTSCRIPT interpreter.

Dictionaries are associative tables that consist of key-value pairs. In POSTSCRIPT, dictionaries are used in two ways. In the first use, dictionaries play a role in defining the naming context or *scope* for the names used in programs. In the second use, dictionaries act as associative data structures for programs.

In POSTSCRIPT there is a root dictionary, which is always present, called the *system dictionary*. This dictionary associates each of the POSTSCRIPT operator names (keys) with the procedure (value) that implements the operator. POSTSCRIPT also provides another dictionary, called the *user dictionary*, that is intended to hold names and values global to a particular POSTSCRIPT program.

When the POSTSCRIPT interpreter encounters a name while scanning, it consults a stack called the *dictionary stack*. This stack always contains the system dictionary at the bottom and the user dictionary immediately above it, but it may also contain other dictionaries as required by the application. The interpreter looks for the name in the dictionary on top of the dictionary stack. If it finds the name, then the interpreter uses the associated value. If it cannot find the name in this dictionary, then it searches the other dictionaries in the dictionary stack in order. If the name is not in any dictionary on the stack, then the POSTSCRIPT interpreter executes the error operator **undefined**.

The **def** operator searches only the dictionary on the top of the dictionary stack. In the following:

```
/a 333 def
```

The **def** operator searches the dictionary on top of the dictionary stack for **a**. If it finds the key **a**, then it replaces **a**'s previous value with the number 333, thereby redefining **a**. If it cannot find the key **a** in the top dictionary, then the **def** operator creates a new definition for **a** in the dictionary on the top of the dictionary stack.

POSTSCRIPT has many operators for dealing with dictionaries. Dictionaries can be created. They can be pushed onto the dictionary stack and therefore may be used as name contexts for programs. They can be accessed directly and thus may be used as associative data structures. They can also be enumerated, giving catalogues of key-value pairs. These and other uses of dictionaries are discussed in examples and in section 3.

2.6.3. Control Operators

Thus far we have described POSTSCRIPT operators such as **add**, that compute values on operands and operators such as **def**, that perform assignments. In addition to these operators, POSTSCRIPT has several

operators that provide program control. The simplest of these operators is **repeat**. The repeat operator expects a procedure body on the top and a count (a number) just below it on the operand stack. The repeat operator removes the procedure body and the count from the stack and executes the procedure body "count" times. An example using repeat is the following:

```
1 12 34 -3 66 4 {add} repeat
```

This example is equivalent in function to:

```
1 12 34 -3 66 add add add add
```

Both of the above add 1, 12, 34, -3, and 66, leaving 110 on the stack.

Most of the control operators in POSTSCRIPT are like the **repeat** operator in that they require procedure bodies on the stack and execute them in ways that depend on the semantics of the operator.

Another example of a POSTSCRIPT control operator is the **ifelse** operator. Many computer languages have a construct like

if <boolean> **then** <statement> **else** <statement>

In POSTSCRIPT, the **ifelse** operator provides the if-then-else semantics. This operator expects three objects on the operand stack. These objects consist of one boolean and two procedure bodies. The **ifelse** operator removes these operands from the stack. If the boolean has the value *true*, then the first procedure body pushed onto the stack is executed, otherwise the second procedure body is executed. For example, the line:

```
a b eq {a 22 sub} {b 34 add} ifelse
```

behaves as follows. The **eq** operator removes the top two operands from the operand stack and checks to see if they are equal. If they are, it pushes a boolean object with value *true* onto the operand stack, otherwise it pushes a boolean object with value *false* onto the operand stack. In the above example, if **a** = **b**, the **ifelse** operator will execute **a 22 sub**, otherwise it will execute **b 34 add**.

The above descriptions of **repeat** and **ifelse** give the general flavor of the way control operators work in POSTSCRIPT. Other POSTSCRIPT control operators, **for**, **loop**, and **if**, provide operations analogous to for loops, unconditional loops, and if statements found in other computer languages.

Graphics Operators

We now return to POSTSCRIPT as a page description language. We have already discussed operators that are executed for the results they return on the operand stack. The POSTSCRIPT operators that deal with graphics, text, or images are executed, not so much for the results they return, but for their *side effects*. In particular, they are executed to print something.

The POSTSCRIPT graphics and printing operators provide control over fonts (collections of character shapes), positioning and orientation of text, positioning and orientation of images, and the definition of geometric shapes and areas.

2.7.1. The Graphics State

The POSTSCRIPT interpreter maintains a data structure called the *Graphics State*. This data structure contains the implicit arguments for the imaging operators and holds the following items (among others):

Name	*Type*	*Value Semantics*
CTM	Array	The current transformation matrix; a matrix that maps positions from user coordinates to device coordinates. This matrix is modified by each application of the coordinate system operators. (Initial value: a straightforward matrix transforming default coordinates to device coordinates.)
color	Internal	The internal representation of colors is not exposed to the POSTSCRIPT user. To encode and decode colors among different color models, see color related operators in section 3.5.5. (Initial value: black.)
cp	Numbers	Current position. (Initial value: undefined.)
path	Path	The current path as built up by the path construction operators. Path objects are not directly accessible in POSTSCRIPT. This object is an implicit argument to the **fill**, **stroke**, and **clip** operators. (Initial value: empty.)
clip	Path	The current boundary against which all output is clipped. (Initial value: the entire imageable portion of the output device.)
font	Dictionary	Set of graphic shapes (characters) that define the current typeface. (Initial value: installation dependent.)
line width	Number	The thickness (in user coordinates) of lines to be drawn by the **stroke** operator. (Initial value: 1.)
line cap	Integer	A code that defines the shape of the endpoints of

		any open path that is stroked. (Initial value: 0, for a square butt end.)
line join	Integer	A code that defines the shape of a stroked line at its corners. (Initial value: 0, for mitered joins.)
dash	Array, real	A description of lengths of portions of dashed lines to be rendered by the **stroke** operator instead of the normal solid line. (Initial value: a 0-length array plus a 0 offset, corresponding to a normal solid line.)

2.7.2. Geometric shapes

Before we can put filled areas or lines onto the current page, we must build up the current path in the position where we would like the marks to be. Here, we have an example of a POSTSCRIPT procedure body that can be used to specify a square one inch on a side and one inch in and up from the lower left corner of a page:

```
/sq1
  {newpath
   72 72 moveto          % move to 1",1".
   144 72 lineto         % define edge to 2",1".
   144 144 lineto        % edge to 2",2".
   72 144 lineto         % edge to 1",2".
   closepath             % close back to 1",1".
  } def % define "sq1" to be a proc that builds a path.
```

This example uses POSTSCRIPT's default user space directly, thus 72 1/72 inch units equals 1 inch. Note that this POSTSCRIPT program fragment by itself does not build the path, but only defines a procedure body to do that job and stores that procedure body in the name "sq1". When "sq1" is executed, it will execute its contents in order. The POSTSCRIPT operator **newpath** takes nothing from the stack, but it initializes the current path internal data structure used by POSTSCRIPT to keep track of geometric shapes. The POSTSCRIPT operator **moveto** takes *x* and *y* coordinates (numbers) from the stack, and enters them as a point in the path. The **lineto** operator is like the **moveto** operator except that the point given is *connected* to the previous point in the path. There is also a **curveto** operator that adds curved sections to paths. Finally, the **closepath** operator behaves like **lineto**, but it constructs its line to the point most recently "moved to".

Now, to fill the square with solid color, we can say:

```
sq1 fill
```

To outline the square with a 2 point thick line we can say:

```
sq1 2 setlinewidth stroke
```

To push an image associated with the name "teapot-pic" through the path, we can say:

```
sq1 clip teapot-pic image
```

There are details left out of this series of examples; they are presented to

explain the nature of the imaging mechanisms but not the details of their use. These details are supplied in Section 3.5.

POSTSCRIPT's path operators are used for making internal general purpose geometric constructs. Path structures are used for making lines and curves; for filling areas bounded by lines and curves; and as clipping templates. The topology of a path structure is unrestricted: it can be concave or convex; it can represent multiple regions; it can even intersect itself.

POSTSCRIPT paths allow two kinds of curved segment in addition to the straight line segments introduced in the examples. One simple kind of curved segment is a circle or a piece of a circle. A more general kind of curve is called a *Bezier cubic*, after the French mathematician P. Bezier. These latter curves are specified by four points: two points represent the curve's endpoints, and the other two specify how the curve bends between its ends. In POSTSCRIPT, more complicated curves than these basic types are built by putting several circular arcs and Bezier cubics end-to-end within a path.

2.7.3. Transformations

The ability to translate, scale and rotate any graphic object is quite valuable for graphics applications. This capability is provided by the POSTSCRIPT interpreter through the current transformation matrix (*CTM*) that it maintains. This transformation matrix maps points from a user coordinate system into a device coordinate system when an object is drawn. Modifications to this transformation matrix may be viewed as either modifying the user coordinate system or as modifying the resulting placement of marks on the output device. The POSTSCRIPT coordinate system operators are implemented so as to make the appropriate modifications to this transformation matrix.

A typical application will define procedures that outline prototypical geometric objects. Before painting an instance of such an object, the application will modify its user coordinate system via the coordinate operators. When the path for the object is built, each of its coordinates is mapped through the resulting transformation into the device coordinate system.

For example, suppose we have made the following definition:

```
/triangle                 %define an equilateral triangle.
{newpath
0 0 moveto                %lower left corner at origin.
10 0 lineto               %side of triangle is 10.
5 5 3 sqrt mul lineto     %apex at (5,5*sqrt(3)).
closepath} def
```

Here, the **moveto** operator moves the current point to coordinate (0, 0). The **lineto** operator defines an edge from this point to coordinate (10, 0), establishing (10, 0) as the current point. The next **lineto** operator defines the next edge of the triangle. Finally, the **closepath** operator closes the figure by defining an edge back to the point referenced after the **newpath** operator.

Now, to create a color-filled triangle at (100, 100), we may give the command:

```
100 100 translate triangle fill
```

The **translate** operator modifies the user coordinate system so that subsequently built objects are translated by the given amounts. The **fill** operator actually paints the contents of the triangle with the current color as defined in the Graphics State.

To change the user coordinate system only temporarily, we can say:

```
gsave 100 100 translate triangle fill grestore
```

Here, the **gsave** and **grestore** operators isolate the changes to the values in the Graphics State to the time between execution of these two operators.

2.7.4. Character Shapes (fonts)

A font, in the POSTSCRIPT context, is a dictionary through which the POSTSCRIPT interpreter can obtain path definitions that generate character shapes. The interpreter uses a character's code to select which path definition represents that character.

A character's shape in POSTSCRIPT is a procedure body that generates a path representing that character's outline. To print a character, the POSTSCRIPT interpreter executes the path building procedure corresponding to that character and fills in the path with ink (more or less).

If you have experience with scan conversion of general shapes, then you may be concerned at the amount of computation the above description seems to imply. Relax. The above description tells you how to think about character shapes and fonts. It does not tell you how fonts are implemented. In fact, the implementation of the POSTSCRIPT interpreter makes character rendering quite efficient.

To see how all of the above hangs together, the following examples are instructive.

Example 2: Print the word "PostScript" ten inches from the bottom of the page, and 4 inches from the left edge.

```
288 720 moveto     % set current point to 4*72, 10*72
(PostScript) show % output "PostScript" in the
                  % current font
```

In example 2, we are still using the default coordinate system. The **moveto** operator is used to specify the *current position* for character printing. The **show** operator uses the current font (here, the default font) to print its argument "PostScript".

A font is made up of descriptions of its character shapes and other metric information for that font. For a POSTSCRIPT application programmer's convenience, each POSTSCRIPT installation maintains a dictionary of commonly used names associated with its available fonts. For instance, to use the font "Helvetica", we can enter:

```
/Helvetica findfont
```

The **findfont** operator takes the font name and returns a dictionary containing all the information that the POSTSCRIPT interpreter needs to generate all of that font's characters.

A font specifies the shape of its characters for one standard size. This standard is arranged so that the height of a singly spaced line of text is 1 unit. In the default coordinate system, this means that the standard font size is one point. Since nobody can read one point type, the font must be *scaled* to be usable. We could scale the font with the coordinate system operators, but it is usually more convenient to modify the size of the font itself, rather than change the current transformation matrix. This latter operation is provided by the POSTSCRIPT operators **scalefont** and **makefont**. **scalefont** scales a font uniformly; **makefont** applies more complicated general transformations to a font. These operators accept on the operand stack the nominal font dictionary and the desired modification, and they return a new font that will render character shapes in the desired size. For example, the sequence:

```
/Helvetica findfont 10 scalefont
```

returns a 10 point Helvetica font on the stack.

To print ''PostScript'' in Helvetica 14, we could use the following sequence:

```
/Helvetica findfont        % push 1pt font dictionary
                           % onto the operand stack.
14 scalefont               % push a 14pt scaled font
                           % onto the operand stack.
setfont                    % make the scaled font the
                           % current font.
288 720 moveto             % set current position to
                           % 4*72, 10*72.
(PostScript) show          % Typeset "PostScript"
                           % in the current font
                           % (Helvetica 14pt).
```

The above example uses POSTSCRIPT operators in a direct way. However, it is desirable in most applications to define new operators to help with the application. To illustrate this point, assume that an application requires that switching frequently between three fonts: Helvetica, Helvetica-Oblique, and Helvetica-Bold.

Example 3: Print several sentences down the page, alternating fonts between Helvetica 10, Helvetica-Oblique 10, and Helvetica-Bold 10.

```
% Start the prologue section.
% First make some font definitions.

% define "fnr" to be 10 pt Helvetica.
/fnr /Helvetica findfont 10 scalefont def

% define "fni" to be 10 pt Helvetica-Oblique.
/fni /Helvetica-Oblique findfont 10 scalefont def

% define "fnb" to be 10 pt Helvetica-Bold.
/fnb /Helvetica-Bold findfont 10 scalefont def

% Define some procedures to move to a given
% position, switch fonts, and show the given
% character string.

/shwr {moveto fnr setfont show} def
/shwi {moveto fni setfont show} def
/shwb {moveto fnb setfont show} def

% Start the script section.

(This is in Helvetica.) 288 720 shwr
(This is in Helvetica Oblique.) 288 710 shwi
(This is in Helvetica Bold.) 288 700 shwb
(And more in Helvetica.) 288 690 shwr
 ...
```

Example 3 shows several things. First, it makes the required fonts and associates them with the names **fnr**, **fni** and **fnb**. Next, it defines three operators all of which move the current position to a given position, switch to a particular font, and show the given string. Finally, it sets text using the operators defined earlier.

This last example is a good model for the structure of POSTSCRIPT programs. Notice that there is a section of program at the beginning (the prologue) that makes a number of definitions. Normally, a programmer makes up this part once for an application, which emits it for each document generated by that application. The second part of the program (the script) is straightforward and can be generated by the application program itself. The script is unique to each document.

The above example shows how to get things done easily with POSTSCRIPT. When an application uses a specific number of fonts with given sizes, it should place appropriate definitions for making and using those fonts in the prologue. After this is done the application program can generate calls to its subroutines to switch between the fonts and print the text.

There are some extra facts to know about fonts. Associated with each character is its width (a distance to move to print the next character). In

some fonts this spacing is a constant, i.e., it does not vary from character to character. These fonts are called *fixed pitch fonts*, or *monospaced fonts*. Most fonts, however, have a different width associated with each character. Such fonts are called *variable pitch fonts*. In either case, POSTSCRIPT's **show** operator moves the current position by the amount of the character width after it prints the character. This movement ensures that characters are spaced properly.

The width information for each character is stored in the POSTSCRIPT dictionary that represents the font. A POSTSCRIPT program may access this information to obtain a character's width, and the program may use any of a variety of character printing operators (**show, widthshow, ashow,** and **awidthshow**) to obtain a variety of width modification effects. For complete control over character placement, a POSTSCRIPT program may even place each character individually, based on this width information and the program's own placement algorithm.

Summary

We are now in a position to evaluate POSTSCRIPT along the lines of the questions given earlier in this chapter.

Is it complete? (Can it describe any printed page?)

The answer is a qualified "Yes". Certainly, any page consisting of marks on paper can be described in POSTSCRIPT. However, this could be claimed for any description language that allowed individual dots on a page to be described, even if tied to a particular device at a particular resolution. The POSTSCRIPT model of the printed page description is one in which the page image is ideal, is described once, and is rendered as well as possible on any raster printer. Even with this stricter model of page images, the answer for the POSTSCRIPT language is "Yes", as its imaging model is rich enough to describe all shapes that may be placed on the page.

So, it becomes necessary to rephrase the question to: What pages can *reasonably* be described in this language? Here we can be more specific. Pages consisting of text (in any typeface, with any linear transformations), line graphics, and filled area graphics are easy. The simple text handled by other print formats is very simple in POSTSCRIPT. Pages that contain photographic images are also easy, provided the program source contains a sampled description of those photographs. Fine typography, suitable for

books, advertising, documentation, and general printing is where POSTSCRIPT shines. Due to its programmable nature, precise alignment, tuning to output device quirks, synthetic images, etc. are all possible.

Although the POSTSCRIPT language is Turing-equivalent (it can perform any computation that any other programming language can express) this can lead us into the trap that it is reasonable to do everything in POSTSCRIPT. Some calculations are more reasonably performed by other systems, whose operations are oriented to other applications. While one could express the calculations necessary to render a fractal mountain vista in POSTSCRIPT, it is probably not practical to do so. Such synthetic graphics are more appropriately performed by specially set up systems. However, if such systems can output gray-scale sampled images, POSTSCRIPT is a most appropriate vehicle for printing the results on any raster output device.

Is it easy to generate?

POSTSCRIPT has been designed to be easy to generate by both programmers and by programs. A POSTSCRIPT program's syntactic form is printable characters, so that it is readable and editable with existing tools. Unit sizes and coordinate systems are all modifiable to be convenient for the application. Grouping elementary operations together in procedures allows levels of abstraction to be built up so that the operations required for a particular application can be expressed in ways appropriate to that application.

POSTSCRIPT can be generated by programs with access to few resources or by those with access to many resources. Programs that have very little resources, such as those running on small computers, may proceed by inserting a clever preamble at the beginning of a POSTSCRIPT file with enough procedures to make the output of the page description very simple. The postfix syntax of POSTSCRIPT files requires a generating program to carry very little state about the POSTSCRIPT file as it is being generated. Thus, even very limited systems can generate high quality output by taking advantage of the processing power available in POSTSCRIPT.

Depending on the resources available, a program generating POSTSCRIPT programs can make most of the decisions regarding the appearance of printed pages itself and express these in precise POSTSCRIPT operations, or it can allow procedures written in POSTSCRIPT and included in the generated program to make those decisions at printing time. The first case will generally result in more time required to generate the POSTSCRIPT program, with the POSTSCRIPT execution going very fast. The second case reverses these efforts, and is very suitable for a resource-limited generating program. In any case, the computation trade-off between effort devoted while producing POSTSCRIPT versus effort in executing POSTSCRIPT is available, and well thought out programs can make use of these trade-offs to their advantage.

Is it easy to interpret?

The surface design of the POSTSCRIPT language is very simple, and thus an interpreter for its basic structure is easy. Making that interpreter run fast is another matter requiring considerable work and insight. As for the interpretation of POSTSCRIPT's graphics, its very general model (self intersecting paths, etc.) requires sophisticated implementation. A more restricted subset of POSTSCRIPT graphics can be handled with a simpler implementation, but surprisingly innocuous page designs require the full power of POSTSCRIPT.

Rendering fonts on raster printers pushes the graphics implementation to its limits. Whereas small inaccuracies in general graphics may not be noticed, even the slightest imperfections in rendered characters can be very offensive.

So while simple interpretation of POSTSCRIPT is easy, sophisticated interpretation of POSTSCRIPT with graphics is not. Fortunately, a fast POSTSCRIPT interpreter with excellent graphics rendering capablities is available.

Is it easy for a person to understand it?

This question can be understood in two ways: is it easy to understand the imaging model?, and is it easy to understand page descriptions written in this format? In the first case, POSTSCRIPT's imaging model is a simple one with much expressive power. People with graphic arts or computer science backgrounds should have no difficulty in understanding how pages are put together in POSTSCRIPT.

Programs in any language are as easy or as hard to understand as the structure of those programs allow. POSTSCRIPT is written in printable characters, so at least the surface structure of a POSTSCRIPT program is easy to read. When things go wrong in a POSTSCRIPT program (e.g., during the debugging phase of bringing up a new application program that emits POSTSCRIPT) being able to read the program source directly is a great help. Subtle problems can be eliminated by using standard debugging techniques and running POSTSCRIPT programs interactively. In fact, the highly interpretive nature of POSTSCRIPT allows debugging tools to be written in the POSTSCRIPT language itself.

Is it valid for more than one printer?

The very nature of POSTSCRIPT is that it is a device independent page description language. The POSTSCRIPT interpreter is almost entirely independent of any specific output device. Sometimes a small amount of device specific software is needed to interface POSTSCRIPT to a new raster printer; in practice this is quite simple. Prior to publication of this manual, POSTSCRIPT has already driven many raster printers, from several different manufacturers and in a wide variety of resolutions.

Is it easy to transmit?

The POSTSCRIPT syntax deliberately avoids any machine dependent quirks in representation by staying within the printable character set. Any computer file system or communication system worthy of the name must be able to handle simple character files such as POSTSCRIPT programs.

Is it compact?

POSTSCRIPT programs can be verbose or compact, depending on the methods used to generate them. Descriptions of very complicated pages of graphics can be shortened substantially through the appropriate use of procedure definitions. Simple text pages also do not require much overhead, since with short names defined for common compound operations on these pages (such as move-to-next-line), the characters in the POSTSCRIPT program that are actually data (the characters to be printed) can be more than 90 per cent of the characters in the program.

When generating POSTSCRIPT programs for interpreters that have no file system, so that all data must be in the POSTSCRIPT program itself, scanned image source (for photographs) can take double the space it might otherwise need. This is due to POSTSCRIPT's representation of binary data in hexadecimal form, so that 8 bits of binary data requires two hexadecimal characters (16 bits worth) for their representation.

Can pieces of a description already built be used to compose more complicated pages?

Emphatically yes. The design of POSTSCRIPT encourages building pieces and templates that are used and reused to build up a page image. Not only can pieces be reused in exactly their original form, but with parameters, executable forms, translation, rotation and scaling, previously defined pieces can serve in a myriad of ways for making new composite pages.

Can it emulate other formats?

There are two distinct ways in which other print formats can co-exist with POSTSCRIPT. One is off-line translation, the other is direct emulation. Off-line translation means that some other program translates a different print format into POSTSCRIPT. Whenever possible, this is the preferred technique, since each print file need be translated only once. Direct emulation means that a POSTSCRIPT preamble that implements an interpreter for the other format be inserted before the other print file. This combination file is then sent to the POSTSCRIPT interpreter, which in executing the POSTSCRIPT program actually interprets the other format.

Several popular pre-POSTSCRIPT print file formats have already been emulated in POSTSCRIPT by these techniques. In general, a print file format to POSTSCRIPT translator is easy, since the POSTSCRIPT imaging model is so rich, and the POSTSCRIPT print file is executable.

Some print formats (derived from interactive screen raster graphics) include operations that require reading bits that were previously written (as with flood-fill operations or exclusive or-ing existing partial pages.) These operations are not device independent, and have no analog in POSTSCRIPT's painting model. Thus, direct emulation of such operations is not possible with POSTSCRIPT. These operations are usually parts of some larger sequence of operations that can be succesfully translated into POSTSCRIPT, e.g., make outline, draw outline, flood-fill can be translated to make-path, fill in POSTSCRIPT.

In summary, POSTSCRIPT is an interpretive, dynamic, page description programming language. It has features that fit together well, with the programming features designed to be convenient for the page description application. The design of the language was also influenced by the need to answer all of the questions considered here in the affirmative. A careful examination of the complete problem of computerized raster printing will reveal why the choices made for the POSTSCRIPT language were made.

3

REFERENCE SECTION

Data Structures and Types

3.1.1. POSTSCRIPT Stacks

The POSTSCRIPT interpreter manages four distinct stacks. The *operand stack*, the *dictionary stack*, and the *execution stack* each contain POSTSCRIPT objects. The *graphics state stack* contains snapshots of the graphics state.

Section 2.6.1 introduced the operand stack. It holds any kind of POSTSCRIPT object, and it is the temporary holding area for arguments and results of operators. Section 2.6.1 also introduced the dictionary stack, which provides the naming context for POSTSCRIPT programs. The dictionary stack can hold only POSTSCRIPT dictionary objects. A POSTSCRIPT program may access (read and write) both the operand stack and the dictionary stack through POSTSCRIPT operators already introduced.

The execution stack is a structure internal to the POSTSCRIPT interpreter. This stack holds procedure bodies (array objects) and other executable objects. At any point in the execution of a POSTSCRIPT program, the execution stack is the *call stack* of the program. Because of the intimate relationship between the execution stack and the correct operation of the interpreter, POSTSCRIPT programs are prohibited from writing in the execution stack; i.e., POSTSCRIPT provides no operators for writing into this stack directly.

The graphics state stack is also an internal structure. This stack holds instances of the graphics state, briefly outlined in section 2.7.1, and explained in detail in section 3.5. Values in the top-most graphics state may be examined and modified with many of the graphics operators in section 3.5. The stack may be pushed and popped with the **gsave**, **grestore**, and **grestoreall** operators.

3.1.2. POSTSCRIPT Objects

The complete list of object types supported by POSTSCRIPT is:

1. Integer
2. Real
3. Boolean
4. Array
5. Dictionary
6. String
7. Name
8. Operator
9. File
10. FontID
11. Mark
12. Null
13. Save

3.1.2.1. Integer and Real

POSTSCRIPT provides *integer* and *real* (floating point) numbers. Most arithmetic and mathematical operators can be freely applied to numbers of both types, with automatic type conversion taking place. In addition, explicit type conversion may be performed when the operand is in range. Other operators expect only integers (or a subrange of the integers) as proper arguments. The binary representation of floating point numbers is not exposed to the user, while the (machine dependent) representation of integers is exposed through bitwise operations.

3.1.2.2. Boolean

POSTSCRIPT provides objects with *boolean* values (*false* and *true*) for use in conditional and logical expressions. Booleans are the results of the relational (comparison) operators, logical operators, and are also returned as status from a variety of operations. The names **true** and **false** return the two values of this type.

3.1.2.3. Array

POSTSCRIPT *arrays* are one-dimensional collections of objects. The main difference between dictionaries and arrays is that dictionaries are accessed by name of element whereas arrays are accessed by numeric index. POSTSCRIPT arrays are different from arrays in most other computer languages. POSTSCRIPT arrays may be heterogeneous; that is, an array's elements may be any combination of numbers, strings, dictionaries, other arrays, or any other POSTSCRIPT objects. POSTSCRIPT directly provides only linear arrays, i.e., *vectors* – arrays with one dimension. Arrays of higher dimension may be constructed by using arrays as elements of arrays, nested arbitrarily deeply. All POSTSCRIPT arrays are indexed from 0, so an

array of *n* elements has indices from *0* through *n-1*. Note that all accesses to POSTSCRIPT arrays are bounds checked, and improper references result in an error. POSTSCRIPT arrays may also be protected to read-only or execute-only access. POSTSCRIPT's procedure bodies are executable arrays.

The POSTSCRIPT interpreter distinguishes between *array storage* and an *array object*. Array storage is the portion of POSTSCRIPT's virtual memory where the array's elements are stored. An array object contains a description of the array's length and a pointer to its associated array storage. Several array objects may point to the same (or portions of the same) array storage.

3.1.2.4. Dictionary

POSTSCRIPT has operators that manipulate general associative tables called *dictionaries*. A dictionary is a table whose elements are pairs of POSTSCRIPT objects. We call the first element of a pair the *key* and the second element the *value*. Though it is most common to use a name for a key, any POSTSCRIPT object except a string and null may be used as a key. A string is automatically converted into a name when used as a key.

The POSTSCRIPT interpreter always keeps one dictionary called the *system dictionary*. This dictionary associates the basic operator names (those defined in this document) with the internal operations that implement them. A user cannot modify this dictionary. The interpreter also keeps a second dictionary called the *user dictionary*. Any user may freely modify this dictionary; it provides the outermost modifiable naming context for POSTSCRIPT programs.

The POSTSCRIPT interpreter also maintains a stack called the *dictionary stack*. This stack always contains the system dictionary as its bottommost element and the user dictionary as the element above that. Other dictionaries that provide additional naming context for a POSTSCRIPT program may also be present on the dictionary stack. We call the topmost dictionary on this stack the *current dictionary*. When the POSTSCRIPT interpreter encounters an executable name, it searches for that name as a key in the current dictionary. If the interpreter finds this key in that dictionary, it proceeds, using the associated value. If the key is not in the current dictionary, the interpreter searches each successive dictionary in the dictionary stack (ending, if necessary, with the system dictionary) until it finds that key. If the interpreter cannot find the name in any of these dictionaries, it executes the error operator **undefined**.

3.1.2.5. String

POSTSCRIPT provides a general mechanism for operating on *strings* of characters. POSTSCRIPT's string implementation distinguishes between *string bodies*, which hold the characters contained in a string, and *string objects*, each of which contains the string's length and a pointer to an as-

sociated string body. The POSTSCRIPT interpreter never puts a string body on the operand stack; the only access a POSTSCRIPT program has to a string body is via operators that manipulate string objects.

As with an array, the elements of a string (its characters) are indexed starting at 0. Thus, a string whose length is *n* has valid character indices *0* through *n-1*. Note that all string accesses are bounds checked, and improper references result in an error. POSTSCRIPT uses non-negative integers in the range from 0 to 255 to represent characters, the elements of strings. POSTSCRIPT has no distinguished object type for characters; neither does it have any syntax specifically designed for representing character values. Generally, an implementation of POSTSCRIPT uses the ASCII character codes for characters; however, an installation may wish to use many non-standard characters, such as ligatures, foreign characters, accents, unusual punctuation, etc. Therefore, the POSTSCRIPT language does not enforce any particular character set encoding; it remains flexible to accommodate the different requirements of different installations.

3.1.2.6. Name

Names are the most common kind of keys in POSTSCRIPT dictionaries. Strings cannot be keys, but are converted to names when used as keys. Key-value pairs are the closest thing POSTSCRIPT has to variables and values, and names are POSTSCRIPT's most common keys.

A name object is much like a string; and in fact, names and strings may be used interchangeably in many contexts. However, a name has an important additional property; uniqueness. Any two names that are lexically the same are in fact the same name object.

3.1.2.7. Operator

POSTSCRIPT's built-in commands or functions are *operators*. Most of this manual is dedicated to describing the semantics of these built-in operators. Operators have names (most often they are defined in the system dictionary) and their values are procedures in the implementation of POSTSCRIPT itself that realize their desired function. POSTSCRIPT also has some internal operators, not documented in this manual, which may be encountered if a program reads the execution stack.

3.1.2.8. File

Files are readable or writable streams of characters (bytes). They may be used for running stored POSTSCRIPT programs, reading scanned images, or almost any other purpose. The exact number of allowed file streams, and the file naming conventions, tend to be quite implementation and site specific. POSTSCRIPT always provides standard input and output streams, however.

3.1.2.9. FontID

FontIDs are POSTSCRIPT's internal method for identifying and keeping track of fonts (typefaces). The IDs are necessary for POSTSCRIPT's font bitmap caching mechanism to operate properly. Most user programs need not be concerned with (or even aware of) objects of this type. The **definefont** operator creates objects of this type.

3.1.2.10. Mark

Mark objects only allow one value; all marks are equal to each other. Marks may be used to mark a position on the operand stack, and are described in detail in the section on stack manipulation operators. Marks are created with the [and **mark** operators.

3.1.2.11. Null

The POSTSCRIPT interpreter uses *null* objects to fill empty or uninitialized positions in composite objects. The **array** operator creates an array object whose elements are initialized to null objects. The key **null** (defined in the system dictionary) returns a null object.

3.1.2.12. Save

Save objects reference POSTSCRIPT interpreter state snapshots that are manipulated by the POSTSCRIPT **save** and **restore** operators, which are described in section 3.4.11.

3.1.3. Composite Objects

POSTSCRIPT arrays, dictionaries, and strings are called *composite* objects, in that they require extra storage (the "*body*" part). In the current implementation of the POSTSCRIPT interpreter, each object consists of a fixed length part containing its type, some bookkeeping information, and either a value or pointer. This fixed length part is called the *primary part.* For all other objects, the value of the object is carried in the primary part. For composite objects, the primary part carries a pointer to the value.

The POSTSCRIPT operand stack, for example, is implemented as a linked list of objects. When the POSTSCRIPT interpreter puts an object onto the stack, it only stores the primary part. The POSTSCRIPT operator **dup** performs the action of duplicating the top element on the operand stack, i.e., it pushes onto the operand stack an additional object identical to its former top element. Since objects may contain pointers, the **dup** operator does not necessarily duplicate an object's data. In other words, if an object contains a pointer to its data, only the pointer is copied, not the data that it points to; i.e., it only duplicates the primary part.

Similarly, when the interpreter puts an object into an array or dictionary, it only stores the primary part. Thus, the POSTSCRIPT interpreter deals more with references than with values. It is important to remember this

when writing programs. All object primary parts are the same size; therefore, the implementation of stacks and arrays is quite straightforward. Note that the elements of an array or dictionary are themselves POSTSCRIPT objects, while the elements of a string are individual characters which have no direct representation in POSTSCRIPT. Arrays are implemented as a linear sequence of objects. Since all object primary parts have the same length, indexing into an array is straightforward. POSTSCRIPT procedure bodies are merely executable arrays. The execution rule for arrays specifies that each element will be executed in turn (see section 3.3).

Argument and Error Handling

In general, each POSTSCRIPT operator removes the arguments it needs from the operand stack before it carries out its operation. All of the intrinsic operators then *type check* their arguments before doing any computing. If an operator discovers a mismatch between the type of an argument on the operand stack and the type it expects for that argument, it puts all of its arguments back onto the operand stack and executes the error operator **typecheck**. If the operand stack becomes empty prematurely while an operator is removing its arguments, the operator restores the operands it removed so far and executes the error operator **stackunderflow**. When a POSTSCRIPT operator finishes its execution, it pushes its return values onto the operand stack. If the stack becomes full during this process, the operator executes the error operator **stackoverflow**. (For the maximum allowed size of the operand stack, see Appendix B.)

Numerical arguments are a tricky issue in any programming language. POSTSCRIPT numbers include both integers and floating point reals. Internally, the POSTSCRIPT interpreter does make a distinction between these two representations for numbers. Certain POSTSCRIPT operators require numeric arguments; some of these further expect arguments to be integers, non-negative integers, or within some other restricted subrange of the integers. Some operators will perform implicit conversions from real numbers to integers; the descriptions of these operators indicate which conversions they perform and what errors may result. When an argument has the wrong type, e.g., string supplied but number expected, then an operator will execute the error operator **typecheck**. If a conversion is attempted but cannot be carried out, e.g., negative number supplied but non-negative integer expected or very large floating point number (out of the integer range) supplied but integer expected, then the operator will execute the error operator **rangecheck**.

Note that the POSTSCRIPT operators may perform implicit conversion only on numbers. In particular, most operators will not convert strings to numbers. POSTSCRIPT does contain a set of explicit type conversion operators that can convert values across a wide range of type boundaries. These are explained in section 3.4.9.

Immediate and Delayed Execution

The POSTSCRIPT interpreter scans POSTSCRIPT programs in a strict left-to-right manner. As the interpreter encounters a syntactic entity, it takes some action immediately. When it encounters a number or a string, it normally pushes that object onto the operand stack. When it encounters a name, it normally looks up the name in the dictionary stack to determine whether its value is *executable*. If the value is executable (operators and procedure bodies are executable), it executes that value immediately. Otherwise, it pushes that value onto the operand stack.

The curly brace characters, "{" and "}" delimit a range of POSTSCRIPT source code that the POSTSCRIPT interpreter does not execute immediately. Instead, the interpreter builds an executable object composed of the entire contents of the matching braces, and it pushes this object onto the operand stack. Subsequent operators may treat this object as they wish: some operators like **if** may execute the object; others, like **def**, may just store the executable object somewhere. To clarify these points, consider the following examples. If the POSTSCRIPT interpreter encounters

```
/abc dup def
```

it will push the literal name "abc" onto the operand stack (without interpreting it further, this is the significance of the "/"), push a duplicate of this name on top of that, and define the name "abc" to have the name object "abc" as its value. On the other hand, if the POSTSCRIPT interpreter scans

```
/abc {dup} def
```

it will push the literal name "abc" onto the operand stack as before, push the executable object consisting of the name **dup** onto the operand stack, and define the name "abc" to have the procedure body consisting of a single **dup** as its value. If the POSTSCRIPT interpreter subsequently encounters an **abc**, its behavior will be different depending on which of the above definitions was used. In the first case, the interpreter will push the literal name value "abc" onto the operand stack. In the second case, the interpreter will execute a **dup** operation.

Using the curly brace construct is the standard method for defining functions in POSTSCRIPT. For example, the sequence

```
/ave {add 2 div} def
```

associates an executable procedure body that averages two numbers on the operand stack with the name "ave". A later execution of the sequence

```
40 60 ave
```

results in the execution of the procedure body on the two arguments, 40

and 60. The POSTSCRIPT interpreter will execute the contents of the procedure body, **add**ing to get the value 100, and **div**ing by 2 to end up with the result, 50, on the operand stack.

The POSTSCRIPT interpreter implements executable procedure bodies as arrays that have their executable flag set. Thus, arrays can be converted to procedure bodies, and procedure bodies can be converted back to non-executable arrays quite easily. These facts are important if you wish to treat executable objects in POSTSCRIPT as data. Also, the POSTSCRIPT interpreter looks up the current associations of the names contained in a procedure body only when executing that procedure. Thus, in the example we have been using in this section, after defining **ave** to be **{add 2 div}**, we could change the definition of **div**, and subsequent execution of **ave** would use the new definition of **div**.

POSTSCRIPT draws another fine distinction between *names*, *commands*, and *executable functions*. The first point to understand is that the executable property is independent of the type of an object. Thus, executable is not itself a type, but any object of any type may be either executable or non-executable. The executable property really matters for arrays, commands, strings, files, and nulls; it is irrelevant when applied to objects of others types. For example, an executable integer, when executed, merely puts itself on the operand stack.

Operators are sections of code written in the C Language (or other implementation language) that actually implement the operators built into the POSTSCRIPT language. New functions defined by POSTSCRIPT source code, using curly braces and the **def** operator, are implemented as array objects that are executable. Both of these kinds of executable code are generally referenced in POSTSCRIPT programs by names, which are the identifiers in the actual POSTSCRIPT source.

For example, consider the POSTSCRIPT source code:

```
/plus1 {1 add} def
```

The POSTSCRIPT scanner will recognize three items at the top level; these are a literal name, an executable array, and a name. The executable array in turn is recognized to be composed of two elements, an integer and a name. The interpreter executes this code by executing each item in turn. It pushes the literal name "**plus1**" on the operand stack. It pushes the array "**{1 add}**" on the operand stack. Remember that curly braces indicate delayed execution, i.e., when encountered by the interpreter, the procedure body is placed on the operand stack, not on the execution stack. Finally, the interpreter encounters the name "**def**". It looks up the name, finds it in the system dictionary, and discovers that its value is an executable operator. When a name is looked up, and its value is executable, the interpreter executes that value. In the case of an operator, it executes it directly. In the case of an executable array, it pushes that array on the execution stack. In this case, the **def** operator is executed immediately, resulting in the name **plus1** being defined as **{1 add}** in the current dictionary. Now consider the execution of:

```
5 plus1
```

The scanner recognizes two items, an integer and a name. The interpreter pushes the integer 5 on the operand stack, and then it processes the name **plus1**. It looks up **plus1** and finds its value to be the executable array, **{1 add}**. Because this array was found as a result of the name lookup, the interpreter pushes this array onto the execution stack. Now the interpreter deals with the top item on the execution stack, which is the array just pushed. It processes the elements of this array in order, by removing the array's first element, pushing the remainder of the array (if any) back onto the execution stack, and dealing with the element it took as if it came from the input stream. So, in the example, the interpreter pushes the integer 1 onto the operand stack, and it processes the name **add**. The name **add** is in the system dictionary, and its value is an executable operator, which the interpreter executes, resulting in the 5 and 1 being popped from the operand stack and a 6 being pushed onto it.

Any object that is not executable (even an operator can be set to non-executable) that is pushed onto the execution stack (as by the **exec** operator) merely moves itself to the operand stack. Executable objects other than operators, arrays, strings, files, and nulls also just move themselves to the operand stack. The above example demonstrated how executable arrays and executable operators are handled as results of a name lookup. It remains for us to describe how executable strings are handled.

The POSTSCRIPT interpreter pushes an executable string that is the value of a name lookup onto the execution stack. From there, the interpreter removes the string, scans the first token out of the string, pushes the remainder of the string (if any) onto the execution stack, and deals with the token just obtained as if it had just been read out of the input buffer. Thus, executable strings and executable arrays are treated identically, except that the strings are scanned at execution time, whereas the arrays were pre-scanned.

File streams may also be executable (the standard input stream is an example). If an executable (readable) file is moved to the execution stack, the effect is just as if the file had been **run**, except that the starting position in the file is the current stream position. The contents of the file are scanned and interpreted until an end of file is reached. The file stream is then closed, and the file is removed from the execution stack.

Null objects may also be executable. Executing a null has no effect (i.e., is a "no-op").

Thus, at any given point in a computation, the execution stack may contain the remainders of many executable arrays, strings, and file objects. Non-executable objects may exist on this stack only ephemerally, as their immediate execution removes them to the operand stack.

POSTSCRIPT Operators

The POSTSCRIPT operators divide naturally into groups, corresponding to both the functions and the objects that POSTSCRIPT supports. This section is divided into subsections according to these groups. These subsections describe all of the intrinsic POSTSCRIPT operators. Each operator description is presented in the following format:

arg_1 arg_2 ... arg_N **operator** $result_1$ ... $result_M$

The block of text in this position explains the operator. Arg_1 through arg_N are the arguments expected by the operator, with arg_N being the topmost element on the operand stack. $Result_1$ through $result_M$ are the objects left on the stack as a result of executing the operator with $result_M$ being the topmost element left on the stack. Normally the names indicate the type of the operand or result. For example, the name num_1 indicates that the argument or result is a number.

Example:
```
an example of the use of this operator is given here
```

Errors: a list of the error operators that this operator might execute is given in this position.

In addition, the notation "⊢" indicates the bottom of the stack. The notation "–" in the arguments position indicates that the operator expects no arguments, and a "–" in the results position indicates that the operator returns no results.

When the notation *proc* is used for an argument, you must be careful to supply an executable array (procedure body) without actually executing it prematurely. Thus, if you have defined **myop** to be some executable procedure, use

```
7 {myop} repeat
```

rather than

```
7 myop repeat.
```

3.4.1. Stack Operators

The operand stack is the POSTSCRIPT interpreter's mechanism for passing arguments to operators and for gathering results from operators. POSTSCRIPT provides a variety of operators that rearrange elements on this stack. Such rearrangement is often required when the results of one operator are to be passed as arguments to another operator that expects its operands in a different order. The group of stack manipulation operators, in addition to providing the obvious stack operations, allow duplicating portions of the operand stack (**copy**), treating a portion of the operand stack as a circular queue (**roll**), and treating the operand stack as an indexable array (**index**).

♦ **pop**

`any` **`pop`** –

removes the top element from the operand stack.

Example:
```
1 2 3 pop => 1 2
1 2 3 pop pop => 1
```

Errors: stackunderflow.

♦ **dup**

`any` **`dup`** `any any`

duplicates the top element on the operand stack. Note that **dup** only copies the primary part of a composite object, not the storage it refers to, so array, dictionary, and string bodies are not duplicated. See section 3.1.2.

Errors: stackoverflow, stackunderflow.

♦ **exch**

any_1 any_2 **`exch`** any_2 any_1

exchanges the top two elements on the operand stack.

Example:
```
1 2 exch => 2 1
```

Errors: stackunderflow.

♦ roll

any_{N-1} .. any_0 N J **roll** $any_{(J-1) (mod\ N)}$ .. any_0 any_{N-1} .. $any_{J (mod\ N)}$

rotates the positions of the arguments any_{N-1} through any_0 on the operand stack by the amount *J*. *N* must be a non-negative integer, and *J* must be an integer. **roll** first removes the the top two arguments from the operand stack. **roll** is a circular shift of the top *N* elements on the operand stack. Consider the top *N* elements to be a substack. Then, if *J* is positive, **roll** shifts the elements from the top to the bottom of the substack. If *J* is negative, **roll** shifts the elements from the bottom of the substack to the top.

Example:

```
(a) (b) (c)  3 -1 roll => (b) (c) (a)
(a) (b) (c)  3 1 roll => (c) (a) (b)
(a) (b) (c)  3 0 roll => (a) (b) (c)
```

Errors: rangecheck, stackoverflow, stackunderflow, typecheck.

♦ index

any_N ... any_0 I **index** any_N ... any_0 any_I

removes *I* from the operand stack, counts down to the *I*th element from the top of the stack, and pushes a copy of that element onto the stack. *I* must be a non-negative integer.

Example:

```
(a) (b) (c) (d)  0 index => (a) (b) (c) (d) (d)
(a) (b) (c) (d)  3 index => (a) (b) (c) (d) (a)
```

Errors: rangecheck, stackunderflow, typecheck.

♦ clear

⊢ any_1 ... any_N **clear** ⊢

removes all objects from the operand stack.

Errors: (none).

♦ count

⊢ any_1 ... any_N **count** ⊢ any_1 ... any_N N

counts the number of items on the operand stack and pushes this count onto the stack.

Example:
```
clear count => 0
clear 1 2 3 count => 1 2 3 3
```

Errors: stackoverflow.

♦ mark

– **mark** mark

pushes a mark (an object of type **marktype**, not the **mark** operator itself) onto the operand stack. All marks are identical, and the operand stack may contain many of them at any given time. Marks are useful for flagging the end of an arbitrarily long list of arguments that may be passed to some procedures. Another common use for marks is for debugging or protecting against POSTSCRIPT code that is suspected of tampering with the operand stack below the level at which it is called. This technique is not guaranteed to reveal all problems with suspect code, but it often causes a typecheck error when faulty stack manipulation occurs. Such debugging code should explicitly pop a mark off of the operand stack after it has served its purpose.

Errors: stackoverflow.

♦ cleartomark

mark ~$mark_1$ ... ~$mark_N$ **cleartomark** –

pops the operand stack repeatedly until it encounters a mark, which it also removes from the stack. The notation *~mark* stands for an object of any type except **marktype**.

Errors: unmatchedmark.

♦ counttomark

`mark ~mark`$_1$ `...` `~mark`$_N$ **`counttomark`** `mark ... ~mark`$_N$ `N`

counts the number of objects on the operand stack starting with the top element, down to but not including the first mark encountered. The notation *~mark* stands for an object of any type except **marktype**.

Example:

```
1 mark 2 3 counttomark => 1 mark 2 3 2
1 mark counttomark => 1 mark 0
```

Errors: stackoverflow, unmatchedmark.

3.4.2. Arithmetic and Math Operators

Since POSTSCRIPT is a general purpose programming language, it provides the usual complement of arithmetic and mathematical operators. Although the numeric types Integer and Real are visible to the user, most arithmetic POSTSCRIPT operators accept either of these numeric representations. Thus, the descriptions that follow indicate that arguments and results have the type **number**; for the most part, a POSTSCRIPT program need not concern itself with which internal representation is used at any given time.

The POSTSCRIPT arithmetic operators automatically convert internal numeric representations from integer to real and vice versa, depending on how their arguments and results approach the limits of their representations. Depending on their input arguments, these operators can generate undefined results. When this happens, they execute the error operator **undefinedresult**.

♦ **abs**

num **abs** |num|

return the absolute value of *num*, the number on the top of the stack. When in range, the type of the result is the same as the type of the argument; otherwise the result is real.

Example:

```
4.5 abs => 4.5
-3 abs => 3
0 abs => 0
```

Errors: stackunderflow, typecheck.

♦ **add**

num_1 num_2 **add** (num_1 + num_2)

adds the top two elements on the stack. If both arguments are integers and the result is in range, the result is an integer; otherwise, the result is a real.

Example:

```
3 4 add => 7
-3 abs -4 add => -1
9.9 1.1 add => 11.0
```

Errors: stackunderflow, typecheck, undefinedresult.

♦ div

num_1 num_2 **div** (num_1 / num_2)

divides the element below the top element on the operand stack by the top element on the stack. The result is a real.

Example:

```
3 2 div => 1.5
4 2 div => 2.0
```

Errors: stackunderflow, typecheck, undefinedresult.

♦ idiv

int_1 int_2 **idiv** integer-part(int_1 / int_2)

divides the element below the top element on the operand stack by the top element on the stack, and returns only the integer part of the result onto the stack. Both operands of **idiv** must be integers. The result is an integer.

Example:

```
3 2 idiv => 1
4 2 idiv => 2
-5 2 idiv => -2
```

Errors: rangecheck, stackunderflow, typecheck, undefinedresult.

♦ mod

int_1 int_2 **mod** (int_1 MOD int_2)

returns the integer remainder that results from dividing int_1 by int_2. The sign of the result is the same as the sign of the dividend int_1. Both operands must be integers. The result is an integer.

Example:

```
5 3 mod => 2
5 2 mod => 1
```

Errors: stackunderflow, typecheck, undefinedresult.

♦ mul

num_1 num_2 **mul** (num_1 * num_2)

multiplies the top two elements on the stack. If both arguments are integers and the result is in range, the result is an integer; otherwise, the result is a real.

Errors: stackunderflow, typecheck, undefinedresult.

♦ neg

`num` **`neg`** `-num`

reverses the sign of the top element on the stack. When in range, the type of the result is the same as the type of the argument; otherwise, the result is a real.

Errors: stackunderflow, typecheck.

♦ sub

num_1 num_2 **`sub`** $(num_1 - num_2)$

subtracts the top element on the operand stack from the element below it on the stack. If both arguments are integers and the result is in range, the result is an integer; otherwise, the result is a real.

Errors: stackunderflow, typecheck, undefinedresult.

♦ sqrt

`num` **`sqrt`** `SquareRoot(num)`

returns the square root of the argument. *num* must be a non-negative number. The result is a real.

Errors: rangecheck, stackunderflow, typecheck.

♦ exp

num_1 num_2 **`exp`** $num_1^{num_2}$

raises num_1 (element below the top element on the operand stack) to the num_2 (top element on the stack) power. The result is a real.

Example:

```
9 0.5 exp => 3.0
9 -1 exp => 0.111111
```

Errors: stackunderflow, typecheck, undefinedresult.

♦ ceiling

num **ceiling** Ceiling(num)

returns the least integer value greater than or equal to *num* (the *ceiling* of *num*). The type of the result is the same as the type of the argument (to preserve the range of reals). **ceiling** is a no-op for arguments of type **integertype**, and returns a real for real arguments.

Example:

```
3.2 ceiling => 4.0
-4.8 ceiling => -4.0
99 ceiling => 99
```

Errors: stackunderflow, typecheck.

♦ floor

num **floor** Floor(num)

returns the greatest integer less than or equal to *num* (the *floor* of *num*). The type of the result is the same as the type of the argument (to preserve the range of reals). **floor** is a no-op for arguments of type **integertype**, and returns a real for real arguments.

Example:

```
3.2 floor => 3.0
-4.8 floor => -5.0
99 floor => 99
```

Errors: stackunderflow, typecheck.

♦ round

num **round** Round(num)

rounds *num* to the nearest integer value without type conversion. The type of the result is the same as the type of the argument (to preserve the range of reals). The effect of **round** on an integer is a no-op.

Example:

```
3.2 round => 3.0
6.5 round => 7.0
-4.8 round => -5.0
-6.5 round = -6.0
99 round => 99
```

Errors: stackunderflow, typecheck.

♦ truncate

`num` **`truncate`** `Truncate(num)`

truncates *num* toward zero by removing its fractional part. The type of the result is the same as the type of the argument (to preserve the range of reals). **truncate** is a no-op for arguments of type **integertype**, and returns a real for real arguments. The **cvi** operator (described on page 83) does truncation and type conversion to the nearest integer.

Example:

```
3.2 truncate => 3.0
-4.8 truncate => -4.0
99 truncate => 99
```

Errors: stackunderflow, typecheck.

♦ atan

num_1 num_2 **`atan`** `ArcTangent(`num_1 `/` num_2`)`

returns the angle (in degrees between 0 and 360) whose tangent is num_1/num_2. Note: num_1 or num_2 may be zero, but not both. The signs of num_1 and num_2 determine the quadrant in which the result will lie. The result is a real.

Example:

```
0 1 atan =>   0.0
1 0 atan =>  90.0
-100 0 atan => 270.0
4 4 atan => 45.0
```

Errors: stackunderflow, typecheck, undefinedresult.

♦ cos

`num` **`cos`** `Cosine(num)`

returns the cosine of the top element of the stack (taken as an angle in degrees). The result is a real.

Example:

```
0 cos => 1.0
90 cos => 0.0
```

Errors: stackunderflow, typecheck.

♦ sin

`num` **`sin`** `Sine(num)`

returns the sine of the top element of the stack (taken as an angle in degrees). The result is a real.

Errors: stackunderflow, typecheck.

♦ ln

`num` **`ln`** `Ln(num)`

returns the natural logarithm (base *e*) of the top element of the stack. The result is a real.

Example:
```
10 ln => 2.30259
100 ln => 4.60517
```

Errors: stackunderflow, typecheck, undefinedresult.

♦ log

`num` **`log`** `Log(num)`

returns the common logarithm (base 10) of the top element of the stack. The result is a real.

Example:
```
10 log => 1.0
100 log => 2.0
```

Errors: stackunderflow, typecheck, undefinedresult.

♦ rand

– **`rand`** `int`

returns a random number. The **rand** operator uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to 2^{31}-1. The generator is reinitialized by executing **srand** with 1 as its argument. It can be set to any other starting point by executing **srand** with any other integer argument. The current seed may be interrogated with the **rrand** operator.

Errors: stackoverflow.

♦ srand

`int` **`srand`** –

Initialize random number generator with seed *int*. The generator is reinitialized by executing **srand** with 1 as its argument. It can be set to any other starting point by executing **srand** with any other argument. *int* must be an integer.

Errors: stackunderflow, typecheck.

♦ rrand

– **`rrand`** `int`

returns the current state of the random number seed (see **rand** and **srand**).

Errors: stackoverflow.

3.4.3. Polymorphic Operators

POSTSCRIPT has several operators that apply to objects of different types. Some of the operators described in this section operate on dictionaries, arrays, and strings in analogous ways. Type-specific operators are described in the sections related only to each type.

♦ **copy**

any_1 ...any_N N **copy** any_1 ... any_N any_1 ...any_N

duplicates the top *N* elements on the operand stack as a group. *N* must be a non-negative integer.

Example:

```
1 2 3 2 copy => 1 2 3 2 3
1 2 3 0 copy => 1 2 3
```

Errors: rangecheck, stackunderflow, stackoverflow, typecheck.

$array_1$ $array_2$ **copy** $subarray_2$
$dict_1$ $dict_2$ **copy** $dict_2$
$string_1$ $string_2$ **copy** $substring_2$

copies all the elements of the first composite object into the second, where the composite objects must be of the same type. In the case of an array or string, the length of the second object must be at least as great as the first; **copy** pushes a descriptor for the initial *subarray* or *substring* of the second object containing the copied contents of the first. In the case of a dictionary, $dict_2$ must have a **length** of zero and a **maxlength** at least as great as the length of $dict_1$. **copy** provides a "top-level" copy of the values in a dictionary or an array, copying only references to contained composite objects, not their bodies. Note that a **copy** of a composite object is not the same as a **dup**. **dup** merely copies the descriptor (reference) to the same storage (body), while **copy** makes a top level copy of the storage into a specified destination. **copy** also copies the protection attributes of the source object.

Example:

```
/a1 [1 2 3] def
a1 dup length array copy => [1 2 3]
```

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ length

```
  array  length  int
   dict  length  int
 string  length  int
```

depends on the type of its argument. If the argument is an array or string, **length** returns its length. If the argument is a dictionary, **length** returns the current number of key-value pairs it contains. (See also the **maxlength** operator in section 3.4.5 which returns the capacity of a dictionary.)

Example:

```
[1 2 4] length => 3
[] length => 0 % an array of zero length
/ar 20 array def ar length => 20

/mydict 5 dict def
mydict length => 0
mydict /firstkey (firstvalue) put
mydict length => 1

(abc\n) length => 4 % the "\n" is one character
() length => 0
```

Errors: invalidaccess, stackunderflow, typecheck.

♦ forall

```
 array proc  forall  –
  dict proc  forall  –
string proc  forall  –
```

enumerates the elements of the argument, executing the procedure body *proc* for each element. If the first argument is an array or string, **forall** pushes *element* and executes *proc* for each element in sequence. If the first argument is a dictionary, **forall** pushes *key* and *value* and executes *proc* for each key-value pair in the dictionary. The order in which **forall** enumerates the entries in the dictionary is arbitrary. (New entries in the dictionary created during the execution of **forall** may or may not be included in the enumeration.) If the first argument is a string, **forall** enumerates the elements of the string as integer character codes (between 0 and 255, inclusive), *not* as one-character strings. If the array, dictionary, or string is empty (i.e., has **length** 0), **forall** does not execute *proc* at all. Although **forall** does not leave any results on the operand stack when it is finished, the execution of *proc* may leave arbitrary results there (and may raise any error). In particular, if *proc* does not remove the enumerated arguments from the operand stack, they will accumulate there.

Example:

```
5 [1 2 3 4 5] {add} forall => 20
```

Errors: invalidaccess, stackunderflow, typecheck.

♦ get

```
  array index  get  array_index
     dict key  get  value
string index  get  string_index
```

gets a single element from an array, dictionary, or string. If the first argument is an array or string, **get** pushes the element of that array or string specified by *index* (counting from zero). If *index* is not a valid integer index for the array or string, **get** executes the error operator **rangecheck**. If the arguments are *dict key*, **get** searches *dict* for an entry with key *key* and pushes the associated *value* onto the operand stack. If *key* is not defined in *dict*, **get** executes the error operator **undefined**. Elements of arrays and dictionaries are POSTSCRIPT objects; elements of strings are integer character codes.

Example:

```
[31 41 59] 0 get => 31
[0 (a mixed-type array) [] {add 2 div}]
        2 get => [] % an empty array

/mykey (myvalue) def
currentdict /mykey get => (myvalue)

(abc) 1 get => 98   % ascii character 'b'
(a) 0 get => 97
```

Errors: invalidaccess, rangecheck, stackunderflow, typecheck, undefined.

♦ put

```
  array index value  put  –
     dict key value  put  –
string index value  put  –
```

stores an element into a specific object. If the first argument is an array or string, **put** stores *value* as the element of that array or string specified by *index* (counting from zero). *index* must be a valid integer index for the array or string. If the arguments are *dict key value*, **put** stores *value* into *dict* with *key*. *dict* need not be on the dictionary stack. If *dict* is full and has no current value for *key*, **put** executes the error operator **dictfull**. The argument *value* may be of any type for an array or dictionary, but it must be an integer character code for a string.

Example:

```
/ar [1 2 3 4] def
ar 2 (abcd) put % remember arrays are indexed from zero
ar 2 aload pop => 1 2 (abcd) 4

/st (abc) def
st 0 8#101 put % modify 'st' to be "Abc", using octal constant
```

Errors: dictfull, invalidaccess, rangecheck, stackunderflow, typecheck.

♦ getinterval

```
 array beg len  getinterval  subarray beg..beg+len-1
string beg len  getinterval  substring beg..beg+len-1
```

constructs a "subobject" of *len* elements, whose element values are the elements indexed by *beg* through *beg+len*-1 of *array* or *string*, and pushes this new object on the operand stack. This object does not have its own copy of the elements of the argument; it points within the same storage as the argument object. Thus if an element is modified through one of these objects, the corresponding element in the other object changes as well. Like all POSTSCRIPT arrays and strings, the subobject's indices start at 0. Thus, its indices are 0 through *len*-1, where

$$subobj_0 = obj_{beg}$$
$$subobj_1 = obj_{beg+1}$$
$$\ldots$$
$$subobj_{len-1} = obj_{beg+len-1}$$

The getinterval operator expects *beg* to be a valid index in *array* or *string*, and *len* to be a non-negative integer such that *beg+len*-1 is also a valid index in *array* or *string*.

Example:

```
[1 2 3 4 5] 1 3 getinterval => [2 3 4] % the middle three elements

(abcde) 1 3 getinterval => (bcd)
(abcde) 0 0 getinterval => () % an empty string
```

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ putinterval

```
  array1 beg array2  putinterval  –
string1 beg string2  putinterval  –
```

stores all the elements of $array_2$ ($string_2$) into the storage of $array_1$ ($string_1$) starting at the element of obj_1 indexed by *beg*. *beg* must be a non-negative integer in the range of valid indices of obj_1 such that *beg*+(length of obj_2)-1 is also a valid index of obj_1.

Example:

```
/ar [1 2 3 4 5] def
ar 1 [(a) (b) (c)] putinterval % 'ar' is now [1 (a) (b) (c) 5]

/st (abc) def
st 1 (de) putinterval % 'st' is now "ade".
```

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

3.4.4. Array Operators

PostScript provides operators to create and access arrays of PostScript objects. In the descriptions in this section, the notation $array_i$ means "the element of array at position i".

The polymorphic operators **copy**, **forall**, **get**, **getinterval**, **length**, **put**, and **putinterval**, described in section 3.4.3, may also be applied to arrays.

♦ **array**

```
int  array  array-of-size-int
```

creates an array of size *int*, each of whose elements is initialized to contain the Null object, and pushes this array onto the operand stack. The **array** operator expects *int* to be a non-negative integer.

Errors: rangecheck, stackunderflow, typecheck.

♦ **[**

```
– [  mark
```

marks the stack, expecting that the elements of an array to be constructed will follow, followed by a "]" operator that does the array construction. This operator is equivalent to the **mark** operator. Note that the "[" character is self-delimiting. This implies that the "[" operator need not be surrounded by blanks (or other delimiters) when used in a PostScript program.

Errors: stackoverflow.

♦ **]**

```
mark ~mark0 ... ~markN-1  ]  array
```

constructs an array of N elements, with the elements initialized to $\sim mark_0$, ... , $\sim mark_{N-1}$, and pushes this array onto the operand stack. The notation *~mark* means an object of any type except **marktype**. This operator is equivalent to the sequence:

```
counttomark array astore exch pop
```

Like the "[" character, "]" is also self-delimiting.

Example:

```
[5 4 3] => % a 3-element array, with elements 5, 4, 3
[1 2 add] => % a 1-element array, with element 3
```

Note that the PostScript interpreter acts on all the array elements as it encounters them (unlike its behavior with curly braces), so the **add** operator was executed.

Errors: unmatchedmark.

♦ aload

`array-of-size-N` **`aload`** $\texttt{array}_0$ ... $\texttt{array}_{N-1}$ `array-of-size-N`

successively pushes all *N* elements of the argument array onto the operand stack and finally pushes the array itself.

Errors: invalidaccess, stackoverflow, stackunderflow, typecheck.

♦ astore

$\texttt{any}_0$ ... $\texttt{any}_{N-1}$ `array-of-size-N` **`astore`** `array-of-size-N`

stores the arguments any_0 through any_{N-1} from the operand stack into the array storage pointed at by the *array-of-size-N* argument, leaving this array object on the operand stack. The **astore** operator first removes the top argument from the stack and determines its length. It then removes that number of objects from the operand stack, storing them, highest on the stack into the highest indexed element of the array through the lowest on the stack into the 0th element of the array.

Example:

```
(a) (b) (c) 3 array astore => [(a) (b) (c)]
```

This creates a three element array, stores the strings "a", "b", and "c" into its 0th, 1st, and 2nd elements respectively, and leaves the array object on the operand stack.

Errors: invalidaccess, stackunderflow, typecheck.

♦ null

– **`null`** `null`

returns a literal object of type **nulltype**.

Errors: stackoverflow.

3.4.5. Dictionary Operators

The dictionary related operators allow a POSTSCRIPT program to create dictionaries, to add key-value pairs to dictionaries, to look up a key in a dictionary, to enumerate the key-value pairs in a dictionary, to push dictionaries onto the dictionary stack, and to remove dictionaries from the dictionary stack. There are no operators that explicitly remove items from dictionaries. (However, see the description of the **save** and **restore** operators, which may restore dictionaries to a previous state.)

In the operator descriptions that follow, the arguments designated *key* and *value* designate key and value arguments respectively. The *value* arguments may be of any POSTSCRIPT object type. The *key* argument may be of any type except *null*; but if *key* is a string, it is converted to a name before being used.

The polymorphic operators **copy**, **forall**, **get**, **length**, and **put**, described in section 3.4.3, may also be applied to dictionaries.

♦ **dict**

```
int  dict  dict
```

creates a dictionary with a maximum capacity of *int* elements and pushes the created dictionary object onto the operand stack. *int* is expected to be a non-negative integer. If a subsequent dictionary operation attempts to create a new element within a dictionary that is already full, it will execute the error operator **dictfull**.

Errors: rangecheck, stackunderflow, typecheck.

♦ **begin**

```
dict  begin  –
```

pushes *dict* onto the dictionary stack, making it the current dictionary. Remember that the dictionary stack constitues a naming context for POSTSCRIPT programs, so **begin**ing a dictionary may establish a new context.

Errors: dictstackoverflow, invalidaccess, stackunderflow, typecheck.

♦ end

– **end** –

pops the current dictionary off of the dictionary stack, making the dictionary below it the current dictionary. If **end** tries to pop the bottommost instance of the user dictionary, it executes the error operator **dictstackunderflow**.

Errors: dictstackunderflow.

♦ def

key value **def** –

stores *value* with *key* in the current dictionary. If *key* is already in the current dictionary, **def** simply replaces its value. Otherwise, **def** creates a new entry for *key* and stores *value* with it.

Example:

```
/i 1 def   % define i to have value 1 in current dictionary
/i i 1 add def   % i now has value 2
```

Errors: dictfull, invalidaccess, limitcheck, stackunderflow, typecheck.

♦ store

key value **store** –

searches the dictionary stack from the current dictionary down to the system dictionary, until it finds *key*. When the **store** operator finds this key, it replaces any previous value associated with the key by *value*. If **store** cannot find the key in any dictionary on the dictionary stack, it creates a new entry in the current dictionary with *key* and *value*. **store** differs from **def** in that **store** may search the dictionary stack to any depth, whereas **def** searches only the current dictionary.

Errors: dictfull, invalidaccess, limitcheck, stackunderflow.

♦ known

```
dict key  known  boolean
```

returns the boolean value *true* if *key* is a key in the dictionary *dict*; otherwise returns *false*. *dict* does not have to be on the dictionary stack.

Example:

```
/mydict 5 dict def
mydict /total 0 put
mydict /total known => true
mydict /badname known => false
```

Errors: invalidaccess, stackunderflow, typecheck.

♦ load

```
key  load  value
```

searches the dictionary stack (from the top down) for *key* and returns the value associated with it. If *key* is not defined in any dictionary on the dictionary stack, **load** executes the error operator **undefined**.

Example:

```
/avg {add 2 div} def
/avg load => {add 2 div}  % the executable array of 3 elements
```

Errors: invalidaccess, stackunderflow, typecheck, undefined.

♦ where

```
key  where  if found: dict true
            if not found: false
```

searches the dictionary stack from the current dictionary down to the system dictionary until it finds *key*. If **where** finds the key, it returns the dictionary in which it found the key, and it returns the boolean value *true*. If it cannot find this key in any dictionary on the dictionary stack, it returns the boolean value *false*. Note that this operator returns either one or two result objects on the operand stack, depending on the boolean value returned.

Errors: invalidaccess, stackoverflow, stackunderflow.

♦ maxlength

`dict` **`maxlength`** `int`

returns the maximum number of keys that *dict* may hold, as defined by the **dict** operator. (See also the **length** operator in section 3.4.3 which returns the number of entries a dictionary contains.)

Example:

```
/mydict 5 dict def
mydict length => 0
mydict maxlength => 5
```

Errors: invalidaccess, stackunderflow, typecheck.

♦ systemdict

– **`systemdict`** `system-dictionary`

pushes the system dictionary onto the operand stack. That is, a new dictionary object containing a primary part that points to the system dictionary is pushed onto the operand stack. The dictionary object residing on the dictionary stack that points to the system dictionary remains there.

Errors: stackoverflow.

♦ userdict

– **`userdict`** `user-dictionary`

pushes the user dictionary onto the operand stack. That is, a new dictionary object containing a primary part that points to the user dictionary is pushed onto the operand stack. The dictionary object residing on the dictionary stack that points to the user dictionary remains there.

Errors: stackoverflow.

♦ currentdict

– **`currentdict`** `dict`

pushes the current dictionary (the dictionary on top of the dictionary stack) onto the operand stack. That is, a new dictionary object that points to the current dictionary is pushed on the operand stack. The dictionary object on top of the dictionary stack remains there.

Errors: stackoverflow.

♦ countdictstack

– **countdictstack** num

returns the number of dictionaries currently on the dictionary stack. This command is most often used to compute the size of the array parameter for the **dictstack** command described below.

Errors: stackoverflow.

♦ dictstack

array **dictstack** subarray

stores as many elements as the dictionary stack has dictionaries into the argument *array* and returns a object describing the initial N-element subarray of *array*, where N is the current depth of the dictionary stack. The dictionaries are placed in this array in order, with the system dictionary in element 0 and the current dictionary in element N-1.

Errors: rangecheck, stackunderflow, typecheck.

3.4.6. String Operators

The POSTSCRIPT string operators provide basic string manipulation facilities. POSTSCRIPT's string operators create strings of a given length, copy existing strings, build string objects that point to substrings of existing string bodies, search for substrings in a given string, and enumerate the characters of a given string. In the following descriptions, the notation $string_i$ denotes the character stored at the i'th position in the string body pointed at by string.

The polymorphic operators **copy, forall, get, getinterval, length, put,** and **putinterval,** described in section 3.4.3, may also be applied to strings.

♦ string

```
int  string  string
```

creates a string body whose length is *int*, initializes its character values to zeros (ASCII Nulls), and returns a newly created string object that references this string body. *int* is expected to be a non-negative integer.

Errors: limitcheck, rangecheck, stackunderflow, typecheck.

♦ anchorsearch

```
string seek  anchorsearch  if found: s-post s-match true
                           if not found: string false
```

(anchored search) is similar to **search**, but **anchorsearch** succeeds only if *seek* is an initial substring in *string*. If the initial substring of *string* with length equal to that of *seek* matches *seek*, the **anchorsearch** operator splits *string* into only two segments, *s-post*, the portion of *string* occurring after the initial *seek*, and *s-match*, the portion of *string* that matches *seek*. Like **search**, if the initial match fails, **anchorsearch** pushes the original *string* back onto the operand stack, and in any case, **anchorsearch** returns a boolean value on the top of the stack that indicates whether the search succeeded or not.

Example:

```
(abbc) (ab) anchorsearch => (bc) (ab) true
(abbc) (bb) anchorsearch => (abbc) false
(abbc) (bc) anchorsearch => (abbc) false
(abbc) (B) anchorsearch => (abbc) false
```

Errors: invalidaccess, stackoverflow, stackunderflow, typecheck.

♦ search

```
string seek  search  if found: s-post s-match s-pre true
                     if not found: string false
```

looks for the first occurrence of the string *seek* within the string *string*, returning results of this search on the operand stack. The result that **search** leaves on top of the operand stack is a boolean object that indicates whether the search succeeded or not. The **search** operator performs a simple equality comparison of *seek* with successive substrings of *string*. If **search** finds *seek* within *string*, it splits *string* into three strings, *s-post*, the substring of *string* following the portion that matches *seek*, *s-match*, the substring of *string* that matches *seek*, and *s-pre*, the substring of *string* that precedes the portion that matches *seek*. When **search** succeeds, it pushes these three string objects followed by the boolean value *true* onto the stack. If the search fails, **search** pushes the original *string* followed by the boolean value *false* onto the operand stack.

Example:

```
(abbc) (ab) search => (bc) (ab) () true
(abbc) (bb) search => (c) (bb) (a) true
(abbc) (bc) search => () (bc) (ab) true
(abbc) (B) search => (abbc) false
```

Errors: invalidaccess, stackoverflow, stackunderflow, typecheck.

♦ token

```
string  token  if found: s-post token true
               if not found: false
```

strips a token from the argument *string* (interpreted as POSTSCRIPT source code). If a valid POSTSCRIPT token is found, **token** pushes *s-post* (the substring of *string* following the token) the compiled *token*, and the boolean value *true*. If the token was self-delimiting (i.e., ended in a "}" or "]", etc.), then *s-post* will include all delimiters which followed the token; otherwise one delimiting character (e.g., a space) will be missing from *s-post*. If no valid POSTSCRIPT token was found in *string* (if *string* contained only delimiters and comments, for example), **token** pushes the boolean value *false*.

The **token** operator behaves like the POSTSCRIPT interpreter's scanner. It removes the initial portion of the argument string corresponding to a single POSTSCRIPT syntactic entity. This entity may be simple: a string, a number or a name; or it may be composite: an executable array extending from an initial left curly brace through its matching right curly brace. The *token* pushed onto the operand stack is a *compiled* POSTSCRIPT object that corresponds to the syntactic entity. Thus, the *token* pushed for a number is a number object, not its string representation. The *token* pushed for a curly brace delimited section of code is an executable array, all of whose components are similarly compiled. The **token** command does not execute this token; it merely pushes it onto the operand stack. The **token** command does *not* interpret backslash ("\") escape sequences inside string bodies, since a string object returned by **token** is a substring of its argument.

See also the description of the **token** operator for file arguments in section 3.4.10.

Example:

```
(15 /abcd def) token => (/abcd def) 15 true
((St1) {1 2 add}) token => ( {1 2 add}) (St1) true
```

Errors: invalidaccess, rangecheck, stackoverflow, stackunderflow, syntaxerror, typecheck, undefinedresult.

3.4.7. Relational, Boolean, and Bitwise Operators

The POSTSCRIPT boolean and relational operators create boolean objects, provide logical operations on boolean operands, and compare or test operands. The bitwise operators provide boolean and other operations on the (machine dependent) binary representations of integers (i.e., patterns of bits).

♦ **eq**

any_1 any_2 **eq** (boolean: any_1 = any_2)

tests the top two elements on the operand stack for equality and pushes the boolean value *true* if so, *false* if not. Some type conversions are performed by **eq**: integers and reals can be compared freely, as can names and strings. If the two arguments have other differing types, **eq** pushes the value *false*. If the arguments are strings or names, **eq** compares their lengths and contained characters for equality. **eq** compares other composite objects for equality of object (pointer and length) only, not for equality of objects they point to. If any_1 and any_2 are both arrays, for example, **eq** tests whether they both point to the same array body, not whether their elements are equal. The one exception to this rule is that all empty (zero length) arrays are equal.

Example:

```
4.0 4 eq => true % a real and an integer
[1 2 3] dup eq => true % an array is equal to itself
[1 2 3] [1 2 3] eq => false % distinct array objects not equal
```

Errors: invalidaccess, stackunderflow.

♦ **ne**

any_1 any_2 **ne** (boolean: any_1 ~= any_2)

tests the top two elements on the operand stack for equality and pushes the boolean value *true* if not equal, *false* if equal. The remarks for the **eq** operator regarding operand types, strings and other composites also apply to **ne**.

Errors: invalidaccess, stackunderflow.

♦ **ge**

```
      num1 num2  ge  (boolean: num1 >= num2)
string1 string2  ge  (boolean: string1 >= string2)
```

pushes the boolean value *true* if the first argument (*num*$_1$ or *string*$_1$) is greater than or equal to the second (*num*$_2$ or *string*$_2$), *false* otherwise. The arguments must have the same type, which must be either number or string. **ge** executes the error operator **typecheck** otherwise. If both arguments are strings, **ge** returns the result of comparing the two strings character by character (by comparing the character code values) to check whether the first string is lexically greater than or equal to the second string.

Example:

```
4.2 4 ge => true
(abc) (d) ge => false
(aba) (ab) ge => true
(aba) (aba) ge => true
```

Errors: invalidaccess, stackunderflow, typecheck.

♦ **gt**

```
      num1 num2  gt  (boolean: num1 > num2)
string1 string2  gt  (boolean: string1 > string2)
```

similar to **ge**, except **gt** checks whether the first argument is greater than the second.

Errors: invalidaccess, stackunderflow, typecheck.

♦ **le**

```
      num1 num2  le  (boolean: num1 <= num2)
string1 string2  le  (boolean: string1 <= string2)
```

similar to **ge**, except **le** checks whether the first argument is less than or equal to the second.

Errors: invalidaccess, stackunderflow, typecheck.

♦ **lt**

```
      num1 num2  lt  (boolean: num1 < num2)
string1 string2  lt  (boolean: string1 < string2)
```

similar to **ge**, except **lt** checks whether the first argument is less than the second.

Errors: invalidaccess, stackunderflow, typecheck.

♦ true

– **true** true

pushes a boolean object whose value is *true* onto the operand stack.

Errors: stackoverflow.

♦ false

– **false** false

pushes a boolean object whose value is *false* onto the operand stack.

Errors: stackoverflow.

♦ not

bool **not** NOT(bool)
int **not** bitwiseNOT(int)

If the argument is a boolean, **not** pushes its *logical negation*. If the argument is an integer, **not** pushes its bitwise complement (the *one's complement* of its binary representation).

Example:

```
true not => false
false not => true

99 not => -100 % that's 16#FFFFFF9C
52 not => -53 % 16#FFFFFFCB
```

Errors: stackunderflow, typecheck.

♦ and

```
bool1 bool2  and  (bool1 AND bool2)
  int1 int2  and  (int1 bitwiseAND int2)
```

If the arguments are booleans, **and** pushes their *logical conjunction* on the operand stack. If the arguments are integers, **and** pushes the *bitwise and* of their binary representations.

Example:

```
% a complete truth table
true true and => true
true false and => false
false true and => false
false false and => false

99 1 and => 1
52 7 and => 4
```

Errors: stackunderflow, typecheck.

♦ or

```
bool1 bool2  or  (bool1 OR bool2)
  int1 int2  or  (int1 bitwiseOR int2)
```

If the arguments are booleans, **or** pushes their *logical disjunction* (inclusive or) on the operand stack. If the arguments are integers, **or** pushes the *bitwise inclusive or* of their binary representations.

Example:

```
% a complete truth table
true true or => true
true false or => true
false true or => true
false false or => false

17 5 or => 21
```

Errors: stackunderflow, typecheck.

♦ xor

```
bool1 bool2 xor   (bool1 XOR bool2)
 int1 int2  xor   (int1 bitwiseXOR int2)
```

If the arguments are booleans, **xor** pushes their *logical exclusive or* on the operand stack. If the arguments are integers, **xor** pushes the *bitwise exclusive or* of their binary representations.

Example:

```
% a complete truth table
true true xor => false
true false xor => true
false true xor => true
false false xor => false

7 3 xor => 4
12 3 xor => 15
```

Errors: stackunderflow, typecheck.

♦ bitshift

```
int shift   bitshift   (bitshift(int,shift))
```

pushes the logical shift (left: if *shift* > 0, right: if *shift* < 0) of *int* by *shift* bits. *shift* and *int* must be integers.

Example:

```
7 3 bitshift => 56
142 -3 bitshift => 17
```

Errors: stackunderflow, typecheck.

3.4.8. Control Operators

POSTSCRIPT contains several operators that modify its default left-to-right control flow. These operators provide analogues to the for-loop, do-loop, repeat-loop, if-then conditional and if-then-else conditional found in more structured programming languages. Notably absent from POSTSCRIPT's set of control operators is any general label-goto mechanism.

♦ **exec**

`any` **`exec`** –

pushes the argument onto the execution stack, where it will be executed. If the argument is non-executable, then the POSTSCRIPT interpreter will just push the object back onto the operand stack. If the argument is executable, then the POSTSCRIPT interpreter will execute that object. The **load**, **get**, and **forall** operators all push a result onto the operand stack without executing it, even if it is executable. From there, you may use the **exec** operator to execute it. Also, enclosing a name in curly braces will cause an executable array consisting of only that name to be pushed on the operand stack without executing it. However, writing a name in POSTSCRIPT source code without quoting it as a literal (preceding it with a slash) or surrounding it with curly braces will cause the interpreter to execute it immediately; this is not a suitable way to provide that name's value as an argument to **exec**. A non-executable argument may be converted to executable prior to an **exec** operator by using the **cvx** operator. See section 3.3 for more details on executable and non-executable objects.

Example:

```
(3 2 add) cvx exec => 5
[3 2 /add cvx] cvx exec => 5
```

In this example, the string "3 2 add" is made executable and then executed (scanned and interpreted). While executing the string, a 3 and a 2 are scanned and pushed on the operand stack, the name **add** is scanned, looked up and executed, resulting in the sum, 5, being left on the stack. The second line creates an executable array and executes it. Note that a **cvx** is performed on the name **add** so that name lookup will take place.

Errors: stackunderflow.

♦ if

`boolean proc` **`if`** `–`

executes *proc* if *boolean* is *true*. Otherwise, *proc* is ignored. The **if** operator pushes no results of its own on the operand stack, but the *proc* may do so.

Example:

```
3 4 lt {(3 is less than 4)}if => (3 is less than 4)
```

Errors: stackunderflow, typecheck.

♦ ifelse

`boolean` $proc_1$ $proc_2$ **`ifelse`** `–`

executes $proc_1$ if *boolean* is *true*; or $proc_2$ if *boolean* is *false*. The **ifelse** operator leaves no results of its own on the operand stack, but the procedure it executes may do so.

Example:

```
4 3 lt {(TruePart)}{(FalsePart)} ifelse
        => (FalsePart)  % since 4 is not less than 3
```

Errors: stackunderflow, typecheck.

♦ repeat

`n proc` **`repeat`** `–`

executes *proc n* times. The **repeat** operator removes both arguments from the operand stack before executing *proc* for the first time. **repeat** leaves no results of its own on the operand stack, but *proc* may do so. *n* must be a non-negative integer.

Example:

```
4 {(abc)} repeat => (abc) (abc) (abc) (abc)
1 2 3 4 3 {pop}repeat => 1  % pops 3 values - down to the 1
4 {} repeat => % does nothing four times
mark 0 {(won't happen)} repeat => mark
```

In the last example, a zero repeat count meant that the body is not executed at all, hence the mark is still top-most on the stack.

Errors: rangecheck, stackunderflow, typecheck.

♦ for

```
initial increment limit proc  for  –
```

executes *proc* repeatedly as with ALGOL for-loops, i.e., for *initial* step *increment* until *limit* do *proc*. The **for** operator expects *initial*, *increment* and *limit* to be numbers, and it maintains an internal *loop counter* with *initial* as its initial value, *increment* as the increment to the counter each time around the loop, and *limit* as the termination value against which the **for** operator checks the loop counter. The **for** operator pushes the current value of the loop counter onto the operand stack before it executes *proc* each time. If *increment* is positive, the loop terminates when the loop counter exceeds *limit*; if *increment* is negative, the loop terminates when the loop counter becomes less than *limit*.

Example:

```
0 1 1 4 {add} for => 10
1 2 6 {} for => 1 3 5
3 -.5 1 {} for => 3.0  2.5  2.0  1.5  1.0
```

In the first example, the value loop counter is added to whatever is on the stack; so 1, 2, 3, and 4 are added to 0 in turn. The second example has an empty loop body, so the values of the loop counter (1, 3, and 5) are left on the stack. The last example counts backwards from 3 to 1 by halves, leaving the values (3.0, 2.5, 2.0, 1.5, and 1.0) on the stack.

Errors: stackoverflow, stackunderflow, typecheck.

♦ loop

```
proc  loop  –
```

repeatedly executes *proc* until *proc* executes a **stop** (not embedded within an inner **stopped**) or an **exit** (not embedded within an inner looping construct). If *proc* does not execute an **exit** or **stop**, an infinite loop results (which may be broken only via an external interrupt; see the **interrupt** error operator).

Errors: stackunderflow, typecheck.

♦ exit

— **exit** —

transfers control to just beyond the innermost dynamically enclosing instance of a looping construct, without regard to lexical relationship. The looping constructs are: **for**, **loop**, **repeat**, **forall**, and **pathforall**. If **exit** would cause the context of a **run** or **stopped** operator to be left, the **exit** terminates and the **invalidexit** operator is executed (still in the context of the **run** or **stopped**). If there is no enclosing looping construct, POSTSCRIPT prints an error message and executes the built-in operator **quit**.

Errors: invalidexit.

♦ stop

— **stop** —

unwinds the execution stack to the innermost dynamically enclosing instance of a **stopped** context (without regard to lexical relationship), which returns *true*. If there is no active **stopped** context, POSTSCRIPT prints an error message and executes **quit**. Note that **start** may execute a **stopped** context.

Errors: (none).

♦ stopped

any **stopped** boolean

executes *any*. If *any* terminates normally, **stopped** pushes *false*. If *any* terminates because **stop** was executed, **stopped** pushes *true*. In any event, control continues at the command after **stopped**; propagation of the **stop** does not proceed any further. Most typically, *any* will be a procedure body, an executable string, or an executable file stream.

This mechanism provides an effective way for a POSTSCRIPT program to *catch* certain error conditions and retain control. The error operators might all execute the **stop** operator (after saving important information), and allow programs to recover. Note that there is no actual connection between the **stop/stopped** mechanism and error handling. If information needs to be passed from the point of the error to the code that catches the **stop**, this must be performed by explicit communication.

Errors: stackunderflow, typecheck.

♦ countexecstack

– **countexecstack** num

counts the number of objects on the execution stack and pushes this count onto the operand stack.

Errors: stackoverflow.

♦ execstack

array **execstack** subarray

store as many elements as the execution stack contains into the argument *array* and returns a object describing the initial N-element subarray of *array*, where N is the current depth of the execution stack. The elements of the execution stack are placed in this array in order, with the bottom element at index 0 and the top element at index N-1.

Errors: rangecheck, stackunderflow, typecheck.

♦ quit

– **quit** –

The definition of **quit** may be environment or installation dependent. When POSTSCRIPT is run on a computer with an operating system and a file system, **quit** terminates execution of the POSTSCRIPT interpreter, returning to the operating system under which POSTSCRIPT is run. The interpreter may save the current state of the VM, to be restored the next time that POSTSCRIPT is run.

Errors: (none).

♦ start

– **start** –

is executed by the POSTSCRIPT interpreter when it starts up. The definition of **start** may be environment or installation dependent. By default, **start** is defined as "{}", i.e., it does nothing. However, **start** may be redefined to do more. Depending on system configuration, the definition of **start** may persist from one invocation of POSTSCRIPT to the next, so the **start** operator may be used to set up a useful working environment that will be installed each time POSTSCRIPT is run. **start** may install a device, define error operators, etc.

Errors: (depends on **start**'s definition).

3.4.9. Type, Conversion, and Property Operators

POSTSCRIPT deals with objects of many different types. Accordingly, POSTSCRIPT contains several operators that deal directly with these types. Some of these operators convert objects of one type to objects of another type. Another operator allows a POSTSCRIPT program to determine the type of any given object.

The type operators give a POSTSCRIPT program a view of the innermost workings of the POSTSCRIPT interpreter. The types presented here are more detailed than the types presented elsewhere in this document. For instance, whereas the rest of this document refers to the Number type, the **type** operator returns the finer distinctions of Integer and Real types that the POSTSCRIPT interpreter actually maintains. Most POSTSCRIPT programs will not need this power; those that do can have it.

The property operators allow restriction of access to certain POSTSCRIPT objects, allowing protection of sensitive data or program components. Composite objects (arrays, dictionaries, and strings) may have the access restrictions *readonly* or *executeonly* imposed on them. Note that *executeonly* and *executable* are distinct attributes. Access restrictions are properties of a string or array *object* (not the storage it references), but of a dictionary *body*.

♦ **type**

```
any  type  name
```

removes the argument from the operand stack and pushes an object of type **nametype** whose value corresponds to the argument's type. The possible results are:

integertype
realtype
booleantype
stringtype
operatortype
nametype
arraytype
filetype
fonttype
dicttype
marktype
nulltype
savetype

Errors: stackunderflow.

♦ cvi

```
   num  cvi  integer
string  cvi  integer
```

(convert to integer) converts the string, integer or real number on the stack to its integer representation. The **cvi** operator truncates any fractional part to obtain the integer result. (See the **round, truncate, floor**, and **ceiling** operators in section 3.4.2 which remove fractional parts without type conversion.)

Example:

```
(3.3E1) cvi => 33
-47.8 cvi => -47
520.9 cvi => 520
```

Errors: rangecheck, stackunderflow, syntaxerror, typecheck, undefinedresult.

♦ cvlit

```
any  cvlit  Literal(any)
```

(convert to literal) makes the object on top of the operand not executable.

Errors: stackunderflow.

♦ cvn

```
string  cvn  name
```

(convert to name) converts the string argument on the stack to a name object that is lexically the same as the string. The name object is executable if the string was.

Example:

```
(abc) cvn => /abc
(abc) cvx cvn => abc
```

Errors: rangecheck, stackunderflow, typecheck.

♦ cvr

```
   num  cvr  real
string  cvr  real
```

(convert to real) converts the string, integer or real number on the stack to its floating point representation.

Errors: rangecheck, stackunderflow, syntaxerror, typecheck, undefinedresult.

♦ **cvrs**

`num base string` **`cvrs`** `substring`

(convert to string - with radix) expects a number, base, and string on the operand stack, overwrites the input string with the string representation of *num* in the given base, and returns a descriptor of the prefix *substring*. *base* is expected to be a positive integer between 2 and 36, inclusive. Digits in the resulting string greater than 9 are represented with the letters "A" through "Z". If the input string is too small to hold the result of conversion, **cvrs** executes the error operator **rangecheck**.

Example:

```
100 8 5 string cvrs => (144)   % 100 (10) is 144 (8)
200 16 (   ) cvrs => (C8)
```

Errors: rangecheck, stackunderflow, typecheck.

♦ **cvs**

`any string` **`cvs`** `substring`

(convert to string) converts an object to a string, overwrites the prefix portion of its string argument with the conversion result, and returns a descriptor to the prefix substring. If *any* is a number, **cvs** returns a string representation of that number. If *any* is a boolean, **cvs** returns either the string "true" or the string "false". If *any* is a string, **cvs** copies its contents into *string* and returns that substring of *string* containing the characters of *any*. If *any* is a name or operator, **cvs** stores into *string* the text (print representation) of that name. If *any* is any other type, **cvs** stores into *string* the text "--nostringval--". If the input string is too small to hold the result of conversion, **cvs** executes the error operator **rangecheck**.

Example:

```
123 456 add 20 string cvs => (579)
mark (                ) cvs => (--nostringval--)
```

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ **cvx**

`any` **`cvx`** `Executable(any)`

(convert to executable) makes the object on top of the operand stack executable without executing it.

Errors: stackunderflow.

♦ executeonly

```
 array  executeonly  ExecuteOnly(array)
string  executeonly  ExecuteOnly(string)
```

the result object allows no further reading or writing of its top-level elements. Thus, subsequent use of the result as an argument to **get**, **put**, **forall**, etc., will result in execution of the error operator **invalidaccess**. An object may be tested for *executeonly* status with the **rcheck** operator. After execution of **executeonly**, the result may only be executed, either explicitly, as an argument to **exec**, etc., or implicitly, if it is the value of some name that is encountered, looked up, in normal execution sequence. The *executeonly* attribute can only be removed through the **restore** operator, if the object was not *executeonly* in the snapshot reinstated by the **restore**.

Errors: invalidaccess, stackunderflow, typecheck.

♦ readonly

```
 array  readonly  ReadOnly(array)
  dict  readonly  ReadOnly(dict)
string  readonly  ReadOnly(string)
```

the result allows no further writing to the object; that is, its top-level elements may no longer be replaced by operations such as **put**. However, this restriction does not extend to the contents of any of those elements that are in turn composite. For an array or string, the *readonly* attribute applies only to the returned object; for a dictionary, however, the dictionary storage itself becomes *readonly*, regardless of how it is accessed. The *readonly* attribute can only be removed through the **restore** operator, if the object was not *readonly* in the snapshot reinstated by the **restore**.

Errors: invalidaccess, stackunderflow, typecheck.

♦ xcheck

```
any  xcheck  boolean
```

(check whether executable) removes the argument from the operand stack and pushes the boolean value *true* if it is executable or *false* if it is literal. Note that **xcheck** checks for executability, not for *executeonly* status.

Errors: stackunderflow.

♦ rcheck

```
  array  rcheck  boolean
   dict  rcheck  boolean
 string  rcheck  boolean
```

removes the argument from the operand stack and pushes the boolean value *true* if it is readable, or *false* otherwise. The **executeonly** operator returns an result which is not readable. In addition, some system-maintained dictionaries may not be readable.

Errors: stackunderflow, typecheck.

♦ wcheck

```
  array  wcheck  boolean
   dict  wcheck  boolean
 string  wcheck  boolean
```

(check whether writeable) removes the argument from the operand stack and pushes the boolean value *true* if it is writeable, or *false* otherwise. The **readonly** and **executeonly** operators return objects that are not writeable.

Errors: stackunderflow, typecheck.

3.4.10. File Operators

This section describes the POSTSCRIPT operators that read, write, and execute information to, from, and in files. Note that graphics operations and printing are *not* accomplished by writing to files. The operations for generating graphics are discussed in a later section.

POSTSCRIPT files behave like streams; each has an associated current *position* that marks where the next read or write operation will take place. Standard input and output devices, such as the interactive user's terminal, are treated as files using the same mechanisms.

Exception conditions are treated in a uniform manner by operators that access files. During reading, if *end-of-file* is encountered before the desired item has been read, the file is closed and the operation returns an explicit end-of-file indication. This is likewise done if the file has already been closed. All other exceptions during reading and all exceptions (including file already closed) during writing cause **ioerror** to be executed. There is a limit on the number of streams that can be open simultaneously.

Input and output operations in computer languages are typically quite dependent on the operating system under which its programs are run. POSTSCRIPT is no exception. The input and output functions described in this section are accurate for all current POSTSCRIPT implementations. The availability of external files and their naming conventions may be environment dependent.

In addition to the normal files accessible through the operating system (if any), POSTSCRIPT defines five special files whose names begin with the character "*%*" and which may be opened with the **file** operator. These are:

%statementedit
: The command "**(%statementedit)(r) file**" waits for the user to type in one or more lines comprising a complete POSTSCRIPT statement (that is, a sequence of one or more tokens with no "{" or "(" left unmatched, terminated by a newline). Certain editing functions are available during typein, including backspace character (BS), erase line (control-U), and retype line (control-R). The **file** operator then returns a new file object that dispenses the entire statement that was typed in, followed by end-of-file. This file object may be read from in the normal ways, either explicitly by other file operators (e.g., **read**) or implicitly by converting it to executable (**cvx**) and then executing it (**exec**). The file operator executes **undefinedfilename** if the terminal input stream (*%stdin*) reaches end-of-file before any characters have been read.

%lineedit
: works similarly to *%statementedit*, but only one line is returned, regardless of whether or not it comprises a complete POSTSCRIPT statement.

%stdin returns a file object designating the standard input. In interactive POSTSCRIPT configurations this is usually the user's terminal; in server configurations this is a communications interface or file being used as the standard source of POSTSCRIPT program text. This file is unbuffered and (generally) unedited, and should not be confused with *%statementedit* or *%lineedit*. Closing the *%stdin* file has no effect other than to clear any end-of-file indication that may have been set.

%stdout, *%stderr*
return a file object designating standard normal output and standard error output. *%stdout* should be used for all normal output, and is the file automatically used by the **print** operator. *%stderr* is intended primarily for reporting low-level errors; in many POSTSCRIPT configurations, this is the same as *%stdout*.

♦ file

`string1 string2 file file`

creates a file object for the file named *string*$_1$ with access restrictions according to *string*$_2$ and pushes this file object onto the operand stack. Unless *string*$_1$ is one of the special file names mentioned earlier, the **file** operator will interpret the file name and access code in an environment dependent manner. The following access code strings should be available on most systems.

code	*meaning*
(r)	Read only. Sets position to the beginning of the file. The named file must exist; executes **undefinedfilename** otherwise.
(w)	Write only. Sets position to the beginning of the file. Creates a new file if non-existent. Truncates file to the current position when closed.

Errors: invalidfileaccess, limitcheck, stackunderflow, typecheck, undefinedfilename.

♦ closefile

`file closefile –`

closes the file stream *file*, taking actions according to *file*'s access mode. The stream referenced by *file* is no longer a valid file stream (i.e., *file* **status** will return *false*). See the description of **file** below for a discussion of access modes.

Errors: ioerror, stackunderflow, typecheck.

♦ read

```
file  read  if not end-of-file: byte true
            if end-of-file: false
```

reads one byte from *file*, pushes it on the stack (as a number), pushes *true*, and moves the *file* position ahead by one. If the end-of-file condition occurs before a byte has been read, the **read** operator closes the file and returns *false*.

Errors: ioerror, stackunderflow, typecheck.

♦ readhexstring

```
file string  readhexstring  substring boolean
```

works like **readstring**, except that characters in *file* are treated as (the ASCII print representation of) hexadecimal digits, and pairs of them are converted to their 0 through 255 values and stored in successive character positions of *string*. The hexadecimal input may be interspersed with blanks, carriage returns, and other non-hexadecimal digits. These are ignored; only the hexadecimal characters are decoded.

Errors: ioerror, rangecheck, stackunderflow, typecheck.

♦ readline

```
file string  readline  substring boolean
```

reads characters (bytes) from *file* through the next newline character, stores them in a prefix substring of the argument *string*, moves *file*'s position ahead that number of characters and returns both the read *substring* and *true*. The returned substring does *not* include the newline character as its last character. If the input line is longer than the argument string, **readline** executes a **rangecheck**. If **readline** encounters the end of file before a newline, the substring (possibly empty) and *false* are returned. Thus, after executing a **readline** and checking for *false*, the string length should be tested.

Errors: ioerror, rangecheck, stackunderflow, typecheck.

♦ readstring

```
file string  readstring  substring boolean
```

reads up to length(*string*) characters (bytes) from *file* into *string*. **readstring** returns *false* if it encounters end-of-file before *string* is full; otherwise, it returns *true*. Essentially, **readstring** fills the buffer *string* with bytes from *file* until either the buffer is full or an end of file is encountered, returning the filled portion of the buffer and *true* or *false* accordingly. Note that newline characters are not treated specially by **readstring**, they are included among the characters in the buffer when read.

Errors: ioerror, rangecheck, stackunderflow, typecheck.

♦ token

```
file  token  if found: token true
             if not found: false
```

extracts a **token** from the *file* (interpreted as POSTSCRIPT source code). If a token is found, it pushes the compiled token and *true*. If a token is not found, it pushes *false* and closes *file*. The **token** operator behaves like the POSTSCRIPT interpreter's scanner. It extracts from the file stream a character sequence that corresponds to a single POSTSCRIPT syntactic entity. This entity may be simple: a string, a number or a name, or it may be composite: an executable array extending from an initial left curly brace through its matching right curly brace. The *token* pushed onto the operand stack is a *compiled* POSTSCRIPT object that corresponds to the syntactic entity. Thus, the *token* pushed for a number is a number object, not its string representation. The *token* pushed for a curly brace delimited section of code is an executable array, all of whose components are similarly compiled. The **token** command does not execute this token; it merely pushes it onto the operand stack. If the token is terminated by a delimiting space, tab, or newline, the file is left positioned immediately *after* the delimiter. However, if the token is self-delimiting (e.g., "["), the file is left positioned immediately after the self-delimiter.

See also the description of the **token** operator for string arguments in section 3.4.6.

Errors: ioerror, rangecheck, stackoverflow, stackunderflow, typecheck, undefinedresult.

♦ bytesavailable

```
file  bytesavailable  int
```

returns the number of bytes immediately available for reading from *file*; or -1 if that number cannot be determined.

Errors: ioerror, stackunderflow, typecheck.

♦ write

```
file byte  write  –
```

writes a single character (*byte*) into *file* at the current position, and moves the position ahead by one. *byte* must be an integer, ordinarily in the range 0 to 255 inclusive (an integer outside this range is reduced modulo 256). *File* must be a writable file stream.

Errors: ioerror, stackunderflow, typecheck.

♦ writehexstring

```
file string  writehexstring  –
```

writes all the characters of *string* into *file* starting at the current position and moves the position ahead that number of characters. The characters are written as pairs of ASCII characters representing the hexadecimal values of the characters in *string*. Thus, if the argument string is "(abz)", the output to *file* is the six characters: "61627a".

Errors: ioerror, stackunderflow, typecheck.

♦ writestring

```
file string  writestring  –
```

writes all the characters of *string* into *file* starting at the current position and moves the position ahead that number of characters.

Errors: ioerror, stackunderflow, typecheck.

♦ flush

```
–  flush  –
```

causes any buffered output for the standard output stream to be sent immediately. In general, a program requiring that output be delivered *now* should call **flush** after generating that output.

Errors: ioerror.

♦ flushfile

`file` **`flushfile`** –

If *file* is an output stream, **flushfile** causes any buffered output for *file* to be sent immediately. In general, a program requiring that output be delivered *now* should call **flushfile** after generating that output. If *file* is an input stream, **flushfile** will read and discard data from *file* until the end-of-file condition is achieved.

Errors: ioerror, stackunderflow, typecheck.

♦ status

`file` **`status`** `boolean`

return *true* if *file* is still a valid (open) stream, *false* otherwise.

Errors: stackunderflow, typecheck.

♦ run

`string` **`run`** –

executes the contents of file named by *string*. When execution reaches the end-of-file, or **run** terminates for some other reason (e.g., **stop**), the file is closed. **run** leaves no values on the stack, but the result of executing the contents of *string* may do so. Note that **run** is a convenience operator for the operation

```
(r) file cvx exec
```

Errors: ioerror, limitcheck, stackunderflow, typecheck.

♦ currentfile

– **`currentfile`** `file`

creates a file object that references the input stream from which the PostScript interpreter is currently reading program input. This operator is necessary when referencing images or other input that reside in the program file itself.

Errors: stackoverflow.

♦ print

`string` **`print`** –

outputs *string* on the standard output. The **print** operator provides the simplest way to output text to an interactive user.

Errors: stackunderflow, typecheck.

♦ prompt

– **prompt** –

is executed by the POSTSCRIPT interpreter whenever it is ready for a new line of input (in interactive mode). The initial definition of **prompt** is "**{(PS>) print}**".

Errors: (none).

♦ echo

boolean **echo** –

sets whether input characters from the standard input are echoed to the standard output according to the value of *boolean*. By default, the POSTSCRIPT interpreter echoes input to the output while opening the files named *%statementedit* and *%lineedit*. One simple operation for which turning off echoing is appropriate is password input.

Errors: stackunderflow, typecheck.

3.4.11. Virtual Memory Operators

The POSTSCRIPT interpreter keeps most of its basic machine storage, objects, name lookup tables and string character contents, in a memory structure called its *Virtual Memory* or *VM* for short. Depending of system configuration, this VM may be *persistent*, that is, it persists beyond a single execution of the POSTSCRIPT interpreter. When the interpreter returns to the system from which it was run, it saves the current state of its VM, and when the interpreter is re-run, it begins by restoring its VM to that state (slightly modified by start-up).

POSTSCRIPT has a save and restore mechanism that is unique among interactive programming languages. A **save** operation causes the POSTSCRIPT interpreter to remember a snapshot of its complete state: the values of dictionary items, the keys in dictionaries, and the values in arrays. (The characters contained in strings are not remembered, but string objects (length and character position) are). The POSTSCRIPT interpreter does not include the state of stacks, file streams, or graphics output in this snapshot. A **restore** operation causes the POSTSCRIPT interpreter to revert back to the state contained in such a snapshot.

Except for changes to the stacks and side-effects such as file operations and graphics output, the execution of POSTSCRIPT source code between a **save** and its corresponding **restore** is as if the execution had not happened. These semantics can be useful for encapsulating a section of POSTSCRIPT source code that makes wholesale changes to variables for some special purpose. Rather than having to reset each variable individually, a **save** and **restore** pair does the job neatly and efficiently, undoing only those modifications that were made within the scope of the **save** and its corresponding **restore**. Since the POSTSCRIPT interpreter runs other programs within its own environment and these programs are free to modify substantial portions of that environment, **save** and **restore** serve to insulate the interpreter from any unwanted legacy.

The POSTSCRIPT interpreter's implementation of the **save** and **restore** operators keeps typical execution overhead small. These operators are an important part of the POSTSCRIPT language, and we encourage their use. Not only are they convenient to use, but they also conserve resources. There is a large but fixed limit on the size of the POSTSCRIPT interpreter's virtual memory. As objects are created by POSTSCRIPT programs, they accumulate in VM and must be culled from there periodically so as not to run out of space. POSTSCRIPT's save and restore mechanism not only snapshots system state but prunes back VM usage as well. When the POSTSCRIPT interpreter executes a **restore** operator, it quickly reclaims all memory allocated since the corresponding **save**. A POSTSCRIPT interpreter that runs program after program, as in a printer server, would be well advised to wrap a **save** and **restore** around each program execution.

Since the VM may be persistent, it may be used to hold a user's state in terms of new operators defined in dictionaries, default graphics

parameters, etc. When the POSTSCRIPT interpreter is started, it restores its stacks to their initial empty state, it **restores** the VM back to the topmost save level (if no **saves** were performed, this is a no-op), and it executes the **start** operator for any special startup actions.

♦ save

– **save** saveobj

sets up a snapshot of the interpreter's state and returns a Save object that refers to this snapshot. Subsequent **restore** operator execution must use this Save object to restore back to the state saved at this time.

Errors: limitcheck, stackoverflow.

♦ restore

saveobj **restore** –

resets the VM to its state at the time the *saveobj* argument was generated by a **save** operator. The **save** and **restore** operators must be issued in a nested fashion.

Errors: invalidrestore, rangecheck, stackunderflow, typecheck.

♦ vmstatus

– **vmstatus** level used total

returns three integers describing the state of the POSTSCRIPT VM. *level* is the current depth of **save** nesting. *used* and *total* are the number of bytes used, and the total number available in VM. (Note, however, that in certain configurations, *total* may be able to increase dynamically by obtaining more storage from the operating system).

Errors: stackoverflow.

3.4.12. Miscellaneous Operators and Functions

This section describes the few POSTSCRIPT operators that do not easily fit into any other category. In addition, this section also lists several standard key-value pairs which exist in **systemdict** and **userdict** which are pre-defined POSTSCRIPT functions (not built-in operators).

♦ **version**

– **version** string

returns the POSTSCRIPT version identifier for a particular version of the POSTSCRIPT language, implementation, and hardware.

Errors: stackoverflow.

♦ **usertime**

– **usertime** msec

returns time in milliseconds (an integer). This time can be used for interval timing, but may not be accurate for long intervals or time-of-day uses.

Errors: stackoverflow.

♦ **=**

any **=** –

destructively prints the top element of the stack with **cvs**. Thus, if *any* is a string, a name, an operator, a number, or a boolean, = will print its readable (**cvs**) representation. If *any* is an array, dictionary, mark, savelevel, null, file, or fontID, = will print "--nostringval--". = is equivalent to the following code:

```
/= {dup type /stringtype ne
    {(                              ) cvs}if
    print (\n) print
   } def
```

Errors: stackunderflow.

♦ **stack**

⊢ any_1 ... any_N **stack** ⊢ any_1 ... any_N

prints any_N through any_1 using the = routine. **stack** does not destroy the contents of the stack, but copies the entire stack and destructively prints the copy. **stack** is equivalent to the following:

```
/stack {count dup 1 add copy {=} repeat pop} def
```

Errors: stackoverflow.

♦ ==

`any == –`

destructively prints the top element of the stack a little more cleverly than does =. == will print the contents of arrays, will flag literal names, and other nice things.

Errors: stackunderflow.

♦ pstack

`⊢ any1 ... anyN` **`pstack`** `⊢ any1 ... anyN`

prints the entire stack (like **stack**) using ==.

Errors: stackoverflow.

Graphics Operators

The preceding sections completely describe the general computer language aspects of POSTSCRIPT. By themselves, they describe an interpretive programming language of great expressive power. This section describes the standard extension of the POSTSCRIPT language that deals with computer graphics. The facilities and operators described here are intended for both display and printer applications.

The POSTSCRIPT interpreter maintains a data structure called the Graphics State that holds current graphics control parameters. These parameters define the context in which the graphics commands operate. For example, the **show** operator implicitly uses the *current font* parameter in the Graphics State, and the **fill** operator implicitly uses the *current color* parameter in the Graphics State.

Graphics States are maintained in a stack. By pushing a new Graphics State onto this stack (with the **gsave** operator) a new context with many different characteristics may be defined without destroying the Graphics State currently in force. This new context may have a different font, transformation matrix, line style, etc. defined. After some graphics output is performed, the original Graphics State may be restored by popping this new Graphics State off its stack (with the **grestore** operator), making resets of each changed Graphics State parameter unnecessary.

The complete set of Graphics State parameters is:

Name	*Type*	*Value Semantics*
CTM	Array	The current transformation matrix; a matrix that maps positions from user coordinates to device coordinates. This matrix is modified by each application of the coordinate system operators. (Initial value: A straightforward matrix transforming default coordinates to device coordinates.)
color	Internal	The internal representation of colors is not exposed to the POSTSCRIPT user. To encode and decode colors among different color models, see color related operators in section 3.5.5. (Initial value: black.)
cp	Numbers	Current position. (Initial value: undefined.)
path	Path	The current path as built up by the path construction operators. Path objects are not directly accessible in POSTSCRIPT. This object is an implicit argument to the **fill**, **stroke**, and **clip** operators. (Initial value: empty.)
clip	Path	The current boundary against which all output is

		clipped. (Initial value: the entire imageable portion of the output device.)
font	Dictionary	Set of graphic shapes (characters) that define the current typeface. (Initial value: installation dependent.)
line width	Number	The thickness (in user coordinates) of lines to be drawn by the **stroke** operator. (Initial value: 1.)
line cap	Integer	A code that defines the shape of the endpoints of any open path that is stroked. (Initial value: 0, for a square butt end.)
line join	Integer	A code that defines the shape of a stroked line at its corners. (Initial value: 0, for mitered joins.)
screen	several	A collection of POSTSCRIPT objects that define current halftone screen pattern for gray and color output. (Initial value: installation dependent.)
transfer	Array	An executable procedure that maps user gray levels into device gray levels, for tuning output devices's gray response curve. (Initial value: installation dependent.)
flatness	Number	A number that determines the smoothness of Bezier curve renditions on the output device. This number gives the maximum error tolerance (in output device pixels) of a straight line segment approximation of any portion of a Bezier curve. Smaller numbers give smoother curves at the expense of more computation. (Initial value: 0.5.)
miter-limit	Number	A number that determines the maximum length of mitered line joins for the **stroke** operator. This number is the ratio of maximum diagonal through the join over the line width. Line segments that meet at sharp angles that would cause their miter ratio to exceed this number are beveled instead. (Initial value: 10, for a miter cutoff below 11 degrees.)
dash	Array, Real	A description of lengths of portions of dashed lines to be rendered by the **stroke** operator instead of the normal solid line. (Initial value: a 0-length array plus a 0 offset, corresponding to a normal solid line.)
device	Internal	An internal data structure that describes the current output device. Each output device has certain procedures that allow it to print any shapes and halftones specified by the rest of the POSTSCRIPT graphics descriptions. Devices are set through the device setup operators described in section 3.5.6. (Initial value: the null output device.)

Each graphics operator description in the following subsections mentions which Graphics State parameters it uses.

POSTSCRIPT's graphics operators form five major groups:

1. *Graphics state operators*. This group contains operators that manipulate Graphics States as a whole. They provide convenient means of switching between different contexts defined by the following groups of operators.

2. *Coordinate system and matrix operators*. The Graphics State contains a transformation matrix (named *CTM*) that maps user specified coordinates into coordinates appropriate for the output device. The operators in this group manipulate this matrix to achieve any combination of translation, scaling (including mirror imaging), and rotation of user coordinates onto device coordinates.

3. *Character and font operators*. These operators allow the specification, selection, and modification of fonts, and the means to image characters in those fonts on the page.

4. *Path construction operators*. The POSTSCRIPT graphics machinery maintains a *current path* that defines shapes and line trajectories for output. The operators in this group begin a new path, add straight line segments, circular arcs and cubic curves to the current path and close the current path. All of these operators implicitly reference the Graphics State *CTM* parameter.

5. *Output operators*. These operators specify the contents of areas to be output. POSTSCRIPT programs may use a variety of color models to specify output color and halftone screens. Scanned images are equivalent to multi-colored sampled ink. Other operators in this section actually generate images on an output device. After a path is constructed, and colors, images, character fonts, line widths, etc. are set, these operators "push" images or color through the current shape (defined by the current path) or render line trajectories on the output device.

3.5.1. Graphics State Operators

The operators in this group manipulate entire Graphics States on the current Graphics State stack. Whenever a POSTSCRIPT **save** operator executes, it establishes a new stack of Graphics States. The initial Graphics State on this stack is a copy of the Graphics State in effect at the time of the **save**. New Graphics States may be pushed onto and popped off of this Graphics State stack, but only the corresponding **restore** operator can remove the Graphics State that a **save** operator placed at the bottom of this Graphics State stack.

♦ **gsave**

– `gsave` –

is a special case of the **save** operator. **gsave** saves only the current Graphics State, pushing a copy of it onto the Graphics State stack, whereas **save** saves the entire state of the POSTSCRIPT interpreter: values of variables and dictionaries, etc. as well as the values in the current Graphics State. **gsave** is useful for creating instances of predefined shapes with different transformations, making possible a simple restoration of the Graphics State through a matching **grestore**. Often, related transformations need not be created entirely from scratch; they may share some common setup which may be **gsave**ed. Note that unlike **save**, **gsave** returns no Savemark object; **gsave**s and **grestore**s work in a strictly stack-like manner.

Errors: limitcheck.

♦ **grestore**

– `grestore` –

pops the current Graphics State off of the Graphics State stack, installing the Graphics State in effect at the time of the matching **gsave** as the current Graphics State. This operator gives a simple way to undo complicated transformations and state setup without having to undo all Graphics State values individually.

An attempt to **grestore** past the most recent **save** barrier replaces the current Graphics State with a copy of the Graphics State in effect at the time of that **save**.

Errors: (none).

♦ grestoreall

– **grestoreall** –

repeatedly pops the current Graphics State off the Graphics State stack down to the most recent **save** barrier. It then pushes a copy of the Graphics State in effect at the time of that **save** back onto the Graphics State stack.

Errors: (none).

3.5.2. Coordinate System and Matrix Operators

POSTSCRIPT defines a standard, device independent coordinate system called *default user coordinates* or *default user space.* All shapes and images manipulated in a POSTSCRIPT program are relative to this coordinate system. POSTSCRIPT transforms coordinates to achieve translation, scaling, and rotation by means of a 3 x 3 transformation matrix maintained in the Graphics State called the *current transformation matrix* or *CTM.*[2] The default value for this transformation matrix relates the default user coordinate system to a raster device's built-in coordinate system in the following way. The origin of the default user coordinate system maps to the lower left corner of the device's image area when viewed in its "preferred orientation", with the user space's x-axis increasing to the right and the the user space's y-axis increasing upwards. One unit in default user space corresponds to 1/72 of an inch on the output device.

The preferred orientation of a printer that prints on 8.5 x 11 inch paper is its *portrait* orientation, that is, the long side is the y-axis and the short side is the x-axis. The preferred orientation of a display device is x-axis horizontal, y-axis vertical. This may be *portrait* or *landscape* orientation, depending on the display's dimensions. In all cases, the active area of the device is in the first quadrant of the default user coordinate system (non-negative *x*, non-negative *y*).

POSTSCRIPT programs need know nothing about the resolution of the raster output device on which a printed page is rendered, nor do they need to know about the manner in which the output device addresses points in its image area. All placements and measurements are made in user space, and the default transformation matrix in the Graphics State maps these locations to the appropriate locations on the output device. Thus, a POSTSCRIPT program may be used unchanged on any raster output device; only the default transformation matrix (set outside of the program) is different to achieve proper imaging on all devices.

POSTSCRIPT computations are carried out using floating point number representation when necessary. Therefore, the units of the default user coordinate system (1/72 inch) in no way constrain or affect the resolution of the output device. For example, if a POSTSCRIPT program says:

```
72.334 196.121 moveto
```

then the POSTSCRIPT interpreter renders this position as accurately as possible in the device's coordinate system.

By modifying the current transformation matrix, simple shapes expressed in simple orientations can be easily transformed to many varia-

[2]Actually, only the first two columns of POSTSCRIPT matrices are meaningful; the third column of a 3 x 3 matrix always contains 0, 0, 1, and the third element of a row vector is always 1. For this reason, the POSTSCRIPT operators that deal with matrix values require specification of only the first two columns. For a complete mathematical explanation of how such a matrix performs geometrical transformations, see the book *Principles of Interactive Computer Graphics* by W. M. Newman and R. F. Sproull.

tions. Many of the graphics operators described in this section achieve their results by constructing new matrices, postmultiplying them by the current transformation matrix, and establishing the result as the new transformation matrix.

While an accurate description of these operators may be expressed in terms of their effect on the transformation matrix, it is often more useful to think of them in terms of their effect on the current user space. For instance, a 2 2 **scale** operation doubles the size at which objects are rendered. This is achieved by postmultiplying the current transformation matrix to yield one that transforms coordinates into positions whose device coordinate values are double those that would have resulted from the transformation in effect before the 2 2 **scale** operation was applied. Alternatively, we can view the effect of this operation as changing the current user coordinate system, so that now a unit in user space represents twice as much as it did before. We will present the coordinate system transformation operators from both of these points of view.

A longer example (that uses several operators to be discussed in later sections) should make these transformation concepts clear.

```
% Define a procedure to construct a unit square path in
% the current user coordinate system.

/box {newpath
      0 0 moveto
      0 1 lineto
      1 1 lineto
      1 0 lineto
      closepath
     } def

% Modify the current transform matrix so that everything
% subsequently drawn will be 72 times larger,
% that is, each unit will represent an inch.

72 72 scale
% the transform matrix now represents unit
% coordinates as one inch long.

% Draw a 1" X 1" box (72 X 72 default coordinate units).

box fill

% Change the transform matrix again so that the origin
% will be at 2", 2".
% Since the coordinate system is now in inches we say:

2 2 translate

% Draw the box again.
% This box will have its lower left corner two inches up
% from and two inches to the right of the lower left corner
% of the page, and it will be one inch square.

box fill
```

This example shows how coordinates expressed in POSTSCRIPT programs, e.g., the coordinates given to the **moveto** and **lineto** graphics operators, are transformed by the current transformation matrix. By combining translations, scalings, and rotations on the transformation matrix, very simple prototype graphics procedures like **box** in the example can generate a myriad of instances.

Transformation matrices are represented in POSTSCRIPT as six-element array objects. As such, they may be stored, copied, and modified as are other POSTSCRIPT array objects. Such a six-element array object [*a b c d tx ty*] corresponds to a transformation matrix:

a	*b*	0
c	*d*	0
tx	*ty*	1

In the operator descriptions below, an argument given as "*matrix*" indicates a 6-element POSTSCRIPT array, while reference to "*CTM*" indicates the current transformation matrix in the Graphics State.

♦ matrix

– **matrix** matrix

creates a 6-element POSTSCRIPT array object, fills it in with the values of an identity matrix, i.e., [1.0 0.0 0.0 1.0 0.0 0.0], and pushes this array onto the operand stack. This operator is equivalent to the sequence:

6 array identmatrix

Errors: stackoverflow.

♦ initmatrix

– **initmatrix** –

sets *CTM* to the default matrix defined by the current output device. This matrix transforms default user coordinates to their default positions on the output device.

Errors: (none).

♦ identmatrix

matrix **identmatrix** matrix

replaces the contents of *matrix* with the values of the identity transformation matrix, i.e., [1.0 0.0 0.0 1.0 0.0 0.0] and pushes this modified matrix back onto the operand stack.

Errors: rangecheck, stackunderflow, typecheck.

♦ defaultmatrix

matrix **defaultmatrix** matrix

replaces the contents of *matrix* with the values of the default transformation matrix for the current output device and pushes this modified matrix back onto the operand stack.

Errors: rangecheck, stackunderflow, typecheck.

♦ currentmatrix

matrix **currentmatrix** matrix

replaces the contents of *matrix* with the values in *CTM* and pushes this modified matrix back onto the operand stack.

Errors: rangecheck, stackunderflow, typecheck.

♦ setmatrix

```
matrix  setmatrix  –
```

sets the contents of *CTM* to the contents of *matrix*.

Note: *matrix* should be a matrix that has resulted from a previous **currentmatrix** operation or sequence of matrix operations that involved a **defaultmatrix** operation. Only then can the POSTSCRIPT program be sure that the matrix will be reasonable with respect to the current output device.

Errors: rangecheck, stackunderflow, typecheck.

♦ translate

```
        tx ty  translate  –
tx ty matrix  translate  matrix
```

With no *matrix* argument, **translate** builds a temporary matrix:

$$T = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{matrix}$$

and replaces *CTM* by *T* * *CTM*. The effect of this operator on the user coordinate system is to move its origin, (0, 0), to the position (*tx*, *ty*) in the user coordinate system defined at the time this operator is executed. The orientation of the user coordinate axes and the unit scale are unaffected.

If the *matrix* argument is supplied, **translate** replaces the contents of *matrix* by [1.0 0.0 0.0 1.0 *tx ty*] and pushes this modified matrix back onto the operand stack with no effect on *CTM*.

Both *tx* and *ty* must be numbers.

Errors: rangecheck, stackunderflow, typecheck.

♦ scale

```
        sx sy  scale  –
sx sy matrix  scale  matrix
```

With no *matrix* argument, **scale** builds a temporary matrix:

$$S = \begin{matrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{matrix}$$

and replaces *CTM* by *S* * *CTM*. The effect of this operator is to make the x and y units in the user coordinate system the size of *sx* x-units and *sy* y-units in the user coordinate system defined at the time this operator is executed. The location of the user coordinate origin and the orientation of the coordinate axes are unaffected.

If the *matrix* argument is supplied, **scale** replaces the contents of *matrix* by [*sx* 0.0 0.0 *sy* 0.0 0.0] and pushes this modified matrix back onto the operand stack with no effect on *CTM*.

Both *sx* and *sy* must be numbers.

Errors: stackunderflow, typecheck.

♦ rotate

```
        ang  rotate  –
ang matrix  rotate  matrix
```

With no *matrix* argument, **rotate** builds a temporary matrix:

$$R = \begin{matrix} \cos(ang) & \sin(ang) & 0 \\ -\sin(ang) & \cos(ang) & 0 \\ 0 & 0 & 1 \end{matrix}$$

and replaces *CTM* by *R* * *CTM*. The effect of this operator is to rotate the user coordinate system axes about their origin by *ang* degrees (positive is counterclockwise) with respect to the user coordinate system defined at the time this operator is executed. The location of the user coordinate origin and the size of the x and y units are unchanged.

If the *matrix* argument is supplied, **rotate** replaces the contents of *matrix* by [cos(*ang*) sin(*ang*) -sin(*ang*) cos(*ang*) 0.0 0.0], where *ang* is interpreted as an angle in degrees, and pushes this modified matrix back onto the operand stack with no effect on *CTM*.

The argument *ang* must be a number.

Errors: stackunderflow, typecheck.

♦ concat

```
matrix  concat  –
```

replaces *CTM* by *matrix* * *CTM*.

Example:

```
sx sy matrix scale concat

sx sy scale
```

The two examples have the same effect on the current transformation.

Errors: stackunderflow, typecheck.

♦ concatmatrix

$matrix_1$ $matrix_2$ $matrix_3$ **concatmatrix** $matrix_3$

replaces the contents of $matrix_3$ by the result of multiplying $matrix_1$ * $matrix_2$ and pushes the modified $matrix_3$ back onto the operand stack. This operator does not effect *CTM*.

Errors: stackunderflow, typecheck.

♦ transform

```
        x y  transform  xt yt
x y matrix  transform  xt yt
```

With no *matrix* argument, **transform** multiplies the row-vector (*x*, *y*, 1) by *CTM*, i.e., (*x*, *y*, 1) * *CTM*, to yield the row-vector (*xt*, *yt*, 1). If (*x*, *y*) is a coordinate in the current user space, then (*xt*, *yt*) is the corresponding coordinate in the output device space under the current transformation.

If the *matrix* argument is supplied, **transform** multiplies the row-vector (*x*, *y*, 1) by the argument *matrix*, i.e., (*x*, *y*, 1) * *matrix*, to obtain the row-vector (*xt*, *yt*, 1).

The arguments *x* and *y* must be numbers.

Errors: stackunderflow, typecheck.

♦ dtransform

```
        xd yd  dtransform  xdt ydt
xd yd matrix   dtransform  xdt ydt
```

With no *matrix* argument, **dtransform** (delta transform) behaves like **transform**, but uses a copy of *CTM* with its *tx* and *ty* translation components zero. This operator shows how a positionless vector (*xd*, *yd*) in user space is transformed by the current transformation into a positionless vector in output device space. This operator is most useful for determining how distances map from user space to device space.

If the *matrix* argument is supplied, **dtransform** uses it instead of an implicit reference to *CTM*.

The arguments *xd* and *yd* must be numbers.

Errors: stackunderflow, typecheck.

♦ itransform

```
        xt yt  itransform  x y
xt yt matrix   itransform  x y
```

With no *matrix* argument, **itransform** (inverse transform) returns *x* and *y* such that (*x*, *y*, 1) * *CTM* = (*xt*, *yt*, 1). This operator thus returns the position in user space that under the current transformation corresponds to the given position in device space.

If the *matrix* argument is supplied, **itransform** uses it instead of an implicit reference to *CTM*.

The arguments *xt* and *yt* must be numbers.

To achieve uniform line weights across an output page, lines should be positioned at the same relative positions to output device pixels. It is a simple matter to specify positions in device-independent user space, yet achieve device-dependent positioning by adjusting user space positions according to the following method:

Example:

```
transform round exch round exch itransform
```

When given an (x, y) position in user space, these operations transform that position to device space, round it to the nearest output pixel boundary, and inverse transform it back to the user space position corresponding to this device-dependent position.

Errors: stackunderflow, typecheck, undefinedresult.

♦ idtransform

```
        xdt ydt  idtransform  xd yd
xdt ydt matrix   idtransform  xd yd
```

With no *matrix* argument, **idtransform** (inverse delta transform) returns the positionless vector (*xd, yd*) such that (*xd, yd*) * *CTM* = (*xdt, ydt*). This combination of **dtransform** and **itransform** gives the vector in user space that corresponds to the given device space vector.

If the *matrix* argument is supplied, **idtranform** uses it instead of an implicit reference to *CTM*.

The arguments *xdt* and *ydt* must be numbers.

Errors: stackunderflow, typecheck, undefinedresult.

♦ invertmatrix

$matrix_1$ $matrix_2$ **invertmatrix** $matrix_2$

replaces the contents of $matrix_2$ with the result of inverting $matrix_1$ and pushes the modified $matrix_2$ back onto the operand stack.

Errors: stackunderflow, typecheck, undefinedresult.

3.5.3. Character and Font Operators

Fonts are collections of graphical symbols accessible through several POSTSCRIPT operators. In the standard case, a POSTSCRIPT font represents a typeface of one particular design. Each installation has a particular set of fonts which may be used in POSTSCRIPT. Fonts are named with strings. Fonts may be named in arbitrary ways, but typically, a hierarchical scheme with some agreed upon separator is used (e.g., "-"). For example, the names of all fonts which were created from artwork licensed by the International Typeface Corporation may begin with the letters "ITC" followed by the font family (e.g., "Souvenir," "Galliard," "FrizQuadrata,"), followed by the face or weight (e.g., "Medium," "Roman," "BoldItalic"). An entire font name might be **ITC-Souvenir-BoldItalic**.

Such names are used as the argument to the **findfont** operator. **findfont** returns a dictionary (called a *font dictionary*) if the font is known to POSTSCRIPT. Fonts (via, in part, their dictionaries) may be modified by geometrical transformations like any other POSTSCRIPT graphical object. The Graphics State contains a notion of the *current font* which is the set of character descriptions referenced by the various character imaging operators (see below).

POSTSCRIPT's font mechanism and the contents of a font dictionary are given in Appendix A, in addition to the manner in which a users can define their own fonts. The typical user of fonts in POSTSCRIPT need not be concerned with these details.

3.5.3.1. Font Dictionary Operators

The following operators deal with font dictionaries. They are used to create, find, and scale fonts, and to set and return the *current font*; part of the Graphics State.

♦ **currentfont**

– **currentfont** `font-dict`

pushes the font dictionary of the font that is in the current Graphics State.

Errors: stackoverflow.

♦ definefont

```
key dict  definefont  font-dict
```

makes the font-description found in *dict* into a POSTSCRIPT font. **definefont** creates a *FontID* (an object of type **fonttype**) for this font, puts it in *dict* with key "**FID**", and makes the dictionary *readonly*. This dictionary is placed in the global dictionary "**FontDirectory**" with *key*. The modified dictionary is returned on the stack.

Errors: dictfull, invalidfont, stackunderflow, typecheck.

♦ findfont

```
key  findfont  font-dict
```

looks up (in **FontDirectory**) the font whose name is on the top of the stack and pushes its font dictionary on the stack. A detailed description of the contents of a font dictionary may be found in Appendix A.

Errors: invalidfont, stackunderflow, typecheck.

♦ scalefont

```
font-dict scale  scalefont  transformed-font-dict
```

scales the font matrix in *font-dict* by *scale*, creates a copy of *fontdict*, and pushes the resulting font dictionary on the stack. The *font-dict* returned from **findfont** is a one-unit by one-unit (in user space) font. (The choice of default user coordinates having one unit equal to one point results in default fonts in default user space being one-point fonts.) When **scalefont** applies its *scale* argument to such a font, it results in a font scaled to the number of user space units specified. For example, a **12 scalefont** applied to a default font by results in a new font description that is 12 units wide by 12 units high. Any characters shown from this font will take on that size in whatever the current user-space coordinate system (*CTM*) specifies.

Errors: stackunderflow, typecheck.

♦ **makefont**

```
font-dict matrix  makefont  transformed-font-dict
```

transforms the font matrix in *font-dict* by *matrix*, creates a copy of *fontdict*, and pushes the resulting font dictionary on the stack. **makefont** is more general than **scalefont** in that it allows an arbitrary matrix to modify an existing font. To achieve simple, uniformly scaled fonts, use **scalefont**. To achieve non-uniformly scaled, translated, or rotated (in the font itself) fonts, use **makefont**. For example, a **[10 0 0 8 0 0]** matrix applied to a default font by **makefont** results in a new font description that is 10 units wide by 8 units high.

Note that the special effects of **makefont** can also be achieved by using simple scaled fonts with non-uniform scaling and rotation in the coordinate system (via **scale**, **rotate**, and **translate**). **makefont** is essentially a convenience operator that allows the PostScript program to not have to switch coordinate systems often when showing unusual characters. Particularly for rotated characters, it is often more convenient to rotate the coordinate system rather than rotate inside the font.

Errors: stackunderflow, typecheck.

♦ **setfont**

```
font-dict  setfont  –
```

establishes the font to be used for all subsequent character imaging operators and remains in force until the next **setfont**, **grestore**, or **restore** operator is executed.

Example:

```
% find, scale, and set a 10-unit Courier.
/Courier findfont 10 scalefont setfont
```

Errors: stackunderflow, typecheck.

3.5.3.2. Character Imaging Operators

PostScript has several operators for showing strings of characters. The graphics environment within which a **show** command is executed affects both the appearance of the character images (i.e., the current font face specified by **setfont**) and the size of the images (i.e., both the font's size and the current transform). The simplest variant of the character imaging commands is **show** which simply lays down a string of characters in the current font starting at the *current point* and updating the current point by the width data for each character. **widthshow** provides a mechanism useful for setting justified text. **ashow** and **awidthshow** are useful for applications requiring copy-fitting and uniform letter spacing. Finally, **kshow** calls back to the PostScript interpreter between each character, allowing the ultimate in individual letter spacing adjustments.

♦ show

`string` **show** –

images the characters in *string* starting at the current point according to the font face, size and orientation specified by the most recent **setfont**. After each character is imaged, the current point is updated by the amount specified in the width information for the character. Upon completion, the current point remains at the position that resulted from the imaging of the last character in the string. There must be a current point (typically set via the **moveto** operator) when **show** is executed; otherwise is executes the error operator **nocurrentpoint**.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ widthshow

`numx numy char-code string` **widthshow** –

images characters in *string* in a manner similar to **show**. But for each instance of the character *char-code* in *string* the current point is modified by adding the vector (*numx,numy*) in addition to the normal width of *char-code*. This operator enables the setting of a justified string of text in a single command.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ ashow

`numax numay string` **ashow** –

images characters in *string* in a manner similar to **show**. But for each character in *string* the current point is modified by adding the vector (*numax, numay*) in addition to the normal width of the character. This operator enables the fitting of a string of text to a specific width in a single command.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ awidthshow

`numx numy char-code numax numay string` **awidthshow** –

images characters in *string* in a manner similar to **widthshow**. But for each character in *string* the current point is modified by adding the vector (*numax,numay*) in addition to the normal width of the character. This operator enables the fitting of a string of justified text to a specific width in a single command.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ kshow

```
proc string  kshow  –
```

images characters in *string* in a manner similar to **show**, but allows user intervention between characters. If the character codes in *string* are c_0, c_1, ... , c_n, **kshow** will proceed as follows: First it shows c_0 at the current point, updating the current point by c_0's width. Then it pushes the character codes c_0 and c_1 onto the stack and executes *proc*. The *proc* may perform any actions it wishes; typically it will modify the current point somehow to affect the subsequent placement of c_1. If *proc* modifies the Graphics State, such changes will remain in effect through subsequent executions of *proc*. **kshow** continues by showing c_1, pushing c_1 and c_2 onto the stack, executing *proc*, and so on, finishing by pushing c_{n-1} and c_n onto the stack, executing *proc* and finally showing c_n.

The name **kshow** is derived from *kern*-show. (To kern characters is to adjust their spacing on a character pair basis to achieve a more pleasing layout.) While the **kshow** operator allows user-defined kerning operations, it is considerably more powerful than a simple kerning operator, as it allows arbitrary computation between each character pair.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ stringwidth

```
string  stringwidth  wx wy
```

calculates the change in the current point that would occur if *string* were given to the **show** operator with the current font. *wx* and *wy* are the *width* of *string* in user coordinates.

Errors: stackunderflow, typecheck.

3.5.4. Path Construction Operators

A POSTSCRIPT *path* is a general purpose construct that defines a geometric shape. Paths represent outlines of areas to be filled with a color or image, and they represent trajectories along which lines may be drawn. A path is composed of straight and curved line segments. These segments may connect to one another, or they may be discontinuous. A continuous section of a path may be *closed*, that is, its last segment may connect back to its starting point, otherwise it is considered *open*. A single path may contain discontinuous closed sections, thus representing many areas. A path may even intersect itself. All paths that can be created through application of the path construction operators are legal in POSTSCRIPT.

The POSTSCRIPT interpreter allows one path to be constructed at a time; this path is called the *current path*. (Remember, it may have several discontinuous parts.) Since the current path is built by executing POSTSCRIPT operators, other paths may be saved and modified by treating them as executable arrays using the basic mechanisms of the POSTSCRIPT language.

The **newpath** operator initializes the current path to be empty. The path is essentially an ordered list of points, where adjacent points in this list may or may not be connected by a straight line segment, or a Bezier cubic curve. All points and relative distances specified to the path construction operators are interpreted in the current user coordinate system. They are immediately transformed into the corresponding output device coordinates and are remembered as such in the current path. If the current transformation changes during construction of a path, points already entered do not move in device space. The most recently entered point in the current path is called the *current point*. If the current path is empty, there is no current point.

These path construction operators do not actually draw anything on an output device. Instead, the current path is an implicit argument to the output operators discussed in section 3.5.5.

♦ **newpath**

– `newpath` –

initializes the current path to be empty, causing there to be no current point.

Errors: (none).

♦ currentpoint

– **currentpoint** x y

returns the user coordinates (*x*, *y*) of the current point (if the current path is non-empty.) Whenever the current point is set, it is transformed to an output device coordinate through the current transformation. This position remains constant until the current point is set again. If the current transformation changes without the current point being set, the **currentpoint** operator will report a different position if that device coordinate corresponds to a different user space coordinate.

Errors: nocurrentpoint, stackoverflow, undefinedresult.

♦ moveto

x y **moveto** –

starts a new segment in the current path. **moveto** makes the point whose user space coordinate is (*x*, *y*) the current point without adding any line segments to the current path. Both *x* and *y* must be numbers.

Note: if the previous path command in the current path was a **moveto**, then its point is deleted from the current path and the new **moveto** point replaces it.

Errors: stackunderflow, typecheck.

♦ rmoveto

dx dy **rmoveto** –

(relative moveto) starts a new section in the current path, relative to the current point. If the current point is (lx, ly), then **rmoveto** makes the point (lx+*dx*, ly+*dy*) the current point without adding a line segment to the current path. If the current path is empty, **rmoveto** executes the error operator **nocurrentpoint**. Both *dx* and *dy* must be numbers.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ lineto

x y **lineto** –

continues the current path with a straight line segment from the current point to (*x*, *y*) and makes (*x*, *y*) the current point. If the current path is empty, **lineto** executes the error operator **nocurrentpoint**.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ rlineto

`dx dy` **`rlineto`** `–`

(relative lineto) behaves like **lineto**, except the new point is interpreted relative to the last point in the current path. If the last point in the current path was (lx, ly), then **rlineto** adds a straight line segment to (lx+*dx*, ly+*dy*), making (lx+*dx*, ly+*dy*) the new current point. If the current path is empty, **rlineto** executes the error operator **nocurrentpoint**. Both *dx* and *dy* must be numbers.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ arc

`x y r` ang_1 ang_2 **`arc`** `–`

builds a counterclockwise segment of a circular arc with (*x*, *y*) as center, *r* as radius, ang_1 the angle of a vector from (*x*, *y*) of length *r* to the first endpoint of the arc, and ang_2 the angle of a vector from (*x*, *y*) of length *r* to the second endpoint of the arc. If there is a current point, the **arc** operator includes a straight line segment from the current point to the first endpoint of this arc and then adds the arc itself into the current path, making the second endpoint of the arc the new current point. If the current path is empty, the **arc** operator does not produce the initial straight line segment. Angles are measured in degrees counterclockwise from the positive x-axis of the current user coordinate system. The curve produced is circular in user space. Non-uniform **scale** operations executed before an **arc** command will produce elliptical curves on the output device.

Example:

```
newpath 0 0 moveto 0 0 1 0 45 arc closepath
```

This constructs a unit radius 45 degree "pie slice."

Errors: rangecheck, stackunderflow, typecheck.

♦ arcn

`x y r` ang_1 ang_2 **`arcn`** `–`

(arc negative) behaves like **arc**, but **arcn** builds its arc segment in a clockwise direction.

Example:

```
newpath 0 0 2 0 90 arc 0 0 1 90 0 arcn closepath
```

This constructs a 2 unit radius, 1 unit wide 90 degree "windshield wiper swath."

Errors: rangecheck, stackunderflow, typecheck.

♦ arcto

x_1 y_1 x_2 y_2 r **arcto** xt_1 yt_1 xt_2 yt_2

builds a segment of a circular arc of radius *r* between two tangent lines. There must be a current point, (x_0, y_0), in the current path; otherwise **arcto** executes the error operator **nocurrentpoint**. The tangent lines are those defined from (x_0, y_0) to (x_1, y_1) and from (x_1, y_1) to (x_2, y_2).

The center of the arc is located inside the smaller angle defined by these two line segments, and the arc built is the smaller of the two possible arcs from the first tangent point, (xt_1, yt_1) on the first tangent line, to the second tangent point (xt_2, yt_2) on the second tangent line. **arcto** includes a straight line segment from the current point to (xt_1, yt_1) and the circular arc defined above in the current path, and it makes (xt_2, yt_2) the new current point. If the two tangent lines are collinear, **arcto** merely includes a straight line segment in the current path from (x_0, y_0) to (x_1, y_1), considering the arc to be the degenerate single point arc at that point. The return values are for information only; they are the two tangent points. In the collinear case, these two tangent points are identical to (x_1, y_1).

Example:

```
newpath 0 0 moveto
0 4 4 4 1 arcto
4 {pop} repeat
4 4 lineto
```

This constructs a 4 unit wide, 4 unit high right angle with a 1 unit radius "rounded corner."

Errors: nocurrentpoint, stackunderflow, typecheck, undefinedresult.

♦ curveto

x_0 y_0 x_1 y_1 x_2 y_2 **curveto** –

adds a Bezier cubic section to the current path between the current point and (x_2, y_2), using (x_0, y_0) and (x_1, y_1) as the Bezier cubic control points, and it makes (x_2, y_2) the new current point. If the current path is empty, **curveto** executes the error operator **nocurrentpoint**.

The conversion of other cubic spline representations to Bezier cubics is straightforward. If A_x, B_x, C_x, and A_y, B_y, and C_y are the coefficients of a parametric cubic equation for x and y, the equation for x, for example, is:

$$X = A_x*t^3 + B_x*t^2 + C_x*t + \textit{current-x}$$

A similar equation is used for Y. The Bezier control points for the cubic are:

$$x_0 = \textit{current-x} + C_x / 3.0$$
$$x_1 = x_0 + (C_x + B_x) / 3.0$$
$$x_2 = \textit{current-x} + C_x + B_x + A_x$$

and similarly for the y components.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ rcurveto

dx_0 dy_0 dx_1 dy_1 dx_2 dy_2 **rcurveto** –

behaves like **curveto**, but the points are interpreted relative to the current point, (cx, cy). The resulting curved segment will start at (cx, cy) and end at (cx+dx_2, cy+dy_2). (cx+dx_0, cy+dy_0) and (cx+dx_1, cy+dy_1) determine the shape of the curve in between the end points, and (cx+dx_2, cy+dy_2) becomes the new current point.

Errors: nocurrentpoint, stackunderflow, typecheck, undefinedresult.

♦ closepath

– **closepath** –

behaves like **lineto**, but constructs its line to the point last "moved to". If the current path is empty, then **closepath** does nothing.

Errors: (none).

♦ pathbbox

– **pathbbox** `llx lly urx ury`

pushes the bounding box of the current path in the current user coordinate system onto the operand stack. The results pushed are four real numbers: lower left x, lower left y, upper right x, upper right y. If the current path is empty, **pathbbox** executes the error operator **nocurrentpoint**.

Note: the bounding box of the current path in the *device coordinate system* is computed first. This box is then inverse-transformed to the current user space, and the bounding box of this resulting figure is what is returned on the operand stack. For rotated or skewed user coordinate systems, this operator may return a bounding box that is larger than expected.

Errors: nocurrentpoint, stackoverflow.

♦ flattenpath

– **flattenpath** –

replaces the current path with an equivalent path that preserves all straight line segments but has all **curveto** segments replaced by sequences of **lineto**'s. This flattening to **lineto**'s is controlled by the current setting of the flatness parameter in the Graphics State. If the current path does not contain any **curveto** segments, **flattenpath** will leave it unchanged.

Errors: limitcheck.

♦ reversepath

– **reversepath** –

replaces the current path with an equivalent one except that the points in the path are connected in the reverse order. Consider each subsequence of the current path that begins with a **moveto** operation a *subpath*. Each subpath thus represents one connected section of the current path. **reversepath** leaves the order of the subpaths within the current path unchanged, however it does reverse the connection direction within each subpath.

Errors: (none).

♦ strokepath

– **strokepath** –

replaces the current path with a path that if **filled** would produce the same result as would the current path if **stroked**. The current path resulting from the **strokepath** operator is suitable as the implicit argument to the **clip** operator.

Errors: limitcheck.

♦ charpath

string strokepath-bool **charpath** –

behaves like the **show** operator, but instead of printing the characters of *string* into the current output device, it appends to the current path a path that describes the outline(s) of the characters in *string*. The *strokepath-bool* value determines how portions of the character definition that are **stroked** are treated. If true, **charpath** applies the **strokepath** operator to any portions of the character outline descriptions that are **stroked**. If false, these portions are added to the resulting path unchanged. Thus, if the character contains only filled portions, or if the *strokepath-bool* is true, then the path that **charpath** appends to the current path is suitable as the implicit argument to **fill** and **clip**.

If the character contains only filled portions, then the resulting path may be **stroked** to output an outlined representation of the character.

Note: as long as output from the **charpath** operator remains in the current path, the **pathforall** operator is disabled.

Errors: nocurrentpoint, stackunderflow, typecheck.

♦ clippath

– **clippath** –

sets the current path to one that describes the current clipping outline. This operator is quite useful for determining the exact extent of the imaging area on the current output device.

Errors: (none).

♦ pathforall

```
mtproc ltproc ctproc cpproc  pathforall  –
```

enumerates the current path in order, executing one of the four given procedure bodies for each element in the path. The four basic elements of a path are **movetos**, **linetos**, **curvetos**, and **closepaths** (relative variants are converted to absolute positions and arcs are converted to **curvetos** by the path machinery.) The four procedure body arguments to **pathforall** correspond to these four basic elements. **pathforall** reads the current path, and for each element in the path it pushes that element's coordinates (in current user space) and executes the corresponding argument procedure body. It pushes $x\ y$ for both *mtproc* and *ltproc*, it pushes $x_1\ y_1\ x_2\ y_2\ x_3\ y_3$ for *ctproc*, and it pushes no operands for *cpproc*. An **exit** executed outside of any loops in one of the procedures will terminate the **pathforall** enumeration.

Among other uses, **pathforall** allows a POSTSCRIPT program to recast a path constructed during intricate user coordinate space changes as one with coordinates from a single, simple user coordinate space.

Note: the **pathforall** operator is disabled when output from the **charpath** operator is in the current path. In this case **pathforall** operator executes the error operator **invalidaccess**.

Errors: stackoverflow, stackunderflow, typecheck.

3.5.5. Graphics Output Operators

The operators in this group operate on the current path, define limits on the output area, and produce output on the attached raster device. Each output device maintains a *current page*, which accumulates "ink" at the places directed by the **fill**, **stroke**, **show**, and **image** operators. The current page may be cleared at any time by the **erasepage** operator, or it can be printed on the output device by the **showpage** or **copypage** operators.

The POSTSCRIPT Graphics State maintains a separate path, the *current clipping boundary*, that defines the limits on the area of the output device that are to be written on, regardless of the extent of an image to be output. Like the current transformation matrix, the current clipping boundary has a default value that depends on the output device. This clipping boundary may be restricted further through the **clip** operator defined below.

The *inside* of a path to be filled or used as a clipping boundary can have different interpretations when the path intersects itself. POSTSCRIPT normally uses a sophisticated *non-zero winding number rule* to determine what is inside and what is outside a path. This rule determines the "insideness" of a point by drawing a ray from that point in any direction through the path. Starting with zero, we add one each time the ray passes through a path segment that is clockwise, and we subtract one every time the ray passes through a path segment that is counterclockwise. If the result is zero, the point is outside, otherwise the point is inside.

With this rule, a simple convex path yields inside and outside as we would expect. Now consider a five pointed star, drawn with five continuous straight line segments intersecting each other. The entire inside of the star, points and center, are considered inside by the non-zero winding number rule. For two concentric circles, if they are both drawn in the same direction, the insides of both circles are inside according to the rule; if they are drawn in opposite directions, only the "doughnut" shape between the two circles is inside according to the rule.

Another "insideness" rule used by some other graphics systems is the *even-odd rule*. This rule determines the "insideness" of a point by drawing a ray from that point in any direction and counting the number of path segments that the ray passes through. If this number is odd, the point is inside; if even, the point is outside.

With the even-odd rule, a simple convex path yields inside and outside as we would expect just as with the non-zero winding number rule. For the five pointed star drawn with five continuous straight line segments intersecting each other, the points are considered inside, but the center is considered outside. For two concentric circles, only the "doughnut" shape between the two circles is inside according to the even-odd rule, regardless of whether the circles are drawn in the same or opposite directions.

Unless otherwise stated, any POSTSCRIPT output operator that depends on "insideness" uses the non-zero winding number rule. There are two operators however, **eofill** and **eoclip** that use the less useful even-odd rule.

♦ initgraphics

– **initgraphics** –

resets several values in the current Graphics State to their default values:

- the transformation matrix, *CTM* (as per the output device)
- the current path (empty)
- the current point (undefined)
- the current clipping boundary (as per the output device)
- the current color (black)
- the current line width (one user space unit)
- the current line cap style (butt end caps)
- the current line join style (miter joins)
- the current dash description (undashed, i.e., solid lines)
- the current miter-limit (10)

The **initgraphics** operator leaves the other Graphics State parameters untouched; these include the current output device, font, transfer function, halftone screen, and flatness setting. This operator affects Graphics State *parameters* only, it does not cause any output to the current page.

initgraphics is equivalent to the POSTSCRIPT sequence:

```
initmatrix newpath initclip
1 setlinewidth 0 setlinecap 0 setlinejoin
[] 0 setdash 0 setgray 10 setmiterlimit
```

Errors: (none).

♦ erasepage

– **erasepage** –

clears the current output page to user white. User white is typically the same as output device white, but if an atypical transfer function is in force, this may fill the current page with a uniform gray shade. **erasepage** does not affect the current Graphics State, nor does it cause any output to be printed on the physical output device.

Errors: (none).

♦ showpage

– `showpage` –

prints one copy of the current output page on the attached device and then performs an **erasepage** and an **initgraphics**. Exactly how the page is printed depends on the output device; see the description of a particular output device for details on how it handles **showpage**.

Note: **showpage** resets values in the Graphics State. The POSTSCRIPT sequence **copypage erasepage** avoids this action.

Errors: (none).

♦ copypage

– `copypage` –

prints one copy of the current output page on the attached device without clearing its contents or changing the graphics state (as opposed to **showpage**, which effects an **erasepage** and an **initgraphics**). To print multiple copies of a page, enclose **copypage** in a loop.

Note: the non-erasing behavior of **copypage** is device dependent, as not all implementations of POSTSCRIPT can guarantee saving the entire state of a printed page during processing. Low and medium resolution devices generally can support this behavior, but high resolution devices (over 1000 spots per inch) when printing complicated pages may not support **copypage**'s non-erasing behavior. However, all POSTSCRIPT implementations will print the current page when executing **copypage**.

Example:

```
n 1 sub {copypage} repeat showpage
```

Prints *n* copies of a page followed by a clearing of the current page.

Errors: (none).

♦ initclip

– `initclip` –

sets the current clipping boundary path to the default clipping boundary for the output device. This clipping boundary usually corresponds to the maximum image area that the output device can handle.

Errors: (none).

♦ clip

– `clip` –

intersects the inside of the current clipping boundary with the inside of the current path to produce a new (smaller) current clipping boundary. The inside of each path is determined by the normal POSTSCRIPT non-zero winding number rule. The **clip** operator implicitly closes the current path for this intersection if it is not already closed.

Note: Unlike **fill** and **stroke**, **clip** does not implicitly perform a **newpath** after it has finished modifying the current clipping boundary.

Errors: limitcheck.

♦ eoclip

– `eoclip` –

intersects the inside of the current clipping boundary with the inside of the current path to produce a new (smaller) current clipping boundary. The inside of the current path is determined by the even-odd rule, while the inside of the current clipping boundary has been determined by the previous **clips** and **eoclips**. The **eoclip** operator implicitly closes the current path for this intersection if it is not already closed.

Errors: limitcheck.

♦ fill

– `fill` –

paints the inside of the current path (the portion within the current clipping boundary) onto the current page with the current color. **fill** implicitly closes any open sections in the current path. The contents of the filled area are painted completely by the current color; any previous contents of that area on the current page are obscured. Areas may be erased by filling with color set to white. The inside of the current path is determined by the normal POSTSCRIPT non-zero winding number rule.

fill implicitly performs a **newpath** after it has finished painting into the current page. To preserve the current path after a **fill** operation, use the sequence: **gsave fill grestore**.

Errors: limitcheck.

♦ eofill

– `eofill` –

behaves just like **fill**, except the inside of the current path is determined by the even-odd rule.

Errors: limitcheck.

♦ stroke

– `stroke` –

paints a line that follows the current path in the current color into the current clipping boundary on the page. This line is centered over the segments of the path, has sides parallel to the path segments, and has a total width equal (in user space) to the current value of line-width in the Graphics State. Open sections of the path are capped according to the current value of line-cap in the Graphics State, and connected sections of the path are joined according to the current value of line-join in the Graphics State. To obtain a tiny stroke consisting primarily of end-caps, a path extending some non-zero fraction of an output device pixel to give the end-cap an orientation should be used (see **itransform**.) **stroke** can also produce dashed lines (see the description of **setdash**).

stroke implicitly performs a **newpath** after it has finished painting into the current page. To preserve the current path after a **stroke** operation, use the sequence: **gsave stroke grestore**.

Note: The line-width, line-cap, line-join, miter-limit, flat-tolerance and dash Graphics State values are consulted only at the time that the **stroke** operator is executed. If they change during the time that the current path is built, only their final values (at **stroke** time) matter.

Errors: limitcheck.

♦ image

```
scan-length scanlines bits/pixel matrix proc  image  –
```

renders the gray-scale image returned by *proc* onto the current page using halftones. **image** paints the scanned image into a region of the output page according to the *matrix* parameter and the current placement of the user space unit square (clipped by the current clipping boundary). The unit square is that quadrilateral bounded by user coordinates (0, 0), (1, 0), (1, 1), and (0, 1). Prior to executing the **image** operator, this unit square may be positioned, rotated and scaled in any manner. Typically, the *matrix* parameter is chosen so that the scanned image exactly fills this unit square.

The **image** operator will execute its *proc* argument as many times as necessary to receive the gray-scale pixels that comprise the scanned image input. This procedure must leave a POSTSCRIPT string on the operand stack containing the next set of such pixels each time it is executed. The *bits/pixel* argument determines how the pixels are packed into the 8-bit bytes (characters) of the string. Legal *bits/pixel* arguments are 1, 2, 4, 8, and 16. If *bits/pixel* is 8, the pixels fit exactly, one pixel per character. If *bits/pixel* is less than 8, the earlier pixels are taken from the high-order bits of the character and the later pixels are taken from the low-order bits of the character. If *bits/pixel* is 16, two successive characters make up one pixel value, with the earlier character containing the high-order bits of the pixel. A pixel whose value is zero corresponds to black input, while the highest value in a pixel (for 8-bit pixels this is 255) corresponds to white input. This correspondence may be modified on output by suitable modification of the output transfer function (see the **settransfer** operator.)

The **image** operator will expect to receive a total of *scan-length* times *scanlines* number of pixels from its executions of *proc*, and it will terminate once it has received this number. The number of pixels actually returned by *proc* each time is given by the length of the string it leaves on top of the stack (modified by the *bits/pixel*. A returned string of zero length indicates a premature termination of the input, and the **image** operator will terminate. The *proc* need not return a full scanline's worth of pixels, or it may return much more than a scanline. The *proc* thus controls the amount of buffering it provides through the length of the string it returns. Longer strings returned will result in fewer executions of *proc* and vice versa.

The **image** operator imposes an x-axis major indexing order on the pixels it receives. The first pixel's coordinate in *input space* is (0, 0), the next is (1, 0), and so on through (*scan-length*-1, 0). The next pixel received is (0, 1), then (1, 1), etc., until the final pixel whose coordinate is (*scan-length*-1, *scanlines*-1). The *matrix* argument defines a mapping from the unit square in user space into this input space, i.e., a coordinate within the unit square times this *matrix* yields the corresponding position within the input space.

The unit square is closed on the zero edges and open on the one edges, so that the input coordinate corresponding to unit square coordinate (1, 1) is actually outside of the defined input space. This *matrix* arrangement allows any orientation of the input image to be mapped properly into the user space unit square.

Many scanned image input files are laid out so that their first pixel corresponds to the upper left corner of the image, the next pixel is the one to the right of the first on the top scan line, etc., finishing with the bottom scan line with the last pixel corresponding to the lower right corner of the image. If there are *n* pixels per scan line, and *m* scan lines, the correct matrix for this image format is: [*n* 0 0 -*m* 0 *m*]. If the image input file is laid out bottom horizontal scan line first, top horizontal scan line last, then the correct matrix is [*n* 0 0 *m* 0 0].

The *proc* technique for returning pixels to the **image** operator provides a flexible means of dealing with a variety of image formats. A simple format involving non-compressed images may require only a simple **readstring** arrangement for obtaining the pixels. A compressed format will require a decoder in the *proc*. Even a completely synthetic image may be generated by the *proc*, as it may use the full range of POSTSCRIPT. Note: any recursive invocation of the **image** operator from within the *proc* is ignored.

A simple way to include scanned input in a POSTSCRIPT file is to include an Ascii-hexadecimal encoding of the image input directly after the line containing the **image** operator. The **currentfile** operator along with the **readhexstring** operator provide the basic tools to read this input from the POSTSCRIPT file.

N.B. The use of **image** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is an error, and results in a call on the error operator **undefined**. The **imagemask** operator, however, is valid in this context (see section 3.5.7).

Errors: stackunderflow, typecheck, undefinedresult.

♦ imagemask

`scan-length scanlines invert matrix proc` **`imagemask`** –

is similar to the **image** operator, except it renders the binary image (1 bit per input pixel only) returned by *proc* onto the current page using the current color. *invert* is a boolean. If *invert* is *false*, the current color images where 0 bits appear in the source, 1 bits remain transparent. If *invert* is *true*, the current color images where 1 bits appear in the source, 0 bits remain transparent. Note that unlike the **image** operator, which paints opaque color everywhere in its destination, **imagemask** leaves some areas (those corresponding to the transparent source pixels) unchanged.

imagemask is useful for loading raster character masks into the cache device. The **image** command cannot be used in the cache context, as it paints all colors, whereas masks have no color.

Errors: stackunderflow, typecheck, undefinedresult.

♦ setlinewidth

`num` **`setlinewidth`** –

sets the value of line-width in the current Graphics State to *num*. This value is interpreted as a scalar distance (number of units) in the user coordinate system when the **stroke** operator executes. If the current scale in user space is uniform, i.e., x-units are the same length as y-units, then stroked lines in any orientation will be drawn with a uniform width. If the current scale in user space is not uniform, e.g., x-units are scaled to be twice the size of y-units, then stroked lines will be wider or narrower depending on their orientation. If x-units are twice the size of y-units, lines perpendicular to the x-axis will be twice as wide as lines perpendicular to the y-axis.

Errors: stackunderflow, typecheck.

♦ currentlinewidth

– **`currentlinewidth`** `num`

pushes the value of line-width in the current Graphics State onto the operand stack.

Errors: stackoverflow.

♦ setlinecap

`integer` **`setlinecap`** –

sets the shape that the **stroke** command will put at the open ends of any paths when writing strokes to the output device. *integer* corresponds to the following end-cap shapes:

0. Butt caps; square butt end caps perpendicular to the path at each open end.
1. Round caps; Semicircular end caps with diameter equal to the line width centered at each open end.
2. Projecting square caps; similar to butt end caps, but extend out one-half of a line width in the direction of the path at each open end.

Note: round end caps will print if a degenerate line (a single point) is stroked. No output will result if butt or projecting square end caps are specified for degenerate lines, as their orientation is indeterminate.

Errors: rangecheck, stackunderflow, typecheck.

♦ currentlinecap

– **`currentlinecap`** `integer`

pushes the value of line-cap in the current Graphics State onto the operand stack.

Errors: stackoverflow.

♦ setlinejoin

`integer` **`setlinejoin`** –

sets the shape that the **stroke** command will insert at the connected corners of any paths when writing strokes to the output device. integer corresponds to the following line-join shapes:

0. Mitered joins; both edges of the stroke are extended until they meet at an angle at each corner, as in a picture frame. Caution: path segments meeting at very sharp angles (less than 10 degrees) can result in long spikes when mitered. If the ratio of the length of the diagonal line through a mitered join (the spike length) to the width of the line would exceed the value of miter-limit in the current Graphics State, then the **stroke** operator makes a bevel join instead of a miter join.
1. Round joins; circular joins with diameter equal to the line width centered at each corner. Note: **stroke** outputs a full circle at each corner if round joins are specified. If path segments of less than one-half line width meet at sharp angles, unintentional "wrong sides" of these circles may show.
2. Bevel joins; the meeting path segments are finished as with butt end caps, and the resulting notch at the larger angle between these segments is filled with a triangle.

Note: join styles are significant at angles in a path. When Bezier curves are stroked, if the flatness has been set sufficiently smooth, there is no difference in appearance along the curve for all of the join styles.

Errors: rangecheck, stackunderflow, typecheck.

♦ currentlinejoin

– **`currentlinejoin`** `integer`

pushes the value of line-join in the current Graphics State onto the operand stack.

Errors: stackoverflow.

♦ setdash

```
array offset  setdash  –
```

provides the **stroke** operator with length information for rendering subsequent strokes as dashed lines with segment lengths as defined in *array*. The length of *array* determines the interpretation of its contents as follows:

length = 0 An empty array argument turns off dashed strokes. Subsequent strokes will be drawn unbroken.

length > 0 Subsequent strokes will be dashed, with filled and unfilled sections alternating (first section is filled.) The 0'th array element determines the length of the first (filled) section. The array elements are used cyclically for the succeeding section of the stroke. These sections continue to alternate unfilled and filled. *array* must contain non-negative numbers, which will be interpreted as distances (in user space) along the path for each filled and unfilled portion of the stroke. At least one of the elements of *array* must be non-zero.

The *offset* argument is another length that must be non-negative. This length starts the dashing "inside" the repeating pattern. The repeating pattern is cycled, adding up lengths of segments and alternating filled and unfilled as described above, except no output is produced until the *offset* length is exhausted. Output then begins at the beginning of the path, with the remainder of the current dash segment being output first. This *offset* argument can be thought of as setting the phase of the repeating pattern. Note: Dashed lines wrap around corners and curves just as normal strokes do. Each end of a dash section is finished with the current line cap and corners are finished with the current line join. When the stroked path ends, output stops, even if in mid-dash. POSTSCRIPT does not modify the given lengths to fit the stroked path in any way; responsibility for ensuring "correct" dash behavior at stroke ends is entirely up to the user. Each new path sequence in the current path, i.e., each path part starting with a **moveto**, begins the dash sequence over again starting with array element 0 and a filled dash section.

Example:

```
[] 0 setdash % turn dashing off - solid lines
[3] 0 setdash % 3-unit on, 3-unit off, ...
[2] 1 setdash % 1 on, 2 off, 2 on, 2 off, ...
[2 1] 0 setdash % 2 on, 1 off, 2 on, 1 off, ...
[3 5] 6 setdash % 2 off, 3 on, 5 off, 3 on, 5 off, ...
[2 3] 11 setdash % 1 on, 3 off, 2 on, 3 off, 2 on, ...
```

Errors: limitcheck, stackunderflow, typecheck.

♦ currentdash

– **currentdash** array offset

pushes the current dash array and offset as described for setdash onto the operand stack.

Errors: stackoverflow.

♦ setflat

num **setflat** –

sets the value of flat-tolerance in the current Graphics State to *num*. When the **stroke**, **fill** or **clip** operators encounter a curve in the current path, they reduce that curve to a series of straight line segments that approximate that curve on the current output device. The flat-tolerance value determines the maximum error allowed in output device pixels for these approximations. A small flat-tolerance value, e.g., 1, will produce an accurate curve approximation at the expense of more computation, whereas a larger flat-tolerance value may produce a cruder approximation with substantially less computation. A default value for the flat-tolerance value should be set in each POSTSCRIPT installation depending on the characteristics of the output device.

Errors: stackunderflow, typecheck.

♦ currentflat

– **currentflat** num

pushes the value of flat-tolerance in the current Graphics State onto the operand stack.

Errors: stackoverflow.

♦ setmiterlimit

num **setmiterlimit** –

sets the value of miter-limit in the current Graphics State to *num*. This number is the maximum ratio of the length of the diagonal line through a mitered join to the line width. Miter joins at sharp angles that would produce spikes whose length ratio would exceed this value are beveled instead. The value of the angle, A, such that bevels are performed for angles sharper than A is given by the formula: miter-join = 1 / sin(A/2).

Examples of miter-join values are: 1.415 cuts off miters at angles below 90 degrees; 2.0 cuts off miters at angles below 60 degrees, and 10.0, which cuts of miters at angles below 11 degrees. The default value of miter-limit is 10. The miter ratio can never be less than 1. Setting the miter-limit to 1 results in bevel joins always (when miter joins are specified). An attempt to set the miter-join to a value less than 1.0 results in a **rangecheck**.

Errors: rangecheck, stackunderflow, typecheck.

♦ currentmiterlimit

– **currentmiterlimit** num

pushes the value of miter-limit in the current Graphics State onto the operand stack.

Errors: stackoverflow.

♦ setgray

num **setgray** –

sets the color in the current Graphics State to a gray shade corresponding to *num*. *num* is expected to be a number between 0 and 1, with 0 corresponding to black, 1 corresponding to white, and values in between corresponding to shades of gray perceived as changing evenly between black and white as the gray value changes from 0 to 1. Note that different output devices render halftones differently; the **setscreen** and **settransfer** operators allow enough flexibility so that each output device can achieve this smooth change in apparent gray levels.

N.B. The use of **setgray** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is an error, and results in a call on the error operator **undefined** (see section 3.5.7).

Errors: stackunderflow, typecheck.

♦ currentgray

– **currentgray** num

pushes the value of gray in the current Graphics State onto the operand stack. If the current color is not a pure gray, but has some color hue, then the value returned is the brightness component of the current color.

Errors: stackoverflow.

♦ sethsbcolor

hue saturation brightness **sethsbcolor** –

sets the color in the current Graphics State to the given hue, saturation, and brightness components. Each operand is expected to be a number between 0 and 1. A 0 hue corresponds to pure red, 1/3 corresponds to pure green, 2/3 corresponds to pure blue, and 1 corresponds to pure red, with values between these points corresponding to mixtures of the adjacent colors. The saturation component refers to the pureness of the color: 0 corresponds to no color (only brightness or gray); 1 corresponds to pure color with no white light mixed in. Note that a 0 saturation makes the hue component irrelevant. The brightness component corresponds to the vividness of the color, with 0 corresponding to black and 1 corresponding to vivid color. The brightness is also used as the gray value by devices without color capability.[3]

N.B. The use of **sethsbcolor** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is an error, and results in a call on the error operator **undefined** (see section 3.5.7).

Errors: stackunderflow, typecheck.

♦ currenthsbcolor

– **currenthsbcolor** hue saturation brightness

pushes the three components of the color in the current Graphics State as per the Hue-Saturation-Brightness model onto the operand stack.

Errors: stackoverflow.

[3]For a complete explanation of the PostScript color models and the conversions between Hue-Saturation-Brightness and Red-Green-Blue please refer to the paper by Alvy Ray Smith, Color Gamut Transform Pairs, *Computer Graphics*, Vol. 12, No. 3, August 1978. (Our Hue-Saturation-Brightness model is referred to there as Hue-Saturation-Lightness.)

♦ setrgbcolor

`red green blue` **`setrgbcolor`** –

sets the color in the current Graphics State to the given red, green and blue components. Each operand is expected to be a number between 0 and 1, with the amount of colored light in each primary component increasing in proportion to its given value. If all three components are equal, the corresponding color is a pure gray. If not all components are equal, the corresponding gray (brightness value) is computed according to the NTSC video standard.

N.B. The use of **setrgbcolor** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is an error, and results in a call on the error operator **undefined** (see section 3.5.7).

Errors: stackunderflow, typecheck.

♦ currentrgbcolor

– **`currentrgbcolor`** `red green blue`

pushes the three components of the color in the current Graphics State as per the Red-Green-Blue model onto the operand stack.

Errors: stackoverflow.

♦ setscreen

`frequency rotation spot-function` **`setscreen`** –

sets the halftone screen definition in the current Graphics State. The *frequency* operand is a number that specifies the screen frequency, the number of halftone dots per inch on the output page. Each halftone dot will typically comprise many output device pixels. The *rotation* argument specifies the number of degrees by which the grid of halftone dots is to be rotated with respect to the default coordinate system of the output page. This definition of halftone dot size and placement is fixed; halftone dots do not scale, translate or rotate according to the **scale**, **translate** and **rotate** operators. The *spot-function* is a POSTSCRIPT procedure body that will be called with a pair of numbers, x and y, each in the range [-1, 1), and which must return a number that indicates the value of the halftone dot shape solid function at that point. The values of this function may be integers or real numbers in the range [-1, 1]. These values indicate which pixels within a halftone dot are to be blackened for different gray levels. The highest spot function value positions will be blackened first for the lightest grays, and the lowest spot function value positions will be blackened last for the darkest grays.

Each device installation should set up the default screen definition that works well for that device. It is only a rare POSTSCRIPT program that would need to specify its own screen definition.

Errors: rangecheck, stackunderflow, typecheck.

♦ currentscreen

– **`currentscreen`** `frequency rotation spot-function`

pushes all the parameters of the current halftone screen onto the operand stack.

Errors: stackoverflow.

♦ settransfer

`gray-transfer-function` **`settransfer`** –

The *gray-transfer-function* is a POSTSCRIPT procedure body that will be called with single real number in the range [0, 1], and which returns a single real number in the same range. This function maps the apparent gray level (specified to the **setgray** operator) to the actual device gray level (ratio of white pixels to total pixels in the halftone dot). This function allows apparent gray levels to be mapped empirically to the gray reproduction characteristics of a particular output device. For example, the transfer function **{1 exch sub}** will invert the output image. When in doubt, use the empty function, **{}**, for the transfer function; it will pass its argument back unchanged.

Each device installation should set up the default transfer function that works well for that device. It is only a rare POSTSCRIPT program that would need to specify its own transfer function.

N.B. The use of **settransfer** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is an error, and results in a call on the error operator **undefined** (see section 3.5.7).

Errors: stackunderflow, typecheck.

♦ currenttransfer

– **`currenttransfer`** `gray-transfer-function`

pushes the current gray transfer function onto the operand stack.

Errors: stackoverflow.

3.5.6. Device Setup Operators

The Graphics State contains an entry for the current output device. Each output device described in this section renders shapes and halftones onto an output raster in some fashion. Typically, when POSTSCRIPT is started, one of the first operations requested will be the installation of the main output device. During later graphics execution, temporary switching to the null output device or to the cache output device may occur as necessary. It is possible, however, to change main output devices, if the POSTSCRIPT processor is connected to more than one physical output device.

One important feature implemented by each output device is its default transformation matrix. This matrix maps default user coordinates (one unit equals 1/72 inch, origin at lower left corner of the standard output page) to device coordinates. POSTSCRIPT installs that matrix as the current transformation matrix in the Graphics State when it installs the output device, or when it executes an **initmatrix** or **initgraphics** operator.

♦ **nulldevice**

– `nulldevice` –

installs the null device as the current output device. The null device produces no printed output, but it behaves like a normal output device in all other respects. The null device is often used for exercising the POSTSCRIPT graphics machinery to load the character cache, build paths, operate on paths and query their bounding boxes, etc., or work with the built-in transformation matrix machinery without producing output. The default transformation matrix for the null device is the identity transform: [1 0 0 1 0 0].

Errors: (none).

♦ framedevice

`matrix width height proc` **`framedevice`** `–`

installs an output device that writes bits into a full frame buffer as each output operator (**fill**, **stroke**, or **image**) is executed. This operator allocates a frame buffer with dimensions 8 * *width* bits wide by *height* bits high, where width and height are according to the particular physical raster output device. The frame device will use these dimensions for its default clipping boundary. Note that *width* is in bytes while *height* is in bits. The *matrix* argument is the matrix that the frame device will use as the default transformation matrix.

The *proc* argument is a procedure body that will be executed as part of the execution of the **showpage** operator. This procedure body may report progress, etc., but its most important task is to call a special POSTSCRIPT operator that will empty the frame buffer onto the physical output device. Those operators are special for each physical device, and are not documented in this manual.

Errors: stackunderflow, typecheck.

3.5.7. Character Cache Management Operators

The POSTSCRIPT interpreter manages a character cache for the scan-converted (bitmap) representations of character shapes. The operators defined in this section allow management of the character cache and modification of cache behavior. Most POSTSCRIPT users need not concern themselves with these operators, and most POSTSCRIPT programs will not use them. The default operation of the cache is designed for good performance for average applications.

The decision to cache a character is made based on the size of the character and the current state of the character cache. Text in larger sizes will not normally be cached, as it takes up a large amount of space to do so, and such text is typically rarely produced. The purging of items from the cache is done by a *least recently used* algorithm based on typeface and transformation information. Fonts become available for caching when a fontid (FID) is generated for them (by the **definefont** operator). Cached entries for this font will be purged if the fontid is destroyed due to the execution of a **restore**. Thus, if a program wants to print a multi-page document with **save/restore** pairs around each page (to reclaim string storage, for example), the **findfont**s (and **makefont**s if possible) for the document should occur in the program's preamble, outside the scope of the first page itself. The character cache is structured on the basis of four sets of numbers:

- The amount of storage allocated for character bitmaps.
- The number of distinct fonts for which characters will be cached. In this context, a font is identified as a combination of the fontid and the specific matrix which represents the final size and orientation of the characters on the page (the concatenation of the FontMatrix and the CTM at the time of a **show**).
- The number of individual characters cached.
- The maximum size of any single character bitmap.

Once one of these parameters reaches its maximum, an operation is performed to remove items from the cache until the value of that parameter has sufficiently decreased to satisfy the current request for space in the cache. Each such operation discards an entire fontid/FontMatrix sets and all its associated bitmaps.

♦ cachestatus

– **cachestatus** bsize bmax msize mmax csize cmax maxbits

returns the current size and maximum limit for bitmap storage, fonts, and characters, and the maximum size of a single bitmap.

Errors: stackoverflow.

♦ setcachedevice

wx wy llx lly urx ury **setcachedevice** –

is designed for use in font imaging. **setcachedevice** installs a device similar to a frame buffer device, but whose frame storage is in the POSTSCRIPT character cache. Any output directed to this device will be saved for later use by the font operators (**show**, etc.) to achieve faster character output. The POSTSCRIPT program will not be able to access the cached item directly, but only indirectly through the font operators. **setcachedevice** may only be executed within the context of a **BuildChar** call-back (see Appendix A). The **BuildChar** call-back occurs within a **gsave** - **grestore** sequence, so that the output device installed before the **setcachedevice** operation will be reinstated properly. After execution of a **setcachedevice** and until the termination of the **BuildChar** procedure, use of the operators **setgray**, **sethsbcolor**, **setrgbcolor**, **settransfer**, and **image** will result in an error (**undefined**). Use of the **imagemask** operator, however, is permitted. The operands to **setcachedevice** are all numbers in the character coordinate space. *wx* and *wy* comprise the basic width vector for this character. Most Indo-European alphabets (including the Roman alphabet) will have a positive *wx* and a zero *wy*; Semitic alphabets will have a negative *wx*; some Oriental alphabets will have a non-zero *wy*. *llx* and *lly* are the coordinates of the lower left corner of the bounding box of the character, and *urx* and *ury* are the coordinates of the upper right corner of the bounding box. If this bounding box is too small, the cached item will be clipped to the inside of this box.

Errors: stackunderflow, typecheck, undefined.

♦ setcharwidth

`wx wy` **`setcharwidth`** –

is similar to **setcachedevice**. It may only be invoked from within a **BuildChar** call-back. Rather than saving a cached character mask, **setcharwidth** is used to inform the font machinery that **BuildChar** should be called *every time* this character is imaged. There are no restrictions on the use of **setgray**, **sethsbcolor**, **setrgbcolor**, **settransfer** and **image** after a **setcharwidth**. **setcharwidth** may be used, for example, by characters which wish to incorporate opaque white.

Errors: stackunderflow, typecheck, undefined.

♦ setcachelimit

`maxbits` **`setcachelimit`** –

reset the maximum allowable size of a cached bitmap. *maxbits* is in bytes of storage required to hold a bitmap. The argument is a non-negative integers.

Errors: limitcheck, rangecheck, stackunderflow, typecheck.

Error Operators

Execution of a POSTSCRIPT primitive may result in an error. Errors are not actually POSTSCRIPT operators, but are POSTSCRIPT procedures of a special nature. POSTSCRIPT error handlers are defined in a dictionary called **errordict** which is defined in **systemdict**. No other instances of the error operators or the error dictionary are considered. When a POSTSCRIPT operator causes an error, the arguments to the offending operator (if any) are replaced on the operand stack, the name of the offending operator is pushed on the operand stack, and the designated error operator is executed *directly out of* **errordict** *that is in* **systemdict** (rather than being looked up in the context of the entire dictionary stack). Initially, the contents of **errordict** may be changed (to install error handlers appropriate for a given environment), but **errordict** may be made *readonly* (and its contents similarly protected) to prevent further modification of the error handling mechanism.

In the initial startup setting, all of the error operators behave in essentially the same manner. They snapshot the state of the operand, dictionary, and execution stacks, print a message detailing which error occurred, report on the operator that caused the error (when appropriate), and execute a **stop**. The **stop** may be *caught* by a **stopped** construct.

More complex error handlers may wish to print a break page (dumping the contents of the stacks and other helpful information), send more information back over the output stream, implement a break package, re-raise **stop**, etc.

♦ dictfull

```
dictfull
```

dictfull occurs when a **def**, **put**, or **store** operator attempts to define a new entry in a dictionary that cannot hold it (i.e., a dictionary whose **length** and **maxlength** are already equal).

This often occurs when an error in planning underestimated the required size of a dictionary. Increasing the size of the offending dictionary (when the dictionary is created with a **dict** operator) should remove the error.

♦ dictstackoverflow

`dictstackoverflow`

The dictionary stack has grown too large. Too many **begins** (without corresponding **ends**) have pushed too many dictionaries on the dictionary stack. This error places the current contents of the dictionary stack in an array on the operand stack and resets the dictionary stack to contain only the system dictionary and the user dictionary. See Appendix B for the limit on the size of the dictionary stack.

dictstackoverflow often occurs when a program neglects to **end** a dictionary properly.

♦ dictstackunderflow

`dictstackunderflow`

An attempt has been made to remove (**end**) the bottommost instance of the user dictionary from the dictionary stack.

dictstackunderflow often occurs when a program does not balance **begin** and **end** operations properly.

♦ execstackoverflow

`execstackoverflow`

The execution stack has overflowed. Procedure invocation is nested too deep. This may result from a recursive call that goes too deep for POSTSCRIPT to handle. See Appendix B for the size of the execution stack.

♦ interrupt

`interrupt`

Processes an external request to interrupt execution of a POSTSCRIPT program. When POSTSCRIPT is run interactively, it listens to the user's keyboard, and if the user types a ^C (control-C), it executes this operator. The default definition of **interrupt** executes a **stop**.

♦ invalidaccess

`invalidaccess`

An attempt has been made to improperly reference an array, dictionary, string, or current path which has restricted access (e.g., readonly).

♦ invalidexit

`invalidexit`

An **exit** command was encountered outside the body of a loop, or attempted to leave the context of a **run** or **stopped** operator.

♦ invalidfileaccess

`invalidfileaccess`

The access string specification to a **file** command was unacceptable.

♦ invalidfont

`invalidfont`

Either the argument to **findfont** is not a valid font name, or the argument to **makefont** or **setfont** was not a proper font dictionary.

♦ invalidrestore

`invalidrestore`

An improper **restore** was attempted. One or more of the stacks contains dangling references to elements that would be destroyed by the execution of the **restore**.

♦ ioerror

`ioerror`

A system error has occurred while performing an input/output operation. The offending stream is pushed on the stack but is not closed (unless the error was raised by **closefile**).

♦ limitcheck

`limitcheck`

A POSTSCRIPT implementation limit has been exceeded (e.g., too many file streams have been created, or a path has become too complex). See Appendix B for the limits of the POSTSCRIPT implementation.

♦ nocurrentpoint

`nocurrentpoint`

The current path was empty, and thus there was no current point, when an operator requiring a current point was executed (e.g. lineto, curveto, currentpoint, etc.)

♦ **rangecheck**

`rangecheck`

An array or string index is out of bounds, or number was out of range (e.g., negative integer supplied when non-negative integer expected, zero supplied when positive integer expected).

♦ **stackoverflow**

`stackoverflow`

Too many entries on the operand stack. This error leaves a single element on the operand stack: an array containing all of the stack's contents at the time the error occurred. See Appendix B for the limit on the size of the operand stack.

This error may result if some frequently-called procedure is not removing all of its operands from the stack, or if it leaves "garbage" on the stack. **stackoverflow** may also be a symptom of an infinite loop.

♦ **stackunderflow**

`stackunderflow`

An attempt to remove an object from the operand stack failed because the operand stack was empty. Some operator did not have all of its required operands on the stack.

♦ **syntaxerror**

`syntaxerror`

POSTSCRIPT code invoked by a **run** had a syntax error. It ended in the middle of a string (missing ")") or procedure body (missing "}").

♦ **typecheck**

`typecheck`

Some operand to the offending command has the wrong type. This is probably the most frequent error encountered.

♦ **undefined**

`undefined`

A name was not found in some context. This can result by attempting to look up a name that is not known on the dictionary stack (e.g., by **load** or direct name lookup during execution) or by explicitly referencing a key not known in a dictionary (e.g., by **get**). Under special circumstances, certain POSTSCRIPT operators are disabled; attempts to use them will result in an **undefined** error.

♦ undefinedfilename

`undefinedfilename`

A stream cannot be created for the file name given to **file** or **run**. The file does not exist or cannot be opened for reading.

♦ undefinedresult

`undefinedresult`

A numeric computation cannot be performed or has a result that cannot be represented. Possible causes are: numeric overflow or underflow, division by zero, or inverse transformation of a singular matrix. The POSTSCRIPT scanner may raise **undefinedresult** if it attempts to create a numeric object out of range. See Appendix B for the maximum and minimum values of integers and reals.

♦ unmatchedmark

`unmatchedmark`

A mark was sought on the operand stack and none was found. This error can be raised by the **cleartomark** and **counttomark** operators.

♦ unregistered

`unregistered`

A serious system error has occurred inside POSTSCRIPT. This problem should be reported.

♦ VMerror

`VMerror`

A serious system error has occurred inside POSTSCRIPT. This problem should be reported.

A

FONT MACHINERY

POSTSCRIPT provides a powerful mechanism for the specification and use of typographic fonts. Character bitmaps may be built at runtime from analytic descriptions of their shapes, thus allowing arbitrary transformations on the font. These bitmaps may be *cached* for reasons of efficiency. Character descriptions may involve filling and stroking complex paths, or downloading a resolution and size dependent raster (using the **image** operator) for use as a font. This appendix details the specifics of POSTSCRIPT's font building and caching mechanisms, explains how existing fonts may be modified in certain ways, and explains how users may specify their own fonts. Most users need not worry about the details of POSTSCRIPT fonts. The existing font definitions and the basic font-related operators will suffice for the vast majority of needs.

Font Dictionaries

Font dictionaries are just POSTSCRIPT dictionaries, but with certain crucial key-value pairs. POSTSCRIPT has several operators that deal with font dictionaries (see section 3.5.3.1). Some of the contents of a font dictionary are optional and user-definable, while other key-value pairs *must be present and have the correct semantics* for POSTSCRIPT to operate correctly.

Font dictionaries are distinguished by a *font ID*, a key-value pair with key "FID" and value an object of type **fonttype**. This entry is inserted into a candidate font dictionary when that dictionary (and a name for that dictionary) are presented to the **definefont** operator. **definefont** takes a name and a dictionary, checks that the dictionary is a valid font dictionary, inserts a FID-fonttype pair, makes the dictionary *readonly*, and associates the font name with the dictionary in the global dictionary **FontDirectory**.

POSTSCRIPT also expects the following fields to exist in all font dictionaries:

FontMatrix	matrix	This maps the character coordinate system into the user coordinate system. The fonts returned by **findfont** are assumed to be one unit high. The actual characters may be defined in some other coordinate systems (the *character coordinate system*) and the FontMatrix maps that system into one unit in the user coordinate system. For example, built-in POSTSCRIPT fonts are defined in terms of a 1000 unit character coordinate

		system and their initial FontMatrix is [0.001 0 0 0.001 0 0]. When a font is modified by the **makefont** operator, the new matrix is concatenated with the FontMatrix to yield a transformed font. Most often, the FontMatrix is used for uniform scaling of the font.
FontType	number	indicates where the information for the character descriptions is to be found and how it is represented. User-defined fonts should have FontType 3 (the integer 3). See the section below on user-defined fonts.
FontBBox	array	an array of four numbers in the character coordinate system giving lower-left-x, lower-left-y, upper-right-x, and upper-right-y of the font bounding box. The font bounding box is the bounding box of the shape that would result if all of the characters of the font were placed with their origins coincident. This information is used in character-caching and clipping decisions.
Encoding	array	The Encoding entry is a vector of 256 names which maps character codes (the array indices in the range 0 to 255) to character names (the values in the array). This *encoding vector* may be changed by the user (see details below) to impose different character encoding schemes: EBCDIC, ISO, or other character set mappings.

a. POSTSCRIPT Built-In Fonts

POSTSCRIPT's built-in fonts contain the following information:

FontName	name	FontName is the name of the font as specified to **findfont** and **definefont**.
PaintType	integer	The PaintType indicates how the font is imaged. 0 The character descriptions are filled. 1 The character descriptions are stroked. 2 The character descriptions (designed to be filled) are outlined. 3 The character descriptions are respon-

		sible for filling or stroking (or some combination of those operations) themselves.
		Arbitrarily changing a font's PaintType will most likely be disastrous. The only reasonable change is from 0 (filled) to 2 (outlined).
Metrics	dictionary	This entry is not present by default, but provides the means by which users can change width and sidebearing information for the font (see **Changing Things** and **Font Metric Information** below).
FontInfo	dictionary	(See below.)
CharStrings	dictionary	The CharStrings entry associates character names (keys) with shape descriptions (values, stored in a protected, proprietary format). Some characters may appear in the CharStrings dictionary without being present in the Encoding vector; the user can remap the font to access these characters.
Private	dictionary	The Private entry contains other protected information about the font.

The *FontInfo* dictionary may contain the following information:

Notice	string	Trademark or Copyright notice (if applicable).
version	string	Font version number.
FullName	string	The full ''print'' name of the font.
FamilyName	string	The name of the ''font family'' to which it belongs.
Weight	string	The ''weight'' of the font (e.g., Bold, Medium, Light, Ultra, Heavy).
ItalicAngle	number	The angle in degrees counter-clockwise from the vertical of the dominant vertical strokes of the font.
isFixedPitch	boolean	Indicates that the font is a ''typewriter'' font.

UnderlinePosition number	Distance from the baseline for positioning underlining strokes. This number is in units of the character coordinate system.
UnderlineThickness number	Stroke width for underlining. This number is in units of the character coordinate system.

a.1. Changing Things

Occasionally, users may wish to change certain things in a built-in font. One common example is the character encoding vector. Note that fonts which differ in at most their **Encoding**, **FontInfo**, and **FontName** entries will share cache space. The way to go about making a user-specified change is as follows:

1. Make a copy of the font dictionary including all entries except the FID.

```
% assumes the dictionary is on the operand stack
dup length dict /newdict exch def
   {1 index /FID ne
      {newdict 3 1 roll put}
      {pop pop}
    ifelse
   } forall
% newdict now is such a copy
```

2. Install the desired changes. For example, re-map the character codes for printing EBCDIC strings:

```
% assumes ebcdicencoding is an array
% of 256 names which maps EBCDIC
% codes to POSTSCRIPT character names.

% install this array in the new (copied)
% font dictionary
newdict /Encoding ebcdicencoding put

% install this as a font in the system
% under the name "MyEBCDICFont"
/MyEBCDICFont newdict definefont pop
```

The basic font metric information may be changed as well. The **Metrics** information in the font dictionary is the means by which users can change the default width and side bearings of characters on an individual basis. This mechanism differs from **ashow**, etc., in that the new metrics apply to the characters *in the font* for as long as the new font dictionary exists. The entries in this dictionary may be of three different forms. The keys are the character names as they appear in **CharStrings** and **Encoding**. The values may be:

1. A single number, indicating a new width only (the x value of the width vector, with y being zero).

2. An array of two numbers, indicating new left side bearing and new width (again in x, with the y coordinates zero).
3. An array of four numbers, indicating true vectors (x and y components) for left side bearing and width.

All of these values are in the character coordinate system of the font.

Here is an example which changes the widths of the digits (0-9) in an existing font:

1. Make a copy of the existing font dictionary including all entries except FID, and making room for the **Metrics** entry.

```
% assumes the dictionary is on the operand stack
dup length 1 add dict /newdict exch def
   {1 index /FID ne
      {newdict 3 1 roll put}
      {pop pop}
    ifelse
   } forall
```

2. Insert the **Metrics** dictionary and the desired values.

```
newdict /Metrics 10 dict put
newdict /Metrics get begin
  [/zero /one /two /three /four
   /five /six /seven /eight /nine]
  {700 def} forall
end
```

3. Install the new dictionary as a font in the system

```
/myChangedFont newdict definefont pop
```

b. User-Defined Fonts

User-defined fonts must be carefully constructed. POSTSCRIPT assumes that such fonts will be reasonably well-behaved. As mentioned above, user-defined fonts must have **FontType** 3 in the font dictionary. When POSTSCRIPT wants to image a character out of a font, it checks to see if it has that character in its character cache. If so, POSTSCRIPT uses the cached character bitmap and metric information and the rest of the font machinery is not invoked. If POSTSCRIPT does not have the character cached because it is too big to cache or POSTSCRIPT has never encountered the character before, POSTSCRIPT will invoke the character building machinery. POSTSCRIPT pushes the font dictionary and the character code (an integer) of the character to be built onto the operand stack, and calls the procedure **BuildChar** which must be present in the font dictionary. **BuildChar** must use the information at hand (the character code and the current font dictionary) to construct and present a character back to POSTSCRIPT. This typically involves determining the character shape needed, setting the cache device (so that the constructed character will be cached if possible), supplying character metric information, constructing the character shape and imaging it.

When **BuildChar** gets control, the current transformation matrix is the concatenation of the font matrix (**FontMatrix** in the current font dictionary) and the matrix that was the current transformation matrix before the font machinery was invoked (the user coordinate system). Thus **BuildChar**'s coordinate system is the character coordinate system, and the resulting character shapes will be the desired size on the page. Before imaging the character, **BuildChar** must allow POSTSCRIPT to cache it if possible. **BuildChar** *must* make a call on **setcachedevice** or **setcharwidth**. POSTSCRIPT may or may not actually set the cache device; the **BuildChar** code has no way of really knowing. If the cache device was not set, then the character will be imaged onto the page in the current position. If the cache device was set, the character will be imaged into the cache and POSTSCRIPT will transfer the image onto the page at the current position.

Here is a small example of a user font which has only one character, a 1 unit by 1 unit solid box with a width of 2 units along the baseline. This box is imaged no matter what character code is shown. The character coordinate system is on a 1000 unit scale.

```
/ExampleFontDict 8 dict def
/$workingdict 10 dict def
ExampleFontDict begin
/FontType 3 def
/FontMatrix [0.001 0 0 0.001 0 0] def
/FontBBox [0 0 1000 1000] def
/Encoding 256 array def
0 1 255 {Encoding exch /Box put} for
/CharProcs 1 dict dup begin
    /Box {0 1000 lineto 1000 1000 lineto
          1000 0 lineto 0 0 lineto
          closepath fill
         } def
end def
/BuildChar
   {$workingdict begin
    /charcode exch def
    /fontdict exch def
    fontdict /CharProcs get
       fontdict /Encoding get
       charcode get get          % get the CharProc
    gsave
    0 setgray newpath
    2000 0                       % width vector
    0 0                          % lower left
    1000 1000 setcachedevice     % upper right

    exec                         % do the CharProc
    grestore
    end
   } def
end
/ExampleFont ExampleFontDict definefont pop

% now image two such 12 unit boxes at position 35 25
/ExampleFont findfont 12 scalefont setfont
35 25 moveto (AA) show
```

Font Metric Information

a. The Character Coordinate System

The *character coordinate system* is the system in which an individual character shape is defined. The *origin* (or *reference point*) of the character is the point which is mapped to the current point when the character is shown. For example, in the POSTSCRIPT sequence

```
40 50 moveto (ABCD) show
```

the origin of the 'A' is placed at coordinate (40,50) in the user coordinate system. After the 'A' is shown, the current point is updated by the *width* of 'A' (a vector) and the origin of the 'B' is placed at this new location.

The *bounding box* of a character is the smallest rectangle (oriented with the coordinate system axes) which will just enclose the entire character's analytic shape. The bounding box is often expressed in terms of its lower left corner and upper right corner, relative to the character origin.

The *side bearing* of a character is the distance from the character's origin to the left edge of the character bounding box. Note that this distance may be negative for characters that *kern* to the left of their origin.

b. Character Metrics

In POSTSCRIPT, character metric information for built-in fonts may be accessed procedurally, and modified by the user as detailed above. The **stringwidth** operator may be used to obtain character widths. The sequence **charpath flattenpath pathbbox** may be used to determine character bounding boxes and side bearings. The font bounding box appears in the font dictionary with key **FontBBox** and value an array of four numbers. Character metrics may be changed by registering new width and side bearing information in a **Metrics** dictionary within the font dictionary.

The user must be cautioned, however, that arbitrary and wholesale modifications to the default widths and side bearings of a typeface will almost certainly be visually disastrous. Determining pleasing and correct character spacing is a difficult and laborious art. Random stabs at changing a familiar and accepted set of character metrics should be discouraged and avoided.

B

IMPLEMENTATION LIMITS

Implementations of the POSTSCRIPT interpreter may impose certain limits on the number and size of various objects. Typical POSTSCRIPT programs should never have to concern themselves with such implementation limits, but very large or complex programs might encounter them. The following is a description of the limits of the current implementation of POSTSCRIPT.

MaxInteger	2147483647	The largest value in the range of type **integertype**. This value is 2^{32}-1, 16#7FFFFFFF.
MinInteger	-2147483648	The smallest value in the range of type **integertype**. This value is -2^{32}, 16#8000000. Note that the POSTSCRIPT scanner will scan the integer representation of this number as a *real*, but the value can be generated internally, or by scanning a radix constant.
MaxReal	10^{38}	The largest value in the range of type **realtype**. (Real numbers fall in the range $\pm 10^{\pm 38}$.)
MinReal	-10^{38}	The smallest value in the range of type **realtype**.
MaxArrayLength	65535	Maximum length of an array. This number is 2^{16}-1.
MaxStringLength	65535	Maximum length of a string.
MaxDictLength	2000	Maximum length of a dictionary.
MaxNumberDicts	65535	Maximum number of dictionaries allowed.
MaxStreams	6	Maximum number of open file streams. This number includes POSTSCRIPT's standard input, output, and error streams.
MaxNameSize	128	Maximum length of a name (print length in characters).
UserdictSize	200	The maximum number of elements in the user dictionary (i.e., **userdict maxlength**).
OperandStackSize	500	The maximum depth of the operand stack.
DictStackSize	20	The maximum depth of the dictionary stack.

ExecStackSize	250	The maximum depth of the execution stack.
MaxExecLevel	10	maximum number of recursive calls of the POSTSCRIPT interpreter. Certain of the graphics operators that call out to POSTSCRIPT procedures use recursive calls (e.g., **pathforall**).
MaxSaveLevel	15	The maximum number of active **saves**.
MaxGSaves	32	The maximum number of active **gsaves** *plus* the number of active **saves** may not exceed 32.
MaxPathElements	15000	The maximum number of points specified in *all* active path descriptions (this includes those nested by **gsaves**).
MaxDash	11	The maximum size of a dash array specification (as given to **setdash**).

Some installations might want to have these values accessible from within POSTSCRIPT. The following example defines a dictionary containing the values detailed above and places in the the user dictionary under the name "LimitDict."

```
userdict /LimitDict
20 dict dup begin
   /MaxInteger 16#7FFFFFFF def
   /MinInteger 16#80000000 def
   /MaxReal 1.0e38 def
   /MinReal -1.0e38 def
   /MaxArrayLength 65535 def
   /MaxStringLength 65535 def
   /MaxDictLength 2000 def
   /MaxNumberDicts 65535 def
   /MaxStreams 6 def
   /MaxNameSize 128 def
   /UserdictSize userdict maxlength def
   /OperandStackSize 500 def
   /DictStackSize 20 def
   /ExecStackSize 250 def
   /MaxExecLevel 10 def
   /MaxSaveLevel 15 def
   /MaxGSaves 32 def
   /MaxPathElements 15000 def
   /MaxDash 11 def
end put
```

REVISION HISTORY

This section is a short list of the major changes that have been made to the POSTSCRIPT language.

The following changes have taken place since the edition of August 1984:

- This section (the Revision History) has been added.
- **imagemask** (3.5.5) and **setcharwidth** (3.5.7) are new operators which increase the flexibility of the character caching machinery.
- The limit on the total number of names in POSTSCRIPT (**MaxNames**) has been removed, and the limit on the maximum length of a name (**MaxNameSize**) has substantially increased (Appendix B). The associated error operator **nametableoverflow** has been removed (3.6).
- **Encoding** is now required in a font dictionary, as cache entries are made based on *name* rather than character code (Appendix A).
- The character cache management operator **cachestatus** has been changed (the "limit" parameters have been removed). The **trimcache** and **setcachelimits** operators have been removed, and **setcachelimit** has been added (Section 3.5.7).

OPERATOR INDEX

name	*type*	*page*
==	miscellaneous	97
=	miscellaneous	96
abs	arithmetic	49
add	arithmetic	49
aload	array	63
anchorsearch	string	69
and	boolean/relational	75
arc	path construction	119
arcn	path construction	119
arcto	path construction	120
array	array	62
ashow	font	115
astore	array	63
atan	arithmetic	53
awidthshow	font	115
begin	dictionary	64
bitshift	boolean/relational	76
bytesavailable	input/output	91
cachestatus	character cache	145
ceiling	arithmetic	52
charpath	path construction	123
clear	stack manipulation	46
cleartomark	stack manipulation	47
clip	graphics output	128
clippath	path construction	123
closefile	input/output	88
closepath	path construction	121
concat	coordinates and matrix	109
concatmatrix	coordinates and matrix	109
copy	polymorphic	56
copypage	graphics output	127
cos	arithmetic	53
count	stack manipulation	47
countdictstack	dictionary	68
countexecstack	control	81
counttomark	stack manipulation	48
currentdash	graphics output	136
currentdict	dictionary	67
currentfile	input/output	92
currentflat	graphics output	136
currentfont	font	112
currentgray	graphics output	138
currenthsbcolor	graphics output	138
currentlinecap	graphics output	133
currentlinejoin	graphics output	134
currentlinewidth	graphics output	132
currentmatrix	coordinates and matrix	106
currentmiterlimit	graphics output	137
currentpoint	path construction	118
currentrgbcolor	graphics output	139

currentscreen	graphics output	140
currenttransfer	graphics output	141
curveto	path construction	121
cvi	type/conversion/property	83
cvlit	type/conversion/property	83
cvn	type/conversion/property	83
cvr	type/conversion/property	83
cvrs	type/conversion/property	84
cvs	type/conversion/property	84
cvx	type/conversion/property	84
def	dictionary	65
defaultmatrix	coordinates and matrix	106
definefont	font	113
dict	dictionary	64
dictfull	error	147
dictstack	dictionary	68
dictstackoverflow	error	148
dictstackunderflow	error	148
div	arithmetic	50
dtransform	coordinates and matrix	110
dup	stack manipulation	45
echo	input/output	93
end	dictionary	65
eoclip	graphics output	128
eofill	graphics output	129
eq	boolean/relational	72
erasepage	graphics output	126
exch	stack manipulation	45
exec	control	77
execstack	control	81
execstackoverflow	error	148
executeonly	type/conversion/property	85
exit	control	80
exp	arithmetic	51
false	boolean/relational	74
file	input/output	88
fill	graphics output	128
findfont	font	113
flattenpath	path construction	122
floor	arithmetic	52
flush	input/output	91
flushfile	input/output	92
for	control	79
forall	polymorphic	58
framedevice	device setup	143
ge	boolean/relational	73
get	polymorphic	59
getinterval	polymorphic	61
grestore	graphics state	101
grestoreall	graphics state	102
gsave	graphics state	101
gt	boolean/relational	73

identmatrix	coordinates and matrix	106
idiv	arithmetic	50
idtransform	coordinates and matrix	111
if	control	78
ifelse	control	78
image	graphics output	130
image	graphics output	131
imagemask	graphics output	132
index	stack manipulation	46
initclip	graphics output	127
initgraphics	graphics output	126
initmatrix	coordinates and matrix	106
interrupt	error	148
invalidaccess	error	148
invalidexit	error	149
invalidfileaccess	error	149
invalidfont	error	149
invalidrestore	error	149
invertmatrix	coordinates and matrix	111
ioerror	error	149
itransform	coordinates and matrix	110
known	dictionary	66
kshow	font	116
le	boolean/relational	73
length	polymorphic	57
limitcheck	error	149
lineto	path construction	118
ln	arithmetic	54
load	dictionary	66
log	arithmetic	54
loop	control	79
lt	boolean/relational	73
makefont	font	114
mark	stack manipulation	47
matrix	coordinates and matrix	106
maxlength	dictionary	67
mod	arithmetic	50
moveto	path construction	118
mul	arithmetic	50
ne	boolean/relational	72
neg	arithmetic	51
newpath	path construction	117
nocurrentpoint	error	149
not	boolean/relational	74
null	array	63
nulldevice	device setup	142
or	boolean/relational	75
pathbbox	path construction	122
pathforall	path construction	124
pop	stack manipulation	45
print	input/output	92
prompt	input/output	93

pstack	miscellaneous	97
put	polymorphic	60
putinterval	polymorphic	61
quit	control	81
rand	arithmetic	54
rangecheck	error	150
rcheck	type/conversion/property	86
rcurveto	path construction	121
read	input/output	89
readhexstring	input/output	89
readline	input/output	89
readonly	type/conversion/property	85
readstring	input/output	90
repeat	control	78
restore	input/output	95
reversepath	path construction	122
rlineto	path construction	119
rmoveto	path construction	118
roll	stack manipulation	46
rotate	coordinates and matrix	108
round	arithmetic	52
rrand	arithmetic	55
run	input/output	92
save	input/output	95
scale	coordinates and matrix	108
scalefont	font	113
search	string	70
setcachedevice	character cache	145
setcachelimit	character cache	146
setcharwidth	character cache	146
setdash	graphics output	135
setflat	graphics output	136
setfont	font	114
setgray	graphics output	137
sethsbcolor	graphics output	138
setlinecap	graphics output	133
setlinejoin	graphics output	134
setlinewidth	graphics output	132
setmatrix	coordinates and matrix	107
setmiterlimit	graphics output	137
setrgbcolor	graphics output	139
setscreen	graphics output	140
settransfer	graphics output	141
show	font	115
showpage	graphics output	127
sin	arithmetic	54
sqrt	arithmetic	51
srand	arithmetic	55
stack	miscellaneous	96
stackoverflow	error	150
stackunderflow	error	150
start	control	81

status	input/output	92
stop	control	80
stopped	control	80
store	dictionary	65
string	string	69
stringwidth	font	116
stroke	graphics output	129
strokepath	path construction	123
sub	arithmetic	51
syntaxerror	error	150
systemdict	dictionary	67
token	input/output	90
token	string	71
transform	coordinates and matrix	109
translate	coordinates and matrix	107
true	boolean/relational	74
truncate	arithmetic	53
type	type/conversion/property	82
typecheck	error	150
undefined	error	150
undefinedfilename	error	151
undefinedresult	error	151
unmatchedmark	error	151
unregistered	error	151
userdict	dictionary	67
usertime	miscellaneous	96
version	miscellaneous	96
VMerror	error	151
vmstatus	input/output	95
wcheck	type/conversion/property	86
where	dictionary	66
widthshow	font	115
write	input/output	91
writehexstring	input/output	91
writestring	input/output	91
xcheck	type/conversion/property	85
xor	boolean/relational	76
[	array	62
]	array	62

INDEX

radix notation 13

% comment character 14
%lineedit 87
%statementedit 87
%stderr 88
%stdin 88
%stdout 88

/ slash character 15

CTM 98, 103, 105
PostScript Objects 34
PostScript Types 34

\ backslash character 14
\ escape character 14

Argument handling 40
Array objects 35, 62
Arrays
  as vectors 35
  executable 35, 39
  representation of 38
  syntax of 15
ASCII character set 2, 37

Bezier cubics 120
Base conversion 84
Base notation, # 13
Boolean objects 35, 72
Bounds checking 40

Character encoding 155
Character encoding, EBCDIC 157
Character metrics 161
Character set 2
Character widths 161
Characters
  {braces} 41
Clipping 10, 125
  boundary 10, 98, 125, 127, 128
  operators 127, 128
  path 10, 98, 125, 127, 128
Color, in graphics state 138, 139
Comments, syntax of 14
Composite objects 38
Coordinate systems
  character 154
  default 11
  font 154
  origin 11
  scaling 11
Cubic splines 120
Current dictionary 36, 67
Current position 98
Current transformation 98, 103, 105

Dash array 99, 135, 136
Dictionary
  current 36, 67
  definition of 18
  font 112, 154
  objects 36, 64
  operators 64
  representation of 38
  stack 19, 34, 36, 64, 68
  system 19, 36, 67
  user 36, 67

EBCDIC character encoding 157
Encoding vector 155
Error handling 40
Escape character, \ 14
Example
  geometric path 23
  multiple fonts 25
  simple text 24
Executable definitions 18
Executable, checking for 85
Executable, conversion to 84
Execution 41
Execution stack 34, 81

FID 154
File objects 37, 87
File operators 87
Files
  %lineedit 87
  %statementedit 87
  %stderr 88
  %stdin 88
  %stdout 88
  special 87
Flatness, in graphics state 99, 136
Flatness, of Bezier curves 99
Font dictionary 112, 154
Font ID 154
Font objects 38
Fonts
  built-in 154
  changing 26
  changing size of 25
  character metrics 161
  coordinate system 154
  encoding vector 155
  fixed pitch 26
  in graphics state 99
  introduction 24
  metric information 161
  monospaced 26
  point size of 25
  representation of 24
  user defined 154
  user modifications 157
  variable pitch 26
  width information 26

Geometric shapes 22
Graphics operators 21, 98
Graphics State 21, 34, 98, 101
Graphics state stack 34, 101

Hexadecimal constant 13

Imaging model 9

Input/output operators 87
Integer objects 35
Integer, conversion to 83
Interpreter
  basic operation 16

Line cap 99, 133
Line join 99, 134
Line width 99, 132
Literal, checking for 85
Literal, conversion to 83
Logical values 35, 72

Mark objects 38, 47, 62
Matrix, current transformation 98, 103, 105
Miter limit 99, 137

Name binding 18
Name objects 37
Name, conversion to 83
Names
  binding of 18
  literal 15
  syntax of 15
Null objects 38, 62, 63
Numbers
  radix notation 13
  syntax of 13
  type conversion 35, 49, 82
  type of 35

Objects
  array 35, 56, 62
  body part 38
  boolean 35, 72
  composite 38
  determining type of 82
  dictionary 36, 56, 64
  file 37, 87
  font 38
  integer 35
  mark 38, 47, 62
  name 37
  null 38, 62, 63
  operator 37
  primary part 38
  real 35
  save 38, 94
  string 36, 56, 69
Octal constant 13
Operand stack 16, 34
Operator
  = 96
  == 97
  [62
  ] 62
  abs 49
  add 49
  aload 63
  anchorsearch 69
  and 75
  arc 119
  arcn 119
  arcto 120
  array 62
  ashow 115
  astore 63
  atan 53
  awidthshow 115
  begin 64
  bitshift 76
  bytesavailable 91
  cachestatus 145
  ceiling 52
  charpath 123
  clear 46
  cleartomark 47
  clip 128
  clippath 123
  closefile 88
  closepath 23, 121
  concat 109
  concatmatrix 109
  copy 56
  copypage 127
  cos 53
  count 47
  countdictstack 68
  countexecstack 81
  counttomark 48
  currentdash 136
  currentdict 67
  currentfile 92
  currentflat 136
  currentfont 112
  currentgray 138
  currenthsbcolor 138
  currentlinecap 133
  currentlinejoin 134
  currentlinewidth 132
  currentmatrix 106
  currentmiterlimit 137
  currentpoint 118
  currentrgbcolor 139
  currentscreen 140
  currenttransfer 141
  curveto 121
  cvi 83
  cvlit 83
  cvn 83
  cvr 83
  cvrs 84
  cvs 84
  cvx 84
  def 17, 65
  defaultmatrix 106
  definefont 113
  dict 64
  dictfull 147
  dictstack 68
  dictstackoverflow 148
  dictstackunderflow 148
  div 50
  dtransform 110
  dup 45
  echo 93
  end 65
  eoclip 128

eofill 129
eq 72
erasepage 126
exch 45
exec 77
execstack 81
execstackoverflow 148
executeonly 85
exit 80
exp 51
false 74
file 88
fill 128
findfont 25, 112, 113
flattenpath 122
floor 52
flush 91
flushfile 92
for 20, 79
forall 58
framedevice 143
ge 73
get 59
getinterval 61
grestore 24, 101
grestoreall 102
gsave 24, 101
gt 73
identmatrix 106
idiv 50
idtransform 111
if 20, 78
ifelse 20, 78
image 130, 131, 145
imagemask 132, 145
index 46
initclip 127
initgraphics 126
initmatrix 106
interrupt 148
invalidaccess 148
invalidexit 149
invalidfileaccess 149
invalidfont 149
invalidrestore 149
invertmatrix 111
ioerror 149
itransform 110
known 66
kshow 116
le 73
length 57
limitcheck 149
lineto 23, 118
ln 54
load 66
log 54
loop 20, 79
lt 73
makefont 114
mark 47
matrix 106
maxlength 67
mod 50
moveto 23, 24, 118
mul 50
ne 72
neg 51
newpath 117
nocurrentpoint 149
not 74
null 63
nulldevice 142
or 75
pathbbox 122
pathforall 124
pop 45
print 92
prompt 93
pstack 97
put 60
putinterval 61
quit 81
rand 54
rangecheck 150
rcheck 86
rcurveto 121
read 89
readhexstring 89
readline 89
readonly 85
readstring 90
repeat 19, 78
restore 95
reversepath 122
rlineto 119
rmoveto 118
roll 46
rotate 108
round 52
rrand 55
run 92
save 95
scale 108
scalefont 25, 113
search 70
setcachedevice 145
setcachelimit 146
setcharwidth 146
setdash 135
setflat 136
setfont 114
setgray 137, 145
sethsbcolor 138, 145
setlinecap 133
setlinejoin 134
setlinewidth 132
setmatrix 107
setmiterlimit 137
setrgbcolor 139, 145
setscreen 140
settransfer 141, 145
show 24, 115
showpage 127
sin 54
sqrt 51
srand 55
stack 96

stackoverflow 150
stackunderflow 150
start 81
status 92
stop 80
stopped 80
store 65
string 69
stringwidth 116
stroke 129
strokepath 123
sub 51
syntaxerror 150
systemdict 67
token 71, 90
transform 109
translate 107
true 74
truncate 53
type 82
typecheck 150
undefined 150
undefinedfilename 151
undefinedresult 151
unmatchedmark 151
unregistered 151
userdict 67
usertime 96
version 96
VMerror 151
vmstatus 95
wcheck 86
where 66
widthshow 115
write 91
writehexstring 91
writestring 91
xcheck 85
xor 76
Operators 37
arithmetic 49
array 62
assignment 17
bitwise 72
boolean 72
character cache 144
conditional 77
control 19, 77
coordinates and matrix 103
device setup 142
dictionary 64
error 147
file 87
font 112
graphics 21, 98
graphics output 125
graphics state 101
input/output 87
looping 77
math 49
miscellaneous 96
path construction 117
polymorphic 56
property 82
relational 72
stack manipulation 45
string 69
type 82
type conversion 82
virtual memory 94

Parametric cubics 120
Path, current 98
Paths
Bezier curves in 23
circular arcs in 23
creating a path 22
geometric 22
in graphics state 98
introduction to 22
line segments in 22
Primary part, of objects 38
Procedure bodies
syntax of 15
Procedures
defining 18
Prologue, of POSTSCRIPT source 2

Radix conversion 84
Radix notation, # 13
Real objects 35
Real, conversion to 83

Save objects 38, 94
Scope of names 19
Screen, in graphics state 99, 140
Script, of POSTSCRIPT source 2
Shapes, geometric 22
Side bearings 161
Stack
dictionary 34, 64, 68
execution 34, 81
graphics state 34, 101
operand 16, 34, 41
operators 45
representation of 38
Standard error file 88
Standard input file 88
Standard output file 88
String objects 36, 69
String, conversion to 84
Strings
hexadecimal 14
representation of 38
syntax of 13
Syntax 13
System dictionary 36, 67

Transfer function, in graphics state 99, 141
Transformation matrix, in graphics state 98
Transformation, current 98, 103, 105
Transformations
introduction 23
Type
array 35, 56, 62
boolean 35, 72
dictionary 36, 56, 64
file 37, 87

font 38
integer 35, 49
mark 38, 47, 62
name 37
null 38, 62, 63
of PostScript objects 34
operator 37
real 35, 49
save 38, 94
string 36, 56, 69
Type checking 40
Type conversion 35, 82
arithmetic 49
automatic 35, 49
explicit 82
Type determination 82

User coordinates 103
User dictionary 36, 67
User space 103

Vectors 35
Virtual memory 94
VM 94

Colophon

The colophon of a book is traditionally a small design device placed on the last page of a book or manuscript. There is usually some inscription of the scribe or printer listing the date, place, and details of publication.

The word colophon is from the Greek word "Kolophon" (κολοφων), meaning summit or final touch. Or perhaps, colophon is from the Greek word "Kolophos" (κολοφως), which was the name of the very last island in the Greek chain of islands; hence the last page was called the colophon.

This manual was written and edited at Adobe Systems Incorporated. It was produced from a POSTSCRIPT print file which was created using a customized version of the Scribe® Document Production System software, which is marketed by UNILOGIC, Ltd. Camera-ready copy for this manual was printed entirely on a POSTSCRIPT printer at Adobe Systems. The typefaces used in this manual were digitized by Adobe Systems, Inc. The body type is *Times*™, and the fixed-pitch font used in operator definitions and examples is *Courier*.

POSTSCRIPT™ Cheat Sheet

27 September 1984
A quick reference guide to POSTSCRIPT operators.
Copyright © 1984 Adobe Systems, Inc.
POSTSCRIPT is a trademark of Adobe Systems, Inc.

Stack Operators

any	**pop**	—	pop top element off operand stack
any	**dup**	any any	duplicate top element on operand stack
any_1 any_2	**exch**	any_2 any_1	exchange top two elements
$a_{n-1} \cdots a_0$ n j	**roll**	$a_{(j-1)(\mathrm{mod}\ n)} \cdots a_0\ a_{n-1} \cdots a_{j(\mathrm{mod}\ n)}$	roll *n* elements *j* times (+ = 'right')
$a_N \cdots a_0$ ind	**index**	$a_N \cdots a_0\ a_{ind}$	index into operand stack (top = 0)
$\vdash any_1..any_N$	**clear**	⊢	clear the operand stack
$\vdash any_1 \cdots any_N$	**count**	N	count elements on operand stack
—	**mark**	mark	push mark onto operand stack
mark ...	**cleartomark**	—	clear operand stack down through *mark*
... mark ...	**counttomark**	... mark ... n	count stack entries from top to mark

Arithmetic and Math Operators

num	**abs**	|num|	absolute value of *num*
n m	**add**	n+m	add two numbers
n m	**div**	n/m	divide two numbers
i j	**idiv**	integer-part(i/j)	integer divide
i j	**mod**	(i MOD j)	modulus (integer remainder of *i/j*)
n m	**mul**	n*m	multiply two numbers
n	**neg**	-n	change sign of *n*
n m	**sub**	n-m	subtract two numbers
n	**sqrt**	Sqrt(n)	square root
n m	**exp**	n^m	raise *n* to *m*th power
x	**ceiling**	Ceiling(x)	Ceiling of *x*
x	**floor**	Floor(x)	Floor of *x*
n	**round**	Round(n)	round *n* to nearest integer
x	**truncate**	truncate(x)	Truncate *x*
y x	**atan**	ArcTan(y/x)	ArcTangent(y/x in degrees)
angle	**cos**	Cos(angle)	Cosine(*angle* in degrees)
angle	**sin**	Sin(angle)	Sine(*angle* in degrees)
n	**ln**	Ln(n)	natural logarithm (base e)
n	**log**	log(n)	logarithm (base 10)
—	**rand**	int	generate pseudo-random number
int	**srand**	—	set random number seed
—	**rrand**	int	return random number seed

Polymorphic Operators

any_1 .. any_N N	**copy**	any_1..any_N	copy top *N* elements of stack
obj_1 obj_2	**copy**	obj_2	copy complex object
dict|string|array	**length**	n	length of argument
array|dict|string proc	**forall**	—	for each element do *proc*
array|dict|string int|key	**get**	value	get *value* of *int|key* in object
array|dict|string index|key val	**put**	—	put *val* into object
array|string ind count	**getinterval**	subobj	subinterval of *array|string* starting at *ind* for *count* elements
obj_1 ind obj_2	**putinterval**	—	store all of obj_2 into obj_1 starting at *ind*

Array Operators

n	**array**	array	create array of size *n*
—	**[**	mark	start array construction
mark ~$mark_0$..~$mark_{N-1}$	**]**	array-size-N	end array construction
array	**aload**	a_0...a_{N-1} array	get all elements of array
any_0...any_{N-1} array-size-N	**astore**	array-size-N	put elements from stack into array
—	**null**	null	return a null object

Dictionary Operators

int	**dict**	dict	create dictionary with capacity for *int* elements
dict	**begin**	—	push *dict* on dict stack
—	**end**	—	pop dict stack
key value	**def**	—	associate *value* with *key* in top dict
key val	**store**	—	define in topmost dict on stack containing *key* else use def
dict key	**known**	bool	test if *key* in *dict*
key	**load**	val	load *val* of *key* from dict stack (no exec)
key	**where**	[dict true] *or* false	search dict stack for *key*
dict	**maxlength**	int	get capacity of *dict*
—	**systemdict**	dict	put system dict on operand stack
—	**userdict**	dict	put the user dict on operand stack
—	**currentdict**	dict	copy top dict to operand stack
—	**countdictstack**	num	number of dicts on dict stack
array	**dictstack**	subarray	copy dict stack into *subarray*

String Operators

n	**string**	string	create string of length *n*
str patt	**anchorsearch**	[post match true] *or* [str false]	search at front of *str* for *patt*
str patt	**search**	[post match pre true] *or* [str false]	search for *patt* in *str*

str	**token**	[post token true] *or* false	strip token from start of *str*

Relational, Boolean, and Bitwise Operators

any_1 any_2	**eq**	bool	test equality
any_1 any_2	**ne**	bool	test not equal
n|string n|string	**ge**	bool	test greater or equal
n|string n|string	**gt**	bool	test greater than
n|string n|string	**le**	bool	test less than or equal
n|string n|string	**lt**	bool	test less than
—	**true**	true	push boolean value *true*
—	**false**	false	push boolean value *false*
bool|int	**not**	NOT(bool|int)	local|bitwise NOT
$bool|int_1$ $bool|int_2$	**and**	($bool|int_1$ & $bool|int_2$)	logical|bitwise AND
$bool|int_1$ $bool|int_2$	**or**	($bool|int_1$ OR $bool|int_2$)	logical|bitwise inclusive OR
$bool|int_1$ $bool|int_2$	**xor**	($bool|int_1$ XOR $bool|int_2$)	logical|bitwise exclusive OR
int shift	**bitshift**	bitshift(int,shift)	logical shift(+ = left)of *int*

Control Operators

any	**exec**	—	(execute) move to exec stack
bool proc	**if**	—	if *bool* then *proc*
bool $proc_T$ $proc_F$	**ifelse**	—	if *bool* then $proc_T$ else $proc_F$
n proc	**repeat**	—	execute *proc n* times
j k l proc	**for**	—	for i=*j* step *k* until *l* do *proc*
proc	**loop**	—	execute *proc* forever
—	**exit**	—	exit innermost active loop
—	**stop**	—	unwind exec stack to stopped
any	**stopped**	boolean	catch execution of **stop** in *any*
—	**countexecstack**	n	number of elements on exec stack
array	**execstack**	subarray	copy exec stack into array
—	**quit**	—	exit to system
—	**start**	—	executed at system startup

Type, Conversion, and Property Operators

any	**type**	name	return type of operand
num|string	**cvi**	int	convert to integer
any	**cvlit**	literal(any)	turn off executable flag
string	**cvn**	name	convert string to name
num|string	**cvr**	real	convert to real
n radix str	**cvrs**	sstr	convert to string with radix
any str	**cvs**	sstr	convert to string
any	**cvx**	executable(any)	turn on executable flag
array|string	**executeonly**	ExecuteOnly(array|string)	protect top-level elements
dict|array|string	**readonly**	ReadOnly(dict|array|string)	protect top-level elements

any	**xcheck**	bool	check executable flag
dict|array|string	**rcheck**	boolean	check if readable
array|dict|string	**wcheck**	bool	check if writeable

File Operators

filename access	**file**	file	open file with access (rwa+)
file	**closefile**	—	close file stream
file	**read**	[byte true] or false	read *byte* from *file*
file string	**readhexstring**	sstr boolean	read hexadecimal into *string* from *file*
file string	**readline**	sstr boolean	read line from *file*
file string	**readstring**	sstr boolean	read string from *file*
file	**token**	[token true] or false	strip *token* from stream
file	**bytesavailable**	num	return number of bytes available for read
file byte	**write**	—	write *byte* to *file*
file string	**writehexstring**	—	write *string* to *file* in hex
file string	**writestring**	—	write *string* to *file*
—	**flush**	—	flush the standard output stream
file	**flushfile**	—	send buffered output immediately or read to EOF
file	**status**	boolean	return status of stream
filename	**run**	(depends on file)	execute contents of *filename*
—	**currentfile**	file	return file of current execution stream
string	**print**	—	print string on primary output (see show)
—	**prompt**	—	executed when ready for new input: (PS>)print
bool	**echo**	—	turn on/off echoing

Virtual Memory Operators

—	**save**	saveobj	create system state snapshot
saveobj	**restore**	—	restore system-state to snapshot
—	**vmstatus**	level used total	report vm status

Miscellaneous Operators and Functions

—	**version**	string	PS version identifier
—	**usertime**	msec	return time in milliseconds
any	**=**	—	destructively print top of stack with cvs
$\|-\ any_1 \ldots any_N$	**stack**	$\|-\ any_1 \ldots any_N$	print stack using = (nondestructive)
any	**==**	—	destructively print top element
$\|-\ any_1 \ldots any_N$	**pstack**	$\|-\ any_1 \ldots any_N$	print stack using == (nondestructive)

Graphics State Operators

—	**gsave**	—	save graphics state for matching grestore
—	**grestore**	—	restore graphics state from matching gsave
—	**grestoreall**	—	restore to bottom-most graphics state

Coordinate System and Matrix Operators

—	**matrix**	matrix	create identity matrix
—	**initmatrix**	—	set transform matrix to device default
matrix	**identmatrix**	matrix	fill *matrix* with identity transform [1 0 0 1 0 0]
matrix	**defaultmatrix**	matrix	fill in *matrix* with device default transform
matrix	**currentmatrix**	matrix	fill in *matrix* with current transform
matrix	**setmatrix**	—	set current transformation matrix to be *matrix*
tx ty	**translate**	—	move user origin to *(tx,ty)* in current units
tx ty matrix	**translate**	matrix	fill in *matrix* with values that translate by *(tx,ty)*
sx sy	**scale**	—	scale user coords by *sx* in x and *sy* in y
sx sy matrix	**scale**	matrix	fill in *matrix* to scale by *sx,sy*
ang	**rotate**	—	rotate user space about origin by *ang* (degrees, positive = counterclockwise
ang matrix	**rotate**	matrix	fill in *matrix* to rotate by *ang*
matrix	**concat**	—	set current transform to *matrix**currentmatrix
m_1 m_2 m_3	**concatmatrix**	m_3	fill in m_3 with m_1*m_2
x y	**transform**	xt yt	transform *(x, y)* by current transformation
x y matrix	**transform**	xt yt	explicit transform of *(x,y)* by *matrix*
xd yd	**dtransform**	xdt ydt	(delta transform) like transform but no translation
xd yd matrix	**dtransform**	xdt ydt	explicit delta transform
xt yt	**itransform**	x y	inverse transform
xt yt matrix	**itransform**	x y	explicit inverse transform
xdt ydt	**idtransform**	xd yd	inverse delta transform
xdt ydt matrix	**idtransform**	x y	explicit inverse delta transform
m_1 m_2	**invertmatrix**	m_2	fill in m_2 with inverse of m_1

Character and Font Operators

—	**currentfont**	dict	return dict for current font
key dict	**definefont**	f-dict	register *dict* as a font dictionary
key	**findfont**	dict	return dict for font with given name

fdict scale	**scalefont**	tdict	return new scaled font dict
fdict matrix	**makefont**	tdict	return new font dict with transformed matrix
fdict	**setfont**	—	set current font
string	**show**	—	output *string* in current graphics context
nx ny cc string	**widthshow**	—	add *(nx ny)* to width of char *cc* when showing *string*
ax ay string	**ashow**	—	add *(ax ay)* to width of each char when showing *string*
nx ny cc ax ay string	**awidthshow**	—	combined effects of ashow and widthshow
proc string	**kshow**	—	execute *proc* between characters shown from *string*
string	**stringwidth**	wx wy	width (user space) of *string* in current font

Path Construction Operators

—	**newpath**	—	initialize current path to be empty
—	**currentpoint**	x y	return current point in user coordinates
x y	**moveto**	—	set current point to *(x,y)*
dx dy	**rmoveto**	—	relative moveto (currentpoint + *(dx, dy)*)
x y	**lineto**	—	continue path with straight line to *(x, y)*
dx dy	**rlineto**	—	relative lineto
x y r ang_1 ang_2	**arc**	—	add counterclockwise arc to current path
x y r ang_1 ang_2	**arcn**	—	add clockwise arc to current path
x_1 y_1 x_2 y_2 r	**arcto**	xt_1 yt_1 xt_2 yt_2	build tangent arc
x_0 y_0 x_1 y_1 x_2 y_2	**curveto**	—	add Bezier cubic section to current path
dx_0 dy_0 dx_1 dy_1 dx_2 dy_2	**rcurveto**	—	relative curveto
—	**closepath**	—	closes current path with a straight line to last moveto point
—	**pathbbox**	llx lly urx ury	return bounding-box of current path
—	**flattenpath**	—	make current path a polygon
—	**reversepath**	—	reverse direction of current path
—	**strokepath**	—	make current path a fillable object (as if stroked)
string bool	**charpath**	—	add character outline(s) to current path
—	**clippath**	—	sets current path to clipping outline
mtproc ltproc ctproc cpproc	**pathforall**	—	enumerate current path

Graphics Output Operators

—	**initgraphics**	—	reset graphics parameters
—	**erasepage**	—	clear current output page
—	**showpage**	—	output current page; erasepage initgraphics
—	**copypage**	—	output current page
—	**initclip**	—	set clipping path to device default
—	**clip**	—	shrink current clipping boundary to its intersection with current path
—	**eoclip**	—	clip with even-odd inside rule
—	**fill**	—	fill the current path with the current color
—	**eofill**	—	fill with even-odd rule
—	**stroke**	—	stroke the current path with the current color/linejoin/linecap/linewidth/dash
scanlen #lines b/p mtx proc	**image**	—	render the image returned by *proc* onto the current page
scanlen #lines bool mtx proc	**imagemask**	—	render the image returned by *proc* onto the current page
num	**setlinewidth**	—	set the current line width
—	**currentlinewidth**	num	return the current line width
0|1|2	**setlinecap**	—	set the shape of line ends for stroke (butt|round|square)
—	**currentlinecap**	0|1|2	return current line cap
0|1|2	**setlinejoin**	—	set the current line join for stroke (miter|round|bevel)
—	**currentlinejoin**	0|1|2	return the current line join
array offset	**setdash**	—	set the current dash array
—	**currentdash**	array offset	return the current dash array
num	**setflat**	—	set the current flat tolerance
—	**currentflat**	num	return the current flat tolerance
num	**setmiterlimit**	—	set the current maximum miter ratio
—	**currentmiterlimit**	num	return the current maximum miter ratio
num	**setgray**	—	set the current color to a gray value (0=black, 1=white)
—	**currentgray**	num	return the current gray
h s b	**sethsbcolor**	—	set current color given hue, saturation, brightness
—	**currenthsbcolor**	h s b	return current color hue, saturation, brightness
r g b	**setrgbcolor**	—	set current color given red, green, blue
—	**currentrgbcolor**	r g b	return current color red, green, blue
freq ang spot	**setscreen**	—	set halftone screen
—	**currentscreen**	freq ang spot	current halftone screen

xfer-func	**settransfer**	—	set gray transfer function
—	**currenttransfer**	xfer-func	current gray transfer function

Device Setup Operators

—	**nulldevice**	—	install device that does no output
mtx w h proc	**framedevice**	—	install framebuffer device

Character Cache Operators

—	**cachestatus**	bs bm ms mm cs cm maxbits	return size and max for bitmaps mids and chars
wx wy llx lly urx ury	**setcachedevice**	—	install character cache
wx wy	**setcharwidth**	—	inform character cache
maxbits	**setcachelimit**	—	set maxbitmap limit in cache

Error Operators

dictfull	no more room in dictionary
dictstackoverflow	too many begins
dictstackunderflow	too many ends
execstackoverflow	exec nesting too deep
interrupt	executed when ^C typed to server (stop)
invalidaccess	attempt to store into readonly object
invalidexit	exit not in loop
invalidfileaccess	bad access string
invalidfont	bad font name or dict
invalidrestore	improper restore
ioerror	system i/o error occurred
limitcheck	implementation limit exceeded
nocurrentpoint	path is empty
rangecheck	argument out of bounds
stackoverflow	operand stack overflow
stackunderflow	operand stack underflow
syntaxerror	input ended in string or proc body
typecheck	argument of wrong type
undefined	name not known
undefinedfilename	file not found
undefinedresult	number over/underflow
unmatchedmark	expected mark not on stack
unregistered	serious system error
VMerror	serious system error

Appendix B

The Postscript Cookbook

Cookbook Examples

The POSTSCRIPT Cookbook describes various procedures for producing text and graphics on the printed page, and includes example PostScript programs to illustrate these procedures. All these programs (including their comments) are included in the Cookbook Examples folder on the Inside LaserWriter diskette.

You can send any of these files to the LaserWriter by invoking the Downloading Program that you can find on the Programming and Debugging Aids diskette. For detailed instructions, see Appendix F, the section entitled " Instructions for spooling, editing and downloading a Postscript file from a Macintosh application."

You can also experiment with changing these programs to produce different results by editing them with a text editor. If you use MacWrite, be sure to invoke "Save as..." and select the "Text only" option when saving your files to disk.

POSTSCRIPT™ Cookbook

A Guide to Graphic Imaging

Adobe Systems Incorporated

Adobe Systems Incorporated
1870 Embarcadero Road, Suite 100
Palo Alto, California 94303

PostScript™ Cookbook
First Printing, Revised
7 January 1985
Copyright © 1985 by Adobe Systems, Inc.
All Rights Reserved.

PostScript is a trademark of Adobe Systems, Inc.

Times and Helvetica ® are trademarks of Allied Corporation.

The information in this document is furnished for informational use only, and is subject to change without notice and should not be construed as a commitment by Adobe Systems, Inc. Adobe Systems assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

Contents

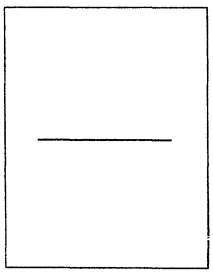
Simple Geometric Shapes	1
Units, Procedures, and the Operand Stack	4
Program 1: Three Squares	9
Program 2: Translated Squares	11
Program 3: Translate, Rotate and Scale	13
Using the arc Operator	14
Program 4: Drawing an Ellipse	17
Program 5: Repeated Lines	19
Program 6: Repeated Shapes	21
Program 7: Expanded and Constant Width Lines	23
Program 8: Drawing Arrows	25
Changing the Appearance of a Stroke	26
Program 9: Centered Dash Patterns	31
Fonts	34
Program 10: Simple Text	37
Program 11: Faces and Sizes	39
Program 12: White Text on a Black Background	41
The makefont Operator	42
Program 13: Condensed, Extended and Obliqued Text	45
Program 14: A Simple Line Breaking Algorithm	47
Program 15: Vertical Text	51
Program 16: Circular Text	53
Program 17: Placing Text Along an Arbitrary Path	57
Miscellaneous	60
Program 18: Drawing a Pie Chart	63
Program 19: Using the image Operator	67
Program 20: Bit Pattern Screens	69
Program 21: Making a Poster	73
Customized Fonts	76
Program 22: Making an Outlined Font	79
Program 23: Re-encoding an Entire Font	81
Program 24: Making Small Changes to Encoding Vectors	85
Program 25: Making a User Defined Font	89
For Further Reference	92
Index	93

Simple Geometric Shapes

This section will introduce some of the basic POSTSCRIPT operators that generate geometric shapes. Concepts covered in this section are *paths, path construction,* and *graphic output.*

In order to "draw" geometric shapes on a page, one must first construct a POSTSCRIPT *path.* A *path* is composed of straight and curved line segments. It can represent either the outline of an area to be filled or a trajectory along which lines can be drawn. POSTSCRIPT accumulates line segments and curves in a path called the *current path.* Later sections will introduce methods for remembering trajectories in paths other than the current path.

After a path has been constructed, it may be "drawn" on the page. (Merely constructing a path does not actually draw it on the output page.) To accomplish this one must use the graphic output operators.



PROBLEM 1: Draw a straight horizontal line which is 400 units long. (Units will be discussed later.)

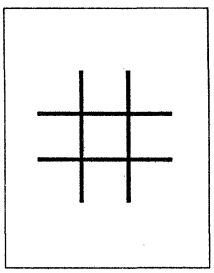
```
newpath
  100 390 moveto
  500 390 lineto
stroke
showpage
```

EXPLANATION: This example demonstrates the basic approach used when drawing geometric shapes with POSTSCRIPT: construct the path first, then "draw" it on the page. **newpath** should be the first operator used when constructing a path. **newpath** initializes the current path to be empty. Next, we use the **moveto** and **lineto** operators to describe the path. The general form of the **moveto** and **lineto** operators is:

<x> <y> **moveto**
<x> <y> **lineto**

moveto starts a new segment in the path; it causes the x,y coordinate of the point specified to be entered as the beginning of the new segment. The point specified with the **moveto** becomes the current point. The most recently entered point is known as the *current point. The first point in a path must always be entered with a moveto.* **lineto** adds a straight line segment from the current point to the x,y

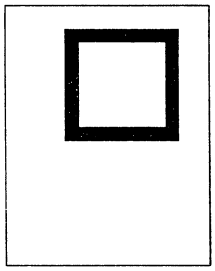
coordinate of the point specified with the **lineto**. **lineto** also redefines the current point to be its coordinate argument. Now we may draw the shape defined in the path onto the page. To do this we use the graphics output operator **stroke**. **stroke** paints a line that follows the trajectory specified in the current path. It also implicitly performs a **newpath** after the stroke is done. In other words, performing a **stroke** operation reinitializes the current path. Finally, we would like to see the page printed on the output device. This is accomplished by the **showpage** operator.



PROBLEM 2: Draw a tic-tac-toe board with lines that are 10 units wide and 400 units long.

```
newpath
  100 470 moveto  % top horizontal
  500 470 lineto  %   line
  100 330 moveto  % bottom horizontal
  500 330 lineto  %   line
  230 600 moveto  % left vertical
  230 200 lineto  %   line
  370 600 moveto  % right vertical
  370 200 lineto  %   line
10 setlinewidth
stroke
showpage
```

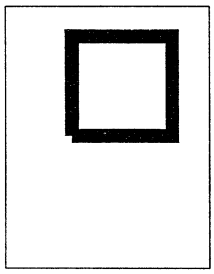
EXPLANATION: This example illustrates the flexibility of the path construct. First, paths need not be made up of continuous shapes. In this example there are four discontinuous shapes (lines). Second, paths may be self intersecting. In this example there are four intersections. This example also illustrates the ability to control the stroke width used when drawing the lines onto paper. This is accomplished by the **setlinewidth** operator. This example program also contains some comments. Comments in POSTSCRIPT begin with a percent sign (%) and are terminated with the newline character (i.e. the end of the line).



PROBLEM 3: Draw a square with sides that are 300 units long. Outline the square with a line that is 40 units wide.

```
newpath
  200 400 moveto  % lower left corner
  200 700 lineto  % left edge
  500 700 lineto  % top edge
  500 400 lineto  % right edge
closepath
40 setlinewidth
stroke
showpage
```

EXPLANATION: We have introduced a new path operator in this example: **closepath. closepath** draws a straight line segment from the current point to the coordinates of the point specified in the most recent **moveto.** (In the above example, **closepath** draws a straight line from (500, 400) to (200, 400)). **closepath** also causes POSTSCRIPT to treat the shape as a closed, continuous shape. The following example appears to have the same effect as the above program yet its graphic output is different:



```
newpath
  200 400 moveto  % lower left corner
  200 700 lineto  % left edge
  500 700 lineto  % top edge
  500 400 lineto  % right edge
  200 400 lineto  % bottom edge
40 setlinewidth
stroke
showpage
```

This example uses an explicit **lineto** instead of a **closepath.** The path is not a closed shape, and as a result, stroking the line leaves a notch in the corner where the path begins and ends. *As a general rule, use* **closepath** *on shapes that should be closed.*

Units, Procedures and the Operand Stack

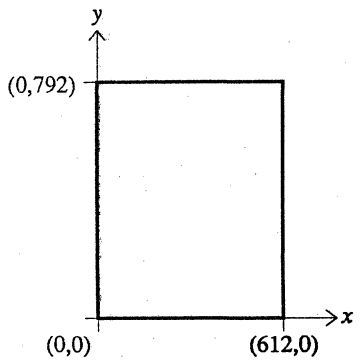


Figure 1

In the previous examples we have been using POSTSCRIPT's *default coordinate system.* The default coordinate system is a Cartesian coordinate system which means that all locations are specified by an (x,y) coordinate pair. The default coordinate system is superimposed on the printed page. The origin is located at the lower left corner of the page with the positive x-axis extending horizontally to the right and the positive y-axis extending vertically upwards.

Each integral unit in the default coordinate system is equal to a printer's measurement called a *point.* There are roughly 72 points to an inch. (No one has been able to agree if there are 72 points per inch or 72.27 points per inch). POSTSCRIPT has adopted the 72 points per inch convention.

Figure 1 shows the 8-1/2 inch by 11 inch output page and how the default coordinate system is superimposed on the page. The extreme corners of the page are marked with their coordinates in points.

Depending upon the application, it may be more convenient to work in units other than points, such as inches or centimeters. This can be done very easily in POSTSCRIPT by defining a procedure to convert from one system of measurement to another. The following procedure definitions will allow us to specify our measurements in inches or centimeters:

```
/inch {72 mul} def
/cm {28.3465 mul} def
```

This is our first example of a procedure definition. The general syntax for a procedure is

/<name> <procedure body> def

A procedure body is a series of POSTSCRIPT operators and operands enclosed in curly braces.

Once the procedure "inch" has been defined, we may use it in a POSTSCRIPT program. The following example illustrates the use of inch:

```
/inch {72 mul} def
1 inch 10 inch moveto
```

EXPLANATION: In order to understand how the procedure "inch" actually works, it is important to understand how the

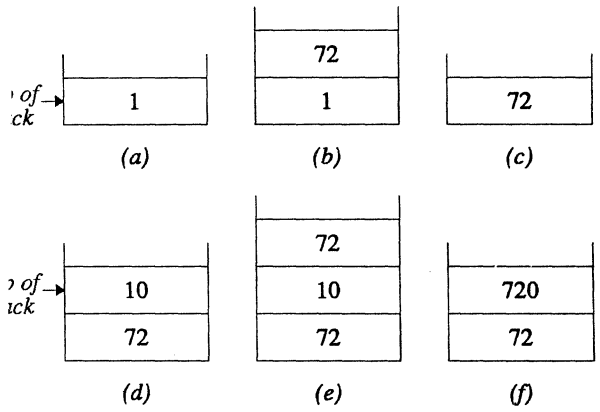
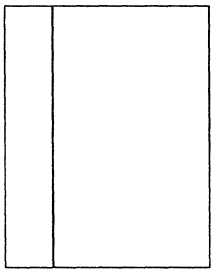


Figure 2: The Operand Stack

POSTSCRIPT operand stack works. The *operand stack* is one of POSTSCRIPT's two user accessible stacks (the other being the dictionary stack). It serves as the mechanism for passing arguments to procedures and operators and as a "scratch" space for doing computations.

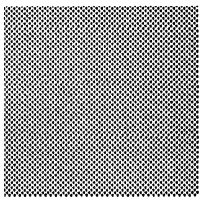
In the above example, the POSTSCRIPT interpreter first encounters the procedure definition for "inch." It associates the procedure body, {72 mul} with the name "inch" (more on associations later in the section on dictionaries). Then the interpreter encounters the number 1 and pushes it onto the operand stack (see Figure 2a). Next, the interpreter encounters the name "inch." It looks up the definition of "inch" and finds the procedure body, {72 mul} and begins executing the procedure body. The first thing encountered in the procedure body is the number 72 which also gets pushed onto the operand stack (see Figure 2b). Next, the operator **mul** is encountered and executed. **mul** removes (pops) the top two elements from the operand stack and pushes their product back onto the operand stack (see Figure 2c). This resulting number represents 1 inch in points (72). The end of the procedure body has been reached and the interpreter resumes interpreting the program. Next, the interpreter encounters the number 10 and pushes it onto the operand stack (see Figure 2d). Once again the name "inch" is encountered, looked up and executed, pushing the number 72 onto the operand stack (see Figure 2e) then multiplying the top two elements on the stack, leaving the number 720 on the stack (see Figure 2f). Finally the operator **moveto** is encountered. Previously we saw that the general form for the **moveto** operator is <x> <y> **moveto**. The <x> and <y> values are actually taken from the operand stack. **moveto** removes the top two elements from the operand stack, treating the topmost value as <y> and the next-to-the-topmost value as <x>. After the **moveto** operation, the operand stack is empty.

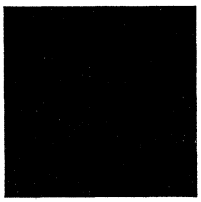


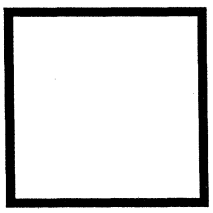
PROBLEM 4: Draw a vertical line 2 inches from the left edge of the paper which spans the length of the paper.

```
/inch {72 mul} def

newpath
  2 inch  0 inch moveto
  2 inch 11 inch lineto
stroke
showpage
```

The illustration of the output page is 1/8th the size of the actual output page.







Three Squares 1

```
/inch {72 mul} def

/inchsquare

  { newpath
      moveto
      0 1 inch rlineto
      1 inch 0 rlineto
      0 -1 inch rlineto
    closepath
  } def
```

In a similar manner to defining "inch", we define a sequence of path commands into a single name, "inchsquare."

Note that the <x> and <y> values for the initial moveto are left out. We expect that they will be supplied as operands before each use of "inchsquare." The edges of the box are specified with the "rlineto" operator so that each path segment is drawn relative to the previous point in the path. Thus, with different initial <x> <y> values, this one procedure will serve to construct a one inch square path anywhere on the page.

```
1 inch 6.5 inch inchsquare
3 setlinewidth
stroke
```

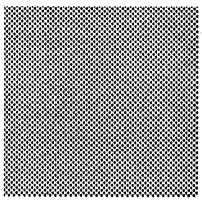
We place the lower left corner of the first square 1 inch from the left edge and 6.5 inches from the bottom of the page, and construct the rest of the path. Stroke the square onto the page.

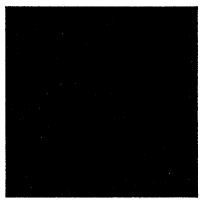
```
3.75 inch 4 inch inchsquare
fill
```

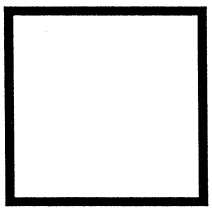
Place another inch square path at a new location
Now fill in this square on the page with black "ink".

```
6.5 inch 1.5 inch inchsquare
0.75 setgray
fill
showpage
```

Place another inch square path at a new location
Change the "ink" to a gray shade. "0 setgray" sets the ink to black, "1 setgray" sets the ink to white.
Finally, print the page on the output device.







Translated Squares 2

```
/inch {72 mul} def
/inchsquare
  { newpath
      0 0 moveto
      0 1 inch rlineto
      1 inch 0 rlineto
      0 -1 inch rlineto
    closepath
  } def
gsave
```

We redefine the "inchsquare" routine from the previous program to always "moveto" to the coordinate system origin (the point (0,0)). It might appear that we will lose the ability to place this box anywhere on the page by doing so but this is not true.

Make a snapshot of all the parameters that control graphics output. This "gsave" will snapshot all of the default parameters, since we haven't done anything yet.

```
1 inch 6.5 inch translate
```

Imagine that a path is first constructed in a stencil, which is then used by "stroke" and "fill" to draw their output on the page. All positions mentioned in paths are relative to the placement of this stencil. Initially, the stencil's origin is placed at the lower left corner of the paper. The "translate" operation moves the stencil to a new position. This position is first located within the current stencil, and a new stencil is positioned with its origin at that position.

```
inchsquare
```

We lay down an "inchsquare" at the stencil's origin. By moving the origin of the coordinate system, we have caused an absolutely positioned path to be moved around on the paper.

```
3 setlinewidth
stroke
grestore
```

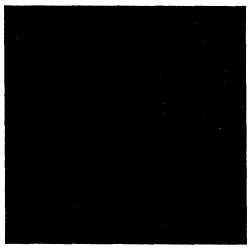
Restore all the graphics parameters to the values they had at the last "gsave". In particular, this restores the stencil position back to the paper's lower left corner.

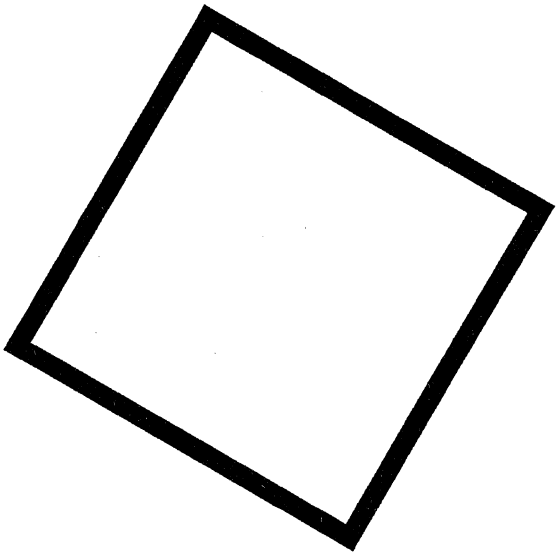
```
gsave
3.75 inch 4 inch translate
```

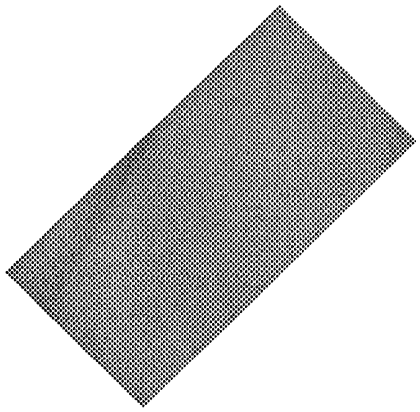
Remember the defaults again.
Move the stencil (coordinate system) to a new place.

```
inchsquare fill
grestore
6.5 inch 1.5 inch translate
0.75 setgray
inchsquare fill
showpage
```

Return to the default coordinate system again.







Translate, Rotate and Scale 3

```
/unitsquare
  { newpath
      0 0 moveto
      0 1 lineto
      1 1 lineto
      1 0 lineto
    closepath
  } def
```

The most convenient way to define a graphic shape is to define it in a unit size. While it may appear that this definition will give only a fixed, small square, by using the PostScript transformation operators, "translate", "rotate" and "scale", we can use this one definition to give us a square shape in any orientation and size, anywhere on the page. Note that this definition places the lower left corner of the square at the coordinate system origin. We could have chosen any corner, or even the center of the square for the origin. When a shape is defined in this manner, placing the origin can affect the ease with which the definition can be used later.

```
/inch {72 mul} def
gsave
1 inch 6.5 inch translate

1.25 inch 1.25 inch scale
```

Take a snapshot of the default graphics state.
Move the origin to one inch from the left edge and 6-1/2 inches from the bottom edge of the page.
Change the units to 1-1/4 inch in both the x and y directions. PostScript allows the x units and y units to be scaled differently (anamorphically), thus the two arguments to the "scale" operator. Although the scale operator allows us this flexibility, most scale operations are uniform (i.e.,- the x and y arguments are equal).

```
unitsquare fill
```

The unit square, translated and scaled, is drawn on the page as shown. Note that we performed the "translate" first, and the "scale" second. Had we performed these operations in the opposite order, the "translate" would have translated the origin in terms of the expanded 1-1/4 inch long units. The 1 inch translation in x would have moved the origin 72 times 1-1/4 inches or 91 inches to the right! Always know your current units when performing these operations.

```
grestore gsave
4.25 inch 3.75 inch translate

60 rotate


2 inch 2 inch scale unitsquare


1 20 div setlinewidth stroke
grestore

4.5 inch 2 inch translate
-45 rotate
1 inch 2 inch scale
unitsquare .75 setgray fill
showpage
```

Get back to normal coordinates, and snapshot them again. Since units are back to 1/72 inch, this sequence does what we'd expect.
Rotate the coordinate axes 60 degrees counterclockwise about the current origin. Now the x axis points up and to the right, and the y axis points up and to the left.
Change the units to two inches long each along the coordinate axes. Translate first, rotate next, and scale last is the simplest order in which to perform these transformations.
Prepare to stroke this path. Careful--the units are 2 inches long and the linewidth is expressed in terms of units. 1/20th of a 2 inch unit will give us a 1/10 inch thick line.

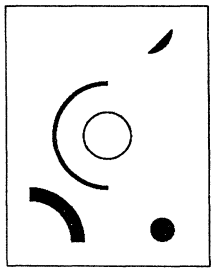
Rotate the axes forty five degrees clockwise.
Let's try an anamorphic scale with the y units twice the length of the x units.

Using the arc Operator

POSTSCRIPT provides a very useful operator, **arc**, which is used for constructing arcs of circles. The general form of the **arc** operator is:

<x> <y> <radius> <start angle> <end angle> **arc**

<x> and <y> are the coordinates of the center of the circle, <radius> is the radius of the circle. <x>, <y>, and <radius> are all specified in the units of the current user coordinate system. Since the **arc** command can draw whole circles or just segments of circular arcs, it is necessary to specify the <start angle> and the <end angle> for the circular arc. These two numbers are specified in degrees (there are a total of 360 degrees in a circle). The arc command draws *counterclockwise* circular arcs.

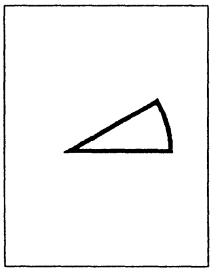


PROBLEM 5: Draw several circular arcs on the page.

```
newpath
  306 396 72 0 360 arc
stroke
newpath
  306 396 160 90 270 arc
10 setlinewidth
stroke
newpath
  470 110 36 0 360 arc
fill
newpath
  72 72 144 0 90 arc
40 setlinewidth
stroke
newpath
  430 720 72 270 360 arc
fill
showpage
```

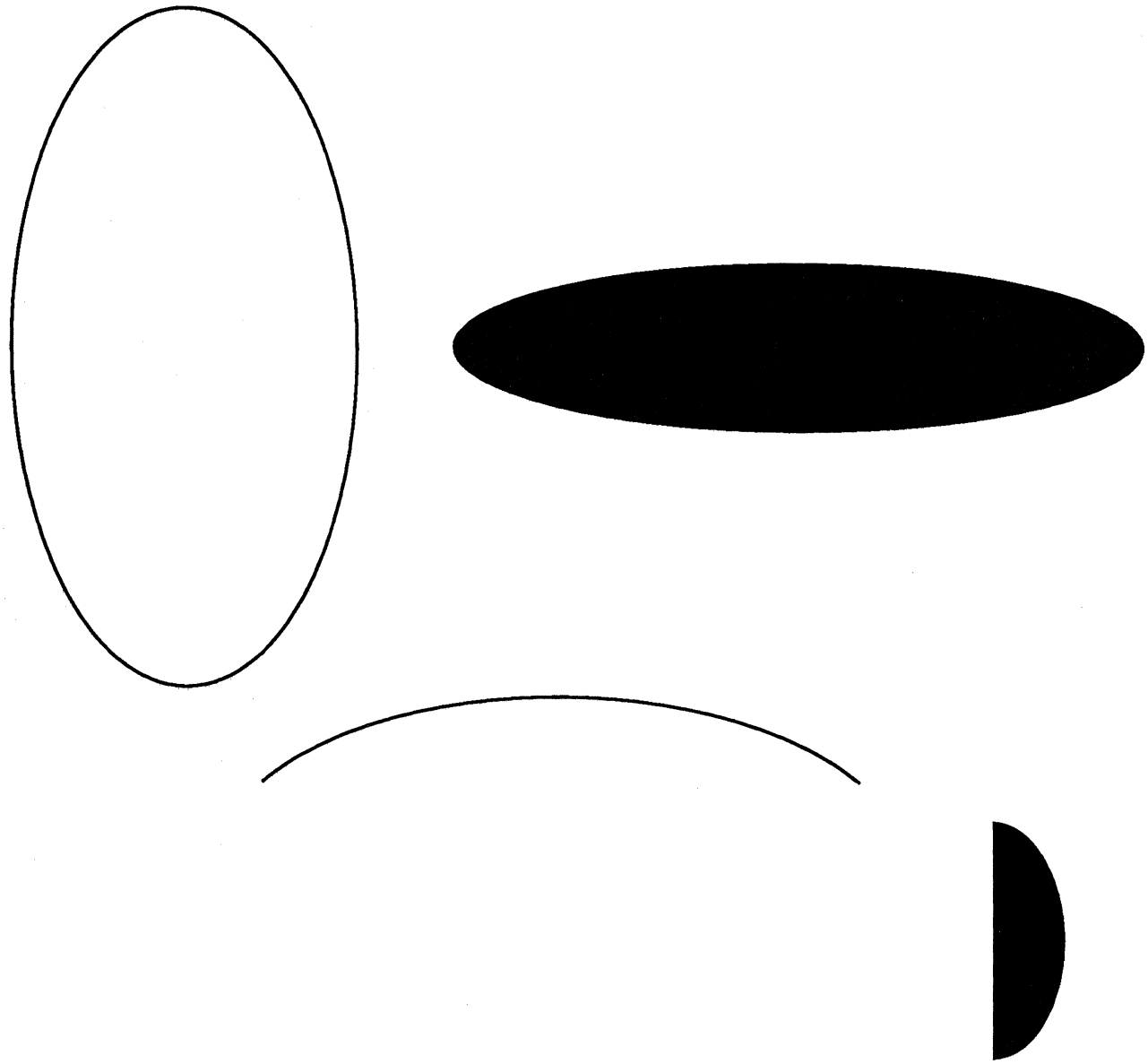
Note that a **moveto** was not necessary at the beginning of each path. The **arc** operator is unusual in this respect; if the current path is empty, **arc** will perform an implicit **moveto**.

The **arc** operator can be used in combination with other path construction operators to produce shapes with curved edges. To make this task easier, the **arc** operator has the following special property: when there is a current point, an implicit **lineto** is inserted from the current point to the beginning point on the arc. By taking advantage of this property, we can easily draw shapes such as pie slices.



PROBLEM 6: Draw an outlined pie slice.

```
newpath
  200 350 moveto
  200 350 300 0 30 arc
closepath
10 setlinewidth
stroke
showpage
```



Drawing an Ellipse 4

```
/mtrx matrix def
/ellipse
  { /endangle exch def
    /startangle exch def
    /yrad exch def
    /xrad exch def
    /y exch def
    /x exch def
```

Allocate a matrix for the save matrix operation below.
ellipse adds a counter-clockwise segment of an elliptical arc to the current path. The ellipse procedure takes six operands: the x and y coordinates of the center of the ellipse (the center is defined as the point of intersection of the major and minor axes), the "radius" of the ellipse in the x direction, the "radius" of the ellipse in the y direction, the starting angle of the elliptical arc and the ending angle of the elliptical arc.

The basic strategy used in drawing the ellipse is to translate to the center of the ellipse, scale the user coordinate system by the x and y radius values, and then add a circular arc, centered at the origin with a 1 unit radius to the current path. We will be transforming the user coordinate system with the translate and rotate operators to add the elliptical arc segment but we don't want these transformations to affect other parts of the program. In other words, we would like to localize the effect of the transformations. Usually the gsave and grestore operators would be ideal candidates for this task. Unfortunately gsave and grestore are inappropriate for this situation because we cannot save the arc segment that we have added to the path. Instead we will localize the effect of the transformations by saving the current transformation matrix and restoring it explicitly after we have added the elliptical arc to the path.

```
    /savematrix mtrx currentmatrix def
    x y translate
    xrad yrad scale
    0 0 1 startangle endangle arc
    savematrix setmatrix
  } def
```

Save the current transformation.
Translate to the center of the ellipse.
Scale by the x and y radius values.
Add the arc segment to the path.
Restore the transformation.

```
newpath 144 400 72 144 0 360 ellipse stroke
```

Full ellipse, stroked. Note that the y-axis is longer than the x-axis.

```
newpath 400 400 144 36 0 360 ellipse fill
```

Full ellipse, filled. Note that the y-axis is shorter than the x-axis.

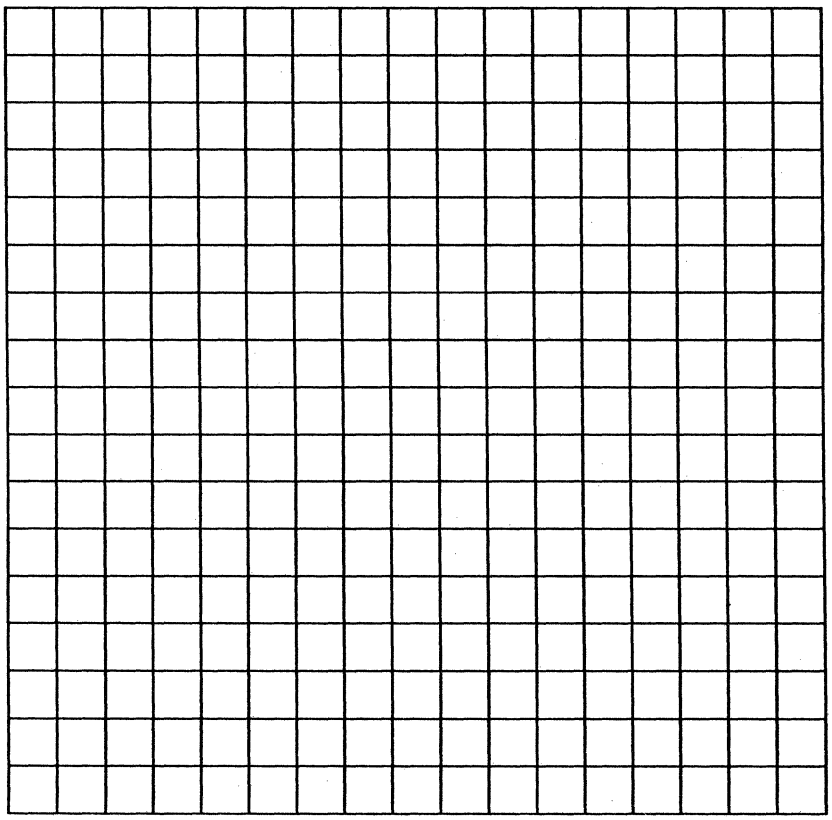
```
newpath 300 180 144 72 30 150 ellipse
  stroke
```

Elliptical arc, stroked.

```
newpath 480 150 30 50 270 90 ellipse fill
```

Elliptical arc, filled.

```
showpage
```



Repeated Lines 5

This program prints "graph paper" by repeatedly drawing horizontal and vertical lines. Each line is moved into place by a "translate" operation. Since the construction of a grid is so regular, the "repeat" operator will come in handy.

```
/inch {72 mul} def
```

```
.25 setlinewidth
2 setlinecap
```

Each line will be 1/4 of a point wide.
Set up projecting square end caps for the lines to be drawn. Each end cap will project half the line width out from the end of the path, or in this case, will project out 1/8 of a point. This projection will fill in the extreme corners of the grid, which with the simple repetition used below, would have had small square notches otherwise.

```
gsave
  2.125 inch 3.5 inch translate
  18
```

Remember the default coordinate system.
Move into position at the lower left corner of the grid.
The first argument for the "repeat" operator is the number of repetitions.

```
    { newpath
        0 0 moveto
        0 4.25 inch lineto
      stroke
      0.25 inch 0 translate
    } repeat
```

The second argument is the sequence of operations to be repeated. This sequence must be enclosed by braces. The code here will draw a 4 1/4 inch vertical line and translate 1/4 inch to the right to set up for the next vertical line.

The second argument is closed with a right brace, and the "repeat" operator executes. The code within the braces will be executed 18 times.

```
grestore
```

Now return to the default coordinate system.

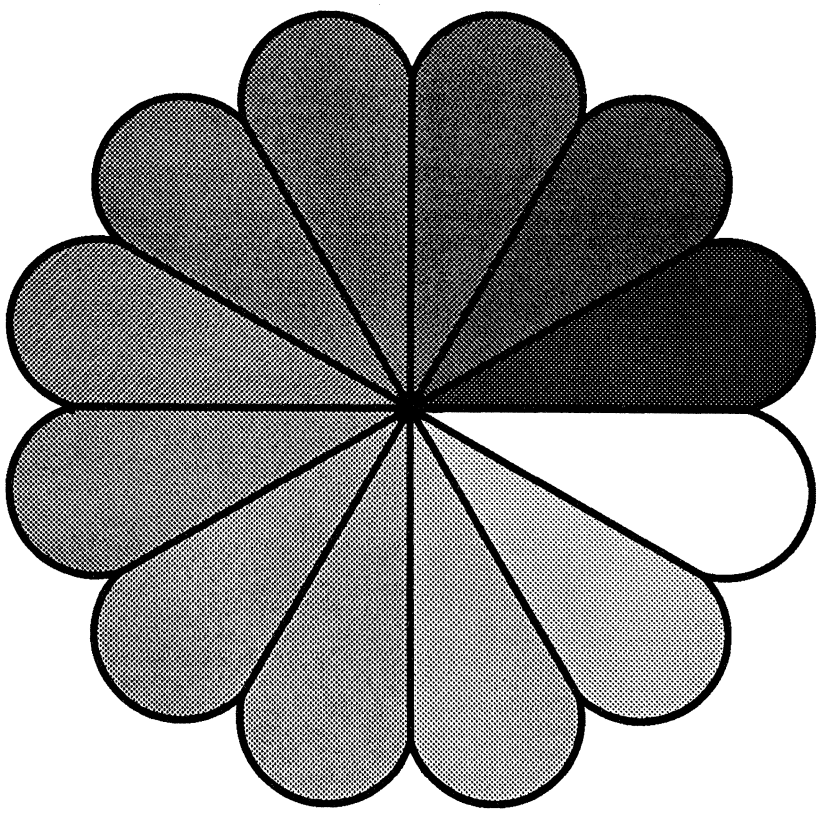
```
gsave
  2.125 inch 3.5 inch translate
  18
    { newpath
        0 0 moveto
        4.25 inch 0 lineto
      stroke
      0 0.25 inch translate
    } repeat
grestore
```

Move into position at the lower left corner again.

This time, the code sequence draws a 4 1/4 inch horizontal line and translates 1/4 inch upward for the next horizontal line.

This time, 18 horizontal lines are drawn.

```
showpage
```



Repeated Shapes 6

This program prints a rosette design by defining a section of that design and printing that section repeatedly. This program illustrates the "for" and "arc" operators, and it shows how coordinate transformations can be nested so as to use the most convenient coordinate system for each part of a design.

```
/inch {72 mul} def

/wedge
  { newpath
      0 0 moveto
      1 0 translate
      15 rotate
      0 15 sin translate
      0 0 15 sin -90 90 arc
    closepath
  } def
```

Define an "ice cream cone" shape by means of the "arc" operator. This shape will have a 30 degree angle topped off with a semicircle. Set the path's first point at the current origin. Next, move the origin to the center of the semicircle by translating to the right 1 unit, rotating counter-clockwise by 15 degrees, and translating "up" in the rotated system by the radius of the semicircle. The "arc" operator includes a straight line to the initial point of the arc and a curved section to the end of the arc. Note that the semicircle goes from -90 degrees to 90 degrees in the rotated coordinate system.

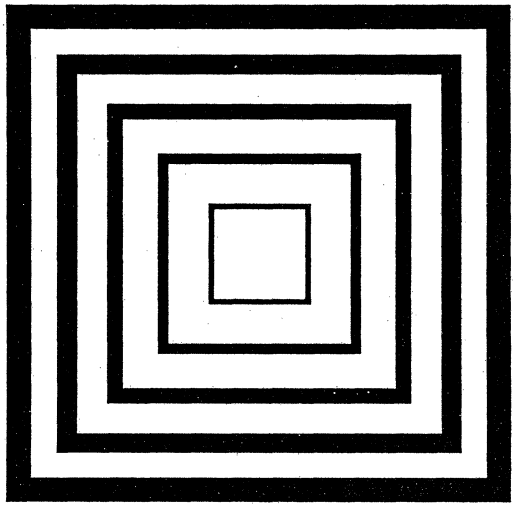
```
gsave
  3.75 inch 7.25 inch translate
  1 inch 1 inch scale
  wedge .02 setlinewidth stroke
grestore
```

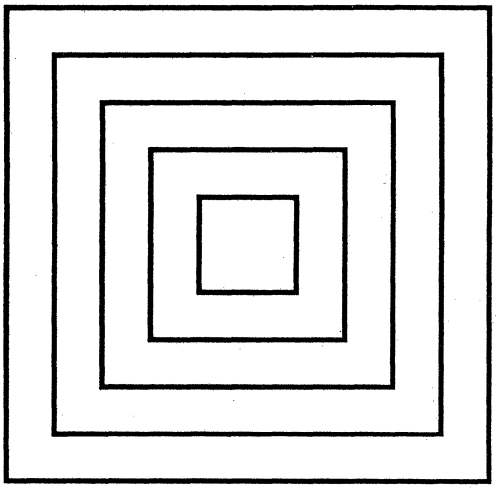
Remember the default coordinate system.
Move into position for a sample of the wedge.
Make the edge of the wedge 1 inch long.
Draw the wedge with a 1/50 inch thick line.
Get back to default coordinates.

```
gsave
  4.25 inch 4.25 inch translate
  1.75 inch 1.75 inch scale
  0.02 setlinewidth
  2 1 13

   {

     13 div setgray
     gsave
       wedge
       gsave
         fill
       grestore
       0 setgray stroke
     grestore
     30 rotate
   } for
grestore
showpage
```

Move into position for the rosette.
Make the edges of the rosette 1 3/4 inches long.
Use a 7/200 inch thick line.
Set up the "for" operator to iterate 12 times, pushing 2 onto the stack the first time, 3 the next time, ... , and 13 the last time.
The last argument for "for" is the sequence of operations to be repeated. This sequence must be enclosed by braces.
Divide the loop index by 13 to set a gray value.
Enclose the "wedge" operation in a "gsave"-"grestore" pair, as it will mess up the coordinate system.
Save the wedge path for use after the "fill".

Draw a black border around the wedge.
Get out of the coordinate system left by wedge.
Set up for the next section.
Close the last argument and execute the "for" operator.





Expanded and Constant Width Lines 7

```
/inch {72 mul} def
```

This program prints two sets of enclosed boxes. Both sets are generated from a unit box definition under different scales. In the left set, the width of the lines increases as the size of the box increases. In the right set, the width of the lines remain constant even as the size of the box increases.

```
/centerbox
  { newpath
      0.5 0.5 moveto
      -.5 0.5 lineto
      -.5 -.5 lineto
      0.5 -.5 lineto
    closepath
  } def
```

A unit box described in terms of its center, rather than in terms of one of its corners, is most convenient for this example. This procedure creates a 1 unit square path around the current coordinate system origin.

```
gsave
  2.25 inch 6.5 inch translate
  1 20 div setlinewidth
```

Remember the original coordinate system.
Place the origin for the expanding linewidth boxes.
Make the lines 1/20 of an inch thick.

```
  1 1 5
    { gsave
        .5 mul inch dup scale
        centerbox stroke

      grestore
    } for
```

Set up a "for" loop to execute five times.
Remember the current coordinate system.
Scale the current units by 1/2 inch times the loop index.
The stroked box has a line width proportional to the current scale, since the line width is expressed in units.
Return to the translated, unscaled coordinate system.

```
grestore
```

Return to the original coordinate system.

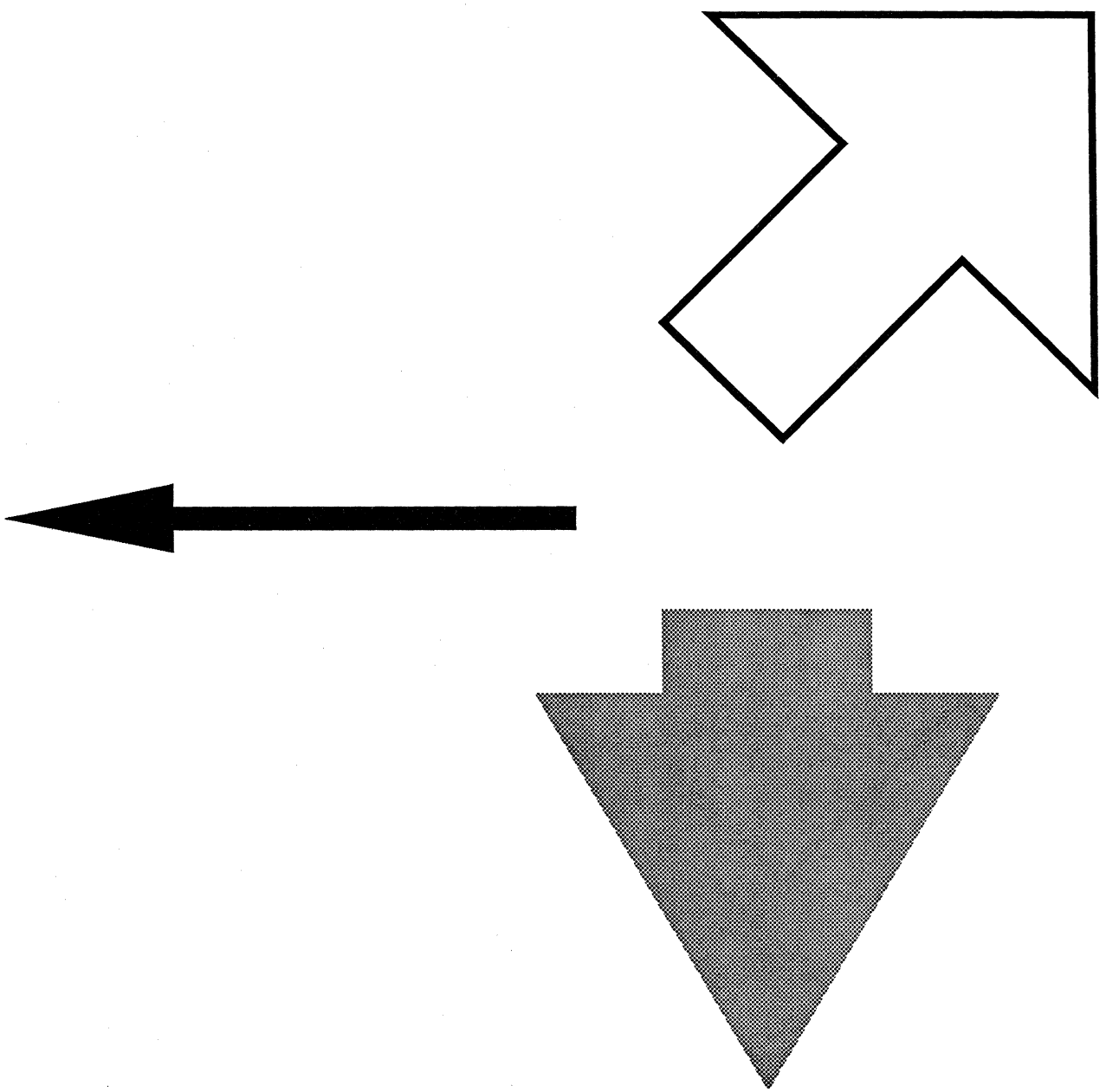
```
/cmtx matrix currentmatrix def
1 50 div inch setlinewidth

6.25 inch 6.5 inch translate
```

Store the current transform matrix, i.e., the current coordinate system, in the variable "cmtx". Set a 1/50 of an inch line width
Place the origin for the constant line width boxes.

```
1 1 5
  { gsave
      .5 mul inch dup scale
      centerbox
      cmtx setmatrix stroke



    grestore

  } for
```

Remember the translated coordinate system.
Scale the boxes as before.
Create the box path, but don't stroke it yet.
Change the coordinate space back to the original one, where the line width is truly 1/20th of an inch thick. We explicitly reset only the coordinate space rather than use a "grestore", since "grestore" resets the current path as well as the current coordinate system.
After stroking the path, return to the translated coordinate system.

```
showpage
```



Drawing Arrows 8

```
/arrowdict 13 dict def
```

Local storage for the procedure "arrow."

```
/arrow
  { arrowdict begin
      /headlength exch def
      /halfheadthickness exch 2 div def
      /halfthickness exch 2 div def
      /tipy exch def /tipx exch def
      /taily exch def /tailx exch def
```

The procedure "arrow" adds an arrow shape to the current path. It takes seven arguments: the x and y coordinates of the tail (imagine that a line has been drawn down the center of the arrow from the tip to the tail, then x and y lie on this line), the x and y coordinates of the tip of the arrow, the thickness of the arrow in the tail portion, the thickness of the arrow at the widest part of the arrowhead and the length of the arrowhead.

```
      /dx tipx tailx sub def
      /dy tipy taily sub def
      /arrowlength dx dx mul dy dy mul add
        sqrt def
      /angle dy dx atan def
      /base arrowlength headlength sub def
```

Compute the differences in x and y for the tip and tail. These will be used to compute the length of the arrow and to compute the angle of direction that the arrow is facing with respect to the current user coordinate system origin.

Compute where the base of the arrowhead will be.

```
      /savematrix matrix currentmatrix def
```

Save the current user coordinate system. We are using the same strategy to localize the effect of transformations as was used in the program to draw an ellipse.

```
      tailx taily translate
      angle rotate
```

Translate to the starting point of the tail.
Rotate the x-axis to correspond with the center line of the arrow.

```
      0 halfthickness neg moveto
      base halfthickness neg lineto
      base halfheadthickness neg lineto
      arrowlength 0 lineto
      base halfheadthickness lineto
      base halfthickness lineto
      0 halfthickness lineto
      closepath
```

Add the arrow shape to the current path.

```
      savematrix setmatrix
    end
  } def
```

Restore the current user coordinate system.

```
newpath
  318 340 72 340 10 30 72 arrow
fill
```

Draw a filled arrow with a thin tail and a long arrowhead.

```
newpath
  382 400 542 560 72 232 116 arrow
3 setlinewidth stroke
```

Draw an outlined arrow with a 90 degree angle at the tip. To get a 90 degree angle, the headthickness should be twice the headlength.

```
newpath
  400 300 400 90 90 200 200 3 sqrt mul 2 div
    arrow .65 setgray fill
showpage
```

Draw a gray filled arrow that has an equilateral triangle for its arrowhead. To get an equilateral triangle, the headlength should be the square root of 3 divided by 2 times the headthickness.

Changing the Appearance of a Stroke

(a) `0 setlinecap`

(b) `1 setlinecap`

(c) `2 setlinecap`

Figure 3: Different Line Cap Styles

(a) `0 setlinejoin`

(b) `1 setlinejoin`

(c) `2 setlinejoin`

Figure 4: Different Line Join Styles

In addition to specifying the trajectory for a stroke, the user can control the way in which the stroke is rendered. We have already been controlling the appearance of a stroke by changing its thickness with the **setlinewidth** operator. Other operators that can change the appearance of a stroke are the **setlinecap, setlinejoin,** and **setdash** operators.

POSTSCRIPT offers three kinds of line caps for a stroke: butt, round, and projecting square. The line caps are placed at the open ends of the current path when the **stroke** operator is executed. Normally when a path is stroked, butt end caps are used. This means that the open ends of the path are "squared off" perpendicular to the path (see Figure 3a). Round end caps are actually semicircles, centered at each open end of the path with diameter equal to the line width (see Figure 3b). Projecting square end caps are similar to butt end caps but they extend out by one-half of a line width in the direction of the path at each open end (see Figure 3c).

The default line cap style is a butt end cap. The **setlinecap** operator allows the user to change the line cap to any of the three different styles. The general form for the **setlinecap** operator is:

<integer> **setlinecap**

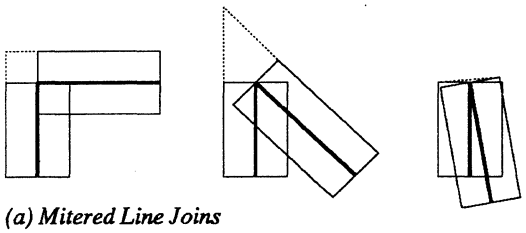
where the integer 0 specifies butt end caps, 1 specifies round end caps and 2 specifies projecting square end caps. Any other integer values will result in a *rangecheck* error.

Another way of changing the appearance of a stroke is to change the line join style. The places in a path where the different segments connect are known as *line joins*. The default line join style is mitered (see Figure 4a). Two additional styles are available: rounded (see Figure 4b) and beveled (see Figure 4c). The different styles are discussed in more detail below.

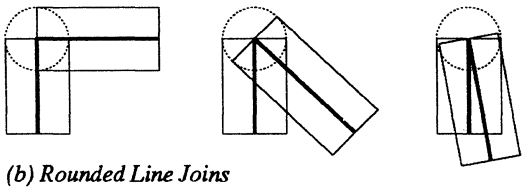
The **setlinejoin** operator allows the user to change the line join style to any of the three styles. The general form for the **setlinejoin** operator is:

<integer> **setlinejoin**

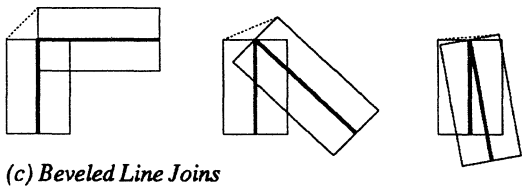
where the integer 0 specifies mitered joins, 1 specifies rounded joins and 2 specifies beveled joins. Any other integer values will result in a *rangecheck* error.



(a) Mitered Line Joins



(b) Rounded Line Joins



(c) Beveled Line Joins

Figure 5

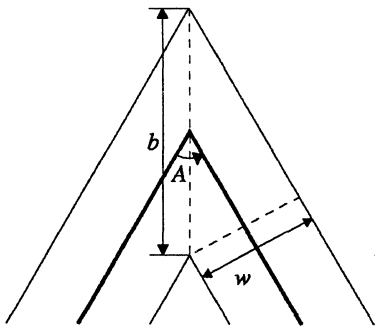


Figure 6: Detail of a Mitered Join

Figure 5 gives more detailed illustrations of how the different line join styles are implemented given different join angles. The dark solid line represents the path, the light solid line represents an outline of the stroke applied to the path and the dotted line represents the line join.

To understand how all the different line join styles are implemented, begin by imagining that all the segments in the path are stroked with butt endcaps. Mitered line joins are slightly more complicated than rounded or beveled joins so they will be discussed last. Rounded joins are formed by centering a circle with diameter equal to the stroke width at each join. Beveled joins are formed by filling in the notches left at each join with a triangle.

Mitered joins are formed by extending the outside lines of the stroke until they intersect. When two segments of a path are joined at a sharp angle (usually less than 11 degrees) it's possible that the mitered join could form a very long "spike" which might project undesireably into other objects on the page or possibly project off the page. To avoid this occurrence there is a value in the graphics state known as the miter limit which is used in limiting the size of miter spikes. If a given spike exceeds the miter limit, then the **stroke** operator makes a bevel join instead. (The miter limit is discussed in more detail in the next paragraph.) This is the case in the third example in Figure 5a which shows different miter joins.

In order to have better control over mitered joins, the user should understand the miter limit value. The *miter limit* is defined to be the maximum ratio of the length of the line bisecting the mitered join to the line width (see Figure 6). Let *b* be the length of the bisecting line and let *w* be the line width. When *b/w* exceeds the miter limit, the line join is beveled instead of mitered.

The default value of the miter limit is 10. This means that any miter joins with a bisecting line length larger than 10 times the line width are beveled instead. The miter limit can be changed with the **setmiterlimit** operator. The default form for the **setmiterlimit** operator is:

<num> **setmiterlimit**

where <num> is the ratio. The miter limit ratio is related to the line join angle by the following formula:

<num> = 1 / sin(*A*/2)

Therefore another way to think about the miter limit is in terms of angles: Any line join angle which is less than *A* will result in a beveled join instead of a mitered join. For the default miter limit ratio of 10, any line joins which meet at an angle of less than 11.4 degrees will be beveled.

Some common miter limit values are: 1.415 which bevels mitered joins at angles less than 90 degrees; 2.0 which bevels mitered joins at angles less than 60 degrees and 10.0, the default, which bevels mitered joins at less than 11.4 degrees. The miter limit ratio can never be less than 1 and setting the miter limit to exactly 1 causes all mitered joins to be beveled.

POSTSCRIPT provides a useful operator for producing dashed lines called the **setdash** operator. **setdash** takes two arguments: an array describing the dash pattern to be repeated and an offset which can be used to control how the pattern is placed on the path.

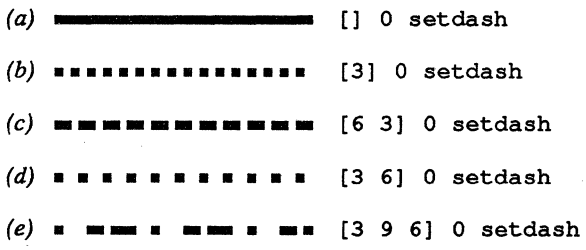


Figure 7: Different Dash Arrays

A dashed stroke consists of alternating filled and unfilled sections. The array describing the dash pattern contains numbers which denote the distances along the path for each filled and unfilled section of the stroke. The array can be of any length. An array of length zero causes dashing to be "turned-off" and hence a solid (undashed) stroke is placed along the path (see Figure 7a). When the array has a length greater than zero, the following procedure takes place: The first section of the stroke is always filled and the zeroeth element of the dash array determines how long it will be. Thereafter, the elements of the array are used cyclically to determine the length of each succeeding section of the stroke.

Figure 7b shows the effect of having a dash array with just one element. Each filled and unfilled section has the same length. Figures 7c and 7d show the effect of having a two element array. The filled sections assume the length of the zeroeth element and the unfilled sections assume the length of the first element. Figure 7e shows a three element array and illustrates how the elements are used cyclically to produce a different pattern.

In the previous examples of the **setdash** operator, we used zero as the offset argument for purposes of clarity. Figure 8 shows the effect of different offset values on the dash line.

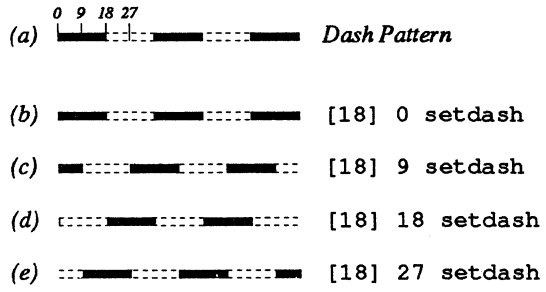


Figure 8: Different Offset Values

Please note that in the figure, the lightly dashed lines exist for clarity only, they are not produced by the **setdash** operator. The offset is a length (it must be non-negative) which can be used to position the beginning of the repeating pattern on the path.

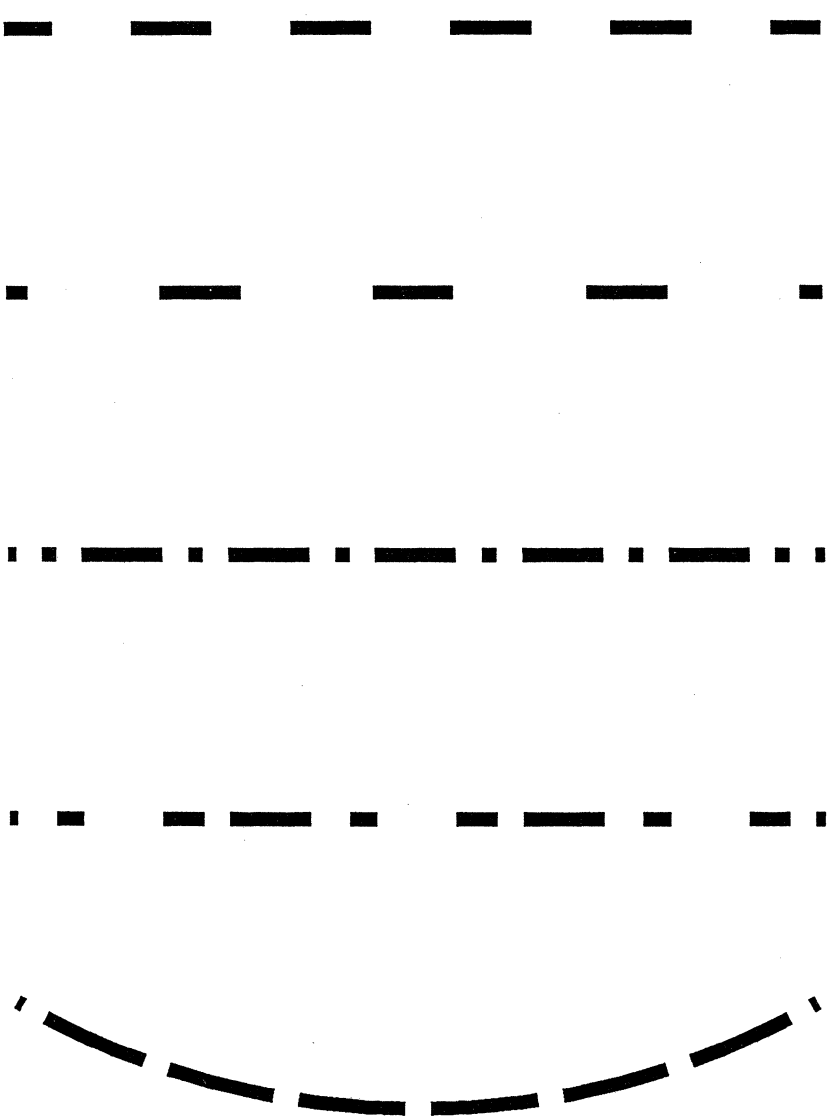
POSTSCRIPT uses the offset in the following manner: Imagine that the dashed line pattern that is placed on the path and the path itself are treated as separate entities. Then the starting point of the dashed line pattern can be shifted on the path and the offset is used to control the shift distance.

Figure 8a shows a dash pattern which is repeatedly filled for 18 units and unfilled for 18 units. Figure 8b shows the dash pattern applied to a path with a zero offset. Figure 8c shows what happens when an offset of 9 is used; the dash pattern is shifted towards the starting point of the path by a distance of 9 units. Figure 8d shows the effect of an offset of 18 units. This time the dash pattern is shifted towards the starting point of the path by 18 units. Figure 8e shows the effect of an offset of 27 units.

Dashed lines, like other stroked lines, assume the current line cap and line join styles. Each filled dash segment in the path is finished with the current line cap style and whenever a filled dash segment coincides with a line join in the path it is given the current line join style.

It is also important to understand that POSTSCRIPT does not modify the dash pattern in any way to fit the path "better." This means that if the end of a dash segment doesn't happen to coincide with the end of the path, only part of the dash segment will be printed; POSTSCRIPT will not make any attempt to "even-out" all the dash segments. This is left up to the user.

Occasionally it is desirable to have identical looking dash segments at the end points of a path. This can be accomplished by adjusting the offset argument to the **setdash** operator. Program 9 presents an algorithm to solve to this problem.



Centered Dash Patterns 9

```
/centerdash
  { /pattern exch def

    /pathlen pathlength def
    /patternlength 0 def
    pattern
      { /segmentlength exch def
        /patternlength patternlength
          segmentlength add def
      } forall

    pattern length 2 mod 0 ne
      { /patternlength patternlength
        2 mul def } if
    /first pattern 0 get def

    /last patternlength first sub def

    /n pathlen last sub patternlength
      idiv def
    /endpart pathlen patternlength n mul
      sub last sub 2 div def
    /offset first endpart sub def



    pattern offset setdash
  } def

/pathlength
  { flattenpath
    /dist 0 def
    { /yfirst exch def /xfirst exch def}
    { /ynext exch def /xnext exch def
      /dist dist ynext yfirst sub dup mul
        xnext xfirst sub dup mul
        add sqrt add def
      /yfirst ynext def /xfirst xnext def}
    {}
    {}
    pathforall
    dist
  } def
```

The procedure "centerdash" will center a dash pattern on a path such that the dashes at the end points are identical. It takes an array describing the dash pattern as its argument. In order to center the dash pattern on the path we need to determine the length of the path. (See definition of "pathlength" below.) First determine the total length of the repeating pattern by summing the elements of the dash array.

If the pattern array is an odd number of elements double its length so that we can get identical end points.

Get the length of the first element in the pattern array to be used in later computations.
Compute length of last part of pattern.

Now we wish to compute the offset provided to the setdash operator such that the dashes at the end points are identical. Think of the path as begin composed of 4 different parts: 2 identical end parts, 1 part which is composed of "n" repeating pattern pieces and 1 part which is the last piece of the pattern. We can compute the values of the last piece and the part composed of "n" repeating pattern pieces and solve for the end part. The amount of offset is then given by the difference in length of the first part and the end part.
Set the dashing for the stroke using the offset computed above.

The procedure "pathlength" computes the length of any given path. It does so by first "flattening" the path with the "flattenpath" operator. "flattenpath" converts any curveto and arc segments in a path to a series of lineto segments. Then the "pathforall" operator is used to access all the segments in the path so that the length of each segment can be determined and added to a total.

The curveto procedure does nothing since there shouldn't be any curveto segments in the path after a flattenpath.

Leave the length of the path on the operand stack.

Centered Dash Patterns, cont. 9

```
5 setlinewidth

newpath
  72 500 moveto 378 500 lineto
[30] centerdash
stroke

newpath
  72 400 moveto 378 400 lineto
[30 50] centerdash
stroke

newpath
  72 300 moveto 378 300 lineto
[30 10 5 10] centerdash
stroke

newpath
  72 200 moveto 378 200 lineto
[30 15 10] centerdash
stroke

newpath
  225 390 300 240 300 arc
[40 10] centerdash
stroke

showpage
```

Set up a line width.

This example illustrates the centering of a very simple dash pattern in which the unfilled dashes have the same length as the filled ones.

This example is similar to the above example except that the unfilled dashes are longer than the filled ones.

This example illustrates the centering of a dot-dash pattern.

This example illustrates the centering of an asymetric pattern.

This final example illustrates the centering of a dash pattern on an arbitrary path, in this case an arc.

Fonts

Times-Roman
Times-Italic
Times-Bold
Times-BoldItalic

Figure 9

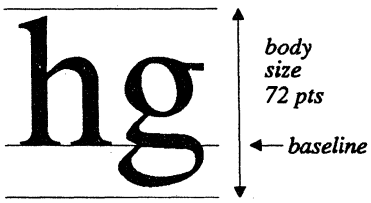


Figure 10

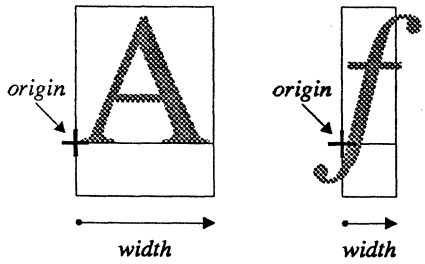


Figure 11

A *font* is a collection of characters (letters, numerals, punctuation marks, reference marks, and symbols) which have a unified design. A *font family* is a group of fonts of similar design created to be used together. The individual fonts in a font family are known as *font faces*.

Every POSTSCRIPT font has a name which is used when referring to that font in a POSTSCRIPT program. The POSTSCRIPT *font name* is usually a combination of the font family name and the font face name. For example, the current Adobe Times font family includes four font faces whose names are: Times-Roman, Times-Italic, Times-Bold, and Times-BoldItalic (see Figure 9).

When we vertically measure a font face we use points (recall that there are approximately 72 points per inch). This measurement is known as the *point size*. The point size refers only to the body size of the font, *not* to the size of any particular character in the font. Figure 10 shows the relationship between body size and actual character size for a font with a body size of 72 points. The point size is the minimum amount of space required between lines of text to ensure that no characters overlap.

Traditional hot metal fonts are most commonly available in a limited number of point sizes from 5 to 72 points. POSTSCRIPT fonts, on the other hand, are available in any point size imaginable; you can print text in an 8.327 point font if you wish. POSTSCRIPT has a unique description for each different font face in a font family. (The description is actually a POSTSCRIPT dictionary called a font dictionary). This unique description has no point size associated with it. Instead, it is one unit high and the user must scale it to get different point sizes.

Each character in a POSTSCRIPT font has an *origin* and a *width* associated with it. Usually the origin is positioned on the baseline of the character and slightly to the left of the character shape. The width of a character usually spans the shape of the character and includes a little white space on either side (see Figure 11a). Figure 11b shows an exception to both of these rules, but nonetheless demonstrates the correct origin and width for that character.

The tendency of the best typography has been and still should be in the path of simplicity, legibility, and orderly arrangement.
Theodore Low De Vinne

Simple Text 10

```
/Times-Roman findfont 12 scalefont
setfont

318 552 moveto
(The tendency of the best) show
318 552 12 sub moveto
(typography has been and) show
318 552 24 sub moveto
(still should be in the) show
318 552 36 sub moveto
(path of simplicity,) show
318 552 48 sub moveto
(legibility, and orderly) show
318 552 60 sub moveto
(arrangement.) show
318 552 72 sub moveto
(Theodore Low De Vinne) show

showpage
```

Before any text can be "drawn" on the output page, a font must be selected and scaled properly. This is accomplished by the first line in the program.

The findfont operator is the first operator used when setting up a font. findfont takes a PostScript font name as its argument. (The PostScript font names available will vary from installation to installation but it is possible to write a PostScript program to find out what they are). When the findfont operator has been executed, it pushes a dictionary, called a "font dictionary," onto the operand stack. This font dictionary contains the information required by PostScript to construct characters in the font 1 unit in size.

Once we have this 1 unit font available, we must scale it to the desired point size. Although this could be accomplished by using the scale operator, it may be inconvenient since it will scale everything including the coordinates to be used for placement of characters on the page. The scalefont operator scales only the characters in the font. scalefont takes a font dictionary (the one left on the stack by the findfont operator) and a number which is used to scale the font. The scalefont operator pushes a new font dictionary that contains the proper scaling information in addition to all the other font dictionary information onto the operand stack.

When PostScript "draws" text onto the page, it uses the "current font." The current font is set using the setfont operator. setfont takes a font dictionary (in this case the result of the findfont-scalefont operations) as its argument.

Finally we wish to draw text on the page. This last step has two parts to it: positioning the text on the page and indicating which characters are to be drawn at that position. Positioning the text is accomplished by using the moveto operator. Indicating which characters are to be drawn is accomplished by using the show operator. The show operator takes a string as its argument. The origin of the first character in the string is placed at the current point and then the current point is shifted by the width of the character. (This is the reason for the moveto operation - it sets the current point). This process is repeated for each character in the string, and when finished, show will leave the current point positioned after the last character drawn. Each new line of text to be printed must be repositioned using the moveto operator.

Architecture

In the sense in which *Architecture*
is an art, **Typography** is an art.

Beatrice Warde

Faces and Sizes 11

```
/smallfont
  /Times-Roman findfont 9 scalefont def
/normalfont
  /Times-Roman findfont 12 scalefont def
/largefont
  /Times-Roman findfont 18 scalefont def
/italicfont
  /Times-Italic findfont 12 scalefont def
/boldfont
  /Times-Bold findfont 12 scalefont def
```

For efficiency reasons, programs that use several different typefaces in different sizes will typically "define" the fonts that are to be used at the beginning of the program. These "defined" fonts are then used later throughout the program in setfont operations. By defining the fonts at the outset, we save time later by not having to perform findfind-scalefont operations each time a setfont is done. Here we are setting up several face and size combinations with findfont-scalefont sequences. The resulting font dictionaries are assigned to suitably named variables. The variable names have been chosen to be as descriptive as possible yet actually we could have chosen any names that adhere to the PostScript name syntax. We will later use these variables with the setfont operator.

```
234 548 moveto
largefont setfont
(Architecture) show
```

Begin with the 18 point Times Roman font.

```
234 518 moveto
normalfont setfont
(In the sense in which) show
italicfont setfont
( Architecture ) show
```

Switch to the normal size, 12 point Times Roman.

Switch to the italic face.
Notice that an additional moveto operation is not necessary. The text is placed where the last show operation left off.

```
234 506 moveto
normalfont setfont
(is an art,) show
boldfont setfont
( Typography ) show
normalfont setfont
(is an art.) show
```

Switch back to the 12 point Times Roman.

Switch to the bold face.

Switch back to the 12 point Times Roman.

```
234 472 moveto
smallfont setfont
(Beatrice Warde) show

showpage
```

Switch to the small point size, 9 point Times Roman.

Rest at pale evening . . .
A tall slim tree . . .
Night coming tenderly
Black like me.

Langston Hughes

White Text on a Black Background 12

```
newpath
  72 561 moveto
  296 561 lineto
  296 440 lineto
  72 440 lineto
closepath fill
```

Create the black rectangular background. Since the default color in the Graphics State is black, there is no need to explicitly set the color.

```
/Helvetica-Bold findfont 14 scalefont
  setfont
```

Usually a bold font will have more presence on a black background than a medium font.

```
gsave
```

Save the current Graphics State so that it will be unaffected by changes in color.

```
  1.0 setgray
```

Change the color to white.

```
  104 520 moveto
  (Rest at pale evening . . .) show
  104 502 moveto
  (A tall slim tree . . .) show
  104 484 moveto
  (Night coming tenderly) show
  104 466 moveto
  (Black like me.) show
```

The show operator uses the current color in the Graphics State. Since we are "drawing" white characters on a black area, it gives the impression of black text on a white background which has been reversed.

```
grestore

104 398 moveto
(Langston Hughes) show

showpage
```

The makefont Operator

Until now, to get different size typefaces, we have used the **scalefont** operator. There is another operator, **makefont**, which can also be used to scale the master font to any size. **makefont** is more general than **scalefont** since it takes a POSTSCRIPT matrix as its argument. By changing the elements of the matrix, we can transform the master font in different ways.

A POSTSCRIPT matrix is really a six element array. The elements of the matrix can be modified to uniformly scale the font or they can be modified to achieve more unusual results. The following two lines of POSTSCRIPT code have the same effect; both will yield a font which is 12 units in size:

```
/Times-Roman findfont 12 scalefont
/Times-Roman findfont [12 0 0 12 0 0] makefont
```

Any **scalefont** operation can also be accomplished with the **makefont** operator but the converse is not true. In cases where a uniform scaling of the font is desired, it is better to use the **scalefont** operator.

The zero-eth and third elements of the matrix (matrix and array elements in POSTSCRIPT are counted from zero) scale the font. The zeroeth element controls scaling in the x-dimension while the third element controls scaling in the y-dimension.

The **makefont** operator can be used with POSTSCRIPT fonts to create fonts which appear condensed or extended. (Usually condensed or extended fonts are different designs not just an optical scaling.) For example, the following will give us the condensed 12 point font shown in Figure 12:

```
/Times-Roman findfont [9 0 0 12 0 0] makefont
```

The third element in the matrix is 12 since we want a 12 point font. The zeroeth element in the matrix is 9, a value less than 12, since we wish to condense the font in the x-dimension. More examples of condensed and extended fonts appear in Program 13, page 44.

Condensed Times Roman

Figure 12

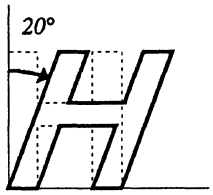


Figure 13

The **makefont** operator can also be used to create obliqued fonts. The degree to which a font is obliqued is measured by the angle between the vertical (90 degrees) and the degree of slant desired (see Figure 13). The general form for the matrix used to create an obliqued font is:

[*a* 0 *b a* 0 0]
where *a* is the point size and *b* = *a* x tan(angle).

EXAMPLE: Determine the matrix that will yield a 12 point font obliqued 20 degrees:

tan(20) = 0.36397,
12 x 0.36397 = 4.36764
the resulting matrix is [12 0 4.36764 12 0 0]
or [12 0 20 sin 20 cos div 12 mul 12 0 0]

Type is one of the most eloquent
means of expression in every epoch
of style. Next to architecture, it
gives the most characteristic
portrait of a period and the most
severe testimony of a nation's
intellectual status. Peter Behrens

Condensed, Extended and Obliqued Text 13

```
/master /Times-Roman findfont def
```
Set up a standard 1 unit font to be used later in makefont-setfont operations.

```
master [12 0 0 12 0 0] makefont setfont
234 504 moveto
(Type is one of the most eloquent) show
```
Create a uniformly scaled 12 point font.

```
master [9 0 0 12 0 0] makefont setfont
234 488 moveto
(means of expression in every epoch) show
```
Create a condensed 12 point font.

```
master [7 0 0 12 0 0] makefont setfont
234 472 moveto
(of style. Next to architecture, it) show
```
Create a very condensed 12 point font.

```
master [15 0 0 12 0 0] makefont setfont
234 456 moveto
(gives the most characteristic) show
```
Create an extended 12 point font.

```
master [17 0 0 12 0 0] makefont setfont
234 440 moveto
(portrait of a period and the most) show
```
Create a very extended 12 point font.

```
master [12 0 4.36764 12 0 0] makefont
  setfont
234 424 moveto
(severe testimony of a nation's) show
```
Create a 20 degree obliqued 12 point font.

```
master [12 0 -4.36764 12 0 0] makefont
  setfont
234 408 moveto
(intellectual status. Peter Behrens) show
```
Create a -20 degree (backward) oblique 12 point font.

```
showpage
```

In every period there have been better or worse types employed in better or worse ways. The better types employed in better ways have been used by the educated printer acquainted with standards and history, directed by taste and a sense of the fitness of things, and facing the industrial conditions and the needs of his time. Such men have made of printing an art. The poorer types and methods have been emplyed by printers ignorant of standards and caring alone for commercial success. To these, printing has been simply a trade. The typography of a nation has been good or bad as one or other of these classes had the supremacy. And to-day any intelligent printer can educate his taste, so to choose types for his work and so to use them, that he will help printing to be an art rather than a trade. Daniel Berkeley Updike.

A Simple Line Breaking Algorithm 14

```
/wordbreak ( ) def
```

Constant used for word breaks (ASCII space).

```
/BreakIntoLines
  { /proc exch def
    /linelength exch def
    /textstring exch def
```

The procedure "BreakIntoLines" takes a string of text and breaks it up into a series of lines, each no longer than the maximum line length. The algorithm breaks lines at word breaks (spaces) only. "BreakIntoLines" takes three arguments: the string of text, the maximum line length and a procedure to be executed each time the end of a line has been found. The procedure should be written so that it takes one argument, a string containing the current line.

```
    /breaklen wordbreak stringwidth pop
       def
```

Get the typeset length of a word break in the current font.

```
    /curlen 0 def
    /lastwordbreak 0 def

    /startchar 0 def

    /restoftext textstring def
```

"curlen" is the current typeset length of the current line.
"lastwordbreak" is the index into the string of text of the most recent word break encountered.
"startchar" is the index of the first character on the current line.
"restoftext" is a temporary variable that holds the remaining results of the "search" operator (see loop below).

```
    { restoftext wordbreak search
        {/nextword exch def pop
         /restoftext exch def
         /wordlen nextword stringwidth
           pop def
         curlen wordlen add linelength lt
           { /curlen curlen wordlen add
             breaklen add def }
           { textstring startchar
             lastwordbreak startchar sub
             getinterval proc
             /startchar lastwordbreak def
             /curlen wordlen breaklen
               add def
           } ifelse
         /lastwordbreak lastwordbreak
           nextword length add 1 add def
        }
        { pop exit }
        ifelse
    } loop
```

The basic strategy for breaking lines is to search the string of text (contained in "restoftext") for the next occurring word break. The pre-string returned by the "search" operator is the word preceding the word break. The post-string returned gets assigned to "restoftext."
If the length of the word returned by the "search" operator exceeds the maximum line length when added to the length of the current line then the substring spanning the current line (from the first character on the line to the most recent word break) is obtained and passed as an argument to the user's procedure. Otherwise the length of the current line is incremented by the width of the word.

The "lastwordbreak" variable is always updated to index into the text string at the position of the most recent word break.
The last word in the text has been found when the "search" operator fails to match the word break pattern. This terminates the loop.

```
    /lastchar textstring length def
    textstring startchar lastchar
      startchar sub getinterval proc
  } def
```

Don't forget to process the last line.

A Simple Line Breaking Algorithm, cont. 14

```
/Times-Roman findfont 12 scalefont setfont
/yline 552 def

(In every period there have been better or\
 worse types employed in better or worse\
 ways. The better types employed in better\
 ways have been used by the educated\
 printer acquainted with standards and\
 history, directed by taste and a sense of\
 the fitness of things, and facing the\
 industrial conditions and the needs of\
 his time. Such men have made of printing\
 an art. The poorer types and methods have\
 been emplyed by printers ignorant of\
 standards and caring alone for commercial\
 success. To these, printing has been\
 simply a trade. The typography of a\
 nation has been good or bad as one or\
 other of these classes had the supremacy.\
 And to-day any intelligent printer can\
 educate his taste, so to choose types for\
 his work and so to use them, that he will\
 help printing to be an art rather than a\
 trade.  Daniel Berkeley Updike.)
   306
   { 236 yline moveto show
     /yline yline 14 sub def}



   BreakIntoLines


showpage
```

Below is an example of the how the "BreakIntoLines" procedure might be used.

Use a line length of 306 points.

The procedure provided to "BreakIntoLines" has been written so that it takes a string as its argument. The procedure uses a global variable "yline" to keep track of vertical positioning on the page. It moves to a specified position on the page, shows the string in the current font and then updates the vertical position.

EXERCISE FOR THE READER: If the user specifies a short enough line length, it is possible for the typeset width of a single word to exceed the maximum line length. Modify this algorithm to handle this event gracefully.

TEXT POSITIONED VERTICALLY

SHOULD BE CENTERED ON

A COMMON CENTER LINE.

VERTICAL TEXT IN CAPITAL

LETTERS HAS MORE EVEN

spacing than lower case letters.

Vertical Text 15

```
/vshowdict 4 dict def
```

Establish a dictionary which can later be used as a local work space for definitions.

```
/vshow
  { vshowdict begin
    /thestring exch def
    /lineskip exch def
    thestring
      {
      /charcode exch def
      /thechar ( ) dup 0 charcode put def

      0 lineskip neg rmoveto
      gsave
        thechar stringwidth pop 2 div neg
          0 rmoveto
        thechar show
      grestore
      } forall
    end
  } def
```

vshow will display text vertically, centering it on a common center line. vshow takes two arguments, the lineskip between letters and the string to be shown vertically.

The forall command allows us to repeat the same procedure for each character in the string.
forall pushes the character code onto the operand stack.
Convert the character code to a string.

Move down by the lineskip amount.

Move left by half of the character width.

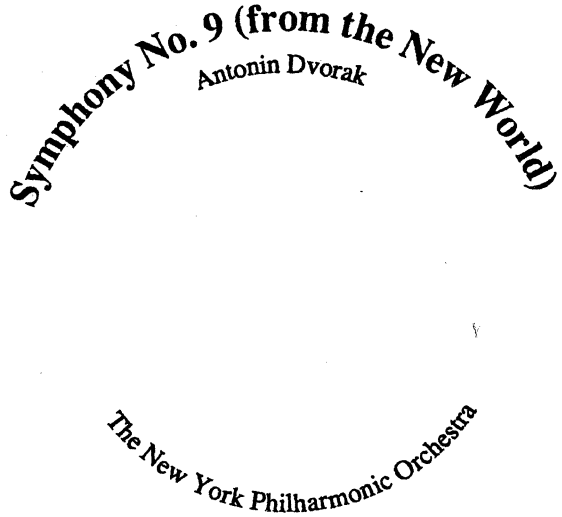
Display the character.

```
/Helvetica findfont 12 scalefont
  setfont

72 555 moveto
12 (TEXT POSITIONED VERTICALLY) vshow
112 555 moveto
12 (SHOULD BE CENTERED ON) vshow
152 555 moveto
12 (A COMMON CENTER LINE.) vshow
192 555 moveto
12 (VERTICAL TEXT IN CAPITAL) vshow
232 555 moveto
12 (LETTERS HAS MORE EVEN) vshow
272 555 moveto
12 (spacing than lower case letters.)
  vshow

showpage
```

Set up the font we wish to use.

Chose the starting position for the string to be shown. The text will be centered around the line x = 72 and it will begin just below the line y = 555.

Notice the order of the arguments in all of the uses of vshow: the lineskip comes first followed by the string.


Symphony No. 9 (from the New World)
Antonin Dvorak
The New York Philharmonic Orchestra

Circular Text 16

```
/outsidecircletext
  { $circtextdict begin
      /radius exch def
      /centerangle exch def
      /ptsize exch def
      /str exch def

      /xradius radius ptsize 4 div add
        def



      gsave
        centerangle str findhalfangle
          add rotate


        str
          { /charcode exch def
            ( ) dup 0 charcode put
              outsideshowcharandrotate
          } forall
      grestore
    end
  } def
```

outsidecircletext places text around a circular arc. The baseline of the text is placed on the outside of the circumference of the circle in a clockwise fashion. outsidecircletext takes four arguments: the string to be printed, the point size of the font being used, the angle around which the text should be centered and the radius of the circular arc. It assumes that the center of the circle is at (0,0).

A radius that is slightly larger than the one specified is used for computations but not for placement of characters. Using a slightly larger radius in the computations places the characters closer together, otherwise the interletter spacing is too loose.

Save the current graphics state.

Find out how much angle the string subtends and then rotate to the appropriate starting position for showing the string. (The positive x-axis now intersects the circle where the text should start.)

For each character in the string, determine its position on the circular arc and show it.

Return to the former graphics state.

```
/insidecircletext
  { $circtextdict begin
      /radius exch def
      /centerangle exch def
      /ptsize exch def
      /str exch def

      /xradius radius ptsize 3 div sub
        def

      gsave
        centerangle str findhalfangle
          sub rotate
        str
          { /charcode exch def
            ( ) dup 0 charcode put
              insideshowcharandrotate
          } forall
      grestore
    end
  } def
```

insidecircletext works very similarly to outsidecircletext except that the baseline of the text is placed on the inside of the circumference of the circle in a counter-clockwise fashion. insidecircletext takes the same four arguments as outsidecircletext.

Here we use a radius which is slightly smaller than the desired radius for computations. This forces the characters to be placed farther apart to avoid overlapping.

Circular Text, cont. 16

```
/$circtextdict 16 dict def
$circtextdict begin
  /findhalfangle
    { stringwidth pop 2 div
        2 xradius mul pi mul div 360 mul
    } def


  /outsideshowcharandrotate
    { /char exch def
      /halfangle char findhalfangle def
      gsave
        halfangle neg rotate
        radius 0 translate
        -90 rotate
        char stringwidth pop 2 div neg
          0 moveto char show
      grestore
      halfangle 2 mul neg rotate
    } def

  /insideshowcharandrotate
    { /char exch def
      /halfangle char findhalfangle def
      gsave
        halfangle rotate
        radius 0 translate
        90 rotate
        char stringwidth pop 2 div neg
          0 moveto char show
      grestore
      halfangle 2 mul rotate
    } def

  /pi 3.1415923 def
end

/Times-Bold findfont 15 scalefont setfont
306 448 translate
(Symphony No. 9 (from the New World))
  15 90 100 outsidecircletext
/Times-Roman findfont 10 scalefont setfont
(Antonin Dvorak)
  10 90 84 outsidecircletext
(The New York Philharmonic Orchestra)
  10 270 84 insidecircletext
showpage
```

findhalfangle takes one argument, a string, and finds the angle subtended by that string. It leaves the value of half of that angle on the stack. The angle is found by computing the ratio of the width of the string to the circumference of the circle and then converting that value to degrees.

This procedure shows a character upright on the outside of the circumference and then rotates clockwise by the amount of angle subtended by the width of the character.

Rotate clockwise by half the angle taken up by the width of the character and translate out to the circumference. Position character upright on outside of circumference. Center the character around the origin.

Rotate clockwise by the amount of angle subtended by the width of the character.

insideshowcharandrotate operates in a similar manner to outsideshowcharandrotate except that the direction of rotation is counter-clockwise and the characters are placed upright on the inside of the circle.

If my film makes one more person feel miserable I'll feel I've done my job. -- WOODY ALLEN

Placing Text Along an Arbitrary Path 17

```
/pathtextdict 26 dict def
```

Local storage for the procedure "pathtext."

```
/pathtext
  { pathtextdict begin
    /offset exch def
    /str exch def
```

"pathtext" will place a string of text along any path. It takes a string and starting offset distance from the beginning of the path as its arguments. Note that "pathtext" assumes that a path has already been defined and after it places the text along the path, it clears the current path like the "stroke" and "fill" operators; it also assumes that a font has been set. "pathtext" begins placing the characters along the current path, starting at the offset distance and continuing until either the path length is exhausted or the entire string has been printed, whichever occurs first. The results will be more effective when a small point size font is used with sharp curves in the path.

```
    /pathdist 0 def
    /setdist offset def
    /charcount 0 def
    gsave
      flattenpath
```

Initialize the distance we have travelled along the path.
Initialize the distance we have covered by setting characters.
Initialize the character count.

Reduce the path to a series of straight line segments. The characters will be placed along the line segments in the "linetoproc."

```
      {movetoproc} {linetoproc}
        {curvetoproc} {closepathproc}
        pathforall
```

The basic strategy is to process the segments of the path, keeping a running total of the distance we have travelled so far (pathdist). We also keep track of the distance taken up by the characters that have been set so far (setdist). When the distance we have travelled along the path is greater than the distance taken up by the set characters, we are ready to set the next character (if there are any left to be set). This process continues until we have exhausted the full length of the path.

```
    grestore
    newpath
    end
  } def
```

Clear the current path.

```
pathtextdict begin
/movetoproc
  { /newy exch def /newx exch def
    /firstx newx def /firsty newy def
```

"movetoproc" is executed when a moveto component has been encountered in the pathforall operation.
Remember the "first point" in the path so that when we get a "closepath" component we can properly handle the text.

```
    /ovr 0 def
    newx newy transform
    /cpy exch def /cpx exch def
  } def
```

Explicitly keep track of the current position in device space.

```
/linetoproc
```

"linetoproc" is executed when a lineto component has been encountered in the pathforall operation.

Placing Text Along an Arbitrary Path, cont. 17

```
  { /oldx newx def /oldy newy def
    /newy exch def /newx exch def
    /dx newx oldx sub def
    /dy newy oldy sub def
    /dist dx dup mul dy dup mul add
      sqrt def
    /dsx dx dist div ovr mul def
    /dsy dy dist div ovr mul def
    oldx dsx add oldy dsy add transform
    /cpy exch def /cpx exch def
    /pathdist pathdist dist add def
    { setdist pathdist le

        { charcount str length lt
            {setchar} {exit} ifelse }
        { /ovr setdist pathdist sub def
          exit }
        ifelse
    } loop
  } def

/curvetoproc
  { (ERROR: No curveto's after flattenpath!)
    print
  } def

/closepathproc
  { firstx firsty linetoproc
    firstx firsty movetoproc
  } def



/setchar
  { /char str charcount 1 getinterval def
    /charcount charcount 1 add def
    /charwidth char stringwidth pop def
    gsave
      cpx cpy itransform translate
      dy dx atan rotate
      0 0 moveto char show
      currentpoint transform
      /cpy exch def /cpx exch def
    grestore
    /setdist setdist charwidth add def
  } def
end
```

Update the old point.
Get the new point.

Calculate the distance between the old and the new point.

dsx and dsy are used to update the current position to be just beyond the width of the previous character.

Update the current position.
Increment the distance we have travelled along the path.
Keep setting characters along this path segment until we have exhausted its length.
As long as there are still characters left in the string, set them.

Keep track of how much we have overshot the path segment by setting the previous character. This enables us to position the origin of the following characters properly on the path.

"curvetoproc" is executed when a curveto component has been encountered in the pathforall operation. It prints an error message since there shouldn't be any curveto's in a path after the flattenpath operator has been executed.

"closepathproc" is executed when a closepath component has been encountered in the pathforall operation. It simulates the action of the operator "closepath" by executing "linetoproc" with the coordinates of the most recent "moveto" and then executing "movetoproc" to the same point.

"setchar" sets the next character in the string along the path and then updates the amount of path we have exhausted.
Increment the character count.
Find the width of the character.

Translate to the current position in user space.
Rotate the x-axis to coincide with the current segment.

Update the current position before we restore ourselves to the untransformed state.
Increment the distance we have covered by setting characters.

Placing Text Along an Arbitrary Path, cont. 17

```
/Helvetica findfont 11.5 scalefont setfont
```

Set up the font we wish to use.

```
newpath
  200 500 50 0 270 arc
  200 80 add 500 50 270 180 arc
```

Define the path along which we wish to place the text.

```
(If my film makes one more person feel\
 miserable I'll feel I've done my job.\
 -- WOODY ALLEN) 40 pathtext
```

Print the string along the path at an offset of 40 points.

```
newpath
  165 360 moveto 315 360 lineto
  315 430 lineto 165 430 lineto
  closepath
  315 390 moveto 355 375 lineto
  355 415 lineto 315 400 lineto
1.5 setlinewidth stroke
```

Draw an outline shape suggestive of a movie camera.
Draw the box part.

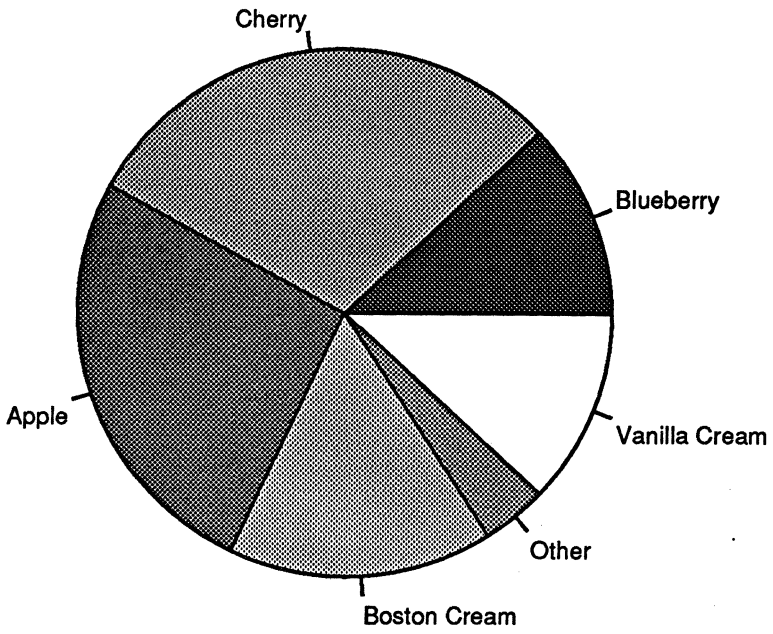
Draw the lens part.

A PROBLEM FOR THE READER: This algorithm places characters along the path according to the origin of each character. Rewrite the algorithm so that the characters are placed according to the center of their width. This will yield better results around sharp curves and when larger point sizes are used.

```
showpage
```

Miscellaneous

The following is a collection of unrelated but very useful POSTSCRIPT programs.



January Pie Sales

Drawing a Pie Chart 18

```
/PieDict 24 dict def
PieDict begin
  /DrawSlice
    { /grayshade exch def
      /endangle exch def
      /startangle exch def
      /thelabel exch def

      newpath
        0 0 moveto
        0 0 radius startangle endangle arc
      closepath

      1.415 setmiterlimit

      gsave grayshade setgray fill grestore
      stroke


      gsave
        startangle endangle add 2 div
          rotate
        radius 0 translate
        newpath
          0 0 moveto labelps .8 mul 0 lineto
        stroke
        labelps 0 translate
        0 0 transform
      grestore
      itransform
      /y exch def /x exch def
      x y moveto




      x 0 lt
        { thelabel stringwidth pop neg
            0 rmoveto
        } if
      y 0 lt { 0 labelps neg rmoveto } if
      thelabel show
    } def
```

Local storage for "DrawPieChart" and its related procedures.

DrawSlice draws an outlined and filled-in pie slice. It takes four operands: the label for this particular pie slice, the starting angle for the slice, the ending angle for the slice and the shade of gray the slice should be.

Create a path which will draw a pie slice.

This guarantees that when we outline the pie slices with a stroke that we will not get a spike on the interior angles.

Fill the pie slice path with the appropriate gray color. By using gsave and grestore we don't lose the current path. Since PostScript paints color onto the page, it is very important that we fill the pie slice first and then outline it with a stroke.

Draw the tick mark and place the label:

Find the center of the pie slice and rotate so that the x-axis coincides with this center.

Translate the origin out to the circumference.

Draw the tick-mark.

Move the origin out a little beyond the circumference.

Next we wish to place the label at the current origin. If we simply draw the text on the page now, it would come out rotated. Since this is not desired we avoid it by returning to the previous unrotated coordinate system. Before returning, though, we would like to remember the position of the current origin on the printed page. We will accomplish this by using the transform and itransform operators. Performing a transform on the origin pushes the coordinates of the origin in device space onto the operand stack. Performing a grestore returns us to the previous unrotated coordinate system. Next we perform an itransform on the two device coordinates left on the stack to determine where we are in the current coordinate system.

Make some adjustments so that the label text won't collide with the pie slice.

Drawing a Pie Chart, cont. 18

```
  /findgray
    { /i exch def /n exch def
      i 2 mod 0 eq
        { i 2 div n 2 div round add n div }
        { i 1 add 2 div n div }
        ifelse
    } def
end

/DrawPieChart
  { PieDict begin
      /radius exch def
      /ycenter exch def /xcenter exch def
      /PieArray exch def
      /labelps exch def /titleps exch def
      /title exch def

      gsave
        xcenter ycenter translate
        /Helvetica findfont titleps
          scalefont setfont
        title stringwidth pop 2 div neg
          radius neg titleps 3 mul sub
          moveto title show
        /Helvetica findfont labelps
          scalefont setfont
        /numslices PieArray length def
        /slicecnt 0 def
        /curangle 0 def

        PieArray
          { /slicearray exch def
            slicearray aload pop
            /percent exch def
            /label exch def
            /perangle percent 360 mul def
            /slicecnt slicecnt 1 add def
            label curangle
              curangle perangle add
              numslices slicecnt findgray
              DrawSlice
            /curangle curangle perangle add
              def
          } forall
      grestore
    end
  } def
```

Procedure findgray calculates the gray value for a slice. It takes two arguments: the total number of slices and the current slice number (Given that there are n pie slices, the slices are "numbered" from 1 to n). The gray values for the pie slices range evenly from white to black (i.e. - the values provided to setgray range from (n/n, n-1/n, ..., 1/n)). Since we don't want similar values of gray next to each other, findgray "shuffles" the possible combinations of gray.

DrawPieChart takes seven arguments: the title of the pie chart, the point size to print the title in, the point size to print the labels for each slice in, a special array (described below where DrawPieChart is called), the (x,y) center of the pie chart and the radius of the pie chart.

Translate the coordinate system origin to center of pie chart. Print the title of the pie chart in Helvetica.

Center the title below the pie chart.

Print the individual pie slice labels in Helvetica

A "loop" variable that keeps track of the angle of arc to begin each pie slice at.
Repeat the following for each element in the PieArray.

Push the label and percentage onto the stack.

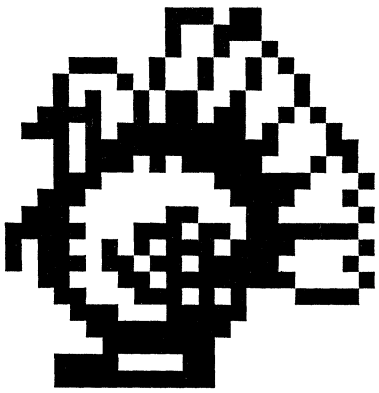
Convert the percentage into degrees of angle.

Update the current starting angle.

Drawing a Pie Chart, cont. 18

```
(January Pie Sales) 18 9
  [ [(Blueberry) .12 ]
    [(Cherry) .30 ]
    [(Apple) .26 ]
    [(Boston Cream) .16 ]
    [(Other) .04 ]
    [(Vanilla Cream) .12 ]
  ] 306 396 100 DrawPieChart
showpage
```

The pie array is an array of arrays. Each array in the pie array contains a string denoting the label to be printed next to the pie slice followed by a real number indicating the percentage of the pie represented by this particular slice.



Using the image Operator 19

```
/concatprocs
  { /proc2 exch cvlit def
    /proc1 exch cvlit def

    /newproc proc1 length proc2 length add
      array def
    newproc 0 proc1 putinterval
    newproc proc1 length proc2 putinterval
    newproc cvx
  } def
/inch { 72 mul } def
/picstr 3 string def

/imageturkey
  { 24 23 1 [24 0 0 -23 0 23]
      { currentfile picstr
        readhexstring pop } image
  } def
```

"concatprocs" takes two procedure bodies as arguments and concatenates them into one procedure body. The resulting procedure body is left on the operand stack. "concatprocs" will be used in constructing a new transfer function below.
Create a new array large enough to accomodate both procedures.
Place the first procedure at the beginning of the new one.
Place the second procedure at the end of the new one.
Now make this array into an executable object.

String to read the hex strings into (each row is 3 bytes long).

The procedure "imageturkey" will read the image (as hex strings) from this file and show it on the page. The image of the turkey is represented as one bit per sample. It is 24 samples wide by 23 samples high and it's first sample is in the upper left corner of the source image.

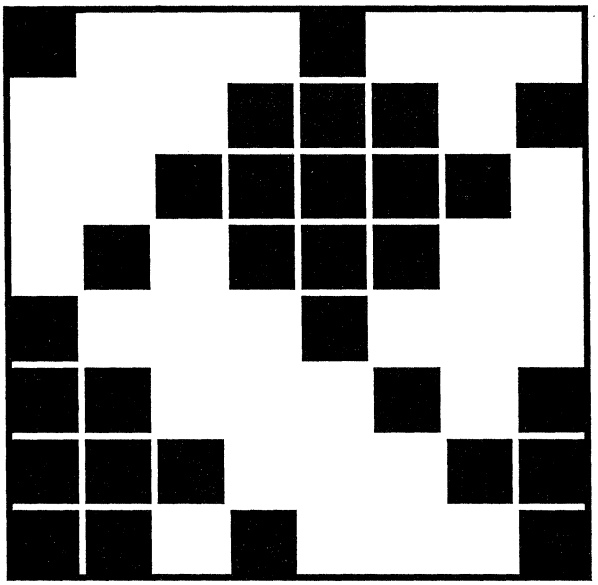
The image we generate will be mapped to the unit square in user space. This unit square has it's lower left corner at the origin and extends 1 unit in the positive x and y directions. If we want the image to appear in the center of the page we must translate the user space origin near the center of the page. If we want an image that is larger than the default unit square, we must scale the user space.

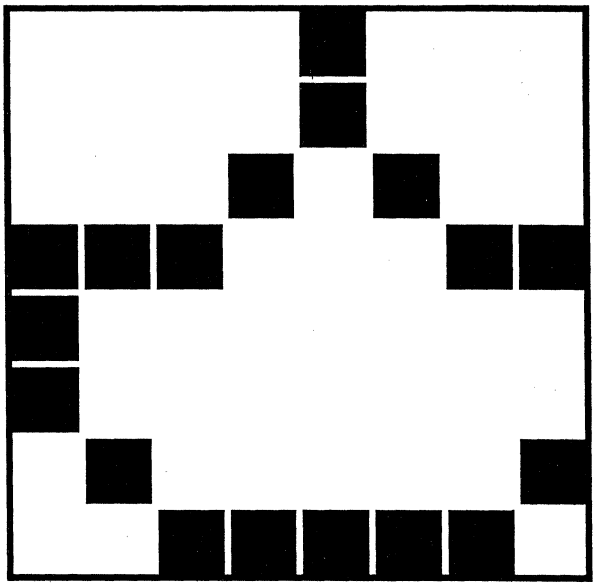
```
gsave
  3 inch 4 inch translate
  2 inch dup scale
  {1 exch sub} currenttransfer concatprocs
    settransfer
```

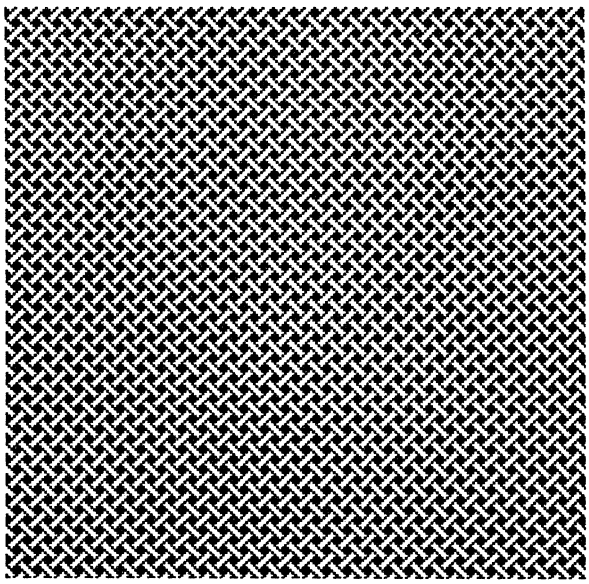
Isolate the effects of the "settransfer."
Position the unit square on the page.
Scale it to be 2 inches square.
Since the source samples for our image specify a reverse image (that is, the samples that correspond to "black" are specified as 1's rather than 0's) we specify a transfer function to reverse this effect. Since some output devices have complex transfer functions we don't simply want to set the transfer function. Instead we want to concatenate our new transfer function with the existing one to achieve our results.

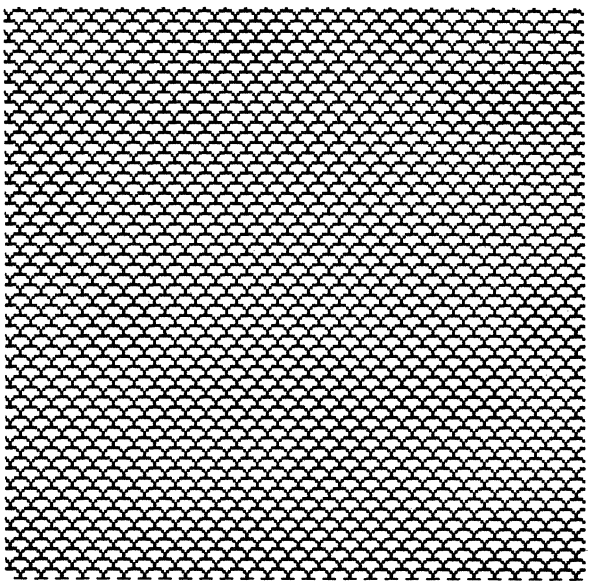
```
  imageturkey
  003B00 002700 002480 0E4940
  114920 14B220 3CB650 75FE88
  17FF8C 175F14 1C07E2 3803C4
  703182 F8EDFC B2BBC2 BB6F84
  31BFC2 18EA3C 0E3E00 07FC00
  03F800 1E1800 1FF800
grestore
showpage
```

As soon as "imageturkey" is executed, the "currentfile ... readhexstring" sequence will begin reading bytes from this file. The safest way to synchronize reading from the program file with the PostScript interpreter's own reading of this file is to embed the reading commmands in a procedure, then place that procedure name followed by a "carriage return" followed by the bytes to be read in the file.
The "image" command reads exactly the number of bytes we supplied, and the interpreter picks up its reading here.









Bit Pattern Screens 20

```
/bitison
  {/ybit exch def /xbit exch def
  bstring ybit bwidth mul
  xbit 8 idiv add get
  1 7 xbit 8 mod sub bitshift
  and 0 ne} def
```

This function does bit addressing within a string whose dimensions and contents have been stored into the variables 'bstring', 'bpside', and 'bwidth'. 'bstring' holds the bit pattern, 'bwidth' is an integer giving the pattern width in bytes, and 'bpside' is an integer giving the width and height of the pattern in bits. This function returns 'true' if the bit at position (xbit, ybit) in bstring is on.

```
/enlargebits
  {/bwidth exch def
  /bpside exch def
  /bstring exch def
  0.08 setlinewidth
  0 1 bpside 1 sub
    {/y exch def
    0 1 bpside 1 sub
      {/x exch def
      x y bitison
        { gsave
            x y translate
            newpath
              0 0 moveto 0 1 lineto
              1 1 lineto 1 0 lineto
            closepath
            gsave 0 setgray fill grestore
            1 setgray stroke grestore
        } if
      } for
    } for
  0 0 moveto 0 bpside lineto
  bpside dup lineto bpside 0 lineto
  closepath 0 setgray stroke
  } def
```

enlargebits prints an enlarged bit pattern, so that we can illustrate the bit patterns that we will use in 'setpattern' below. This routine sets up the global variables needed by 'bitison' defined above, and prints a black square for each on bit. The squares are one unit in size; the caller of this routine should scale the units appropriately. Note that the earlier bits in the pattern are printed in the lower positions. The high order bit of the first byte of the pattern is the lower left bit, and the low order bit of the last byte in the pattern is the upper right bit.

```
/setpattern
  {/freq exch def
   /bwidth exch def
   /bpside exch def
   /bstring exch def
   /onbits 0 def /offbits 0 def
   freq 0
     {/y exch def /x exch def
      /xindex x 1 add 2 div
         bpside mul cvi def
      /yindex y 1 add 2 div
         bpside mul cvi def
      xindex yindex bitison
```

This routine sets up the halftone screen machinery so that a repeating bitmap pattern will be used for subsequent output. The screens are device dependent, i.e., the caller of this routine must understand the device resolution and rotation.

Here, we begin to set up the arguments to 'setscreen'. This begins the screen function argument to 'setscreen'. First, we transform the (x, y) position into a position to address into the bit pattern.

Bit Pattern Screens, cont. 20

```
        {/onbits onbits 1 add def 1}
        {/offbits offbits 1 add def
         0}
        ifelse
      } setscreen
   {} settransfer

   offbits offbits onbits add div
     setgray

  } def

/pat1 <d1e3c5885c3e1d88> def
/pat2 <3e418080e3140808> def

/inch {72 mul} def

/showpattern
  {/pat exch def
  pat 8 1 300 32 div setpattern
  0 0 moveto 3 inch 0 lineto
  3 inch dup lineto 0 3 inch lineto
  closepath fill
  0 3.5 inch translate 3 8 div inch
  dup scale
  pat 8 1 enlargebits
  } def

gsave 1 inch 1.25 inch translate
pat1 showpattern grestore

gsave 4.5 inch 1.25 inch translate
pat2 showpattern grestore

showpage
```

If that bit is on, count it and return a high value.
If that bit is off, count it and return a low value.

Don't allow correction of gray values, because we want to set the gray exactly according to the off-bit, total-bits ratio.

By setting the gray this way, exactly the number of on bits will turn on in the screen.
Finish the definition of 'setpattern'. Use hex-string notation to set the bit patterns.

Define a routine to make a simple demonstration of the above functions. Take a pattern and display it as enlarged bits, and in use filling an area.

On the left, show a weaving pattern.

On the right, show a fish scale pattern.

NOTE: This is not the actual output page produced by the following POSTSCRIPT program. The rectangles are scaled down versions of the 8 1/2" by 11" pages generated by the program.

Making a Poster 21

```
/BigPrint
  { /rows exch def
    /columns exch def
    /bigpictureproc exch def
```

"BigPrint" takes a large picture (larger than 8.5" by 11") and prints it on several pages according to the number of rows and columns specified. Imagine superimposing a grid composed of the specified number of rows and columns on the large image. Then each rectangle in the grid represents an 8.5" by 11" page to be printed. "BigPrint" takes three arguments: a procedure representing the large picture, the number of columns and the number of rows.

```
    newpath
      leftmargin botmargin moveto
      0 pageheight rlineto
      pagewidth 0 rlineto
      0 pageheight neg rlineto
    closepath clip
```

Set up a clipping region for the page we will print on. Since most printers cannot print to the very edge of the paper, we will explicitly set up the clipping boundary so that it lies within the printing boundaries of the printer and we will compensate for this when we print the large image so that all parts of the image are indeed printed.

```
    leftmargin botmargin translate
```

Readjust the origin on the page so that it coincides with the origin of the clipping boundary.

```
    0 1 rows 1 sub
      { /rowcount exch def
        0 1 columns 1 sub
          { /colcount exch def
            gsave
              pagewidth colcount mul neg
                pageheight rowcount mul neg
                translate

              bigpictureproc
              copypage erasepage
            grestore
          } for
      } for
  } def
```

For each row of pages...

For each page within that row...

Translate the large picture so that the desired section will be imaged on the printed page. We must translate the large picture in the negative directions so that the lower left corner of the section to be printed always coincides with the origin.
Execute the large picture.
Since the "showpage" operator has the side effect of executing the "initgraphics" operator (which would reset the clipping region), we perform a "copypage erasepage" sequence instead. The "copypage" prints the page and the "erasepage" clears the current output page.

```
/inch {72 mul} def
/leftmargin .5 inch def
/botmargin .25 inch def
/pagewidth 7.5 inch def
/pageheight 10 inch def
```

These are the dimensions of the clipping boundary.

Making a Poster, cont. 21

```
{ gsave
```

This procedure draws a large sign with a border around it. The sign is 22.5 inches wide and 19.5 inches high so that it will fit comfortably on 6 8.5 inch by 11 inch pages (the final result will be 2 rows of pages high and 3 columns of pages wide).

```
    20 setflat
```

Since the letters being printed are so large, we can increase the flatness parameter used without degrading the quality of the image. This will significantly decrease the computation time required.

```
    /Times-Roman findfont 500 scalefont
      setfont
    2.5 inch 11 inch moveto
    (SALE) show
    /Times-Roman findfont 350 scalefont
      setfont
    1.45 inch 4 inch moveto
    .5 setgray (50%) show
    0 setgray ( OFF) show
    newpath
```

Specify the path for the border.

```
      .5 inch 18 inch moveto
      22 inch 18 inch lineto
      22 inch 2 inch lineto
      .5 inch 2 inch lineto
    closepath
    gsave
      .75 inch setlinewidth stroke
```

First paint the border with a thick black stroke.

```
    grestore
    10 setlinewidth 1 setgray stroke
  grestore
} 3 2 BigPrint
```

Then paint a thin white stroke down the cneter of the border. Print the large picture on a total of 6 pages. The image is three columns of pages wide and 2 rows of pages high.

Customized Fonts

Although a large variety of fonts are available through POSTSCRIPT, there are situations when users may wish to modify the existing fonts or create new fonts of their own. This section presents several examples of modifying existing fonts to change their rendering style or to change the encoding of the characters in a font. There is also an example of creating an entirely new font.

The basic underlying structure of a font is the *font dictionary*. When fonts are modified, the entries in the font dictionary are changed. When new fonts are created, certain crucial entries in the font dictionary must be provided. For details on the entries in a font dictionary and how to modify them, please refer to the "Font Machinery Appendix" of the **POSTSCRIPT Language Manual**.

The basic strategy for modifying an existing font is to create an entirely new font dictionary and to copy all the references to entries in the original font dictionary, except for the FID entry, into the new dictionary. Then the appropriate fields should be modified. A **definefont** operation should then be performed on the new font dictionary to make it into a POSTSCRIPT font.

The most important thing to remember when modifying an existing font is to change the FontName and FID fields in the new font dictionary. The FID field automatically gets created when the **definefont** operator is executed. The FontName field must be explicitly changed. The same name which appears in the FontName field should be provided as an argument to the **definefont** operator. *The FontName should always be a unique name.*

One useful modification which can be made to an existing font is to convert it to an outlined font. This is illustrated in Program 22, page 78. Another common modification is to change the encoding vector. The encoding vector is a mapping of character codes (0-255) to character names. Most POSTSCRIPT fonts are encoded according to the POSTSCRIPT default encoding, although there are cases where other encodings are desireable. One such case is the EBCDIC encoding. An example of re-encoding a font to use the EBCDIC encoding is demonstrated in Program 23, page 80. Some POSTSCRIPT fonts contain characters which are not encoded in the default encoding vector such as accented characters. In order to print text in a foreign language, it's necessary to re-encode the font to include the desired

accented characters in the encoding vector. In this case, only a small portion of the encoding vector needs to be changed; Program 24 on page 84 shows an example of re-encoding a small portion of the encoding vector.

The last programming example, Program 25 on page 88, in this section demonstrates how to build a font from scratch. It shows how to define all the required font dictionary entries and how to define character shapes. The font combines analytic character shapes and bitmap character shapes to demonstrate the flexibility of POSTSCRIPT fonts.

outlineOutline

outlineOutline

outlineOutline

Making an Outlined Font 22

```
/makeoutlinedict 5 dict def
/MakeOutlineFont
  { makeoutlinedict begin
    /strokeweight exch def
    /newfontname exch def
    /basefontname exch def

    /basefontdict basefontname findfont def

    /outfontdict basefontdict maxlength
      1 add dict def

    basefontdict
      { exch dup /FID ne
          {exch outfontdict 3 1 roll put}
          {pop pop}
          ifelse
      } forall
    outfontdict /FontName newfontname put
    outfontdict /PaintType 2 put
    outfontdict /StrokeWidth strokeweight
      put
    newfontname outfontdict definefont pop
    end
  } def

/Helvetica-Bold /Helvetica-Outline0 0
  MakeOutlineFont
/Helvetica-Outline0 findfont 24 scalefont
  setfont 72 542 moveto (outline) show
/Helvetica-Outline0 findfont 36 scalefont
  setfont (outline) show

/Helvetica-Bold /Helvetica-Outline1
  1000 36 div MakeOutlineFont
/Helvetica-Outline1 findfont 24 scalefont
  setfont 72 502 moveto (outline) show
/Helvetica-Outline1 findfont 36 scalefont
  setfont (outline) show

/Helvetica-Bold /Helvetica-Outline2
  1000 24 div MakeOutlineFont
/Helvetica-Outline2 findfont 24 scalefont
  setfont 72 462 moveto (outline) show
/Helvetica-Outline2 findfont 36 scalefont
  setfont (outline) show
showpage
```

Local storage for the procedure MakeOutlineFont. MakeOutlineFont takes one of PostScript's standard filled fonts and makes an outlined font out of it. It takes three arguments: the name of the font on which to base the outline version, the new name for the outline font and a strokeweight to use on the outline.

Get the dictionary of the font on which the outline version will be based.
Create a dictionary to hold the description for the outline font. Make it one entry larger to accomodate an entry for the strokewidth used on the outline.
Copy all the entries in the base font dictionary to the outline dictionary except for the FID.

Ignore FID pair.

Insert the new name into the dictionary.
Change the paint type to outline.
Insert the strokeweight into the dictionary.

Now make the outline dictionary into a PostScript font. We will ignore the modified dictionary returned on the stack by the definefont operator.

We will create an outline font based on Helvetica-Bold named Helvetica-Outline0. By specifying a stroke weight of zero, we will always get a one pixel wide outline around each character, no matter what the font's point size.

Here we are creating a font with a heavier stroke weight. The stroke weight is always specified in the character coordinate system (1000 units). The value specified here, 1000/36 will yield a one point wide outline when the font is scaled to 36 points in size. Note that this outline weight changes with different point sizes.

A strokeweight value of 1000/24 yields a one point wide outline when the font is scaled to 24 points in size. It yields a 1.5 point outline when the font is scaled to 36 points in size (36/24 = 1.5).

Decimal Number	Standard Char	EBCDIC Char
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		
33	!	
34	"	
35	#	
36	$	
37	%	
38	&	
39	'	
40	(	
41	)	
42	*	
43	+	
44	,	
45	-	
46	.	
47	/	
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	

Decimal Number	Standard Char	EBCDIC Char
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	
73	I	
74	J	¢
75	K	.
76	L	<
77	M	(
78	N	+
79	O	|
80	P	&
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	!
91	[	$
92	\	*
93	]	)
94	^	;
95		~
96	‾	-
97	a	/
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	,
108	l	%
109	m	_
110	n	>
111	o	?
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	:
123	{	#
124	|	@
125	}	'
126	~	=
127		"

Decimal Number	Standard Char	EBCDIC Char
128		
129		a
130		b
131		c
132		d
133		e
134		f
135		g
136		h
137		i
138		
139		
140		
141		
142		
143		
144		
145		j
146		k
147		l
148		m
149		n
150		o
151		p
152		q
153		r
154		
155		
156		
157		
158		
159		
160		
161	¡	
162	¢	s
163	£	t
164	/	u
165	¥	v
166	ƒ	w
167	§	x
168	¤	y
169	'	z
170	“	
171	«	
172	‹	
173	›	
174	fi	
175	fl	
176		
177	–	
178	†	
179	‡	
180	·	
181		
182	¶	
183	•	
184	¸	
185	‟	
186	»	
187	…	
188	‰	
189		
190		
191	¿	

Decimal Number	Standard Char	EBCDIC Char
192		
193	`	A
194	´	B
195	^	C
196	~	D
197	¯	E
198	˘	F
199	˙	G
200	¨	H
201		I
202	°	
203	¸	
204	˝	
205	˛	
206	ˇ	
207		
208	—	
209		J
210		K
211		L
212		M
213		N
214		O
215		P
216		Q
217		R
218		
219		
220		
221		
222		
223		
224		
225	Æ	
226		S
227	ª	T
228		U
229		V
230		W
231		X
232	Ł	Y
233	Ø	Z
234	Œ	
235	º	
236		
237		
238		
239		
240		0
241	æ	1
242		2
243		3
244		4
245	ı	5
246		6
247		7
248	ł	8
249	ø	9
250	œ	
251	ß	
252		
253		
254		
255		

Re-encoding an Entire Font 23

```
/reencodedict 5 dict def
/ReEncode
  { reencodedict begin
    /newencoding exch def
    /newfontname exch def
    /basefontname exch def


    /basefontdict basefontname findfont def

    /newfont basefontdict maxlength dict def

    basefontdict
     {exch dup /FID ne dup /Encoding ne and
        { exch newfont 3 1 roll put }
        { pop pop }
        ifelse
     } forall

    newfont /FontName newfontname put
    newfont /Encoding newencoding put

    newfontname newfont definefont pop
    end
   } def
```

Local storage for the procedure "ReEncode."
ReEncode generates a new font given the name of the font to be re-encoded, a new name, and a new encoding vector. ReEncode copies the existing font dictionary, replacing the FontName and Encoding fields, then generates a new FID and enters the new name in FontDirectory with the "definefont" operator. The new name provided can later be used in a "findfont" operation.

Get the dictionary of the font on which the re-encoded version will be based.

Create a dictionary to hold the description for the re-encoded font.

Copy all the entries in the base font dictionary to the new dictionary except for the FID and Encoding fields.

Ignore FID and Encoding pairs.

Install the new name and the new encoding vector in the font.

Now make the re-encoded font dictionary into a PostScript font. We will ignore the modified dictionary returned on the operand stack by the "definefont" operator.

```
/EBCDIC 256 array def
0 1 255 { EBCDIC exch /.notdef put } for
EBCDIC
  dup 64 /space put

  dup 74 /cent put
  dup 75 /period put
  dup 76 /less put
  dup 77 /parenleft put
  dup 78 /plus put
  dup 79 /bar put
  dup 80 /ampersand put

  dup 90 /exclam put
  dup 91 /dollar put
  dup 92 /asterisk put
  dup 93 /parenright put
  dup 94 /semicolon put
  dup 95 /asciitilde put
  dup 96 /hyphen put
  dup 97 /slash put
```

To illustrate how the ReEncode procedure is used, we will re-encode one of the standard PostScript fonts to support the EBCDIC encoding. (The EBCDIC encoding used is referenced in "IBM System/360: Principles of Operation," Appendix F.) The first step in doing this is to define an array containing that encoding. This array is referred to as an "encoding vector." The encoding vector should be 256 entries long. Since the encoding vector is rather sparse, all the entries are initialized to "/.notdef." Then those entries which correspond to characters in the EBCDIC encoding are filled in with the proper character name.

Re-encoding an Entire Font, cont. 23

```
dup 107 /comma put
dup 108 /percent put
dup 109 /underscore put
dup 110 /greater put
dup 111 /question put

dup 122 /colon put
dup 123 /numbersign put
dup 124 /at put
dup 125 /quoteright put
dup 126 /equal put
dup 127 /quotedbl put

dup 129 /a put    dup 134 /f put
dup 130 /b put    dup 135 /g put
dup 131 /c put    dup 136 /h put
dup 132 /d put    dup 137 /i put
dup 133 /e put

dup 145 /j put    dup 150 /o put
dup 146 /k put    dup 151 /p put
dup 147 /l put    dup 152 /q put
dup 148 /m put    dup 153 /r put
dup 149 /n put

dup 162 /s put    dup 166 /w put
dup 163 /t put    dup 167 /x put
dup 164 /u put    dup 168 /y put
dup 165 /v put    dup 169 /z put

dup 193 /A put    dup 198 /F put
dup 194 /B put    dup 199 /G put
dup 195 /C put    dup 200 /H put
dup 196 /D put    dup 201 /I put
dup 197 /E put

dup 209 /J put    dup 214 /O put
dup 210 /K put    dup 215 /P put
dup 211 /L put    dup 216 /Q put
dup 212 /M put    dup 217 /R put
dup 213 /N put

dup 226 /S put    dup 230 /W put
dup 227 /T put    dup 231 /X put
dup 228 /U put    dup 232 /Y put
dup 229 /V put    dup 233 /Z put
```

Continuation of the EBCDIC encoding vector definition.

Re-encoding an Entire Font, cont. 23

```
  dup 240 /zero put   dup 245 /five put
  dup 241 /one put    dup 246 /six put
  dup 242 /two put    dup 247 /seven put
  dup 243 /three put  dup 248 /eight put
  dup 244 /four put   dup 249 /nine put
pop
```

Remove the array from the operand stack.

```
/TR /Times-Roman findfont 7 scalefont def
/Times-Roman /Times-Roman-EBCDIC EBCDIC
  ReEncode
/TRE /Times-Roman-EBCDIC findfont 7
  scalefont def
```

Now we will print a table comparing the standard PostScript character set encoding with the EBCDIC encoding. First we will set up the fonts to be used: Times Roman with the standard encoding and Times Roman with the EBCDIC encoding.

```
TR setfont
0 1 3
  { /count exch def
    72 count 127 mul add 560 moveto
    (Decimal   Standard    EBCDIC) show
    72 count 127 mul add 560 7 sub moveto
    (Number      Char        Char) show
  } for
```

Print each column heading in the standard Times Roman.

```
/yline 538 def
/xstart 82 def
0 1 255
  { /count exch def
    /charstring ( ) dup 0 count put def
    TR setfont
    xstart yline moveto
    count (   ) cvs show
    xstart 28 add yline moveto
    charstring show
    TRE setfont
    xstart 56 add yline moveto
    charstring show
    /yline yline 7 sub def
    count 1 add 64 mod 0 eq
      { /xstart xstart 127 add def
        /yline 538 def
      } if
  } for
```

Print the table of character codes and corresponding characters.
For each character code from 0 to 255, print the corresponding standard and EBCDIC characters.

Print the character code in decimal.

Print the corresponding standard character.

Print the corresponding EBCDIC character.
Move down one line.
If we have gotten to the 64th line, move over by a column and start at the top again.

```
showpage
```

Boktryckarkonsten är källan till praktiskt taget all mänsklig odling.
Printing is the source of practically all human evolution.
Den förutan hade de oerhörda framstegen inom vetenskap
Without it the tremendous progress in the fields of science and
och teknik inte varit möjliga.
technology would not have been possible.
VALTER FALK

Making Small Changes to Encoding Vectors 24

```
/reencsmalldict 12 dict def
```
Local storage for the procedure "ReEncodeSmall."

```
/ReEncodeSmall
  { reencsmalldict begin
    /newcodesandnames exch def
    /newfontname exch def
    /basefontname exch def
```
ReEncodeSmall generates a new font given the name of the font to be re-encoded, a new name, and an array of new character encoding and character name pairs (see definition of the "scandvec" array below for the format of this array). This method has the advantage that it allows the user to make changes to an existing encoding vector without having to specify the entire new encoding vector. It also saves space when the character encoding and name pairs array is smaller than an entire encoding vector.

```
    /basefontdict basefontname findfont def
```
Get the font dictionary on which to base the re-encoded version.

```
    /newfont basefontdict maxlength dict def
```
Create a dictionary to hold the description for the re-encoded font.

```
    basefontdict
      { exch dup /FID ne
          { dup /Encoding eq
            { exch dup length array copy
                newfont 3 1 roll put }
            { exch newfont 3 1 roll put }
            ifelse }
          { pop pop }
          ifelse
       } forall
```
Copy all the entries in the base font dictionary to the new dictionary except for the FID field.

Make a copy of the Encoding field.

Ignore FID pair.

```
    newfont /FontName newfontname put
```
Install the new name.

```
    newcodesandnames aload pop
```
Modify the encoding vector. First load the new encoding and name pairs onto the operand stack.

```
    newcodesandnames length 2 idiv
      { newfont /Encoding get 3 1 roll put}
      repeat
```
For each pair on the stack, put the new name into the designated position in the encoding vector.

```
    newfontname newfont definefont pop
    end
  } def
```
Now make the re-encoded font description into a PostScript font. We will ignore the modified dictionary returned on the operand stack by the "definefont" operator.

Making Small Changes to Encoding Vectors, cont. 24

```
/scandvec [
  192 /Oacute
  201 /Adieresis
  209 /oacute
  210 /Ograve
  211 /Scaron
  212 /ograve
  213 /scaron
  216 /Edieresis
  217 /adieresis
  218 /edieresis
  219 /Odieresis
  220 /odieresis
  224 /Aacute
  226 /Aring
  228 /Zcaron
  231 /Eacute
  240 /aacute
  242 /aring
  244 /zcaron
  247 /eacute
  ] def
```

Define an array of new character encoding and name pairs that will enable us to print the accented characters in the Scandinavian Languages. The array is a series of encoding number and name pairs. The encoding number always preceeds the character name. By definition, there must be an even number of elements in this array. The encoding vector positions for these new characters have been chosen so that they do not actually replace any of the characters in the standard encoding.

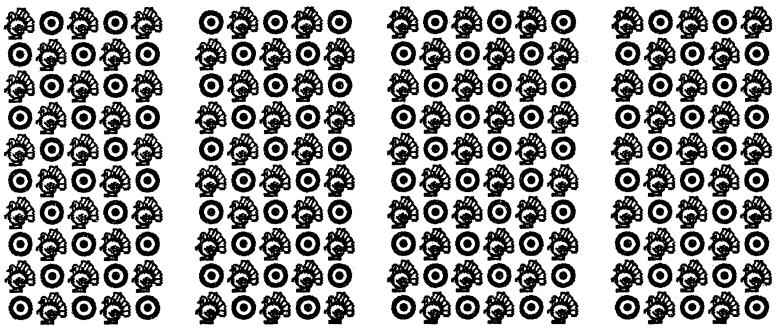
```
/ss { 72 yline moveto show
      /yline yline 28 sub def } def
```

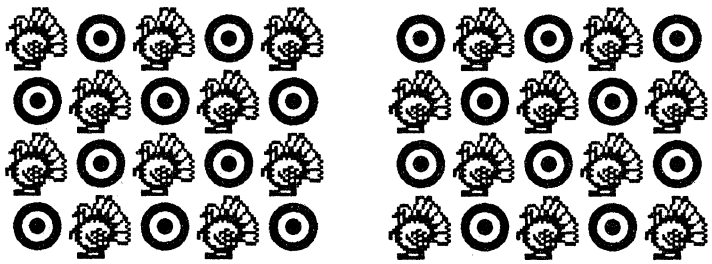
This procedure shows a string and then skips a line.

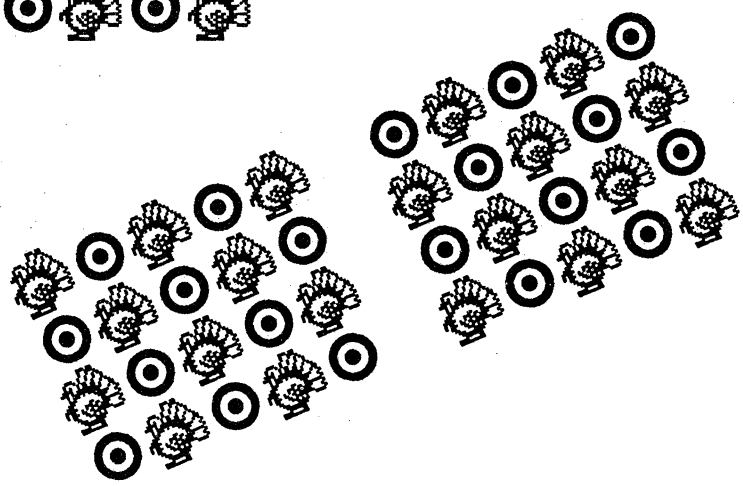
```
/Times-Roman /Times-Roman-Scand scandvec
  ReEncodeSmall
/Times-Roman-Scand findfont 12 scalefont
  setfont
/yline 500 def
(Boktryckarkonsten \331r k\331llan till \
praktiskt taget all m\331nsklig odling.) ss
(Den f\334rutan hade de oerh\334rda \
framstegen inom vetenskap) ss
(och teknik inte varit m\334jliga.) ss
(VALTER FALK) ss
```

Re-encode the standard Times Roman to include the accented characters for the Scandinavian Languages.
Now we will print some text with accented characters. Since the accented characters are in the upper half of the encoding vector we must refer to them by their octal codes.

```
/Times-Italic findfont 10 scalefont setfont
/yline 500 12 sub def
(Printing is the source of practically \
all human evolution.) ss
(Without it the tremendous progress in \
the fields of science and) ss
(technology would not have been \
possible.) ss
showpage
```







Making a User Defined Font

The following program demonstrates the construction of a user defined font. The font will only have two characters defined ("a" and "b") and will illustrate how both bitmaps and analytic shapes may be used as font characters. The character "a" will print a turkey (constructed as a bitmap), and the character "b" will print a bullseye.

```
/BuildCharDict 10 dict def
```
This dictionary is used by the BuildChar procedure for local variables.

```
/ExampleFont 7 dict def
ExampleFont begin
  /FontType 3 def
  /FontMatrix [1 0 0 1 0 0] def
  /FontBBox [0 0 1 1]def
  /Encoding 256 array def
```
Allocate the font dictionary. Leave room for the FontID.
Build the required entries in the font dictionary.
FontType 3 tells PostScript that this is a user defined font.
Use the identity matrix for the font coordinate system.
The largest character in the font will be 1 unit by 1 unit.
Allocate the Encoding array.

```
  0 1 255 {Encoding exch /.notdef put} for
  Encoding (a) 0 get /turkey put
  Encoding (b) 0 get /bullseye put
```
Build the encoding vector that will define "a" and "b".
Initialize all entries in the encoding vector with ".notdef".
Associate the name "turkey" with the character code for "a".
Associate the name "bullseye" with the character code for "b".

```
  /CharacterDefs 3 dict def
```
Define the various character drawing procedures and put them in the CharacterDefs dictionary.

```
  CharacterDefs /.notdef {} put
```
There should always be a description for the undefined character ".notdef" which does nothing.

```
  CharacterDefs /bullseye
    { newpath
        .5 .5 .375 0 360 arc
        .5 .5 .25 360 0 arcn
        .625 .5 moveto
        .5 .5 .125 0 360 arc
      fill
    } put
```
This procedure provides the analytic description for drawing a bullseye. The bullseye is centered within the unit square.

```
  CharacterDefs /turkey

    { 24 23 true [24 0 0 -23 0 23]
      {<003B00 002700 002480 0E4940 114920
        14B220 3CB650 75FE88 17FF8C 175F14
        1C07E2 3803C4 703182 F8EDFC B2BBC2
        BB6F84 31BFC2 18EA3C 0E3E00 07FC00
        03F800 1E1800 1FF800>
      } imagemask
    } put
```
This procedure provides the bitmap description for drawing a turkey.
To print a bitmap as a character in a font, the "imagemask" operator is used. The size of the bitmap is specified (note that this particular bitmap is not perfectly square: it is 24 bits wide by 23 bits high). The bitmap itself is specified as a hex string.

Making a User Defined Font, cont. 25

```
  /BuildChar
    { BuildCharDict begin
        /char exch def

        /fontdict exch def

        /charname fontdict /Encoding get
          char get def
        /charproc fontdict /CharacterDefs get
          charname get def
        1 0 0 0 1 1 setcachedevice

        gsave charproc grestore
      end
    } def
end

/MyFont ExampleFont definefont pop
/showline
  { gsave show grestore
    0 lineskip rmoveto } def

/MyFont findfont 12 scalefont setfont
/lineskip -12 def
72 555 moveto
5 { (ababa babab ababab ababa) showline
    (babab ababa bababa babab) showline
  } repeat

/MyFont findfont 24 scalefont setfont
/lineskip -24 def
72 360 moveto
2 { (ababa babab) showline
    (babab ababa) showline
  } repeat
250 180 moveto
23 rotate
2 { (ababa babab) showline
    (babab ababa) showline
  } repeat
showpage
```

The procedure BuildChar is called everytime a character from this font must be constructed.
The character code is provided as an argument to this procedure.
So is the font dictionary.

Convert the character code to the corresponding name by looking it up in the encoding vector.
Now retrieve the procedure by that name from the CharacterDefs dictionary.
Using the "setcachedevice" operator enables the characters from this font to be cached.
Call the procedure which renders the character.

Now we are done specifying all the information required to build a font.

Register the font with PostScript and name it "MyFont."
This procedure makes it more convenient to show a line of text.

Now use the font we have built. Build a 12 point version of the font.

Note that one of the characters in the string which is shown is the "space" character. Since we have not defined what the "space" character should look like, the definition of the ".notdef" character is printed instead.
Now build a 24 point version.

Rotate the user coordinate space to an arbitrary rotation.
This shows a rotated version of the font.

For Further Reference

Foley, James D. and Van Dam, Andries. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Massachusetts, 1982.

IBM System/360: Priciples of Operation, Ninth Edition, November 1970.

Newman, William M. and Sproull, Robert F. *Principles of Interactive Computer Graphics*. McGraw-Hill, New York, 1979.

POSTSCRIPT Language Manual, Adobe Systems, Inc.

Pratt, Terrence W. *Programming Languages: Design and Implementation*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1975.

Warnock, John and Wyatt, Douglas. "A Device Independent Graphics Imaging Model for Use with Raster Devices," *Computer Graphics* Volume 16, Number 3, July 1982, pp. 313-320.

Index

Accented characters 85
Arc 14, 17, 89
Arcn 89
Arrows, drawing 25

Beveled line joins 26
BuildChar 89
Butt line caps 26

Centimeter 4
Character origin 34
Character width 34
Circular arcs 14
Circular text 53
Clip 73
Closed shapes 3
Closepath, motivation for 3
Condensed text 42
Copypage 73
Current font 37
Current path 1
Current point 1, 14, 37
Currentfile 67
Currentmatrix 17, 23, 25
Currenttransfer 67
Cvlit 67
Cvx 67

Dash patterns 31
Default coordinate system 4
Definefont 76, 81, 89
Dictionaries, as local storage 51, 53

EBCDIC 77, 81
Ellipse 17
Encoding vector 77, 81, 85, 89
End caps 26
Erasepage 73
Exit 47
Extended text 42

Face 34, 39
FID 76, 81, 85
Findfont 37
Flattenpath 57
Font 34
Font dictionary 37, 76, 81, 85, 89
Font family 34
Font name 34, 37
Fonts, defining in prologue 39
For 73

Getinterval 47
Grestore 11, 41, 73
Grid pattern 19
Gsave 11, 41, 73

Image 67
Imagemask 89
Inch 4
Intersecting lines 2
Itransform 57

Line breaks 47
Line caps 26
Line joins 26
Lineto 1
Loop 47

Makefont 42, 45
Miter limit 27
Mitered line joins 26
Moveto 1
Moveto, positioning text 37

Newpath 1

Obliqued text 43, 45
Operand stack 5
Outline font 77
Outlined fonts 79

Path 1
Path construction 1
Pathforall 57
Pie slice 15
Piechart, drawing a 63
Point 4, 34
Point size 34, 39
Procedure, defining a 4
Projecting square line caps 26

Re-encoding a font 77
Readhexstring 67

Repeat 19, 21
Rlineto 9
Rotate 13
Round line caps 26
Rounded line joins 26

Scale 13, 23
Scalefont 37, 42
Search 47
Setcachedevice 89
Setdash 26, 28, 31
Setdash, dash array 28
Setdash, offset 29, 31
Setfont 37
Setgray 9, 41, 73
Setlinecap 19, 26
Setlinejoin 26
Setlinewidth 2
Setmatrix 17, 23, 25
Setmiterlimit 27
Setscreen 69
Settransfer 67, 69
Show 37, 41
Showpage 2
Square path 9, 11
Square, drawing a 3
Stringwidth 47, 51
Stroke 2

Text, condensed 42
Text, extended 42
Text, obliqued 43
Transfer function 67
Transform 57
Transformations order of 13
Translate 11, 13, 19, 73

Vertical text 51

Appendix C

The Adobe Font Manual

Adobe Font Manual

These pages are reproductions
of POSTSCRIPT samples
printed on a 300 dpi
laser printer.

Revision 2
2 October 1984

This manual is intended for informational use only. It is subject to change without notice and should not be construed as a commitment by Adobe Systems, Inc. Adobe Systems assumes no responsibility or liability for errors or inaccuracies that may appear in this document.

The software described in this document is furnished under license and may be used or copied only in accordance with the terms of such license.

Adobe Systems Incorporated
1870 Embarcadero Road, Suite 100
Palo Alto, California 94303

POSTSCRIPT™ Font Manual
Second Edition
2 October 1984
Copyright © 1984 Adobe Systems, Inc.

POSTSCRIPT is a trademark of Adobe Systems, Inc.

Times is a trademark of Allied Corporation.

Helvetica is a registered trademark of Allied Corporation.

Avant Garde Gothic is a registered trademark of International Typeface Corporation.

Table of Contents

Introduction
- General Information . . . 3
- Languages . . . 7

Typefaces
- Courier Family . . . 11
  - Courier . . . 13
  - Bold . . . 15
  - Oblique . . . 17
  - Bold Oblique . . . 19
- Helvetica Family . . . 21
  - Helvetica . . . 23
  - Bold . . . 25
  - Oblique . . . 27
  - Bold Oblique . . . 29
- Symbol . . . 31
- Times Family . . . 35
  - Roman . . . 37
  - Bold . . . 39
  - Italic . . . 41
  - Bold Italic . . . 43

Encodings
- Work Sheet . . . 47
- Standard Encoding . . . 49
- Standard Character Set . . . 51
- Symbol Encoding . . . 57
- Symbol Set . . . 59

Appendix
- Updates . . . 69

Introduction

General Information

A font is a collection of characters (letters, numerals, punctuation marks, reference marks, and symbols) which has a unified design. POSTSCRIPT, as a graphics language, can reproduce a character as easily as any other graphic shape, and has a wide variety of procedures to allow easy handling of characters.

Adobe licenses typefaces from leading designers, such as Mergenthaler Linotype and International Typeface Corporation. Thus, POSTSCRIPT can produce text in such popular typefaces as Times, Helvetica, and Avant Garde Gothic. These typefaces can be scaled, obliqued, and rotated as desired, giving complete control over the resulting image.

A considerable amount of information concerning POSTSCRIPT fonts is available to an application program. Much of this information, such as character widths, sidebearings, and character bounding boxes, is of interest to the typesetter. A program can obtain information about the font by either accessing the POSTSCRIPT font dictionary or using a POSTSCRIPT operator. For details on how to obtain font information, refer to the appendix on Font Machinery in the **POSTSCRIPT Language Manual.**

Font Manual

This manual presents detailed information on the fonts available with POSTSCRIPT. The information supplied allows the graphic artist to design a document and the applications programmer to produce that document. Further information needed for program implementation is found in the **POSTSCRIPT Language Manual** and **POSTSCRIPT Cookbook.**

The **Font Manual** incorporates the following information:

- Samples of each font at various point sizes

- Character widths for each font

- Character encodings

Adobe publishes quarterly updates to this manual. At the end of the manual is an appendix reserved for these updates. Any updates current as of the printing of your manual have been included in this appendix. You should add new updates to the appendix as you receive them.

Typeface Samples

Two kinds of samples are presented in this section of the **Font Manual**.

1. Each font family is presented in bodies of text showing the related faces in that family.

2. Following this is a series of pages showing samples of each face in the family at various point sizes.

Width Tables

The width tables are printed on the back of the typeface samples. They show the character set of each typeface along with information needed for the precise placement of characters on a page. Each character is shown in relation to the cap height, x-height, and descender height of the font. Underneath each character is its width; this is given in points for a character that is one point high. (There are seventy-two points to the inch.)

The one-point character width is used to obtain the width of a character at a particular point size. Simply multiply the one-point width by the point size of the font to get the character's width in points.

Character Encodings

All computer-based systems must internally encode characters as numbers. The last section of this manual contains several charts and tables that present POSTSCRIPT's default character-encoding scheme, which is based on the USASCII and ISO 6937 standards.

Character codes are used to print a specific character, either by

inserting that code (as a decimal number) into a POSTSCRIPT string or by directly including that code (as an octal number) in a string. For more information on printing characters, see "Character and Font Operators" in the **POSTSCRIPT Language Reference Manual** and the text program examples in the **POSTSCRIPT Cookbook.**

POSTSCRIPT's encoding can be easily changed. You can assign any character to any numeric code to produce anything from a variation of Adobe's default encoding to a completely original system of codes. This ability is valuable to many applications, since some of POSTSCRIPT's characters are not encoded in the default system and can be obtained only by explicitly assigning them a code number. For example, this applies to all of the accented characters.

There are two types of charts in this section: an encoding grid and a code list. These charts are given for both the Adobe Standard Character Set and the Adobe Symbol Set.

The encoding grid is a one-page presentation of Adobe's default encoding system, arranged in sixteen columns of sixteen characters each. To obtain the decimal code of a particular character, locate that character on the grid, multiply its column number by sixteen and add its row number to that product.

Grid positions marked in dark gray represent codes that are unused and hence are available for assignment to whatever special characters are needed by an application. Light gray boxes in the default grid indicate positions reserved for control characters; these can also be redefined as necessary, as can any position on the chart.

The code list is a multiple-page chart listing each character with its description, code name, and octal code. The code name is needed for reassigning character codes; the octal code can be directly used in a POSTSCRIPT string.

It should be emphasized that the encoding presented in this section of the manual is only the default scheme; it may be altered to suit an

application program. For your convenience, the encoding section begins with a blank grid which can be used for setting up a customized code system.

Language Samples

POSTSCRIPT can produce a wide selection of accented characters. Since there is not enough room in the standard encoding system for all fifty-six of Adobe's accented characters, these are not assigned default codes.

If an accented character is needed by an application program, that character may be assigned one of the unused codes in the default system. Alternatively, the desired character may replace an already-encoded character in the grid. Once an accented character has an assigned code, it may be used from within POSTSCRIPT like any other character.

The following pages present examples of non-English text produced with POSTSCRIPT.

Languages

German

Zum Bestand wahrhafter Bildung sollte es gehören, daß Jeder, der unserer Lettern sich bedient, über deren Herkunft sich klare Vorstellungen machen kann und, indem er sein Wissen den Nachfahren weiter gibt, eingedenk dessen bleibt, daß in den Lettern ein ewiges Stück Menschheitsgeschichte sich dartut, an dem auch er Teil hat. –F. H. Ehmcke

French

Observons ici que l'œuvre typographique exclut l'improvisation; elle est le fruit d'essais qui disparaissent, l'objet d'un art qui ne retient que des ouvrages achevés, qui rejette les ébauches et les esquisses, et ne connaît point d'états intermédiaires entre l'être et le non être. Il nous donne par là une grande et redoutable leçon. –Paul Valéry

Spanish

Encarada la tipografía de tal manera, deja de ser un arte menor, una artesanía, para asumir el título de ciencia, o de filosofía, pues incluye también a la ética, como condición dignificante del destino del hombre sobre la tierra con sus problemas morales y perfecciones, al fin espiritual de ser algo más que un peso inútil. –Raúl Rosarivo

Italian

È natural vantaggio della stampa il far ciascuna lettera sempre la stessa, avendone le migliaja fuse in matrici percosse da un medesimo punzone. Ma dalla maestria del punzonista dipende che le misure e le parti, che possono esser comuni a più lettere, sieno precisamente ed esattamente le medesime in esse tutte.
–Giambattista Bodoni

Swedish

Boktryckarkonsten är källan till praktiskt taget all mänsklig odling. Den förutan hade de oerhörda framstegen inom vetenskap och teknik inte varit möjliga. Men ej heller boktryckarkonsten, som vi känner den, var möjlig utan uppfinningen av stilgjutningskonsten. Det är denna som är Gutenbergs storhet.
–Valter Falk

Finnish

Jos kerran kirjasimia luotaessa pyritään kaikin keinoin taiteellisuuteen ja tyylikkyyteen, niin kuin on ollut laita, velvoittaa tämä myös kirjapainoammatin harjoittajia ja tällä alalla työskentel eviä pyrkimään töissään samaan arvokkaaseen tulokseen.
–Atte Syvänne

Norwegian

Derfor priser jeg Gutenbergs opfindelse, bogtrykkerkunsten, som den nødvendige forudsætning for kulterens sejrsgang, for den udvikling, der har ført folkene ind i oplysningens verden, ført dem fra mørket og trældom til lyset og friheden.
–Thorvald Stauning

Hungarian

Ama könyvművésze feladatát nagy részben építészeti, főként pedig a szó tiszta értelmében grafikai feladatnak érzi. Ez a felfogásbeli változás a könyvet egészen átalakítja és a fejlődés irányát ismét visszatereli a mult könyvművészetének forrásaihoz.
–Emerich Kner

Typefaces

Courier Family

Courier with Oblique, Bold and Bold Oblique.

10 point text on 11 point linespacing.

Each single letter is a small, well-balanced figure in itself. There are bad types, too; however, in a good type-face *each letter rests complete in itself.* To us, who are used to reading, a letter has become an abstract idea, a mere means of understanding. However, its characteristic forms reveal that originally it meant more than that: *a symbol, simplified to the utmost and representing a given thing.* And, even more, that it held a mysterious meaning, acting as a magic symbol to invoke spirits and subdue powers.
-Romano Guardini

Courier Oblique

The contemporary typographer regards his work from the design point of view and concentrates on the true essence of his task, to create graphic design.
-Emerich Kner

*

Courier Bold

Machines exist; let us then exploit them to create beauty - a modern beauty, while we are about it. For we live in the twentieth century.
-Aldous Huxley

Courier Bold Oblique

Neither may the clarity of the single letter be given up for the sake of rhythm, nor may formal beauty be sacrificed to mere clarity or misconceived utility.
-Jan Tschichold

The Courier family supports all characters in the Standard Character Set except the ligatures (fi and fl), the diphthongs (Æ, Œ, æ and œ) and the per thousand symbol (‰).

Courier

15 cpi
8 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`.',"„',’·”› [¡¿ªº]
{¯˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

13.6 cpi
8.8 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`.',"„',’·”› [¡¿ªº]
{¯˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

12 cpi
10 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`.',"„',’·”› [¡¿ªº]
{¯˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

10 cpi
12 pt.

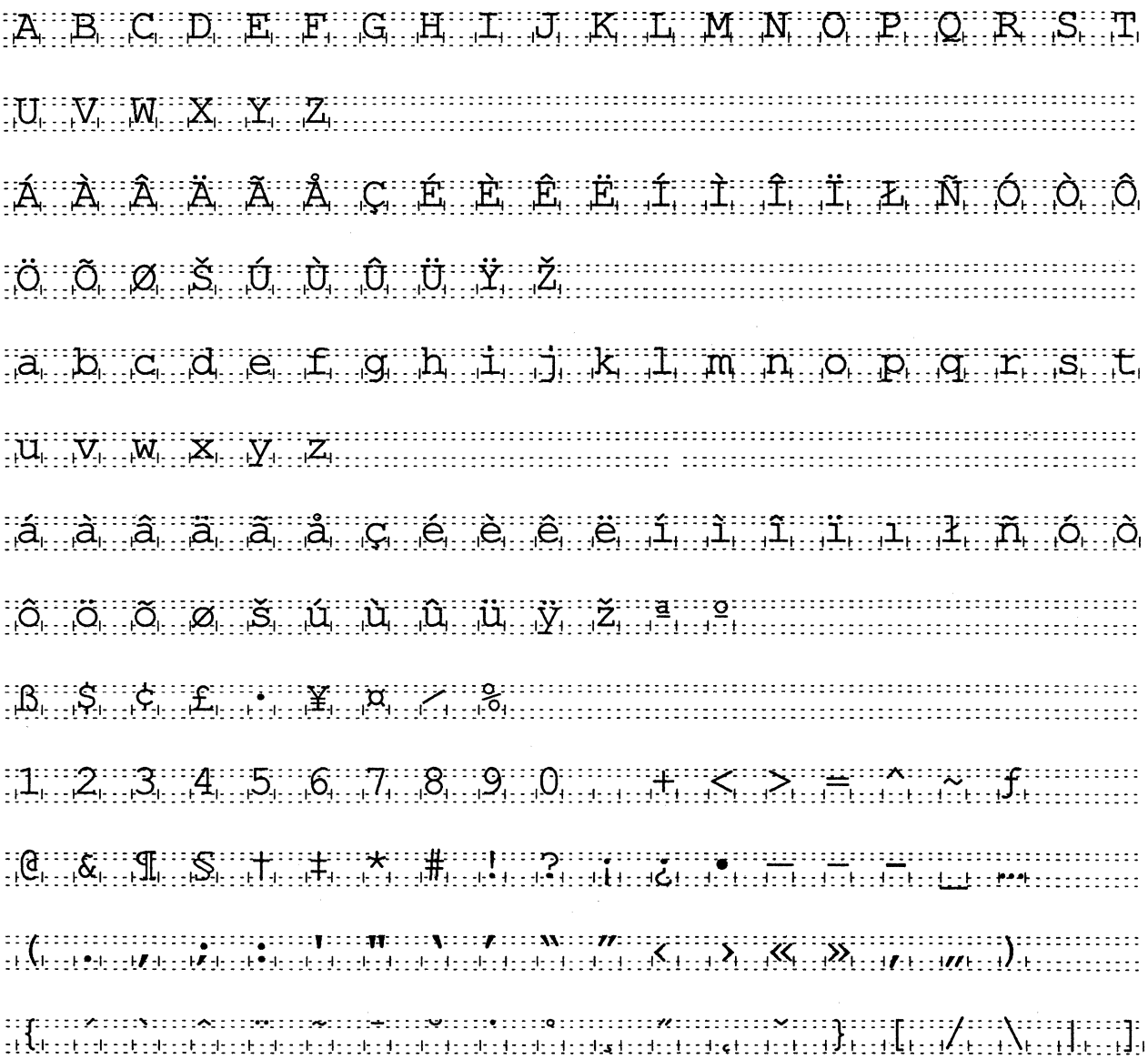
abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`.',"„',’·”› [¡¿ªº]
{¯˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

8 cpi
15 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`.',"„',’·”› [¡¿ªº]
{¯˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

Courier

1-point width of 0.6



Courier Bold

15 cpi
8 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`',"„',’·”› [¡¿ªº]
{˜ˇˆ˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

13.6 cpi
8.8 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`',"„',’·”› [¡¿ªº]
{˜ˇˆ˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

12 cpi
10 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`',"„',’·”› [¡¿ªº]
{˜ˇˆ˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

10 cpi
12 pt.

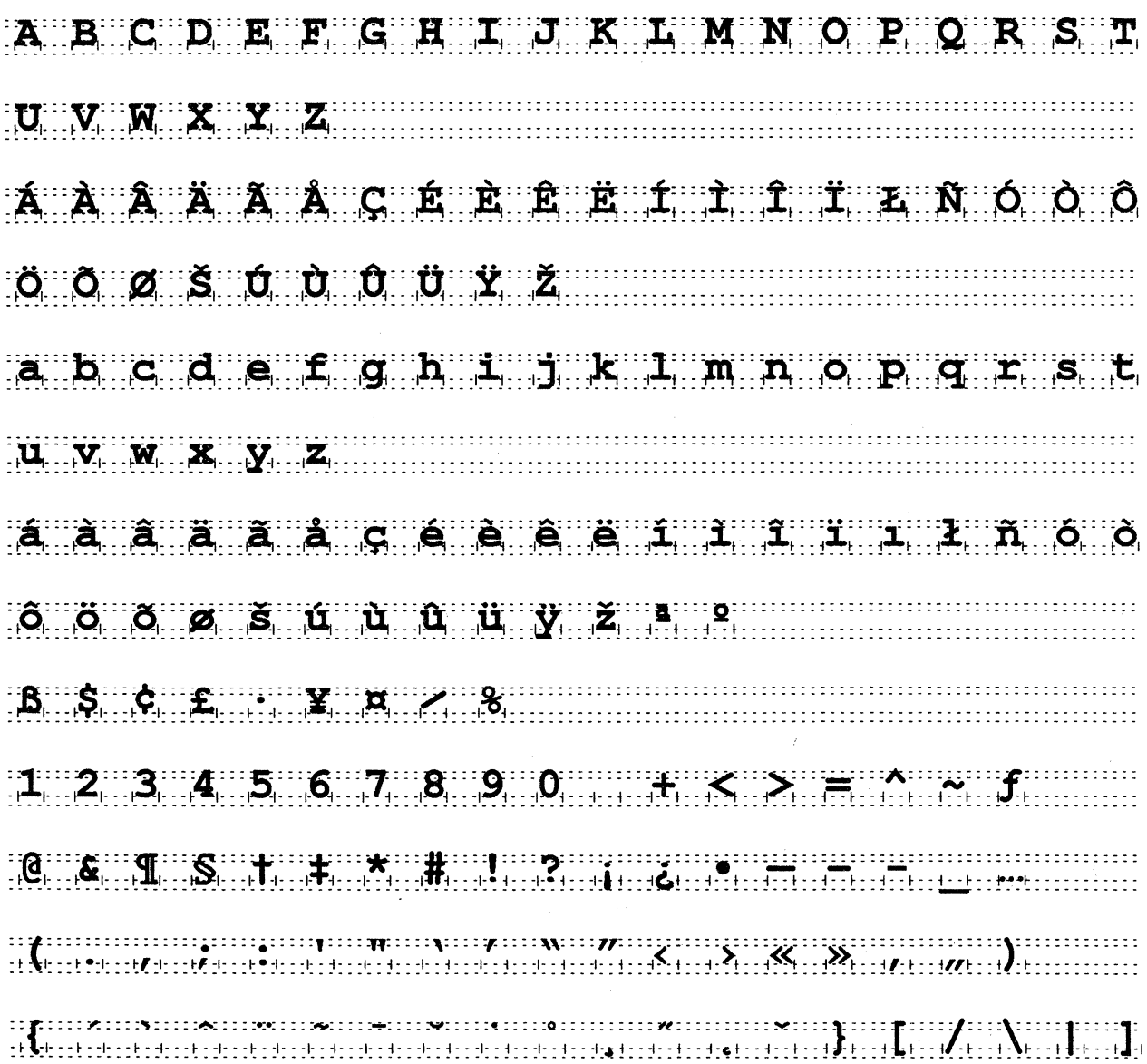
abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`',"„',’·”› [¡¿ªº]
{˜ˇˆ˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

8 cpi
15 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹"`',"„',’·”› [¡¿ªº]
{˜ˇˆ˘˙˝¸}… ÅÇËÎÑÒŠÚáçöû

Courier Bold

1-point width of 0.6



Courier Oblique

15 cpi
8 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹ ". ', „ ', ′ · ″ › [¡¿ª º]
{ ˉ ˘ · ˚ ¸ }… ÅÇËÎÑÒŠÚáçöû

13.6 cpi
8.8 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹ ". ', „ ', ′ · ″ › [¡¿ª º]
{ ˉ ˘ · ˚ ¸ }… ÅÇËÎÑÒŠÚáçöû

12 cpi
10 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹ ". ', „ ', ′ · ″ › [¡¿ª º]
{ ˉ ˘ · ˚ ¸ }… ÅÇËÎÑÒŠÚáçöû

10 cpi
12 pt.

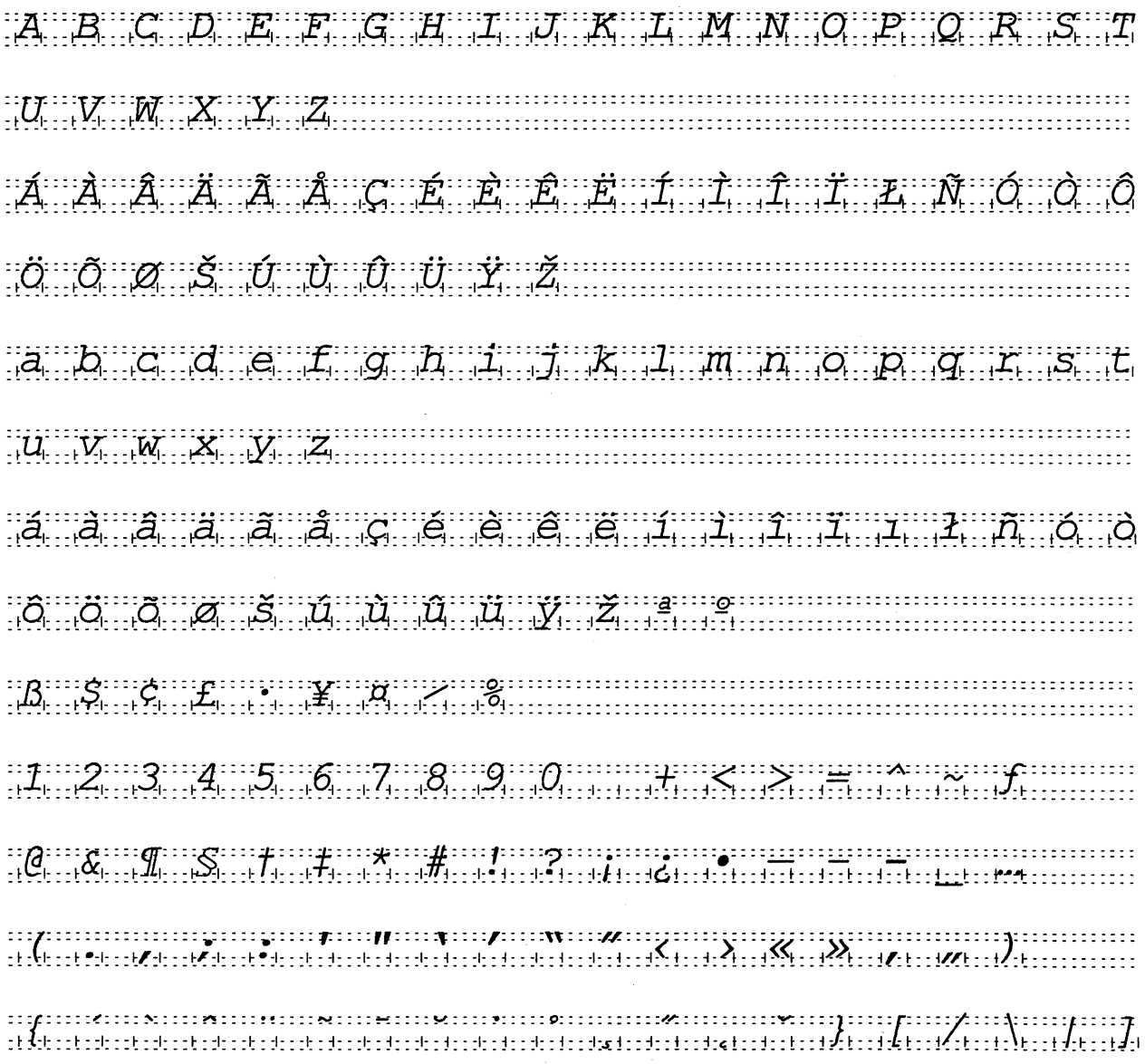
abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹ ". ', „ ', ′ · ″ › [¡¿ª º]
{ ˉ ˘ · ˚ ¸ }… ÅÇËÎÑÒŠÚáçöû

8 cpi
15 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ---
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹ ". ', „ ', ′ · ″ › [¡¿ª º]
{ ˉ ˘ · ˚ ¸ }… ÅÇËÎÑÒŠÚáçöû

Courier Oblique

1-point width of 0.6



Courier Bold Oblique

15 cpi
8 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹".',"„',‘·”› [¡¿ªº]
{˘ˇ˙˚¸}… ÅÇËÎÑÒŠÚáçöû

13.6 cpi
8.8 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹".',"„',‘·”› [¡¿ªº]
{˘ˇ˙˚¸}… ÅÇËÎÑÒŠÚáçöû

12 cpi
10 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹".',"„',‘·”› [¡¿ªº]
{˘ˇ˙˚¸}… ÅÇËÎÑÒŠÚáçöû

10 cpi
12 pt.

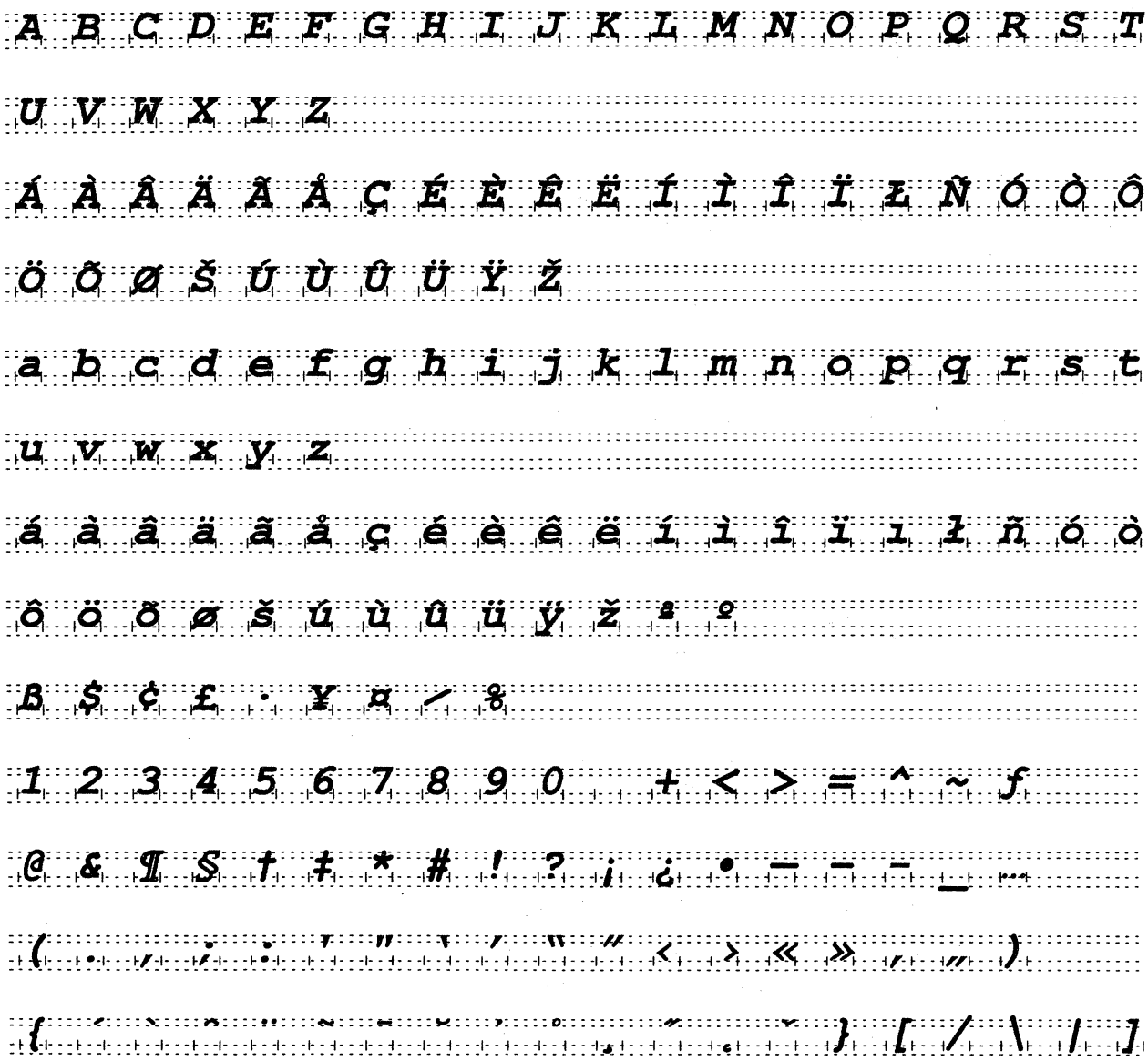
abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹".',"„',‘·”› [¡¿ªº]
{˘ˇ˙˚¸}… ÅÇËÎÑÒŠÚáçöû

8 cpi
15 pt.

abcdefghijklmnopqrstuvwxyz _ ıłøß
ABCDEFGHIJKLMNOPQRSTUVWXYZ & ŁØ
1234567890 • $¢¥£ƒ¤ ——
<%~/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹".',"„',‘·”› [¡¿ªº]
{˘ˇ˙˚¸}… ÅÇËÎÑÒŠÚáçöû

Courier Bold Oblique

1-point width of 0.6



Helvetica Family

Helvetica with Oblique, Bold and Bold Oblique.

11 point text on
12 point linespacing.

I am Type! I bring into the light of day the precious stores of knowledge and wisdom long hidden in the grave of ignorance. I coin for you *the enchanting tale, the philosopher's moralizing, and the poet's phantasies;* I enable you to exchange the irksome hours that come, at times, to every one, for sweet and happy hours with books – golden urns filled with all manna of the past. In books, I present to you a portion of the eternal mind caught in its progress through the world, stamped in an instant, and preserved for eternity. Through me, *Socrates and Plato, Chaucer and the Bards,* become your faithful friends who ever surround and minister to you.
–Frederic Goudy

Helvetica Oblique

The typographer who can serve his art modestly and with a sensitive understanding of the special demands made by each type face will be the one to achieve the finest results.
–Paul Renner

Helvetica Bold

Of all arts, architecture is nearest akin to typography. Both are equally related to their function. In both, that which wholly fulfils its purpose is beautiful.
–Helmut Presser

Helvetica Bold Oblique

No other art is more justified than typography in looking ahead to future centuries; for the creations of typography benefit coming generations as much as present ones.
–Giambattista Bodoni

Helvetica is a registered trademark of Allied Corporation.

The Helvetica family supports all characters in the Standard Character Set.

Helvetica

6 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„,’·”› [¡¿ªº] { ¯˘˝˛}... ÅÇËÎÑÒŠÚáçöû

8 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„,’·”› [¡¿ªº] { ¯˘˝˛}... ÅÇËÎÑÒŠÚáçöû

9 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„,’·”› [¡¿ªº] { ¯˘˝˛}... ÅÇËÎÑÒŠÚáçöû

10 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„,’·”› [¡¿ªº] { ¯˘˝˛}... ÅÇËÎÑÒŠÚáçöû

12 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„,’·”› [¡¿ªº] { ¯˘˝˛}... ÅÇËÎÑÒŠÚáçöû

14 pt.

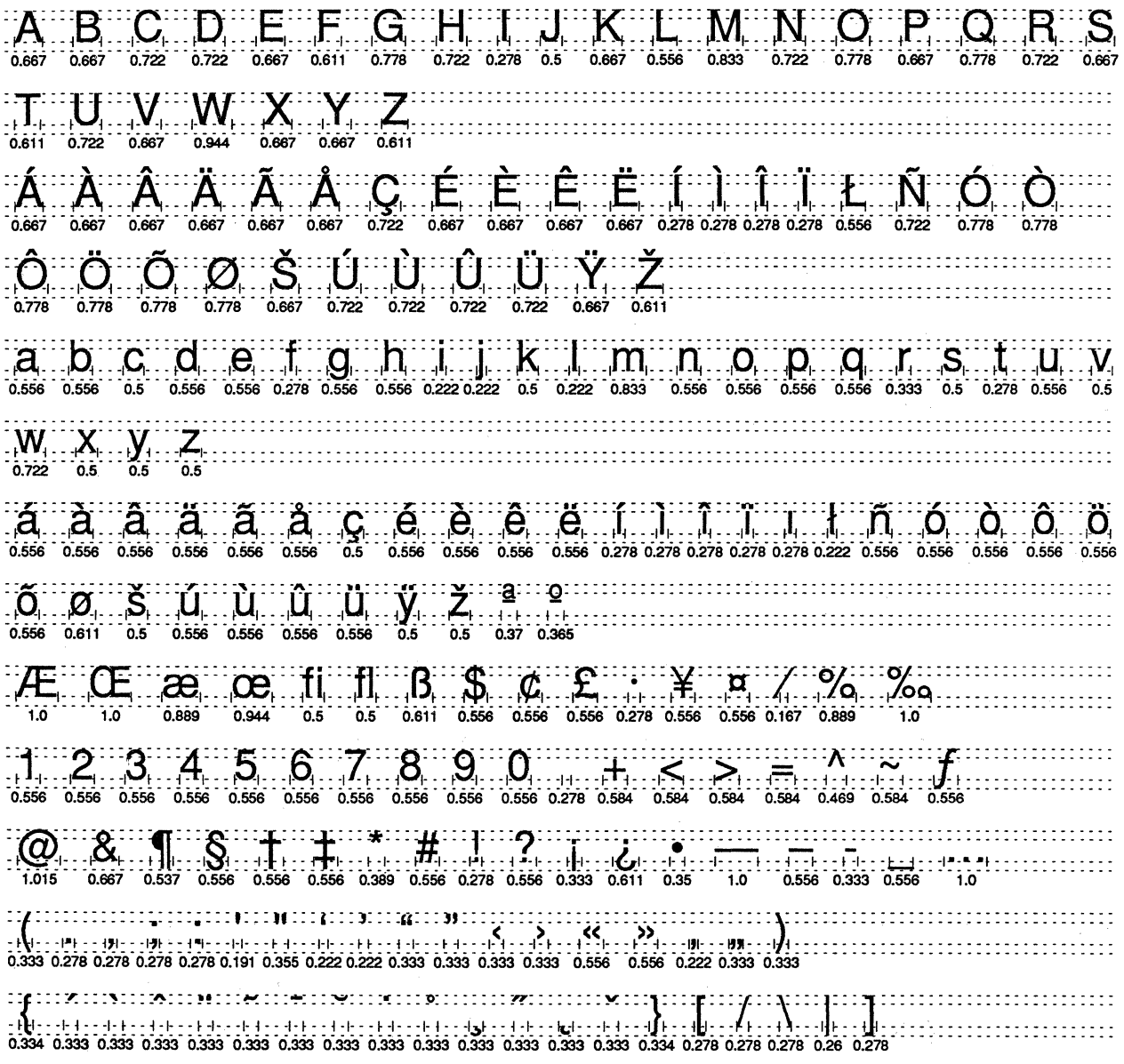
abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„,’·”› [¡¿ªº] { ¯˘˝˛}... ÅÇËÎÑÒŠÚáçöû

18 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*⁄+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„,’·”› [¡¿ªº] { ¯˘˝˛}... ÅÇËÎÑÒŠÚáçöû

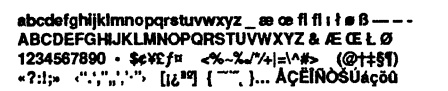
Helvetica

1-point widths

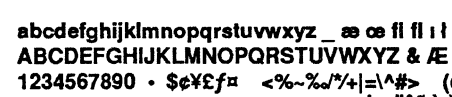


Helvetica Bold

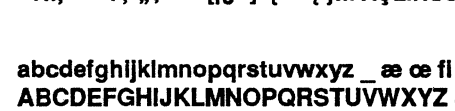
6 pt.



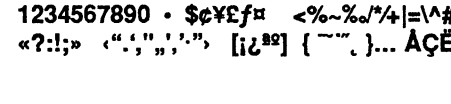
8 pt.



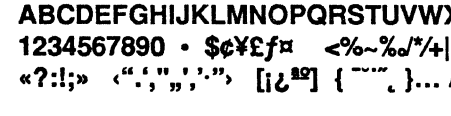
9 pt.



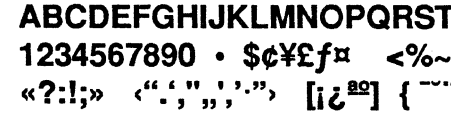
10 pt.



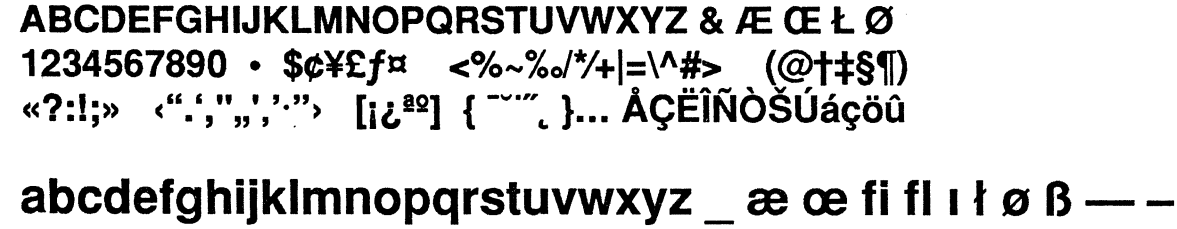
12 pt.



14 pt.

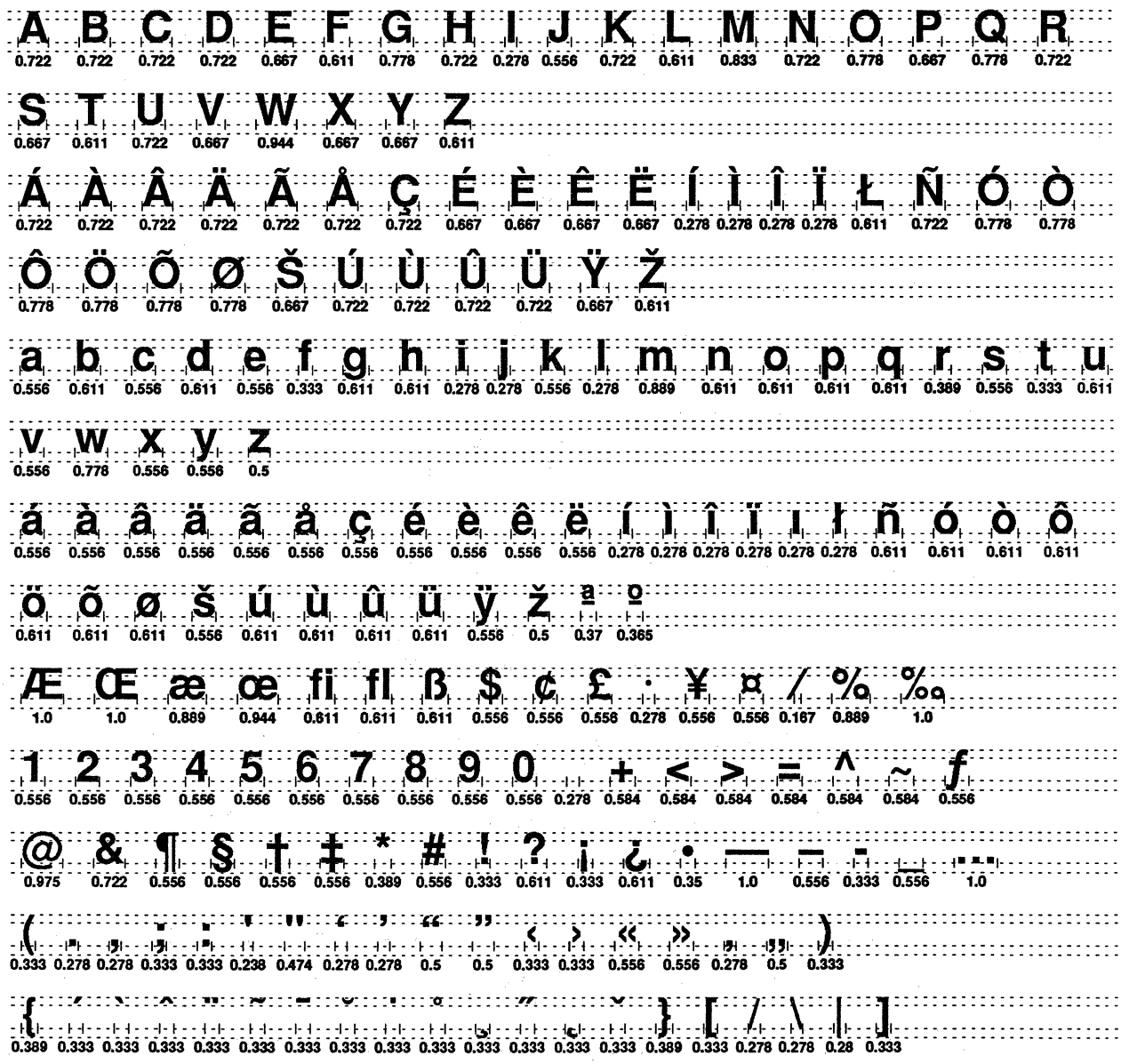


18 pt.



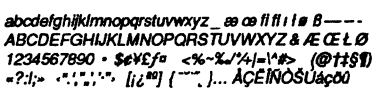
Helvetica Bold

1-point widths

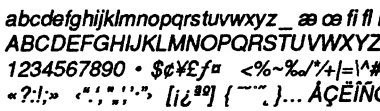


Helvetica Oblique

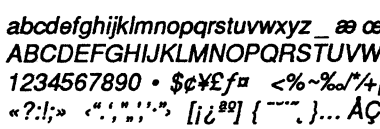
6 pt.



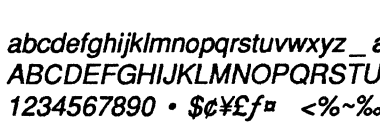
8 pt.



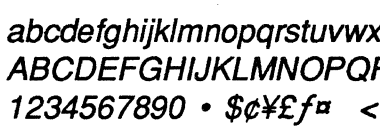
9 pt.



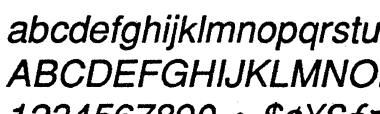
10 pt.



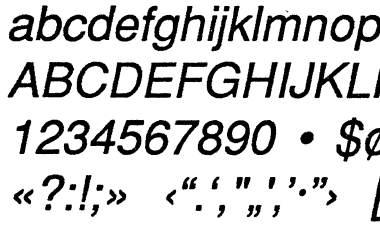
12 pt.



14 pt.

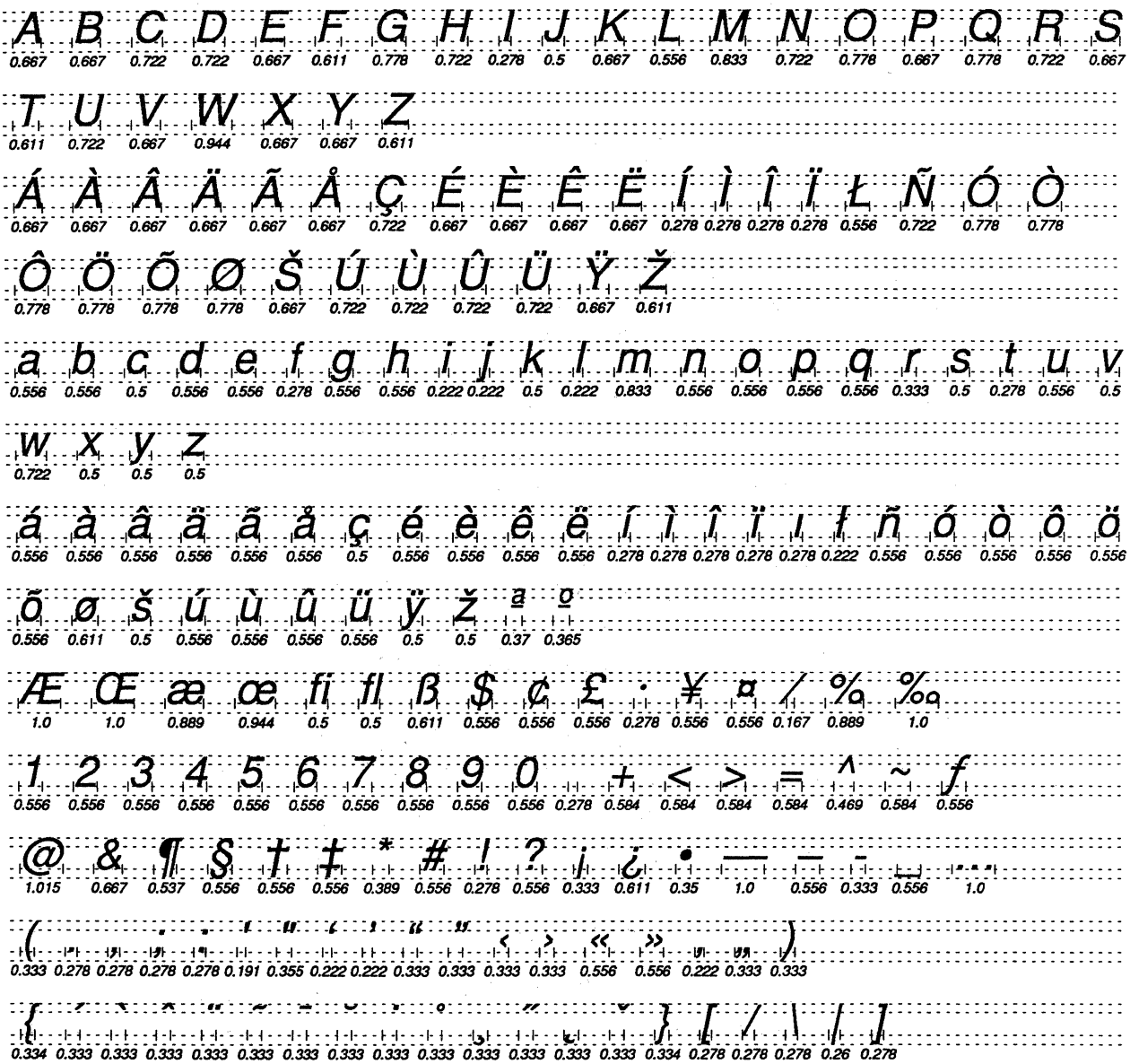


18 pt.



Helvetica Oblique

1-point widths



Helvetica Bold Oblique

6 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„'’·”› [¡¿ªº] { ¯˘˝˛ }... ÅÇËÎÑÒŠÚáçöû

8 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„'’·”› [¡¿ªº] { ¯˘˝˛ }... ÅÇËÎÑÒŠÚáçöû

9 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„'’·”› [¡¿ªº] { ¯˘˝˛ }... ÅÇËÎÑÒŠÚáçöû

10 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„'’·”› [¡¿ªº] { ¯˘˝˛ }... ÅÇËÎÑÒŠÚáçöû

12 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„'’·”› [¡¿ªº] { ¯˘˝˛ }... ÅÇËÎÑÒŠÚáçöû

14 pt.

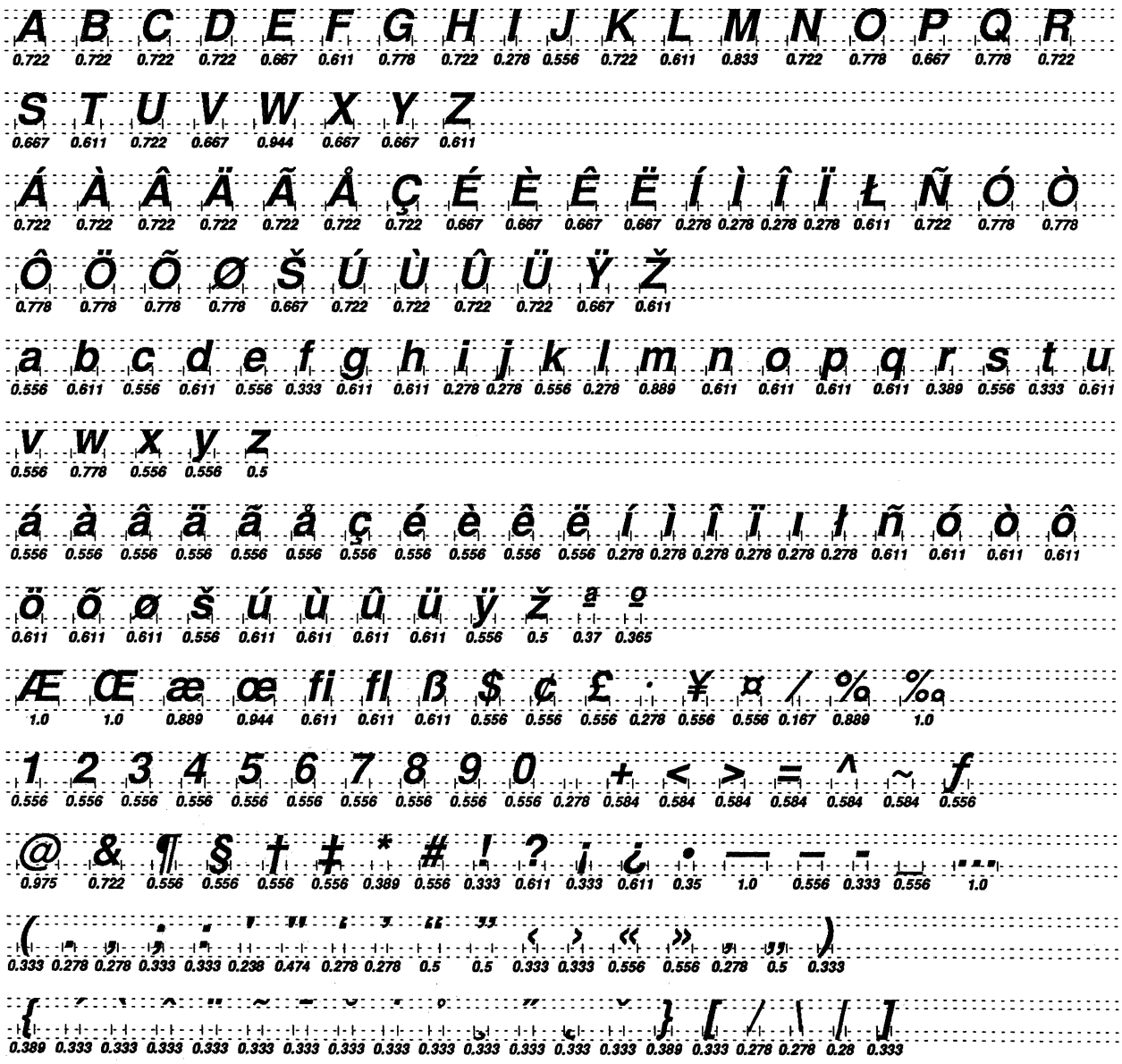
abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„'’·”› [¡¿ªº] { ¯˘˝˛ }... ÅÇËÎÑÒŠÚáçöû

18 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,"„'’·”› [¡¿ªº] { ¯˘˝˛ }... ÅÇËÎÑÒŠÚáçöû

Helvetica Bold Oblique

1-point widths



Symbol

$$a \oplus (b \otimes c) = (a \oplus b) \otimes (a \oplus c)$$

$$\gamma p \supset \Gamma * P$$

$$\neg (p \vee q) = \neg p \wedge \neg q$$
$$\neg (p \wedge q) = \neg p \vee \neg q$$

$$\varepsilon = \min_{x>0} (x \mid (1 + x) \neq 1)$$

$$w(\xi' - \xi'') = \sum_{i=1}^{m} |C_i \cap \{\xi'\}| \cdot |C_i \cap \{\xi''\}| \text{ if } \xi', \xi'' \in L \text{ and } \xi' \neq \xi''$$

$$\bigcup_{i=1}^{n} Z_i(t) \subseteq M$$
$$Z_i(t) \cap Z_j(t) = \varnothing \quad (i \neq j)$$

proposition	true if and only if
$(\forall u)_S(p)$	$S \cap T'_p = \varnothing$
$(\exists u)_S(p)$	$S \cap T_p \neq \varnothing$
$(\forall u)_S(\sim p)$	$S \cap T_p = \varnothing$
$(\exists u)_S(\sim p)$	$S \cap T'_p \neq \varnothing$
$\sim((\forall u)_S(p))$	$S \cap T'_p \neq \varnothing$
$\sim((\exists u)_S(p))$	$S \cap T_p = \varnothing$

$$kp^{k/2} t^{-1} I_k(at) \Leftrightarrow \left[\frac{s + \sqrt{s^2 - 4\lambda\mu}}{2\lambda} \right]^{-k}$$

$$V_C = (1/j\omega C)\, I = (1/\omega C) \underline{/-90°} \cdot I \underline{/\theta} = (1/\omega C)\, I \underline{/\theta - 90°}$$

Symbol supports all characters in the Symbol Set.

Symbol

9 pt.

αβγδεζηθϑικλμνξοπρσςτυφφχψωϖ
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΥΦΧΨΩ ΠΣ
= ≠ ≡ ≅ < > ≤ ≥ ∧ ∨ − + ± × + ≈ ~ ¬ ∞ ∝ ∴
⊗ ⊕ ∅ ∩ ∪ ⊃ ⊇ ⊄ ⊂ ⊆ ∈ ∉ ∋ ∫ ∫ {⋂} [⋂] (⋂) ↑
|• ®©™ ®©™ ♣♦♥♠ | || || || |
↵/◊∠⊥∃∀/ƒ ∂ ⟨ ⟩ _ ∇ ℵ ℑ ℘ ℜ ⌡ {} ⌊⌋ ⋃ ↓
0123456789 °#%′*″ & ·!?:;., … ||
⇓⇔⇑⇐⇒ ↔ ←—→ √‾ ⋃

12 pt.

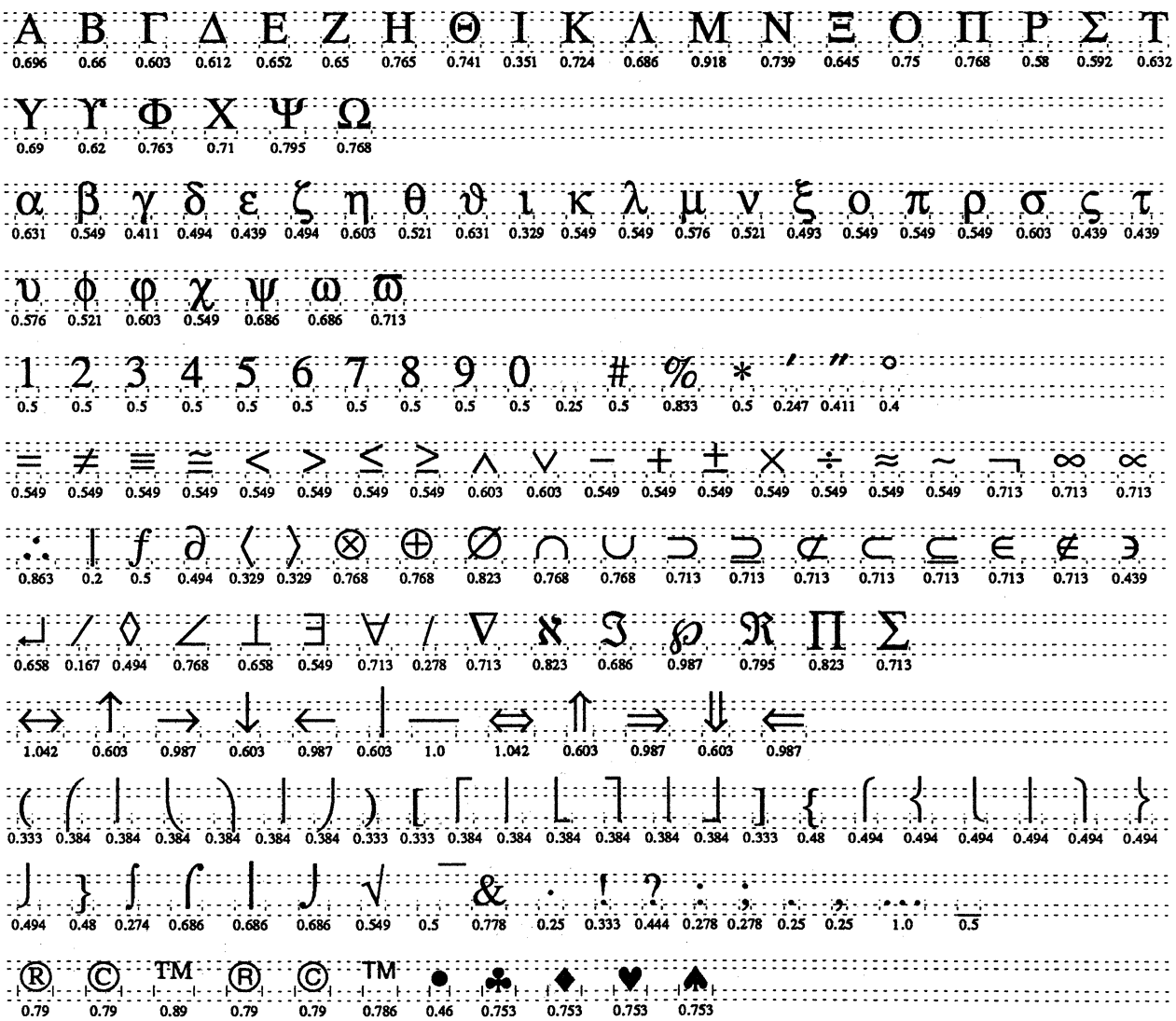
αβγδεζηθϑικλμνξοπρσςτυφφχψωϖ
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΥΦΧΨΩ ΠΣ
= ≠ ≡ ≅ < > ≤ ≥ ∧ ∨ − + ± × ÷ ≈ ~ ¬ ∞ ∝ ∴
⊗ ⊕ ∅ ∩ ∪ ⊃ ⊇ ⊄ ⊂ ⊆ ∈ ∉ ∋ ∫ ∫ {⋂} [⋂] (⋂) ↑
|• ®©™ ®©™ ♣♦♥♠ | || || || |
↵/◊∠⊥∃∀/ƒ ∂ ⟨ ⟩ _ ∇ ℵ ℑ ℘ ℜ ⌡ {} ⌊⌋ ⋃ ↓
0123456789 °#%′*″ & ·!?:;., … ||
⇓⇔⇑⇐⇒ ↔ ←—→ √‾ ⋃

18 pt.

αβγδεζηθϑικλμνξοπρσςτυφφχψωϖ
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΥΦΧΨΩ ΠΣ
= ≠ ≡ ≅ < > ≤ ≥ ∧ ∨ − + ± × ÷ ≈ ~ ¬ ∞ ∝ ∴
⊗ ⊕ ∅ ∩ ∪ ⊃ ⊇ ⊄ ⊂ ⊆ ∈ ∉ ∋ ∫ ∫ {⋂} [⋂] (⋂) ↑
|• ®©™ ®©™ ♣♦♥♠ | || || || |
↵/◊∠⊥∃∀/ƒ ∂ ⟨ ⟩ _ ∇ ℵ ℑ ℘ ℜ ⌡ {} ⌊⌋ ⋃ ↓
0123456789 °#%′*″ & ·!?:;., … ||
⇓⇔⇑⇐⇒ ↔ ←—→ √‾ ⋃

Symbol

1-point widths



Times Family

Times Roman with Italic, Bold and Bold Italic.

11 point text on
12 point linespacing.

The graphic signs called letters are so completely blended with the stream of written thought that their presence therein is as unperceived as *the ticking of a clock in the measurement of time.* Only by an effort of attention does the layman discover that they exist at all. It comes to him as a surprise that these signs should be a matter of concern to any one of the crafts of men. But to be concerned with the shapes of letters is to work in an ancient and fundamental material. The qualities of letter forms at their best are the qualities of a classic time: *order, simplicity, grace.* To try to learn and repeat their excellence is to put oneself under training in a simple and severe school of design.
–William Addison Dwiggins

Times Italic

Architecture began like all scripts. First there was the alphabet. A stone was laid and that was a letter, and each letter was a hieroglyph, and on each hieroglyph there rested a group of ideas.
–Victor Hugo

Times Bold

Decisive, too, for the quality of a letter is, that its various parts, though of limited expressiveness in themselves should combine into a harmonious unity charged with imagination and feeling.
–Albert Windisch

Times Bold Italic

It can be considered a special merit of our time that creative forces are again concerned with the problem of type design – a problem which has been faced by the best artists of every age.
–Walter Tiemann

Times is a trademark of Allied Corporation.

The Times family supports all characters in the Standard Character Set.

Times Roman

6 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

8 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

9 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

10 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

12 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

14 pt.

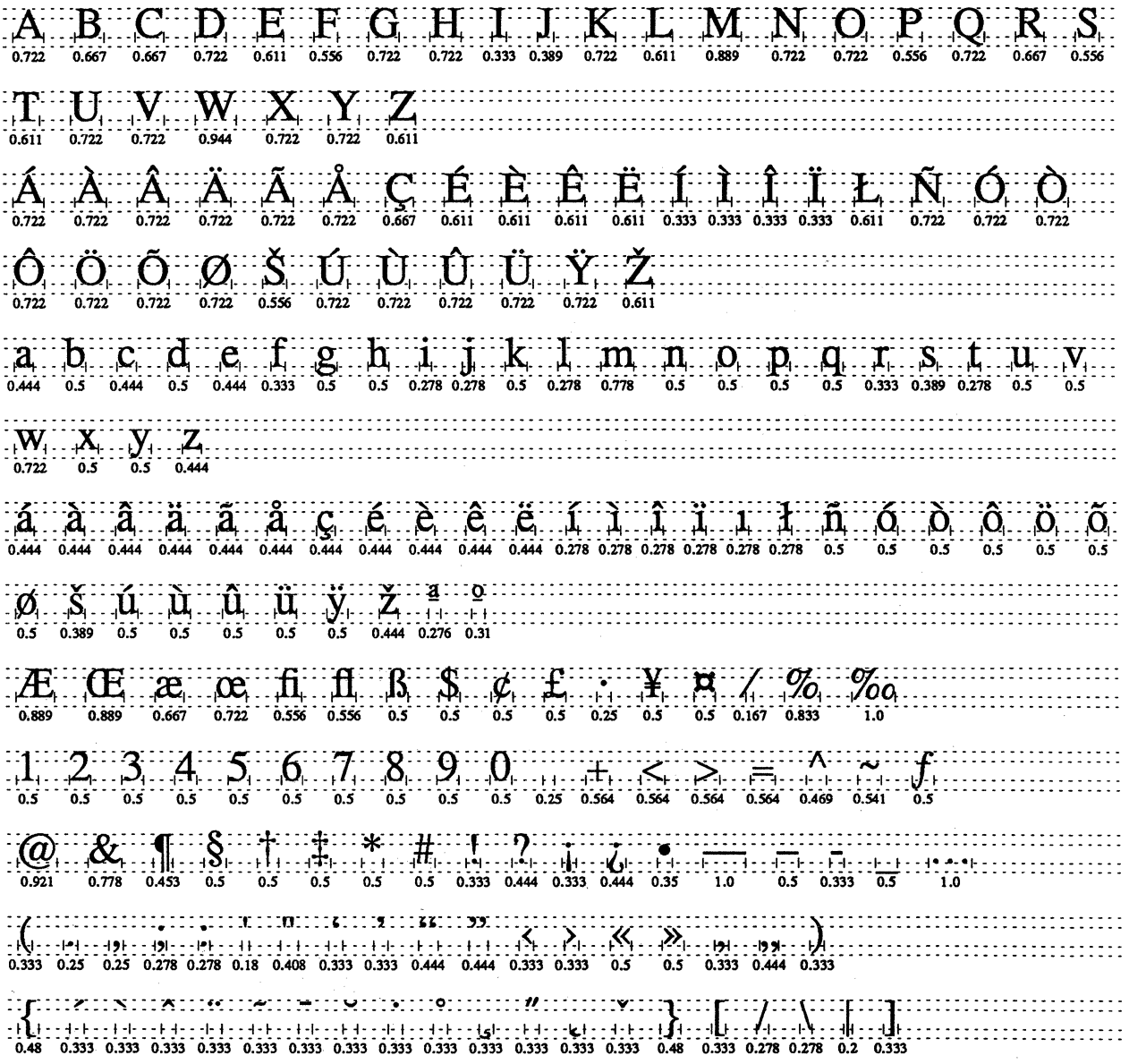
abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

18 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

Times Roman

1-point widths



Times Bold

6 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,',’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

8 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,',’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

9 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,',’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

10 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,',’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

12 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,',’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

14 pt.

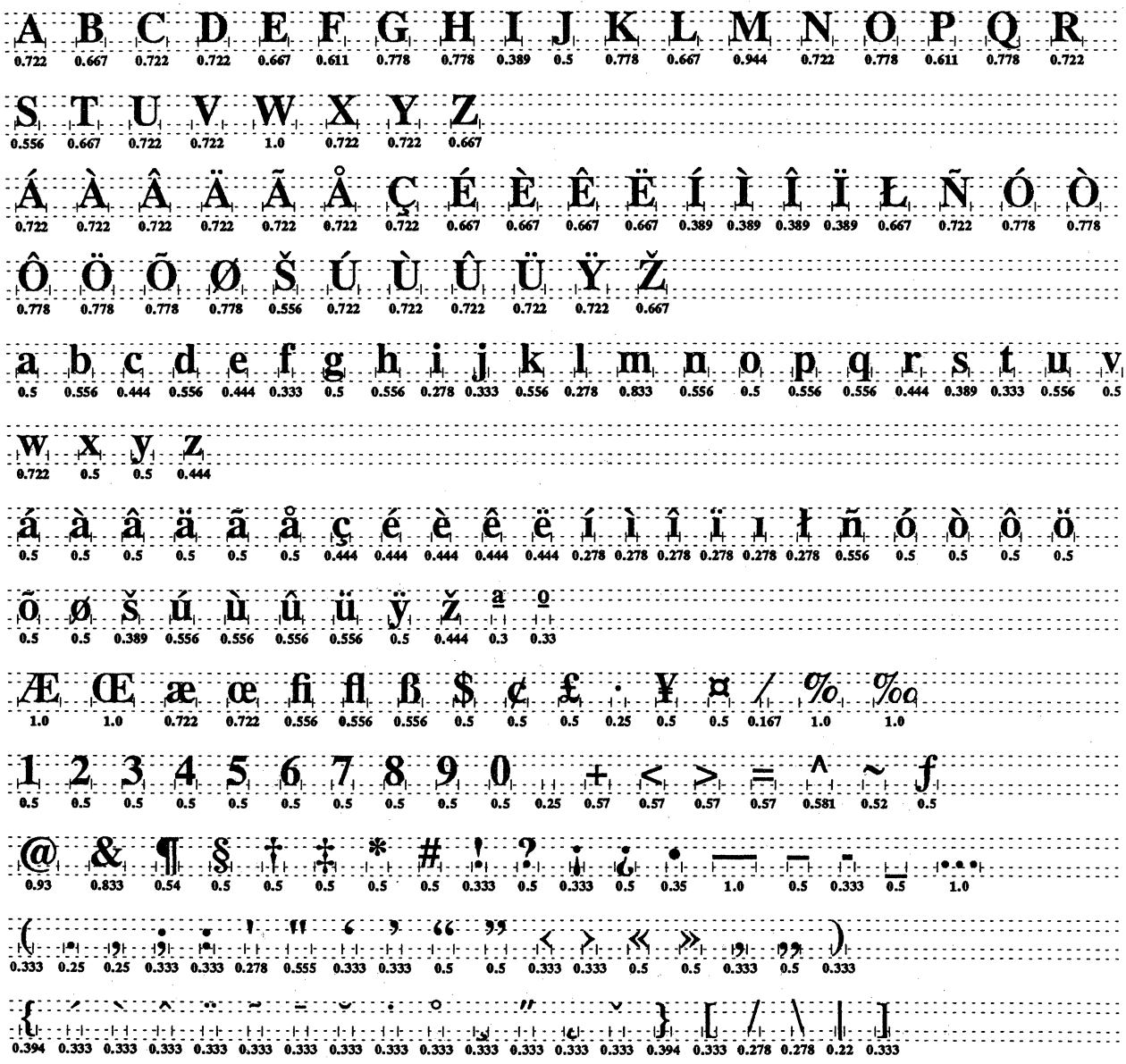
abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,',’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

18 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£ƒ¤ <%~‰/*/+|=\^#> (@†‡§¶)
«?:!;» ‹“.‘,",„,',’·”› [¡¿ªº] { ¯˘·˝˛ }... ÅÇËÎÑÒŠÚáçöû

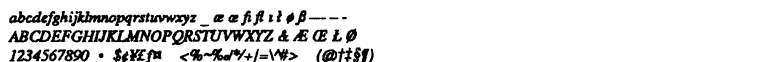
Times Bold

1-point widths

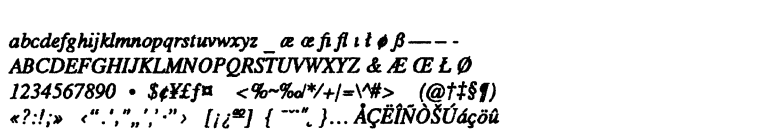


Times Italic

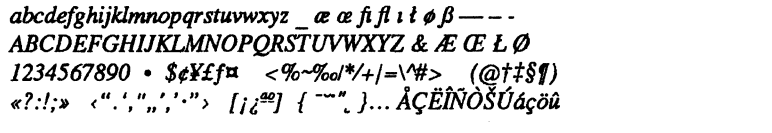
6 pt.



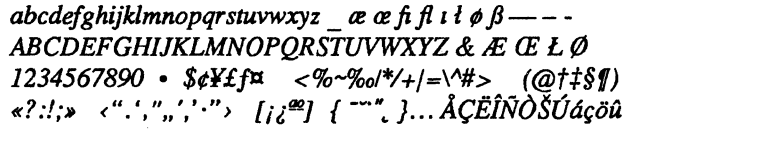
8 pt.



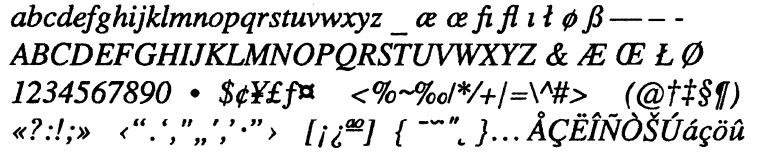
9 pt.



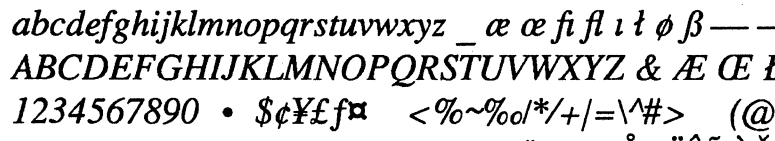
10 pt.



12 pt.



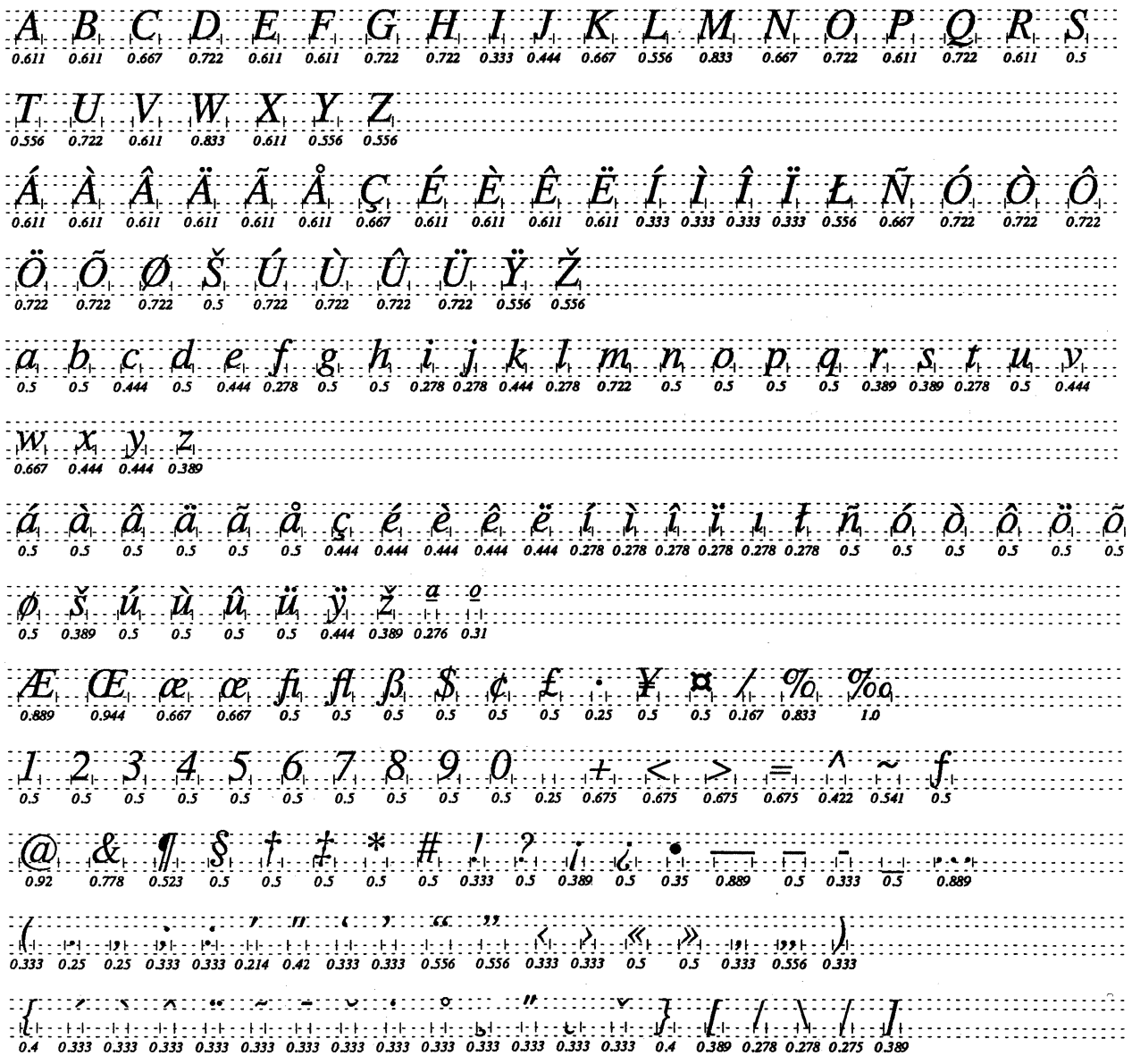
14 pt.



18 pt.

Times Italic

1-point widths



Times Bold Italic

6 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£f¤ <%~‰//+|=\^#> (@†‡§¶)*
«?:!;» ‹“.‘,",„',’·”› [¡¿ªº] { ¯˘˛ }... ÅÇËÎÑÒŠÚáçöû

8 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£f¤ <%~‰//+|=\^#> (@†‡§¶)*
«?:!;» ‹“.‘,",„',’·”› [¡¿ªº] { ¯˘˛ }... ÅÇËÎÑÒŠÚáçöû

9 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£f¤ <%~‰//+|=\^#> (@†‡§¶)*
«?:!;» ‹“.‘,",„',’·”› [¡¿ªº] { ¯˘˛ }... ÅÇËÎÑÒŠÚáçöû

10 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£f¤ <%~‰//+|=\^#> (@†‡§¶)*
«?:!;» ‹“.‘,",„',’·”› [¡¿ªº] { ¯˘˛ }... ÅÇËÎÑÒŠÚáçöû

12 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£f¤ <%~‰//+|=\^#> (@†‡§¶)*
«?:!;» ‹“.‘,",„',’·”› [¡¿ªº] { ¯˘˛ }... ÅÇËÎÑÒŠÚáçöû

14 pt.

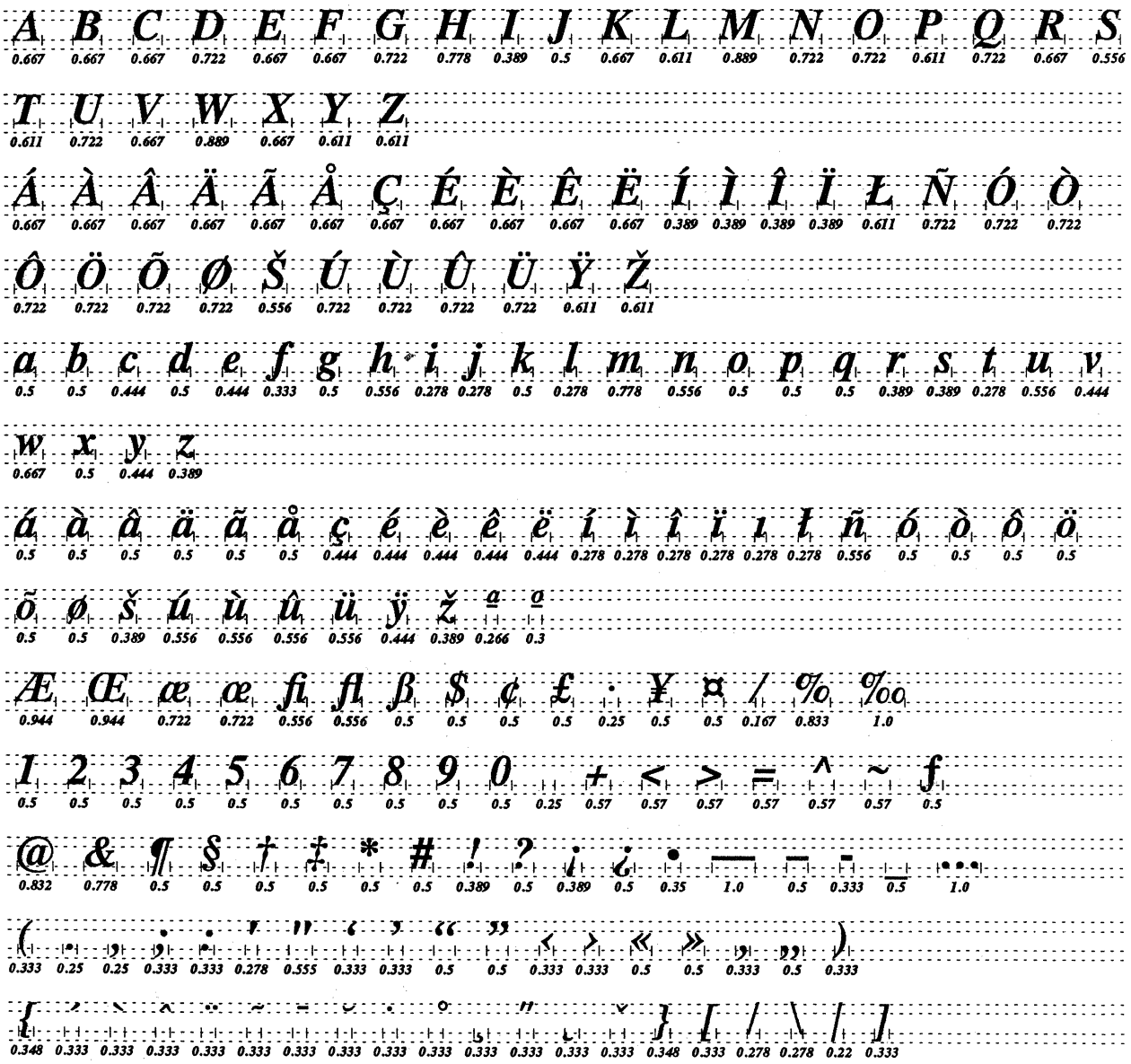
abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£f¤ <%~‰//+|=\^#> (@†‡§¶)*
«?:!;» ‹“.‘,",„',’·”› [¡¿ªº] { ¯˘˛ }... ÅÇËÎÑÒŠÚáçöû

18 pt.

abcdefghijklmnopqrstuvwxyz _ æ œ fi fl ı ł ø ß — – -
ABCDEFGHIJKLMNOPQRSTUVWXYZ & Æ Œ Ł Ø
1234567890 • $¢¥£f¤ <%~‰//+|=\^#> (@†‡§¶)*
«?:!;» ‹“.‘,",„',’·”› [¡¿ªº] { ¯˘˛ }... ÅÇËÎÑÒŠÚáçöû

Times Bold Italic

1-point widths



Encodings

Work Sheet

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																

Standard Encoding

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0			[1]	0	@	P	‘	p						—		
1			!	1	A	Q	a	q			¡	–	`		Æ	æ
2			"	2	B	R	b	r			¢	†	´			
3			#	3	C	S	c	s			£	‡	^ [7]		a	
4			$	4	D	T	d	t			/ [6]	·	~ [8]			
5			%	5	E	U	e	u			¥		¯			ı
6			&	6	F	V	f	v			*f*	¶	˘			
7			’	7	G	W	g	w			§	•	˙			
8			(	8	H	X	h	x			¤	¸	¨		Ł	ł
9			)	9	I	Y	i	y			'	„			Ø	ø
10			* [2]	:	J	Z	j	z			“	”	°		Œ	œ
11			+	;	K	[	k	{			«	»	˛		º	ß
12			,	<	L	\	l	|			‹	…				
13			- [3]	=	M	]	m	}			›	‰	″			
14			.	>	N	^ [4]	n	~ [5]			fi		˛			
15			/	?	O	_	o				fl	¿	ˇ			

|←——— USASCII ———→|

Control character

Not assigned

[1] *Space or blank.*
[2] *Asterisk hangs from capital height.*
[3] *Hyphen.*
[4] *ASCII circumflex.*
[5] *ASCII tilde.*
[6] *Fraction (shallower than slash).*
[7] *Circumflex accent.*
[8] *Tilde accent.*

Standard Character Set

Graphic	Description	Code Name	Octal Code
	Space, blank	space	40
!	Exclamation mark, screamer	exclam	41
"	Double vertical quote	quotedbl	42
#	Number sign, pound, hash mark	numbersign	43
$	Dollar sign	dollar	44
%	Percent	percent	45
&	Ampersand	ampersand	46
'	Right single quote, apostrophe	quoteright	47
(	Left parenthesis	parenleft	50
)	Right parenthesis	parenright	51
*	Asterisk	asterisk	52
+	Plus	plus	53
,	Comma	comma	54
-	Hyphen	hyphen	55
.	Period	period	56
/	Slash, slant, solidus, oblique, stroke	slash	57
0-9	Lining figures	zero-nine	60-71
:	Colon	colon	72
;	Semicolon	semicolon	73
<	Less than	less	74
=	Equal	equal	75
>	Greater than	greater	76
?	Question mark, query	question	77
@	(Commercial) at	at	100
A-Z	Capital alphabet	A-Z	101-132
[	Left bracket	bracketleft	133
\	Backward slash, reverse slash, reverse solidus	backslash	134
]	Right bracket	bracketright	135

Graphic	Description	Code Name	Octal Code
^	ASCII circumflex (large & hangs from top of zero)	asciicircum	136
_	Underscore, underline	underscore	137
‘	Left single quote	quoteleft	140
a-z	Lower-case alphabet	a-z	141-172
{	Left brace	braceleft	173
|	Vertical line or bar	bar	174
}	Right brace	braceright	175
~	ASCII tilde (large & centered like math operator)	asciitilde	176
¡	Inverted exclamation mark	exclamdown	241
¢	Cent sign	cent	242
£	Pound sterling	sterling	243
/	Diagonal fraction bar (shallower than slash)	fraction	244
¥	Yen	yen	245
f	Florin, function (mathematical)	florin	246
§	Section mark	section	247
¤	General currency symbol	currency	250
'	Single vertical quote	quotesingle	251
“	Left double quote	quotedblleft	252
«	Left double angle quote, left guillemet	guillemotleft	253
‹	Left single angle quote, left single guillemet	guilsinglleft	254
›	Right single angle quote, right single guillemet	guilsinglright	255
fi	fi ligature	fi	256
fl	fl ligature	fl	257
–	En dash (medium dash)	endash	261
†	Dagger	dagger	262
‡	Double dagger	daggerdbl	263
·	Period centered vertically, dot	periodcentered	264
¶	Paragraph mark, pilcrow	paragraph	266

Graphic	Description	Code Name	Octal Code
•	Bullet (larger than dot)	bullet	267
‚	Left single quote (on base line)	quotesinglbase	270
„	Left double quote (on base line)	quotedblbase	271
”	Right double quote	quotedblright	272
»	Right double angle quote, right guillemet	guillemotright	273
...	Ellipsis, 3-dot leader	ellipsis	274
‰	Per mill, per thousand	perthousand	275
¿	Inverted question mark	questiondown	277
`	Grave accent	grave	301
´	Acute accent	acute	302
ˆ	Circumflex accent	circumflex	303
˜	Tilde accent	tilde	304
¯	Macron accent	macron	305
˘	Breve accent	breve	306
˙	Dot accent (above)	dotaccent	307
¨	Dieresis or umlaut accent	dieresis	310
°	Ring accent	ring	312
¸	Cedilla accent	cedilla	313
˝	Hungarian umlaut or double acute accent	hungarumlaut	315
˛	Ogonek accent, nasalization sign	ogonek	316
ˇ	Caron or hacek accent	caron	317
—	Em dash (long dash)	emdash	320
Æ	Capital AE diphthong	AE	341
ª	Feminine ordinal indicator	ordfeminine	343
Ł	Capital L with slash (stroke)	Lslash	350
Ø	Capital O with slash	Oslash	351
Œ	Capital OE diphthong	OE	352
º	Masculine ordinal indicator	ordmasculine	353

Graphic	Description	Code Name	Octal Code
æ	Lower-case ae diphthong	ae	361
ı	Dotless lower-case i	dotlessi	365
ł	Lower-case l with slash (stroke)	lslash	370
ø	Lower-case o with slash	oslash	371
œ	Lower-case oe diphthong	oe	372
ß	German double s	germandbls	373

Graphic	Description	Code Name	Octal Code
Á	Capital A with acute accent	Aacute	Unassigned
á	Lower-case a with acute accent	aacute	Unassigned
Â	Capital A with circumflex accent	Acircumflex	Unassigned
â	Lower-case a with circumflex accent	acircumflex	Unassigned
Ä	Capital A with dieresis accent	Adieresis	Unassigned
ä	Lower-case a with dieresis accent	adieresis	Unassigned
À	Capital A with grave accent	Agrave	Unassigned
à	Lower-case a with grave accent	agrave	Unassigned
Å	Capital A with ring accent	Aring	Unassigned
å	Lower-case a with ring accent	aring	Unassigned
Ã	Capital A with tilde accent	Atilde	Unassigned
ã	Lower-case a with tilde accent	atilde	Unassigned
Ç	Capital C with cedilla accent	Ccedilla	Unassigned
ç	Lower-case c with cedilla accent	ccedilla	Unassigned
É	Capital E with acute accent	Eacute	Unassigned
é	Lower-case e with acute accent	eacute	Unassigned
Ê	Capital E with circumflex accent	Ecircumflex	Unassigned
ê	Lower-case e with circumflex accent	ecircumflex	Unassigned
Ë	Capital E with dieresis accent	Edieresis	Unassigned
ë	Lower-case e with dieresis accent	edieresis	Unassigned
È	Capital E with grave accent	Egrave	Unassigned
è	Lower-case e with grave accent	egrave	Unassigned
Í	Capital I with acute accent	Iacute	Unassigned
í	Lower-case i with acute accent	iacute	Unassigned
Î	Capital I with circumflex accent	Icircumflex	Unassigned
î	Lower-case i with circumflex accent	icircumflex	Unassigned
Ï	Capital I with dieresis accent	Idieresis	Unassigned
ï	Lower-case i with dieresis accent	idieresis	Unassigned

Graphic	Description	Code Name	Octal Code
Ì	Capital I with grave accent	Igrave	Unassigned
ì	Lower-case i with grave accent	igrave	Unassigned
Ñ	Capital N with tilde accent	Ntilde	Unassigned
ñ	Lower-case n with tilde accent	ntilde	Unassigned
Ó	Capital O with acute accent	Oacute	Unassigned
ó	Lower-case o with acute accent	oacute	Unassigned
Ô	Capital O with circumflex accent	Ocircumflex	Unassigned
ô	Lower-case o with circumflex accent	ocircumflex	Unassigned
Ö	Capital O with dieresis accent	Odieresis	Unassigned
ö	Lower-case o with dieresis accent	odieresis	Unassigned
Ò	Capital O with grave accent	Ograve	Unassigned
ò	Lower-case o with grave accent	ograve	Unassigned
Õ	Capital O with tilde accent	Otilde	Unassigned
õ	Lower-case o with tilde accent	otilde	Unassigned
Š	Capital S with caron accent	Scaron	Unassigned
š	Lower-case s with caron accent	scaron	Unassigned
Ú	Capital U with acute accent	Uacute	Unassigned
ú	Lower-case u with acute accent	uacute	Unassigned
Û	Capital U with circumflex accent	Ucircumflex	Unassigned
û	Lower-case u with circumflex accent	ucircumflex	Unassigned
Ü	Capital U with dieresis accent	Udieresis	Unassigned
ü	Lower-case u with dieresis accent	udieresis	Unassigned
Ù	Capital U with grave accent	Ugrave	Unassigned
ù	Lower-case u with grave accent	ugrave	Unassigned
Ÿ	Capital Y with dieresis accent	Ydieresis	Unassigned
ÿ	Lower-case y with dieresis accent	ydieresis	Unassigned
Ž	Capital Z with caron accent	Zcaron	Unassigned
ž	Lower-case z with caron accent	zcaron	Unassigned

Symbol Encoding

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0			[1]	0	≅	Π	[4]	π				°	ℵ	∠	◊	
1			!	1	A	Θ	α	θ			ϒ	±	ℑ	∇	〈	〉
2			∀	2	B	P	β	ρ			′	″	ℜ	®	®	∫
3			#	3	X	Σ	χ	σ			≤	≥	℘	©	©	⌠
4			∃	4	Δ	T	δ	τ			/ [6]	×	⊗	™	™	⌡
5			%	5	E	Y	ε	υ			∞	∝	⊕	Π	Σ	⎛
6			&	6	Φ	ς	ϕ	ϖ			*f*	∂	∅	√	⌠	⎞
7			ə	7	Γ	Ω	γ	ω			♣	•	∩	·	⌡	⌡
8			(	8	H	Ξ	η	ξ			♦	÷	∪	¬	⎛	⎝
9			)	9	I	Ψ	ι	ψ			♥	≠	⊃	∧	⌈	⌉
10			* [2]	:	ϑ	Z	φ	ζ			♠	≡	⊇	∨	⌡	⌡
11			+	;	K	[	κ	{			↔	≈	⊄	⇔	⎣	⎦
12			,	<	Λ	∴	λ	|			←	…	⊂	⇐	⌈	⌉
13			− [3]	=	M	]	μ	}			↑	⏐ [7]	⊆	⇑	⎧	⎫
14			.	>	N	⊥	ν	~ [5]			→	— [8]	∈	⇒	⎩	⎭
15			/	?	O	_	o				↓	↵	∉	⇓	⏐ [9]	

Control character

Not assigned

[1] *Space or blank.*
[2] *Asterisk at height of math operator.*
[3] *Minus.*
[4] *Extension for radical.*
[5] *Approximately equal to, similar to.*
[6] *Fraction (shallower than slash).*
[7] *Extension for upward/downward arrow.*
[8] *Extension for leftward/rightward arrow.*
[9] *Extension for left/right brace.*

Symbol Set

Graphic	Description	Code Name	Octal Code
	Space, blank	space	40
!	Exclamation mark, screamer	exclam	41
∀	Universal quantifier, "For every"	universal	42
#	Number sign, pound, hash mark	numbersign	43
∃	Existential quantifier, "There exists"	existential	44
%	Percent	percent	45
&	Ampersand	ampersand	46
∍	Such that	suchthat	47
(	Left parenthesis	parenleft	50
)	Right parenthesis	parenright	51
*	Asterisk (centered like math operator)	asteriskmath	52
+	Plus	plus	53
,	Comma	comma	54
−	Minus	minus	55
.	Period	period	56
/	Slash, slant, solidus, oblique, stroke	slash	57
0-9	Lining figures	zero-nine	60-71
:	Colon	colon	72
;	Semicolon	semicolon	73
<	Less than	less	74
=	Equal	equal	75
>	Greater than	greater	76
?	Question mark, query	question	77
≅	Congruent (same shape and same size)	congruent	100
A	Capital alpha	Alpha	101
B	Capital beta	Beta	102
X	Capital chi	Chi	103
Δ	Capital delta	Delta	104

Graphic	Description	Code Name	Octal Code
E	Capital epsilon	Epsilon	105
Φ	Capital phi	Phi	106
Γ	Capital gamma	Gamma	107
H	Capital eta	Eta	110
I	Capital iota	Iota	111
ϑ	Alternate lower-case theta	theta1	112
K	Capital kappa	Kappa	113
Λ	Capital lambda	Lambda	114
M	Capital mu	Mu	115
N	Capital nu	Nu	116
O	Capital omicron	Omicron	117
Π	Capital pi	Pi	120
Θ	Capital theta	Theta	121
P	Capital rho	Rho	122
Σ	Capital sigma	Sigma	123
T	Capital tau	Tau	124
Y	Capital upsilon	Upsilon	125
ς	Alternate lower-case sigma	sigma1	126
Ω	Capital omega, ohm	Omega	127
Ξ	Capital xi	Xi	130
Ψ	Capital psi	Psi	131
Z	Capital zeta	Zeta	132
[	Left bracket	bracketleft	133
∴	Hence, therefore	therefore	134
]	Right bracket	bracketright	135
⊥	Perpendicular	perpendicular	136
_	Underscore, underline	underscore	137
	Extension for radical	radicalex	140

Graphic	Description	Code Name	Octal Code
α	Lower-case alpha	alpha	141
β	Lower-case beta	beta	142
χ	Lower-case chi	chi	143
δ	Lower-case delta	delta	144
ε	Lower-case epsilon	epsilon	145
ϕ	Lower-case phi	phi	146
γ	Lower-case gamma	gamma	147
η	Lower-case eta	eta	150
ι	Lower-case iota	iota	151
φ	Alternate lower-case phi	phi1	152
κ	Lower-case kappa	kappa	153
λ	Lower-case lambda	lambda	154
μ	Lower-case mu, micron	mu	155
ν	Lower-case nu	nu	156
ο	Lower-case omicron	omicron	157
π	Lower-case pi	pi	160
θ	Lower-case theta	theta	161
ρ	Lower-case rho	rho	162
σ	Lower-case sigma	sigma	163
τ	Lower-case tau	tau	164
υ	Lower-case upsilon	upsilon	165
ϖ	Alternate lower-case omega	omega1	166
ω	Lower-case omega	omega	167
ξ	Lower-case xi	xi	170
ψ	Lower-case psi	psi	171
ζ	Lower-case zeta	zeta	172
{	Left brace	braceleft	173
|	Vertical line or bar	bar	174

Graphic	Description	Code Name	Octal Code
}	Right brace	braceright	175
~	Approximately equal, similar, difference	similar	176
ϒ	Alternate upsilon	Upsilon1	241
′	Foot, minute, prime	minute	242
≤	Less than or equal	lessequal	243
/	Diagonal fraction bar (shallower than slash)	fraction	244
∞	Infinity	infinity	245
f	Florin, function (mathematical)	florin	246
♣	Club	club	247
♦	Diamond	diamond	250
♥	Heart	heart	251
♠	Spade	spade	252
↔	Left-and-right arrow	arrowboth	253
←	Leftward arrow	arrowleft	254
↑	Upward arrow	arrowup	255
→	Rightward arrow	arrowright	256
↓	Downward arrow	arrowdown	257
°	Degree (hangs from cap height)	degree	260
±	Plus or minus	plusminus	261
″	Inch, second, double prime	second	262
≥	Greater than or equal	greaterequal	263
×	Multiplication	multiply	264
∝	Varies directly as, proportional	proportional	265
∂	Partial differential	partialdiff	266
•	Bullet (larger than dot)	bullet	267
÷	Division	divide	270
≠	Not equal	notequal	271
≡	Equivalent, identical with, congruent	equivalence	272

Graphic	Description	Code Name	Octal Code
≈	Nearly or approximately equal	approxequal	273
...	Ellipsis, 3-dot leader	ellipsis	274
|	Extension for upward/downward arrow	arrowvertex	275
—	Extension for leftward/rightward arrow	arrowhorizex	276
↵	Carriage return and line feed	carriagereturn	277
ℵ	Aleph, transfinite cardinal number	aleph	300
ℑ	Fraktur I, imaginary number	Ifraktur	301
ℜ	Fraktur R, real number	Rfraktur	302
℘	Script P, Weierstrass elliptic function	weierstrass	303
⊗	Set (vector) multiplication	circlemultiply	304
⊕	Set (vector) summation	circleplus	305
∅	Null set, empty set	emptyset	306
∩	Intersection, product	intersection	307
∪	Union, sum, join	union	310
⊃	Contains as proper sub-class, implies	propersuperset	311
⊇	Contains as sub-class	reflexsuperset	312
⊄	Not contained as proper sub-class within	notsubset	313
⊂	Contained as proper sub-class within	propersubset	314
⊆	Contained as sub-class within	reflexsubset	315
∈	Member or element of a set	element	316
∉	Not a member or element of a set	notelement	317
∠	Angle	angle	320
∇	Gradient, divergence, curl	gradient	321
®	Registered (serif version)	registerserif	322
©	Copyright (serif version)	copyrightserif	323
™	Trademark (serif version)	trademarkserif	324
∏	Product (larger than capital pi)	product	325
√	Long division, square root, radical	radical	326

Graphic	Description	Code Name	Octal Code
·	Dot (centered like math operator)	dotmath	327
¬	Negation, logical NOT	logicalnot	330
∧	Logical AND	logicaland	331
∨	Logical OR	logicalor	332
⇔	Double left-and-right arrow	arrowdblboth	333
⇐	Double leftward arrow	arrowdblleft	334
⇑	Double upward arrow	arrowdblup	335
⇒	Double rightward arrow	arrowdblright	336
⇓	Double downward arrow	arrowdbldown	337
◊	Lozenge, subtotal, diamond	lozenge	340
⟨	Left angle bracket	angleleft	341
®	Registered (sans-serif version)	registersans	342
©	Copyright (sans-serif version)	copyrightsans	343
™	Trademark (sans-serif version)	trademarksans	344
Σ	Summation (larger than capital sigma)	summation	345
⎛	Extensible left parenthesis: top	parenlefttp	346
⎜	Extension for left parenthesis	parenleftex	347
⎝	Extensible left parenthesis: bottom	parenleftbt	350
⎡	Extensible left bracket: top (ceiling)	bracketlefttp	351
⎢	Extension for left bracket	bracketleftex	352
⎣	Extensible left bracket: bottom (floor)	bracketleftbt	353
⎧	Extensible left brace: top	braceleftp	354
⎨	Extensible left brace: middle	braceleftmid	355
⎩	Extensible left brace: bottom	braceleftbt	356
⎪	Extension for left/right brace	braceex	357
⟩	Right angle bracket	angleright	361
∫	Integral	integral	362
⌠	Extensible integral: top	integraltp	363

Graphic	Description	Code Name	Octal Code
⎮	Extension for integral	integralex	364
⌡	Extensible integral: bottom	integralbt	365
⎞	Extensible right parenthesis: top	parenrighttp	366
⎟	Extension for right parenthesis	parenrightex	367
⎠	Extensible right parenthesis: bottom	parenrightbt	370
⎤	Extensible right bracket: top (ceiling)	bracketrighttp	371
⎥	Extension for right bracket	bracketrightex	372
⎦	Extensible right bracket: bottom (floor)	bracketrightbt	373
⎫	Extensible right brace: top	bracerighttp	374
⎬	Extensible right brace: middle	bracerightmid	375
⎭	Extensible right brace: bottom	bracerightbt	376

Appendix: Updates

Updates

Revision 2

Additions

General Information.

Languages.

Courier Family: width tables and clarification of character set.

Symbol: samples.

Corrections

Standard Encoding and Character Set: the accented characters have no default positions.

The widths of the following characters have been changed in Times, Helvetica and Symbol, to be equal to 50% of the widths of the lining figures (0-9):

Comma
Dot, in Symbol
Period
Period centered vertically, in Times and Helvetica
Space

Courier samples: the point size of the last sample on each page of graduated text is 15, not 13.3.

Symbol width table: the width of the extender for the vertical arrows is 0.603, not 0.247.

Apple Supplement

Apple QuickDraw Encoding 1
Apple QuickDraw Character Set 3

Apple Symbol Encoding 9
Apple Symbol Widths 10
Apple Symbol Set 11

Apple Updates 19

Revision 2
2 October 1984

Apple, the Apple logo, and QuickDraw are trademarks of Apple Computer, Inc.

Apple QuickDraw Encoding

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL	DLE	SP	0	@	P	`	p	Ä	ê	†	∞	¿	–	‡	
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—	·	Ò
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	“	‚	Ú
3	ETX	DC3	#	3	C	S	c	s	É	ì	£	≥	√	”	„	Û
4	EOT	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘	‰	Ù
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈	’	Â	ı
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷	Ê	ˆ
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	Σ	«	◊	À	˜
8	BS	CAN	(	8	H	X	h	x	à	ò	®	∏	»	ÿ	Ë	¯
9	HT	EM	)	9	I	Y	i	y	â	ô	©	π	…	Ÿ	È	˘
10	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫	␣	⁄	Í	˙
11	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª	Á	¤	Î	°
12	FF	FS	,	<	L	\	l	|	å	ú	¨	º	Ã	‹	Ï	¸
13	CR	GS	–	=	M	]	m	}	ç	ù	≠	Ω	Õ	›	Ì	˝
14	SO	RS	.	>	N	^	n	~	é	û	Æ	æ	Œ	fi	Ó	˛
15	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø	œ	fl	Ô	ˇ

Additions

The first 33 symbols and DEL *do not print. They refer to ASCII control codes.*

SP and ␣ indicate word spaces.

The following characters from the Standard Character Set are not included in this encoding: Ł, ł, Š, š, Ž and ž.

Apple QuickDraw Character Set

Graphic	Description	Code Name	Octal Code
. . . . .	Space, blank	space	40
!	Exclamation mark, screamer	exclam	41
"	Double vertical quote	quotedbl	42
#	Number sign, pound, hash mark	numbersign	43
$	Dollar sign	dollar	44
%	Percent	percent	45
&	Ampersand	ampersand	46
'	Single vertical quote	quotesingle	47
(.	Left parenthesis	parenleft	50
)	Right parenthesis	parenright	51
*	Asterisk	asterisk	52
+	Plus	plus	53
, ...	Comma	comma	54
-	Hyphen	hyphen	55
. . . . .	Period	period	56
/	Slash, slant, solidus, oblique, stroke	slash	57
0-9 . . .	Lining figures	zero-nine	60-71
:	Colon	colon	72
;	Semicolon	semicolon	73
<	Less than	less	74
=	Equal	equal	75
>	Greater than	greater	76
?	Question mark, query	question	77
@	(Commercial) at	at	100
A-Z . . .	Capital alphabet	A-Z	101-132
[.	Left bracket	bracketleft	133
\	Backward slash, reverse slash, reverse solidus	backslash	134
]	Right bracket	bracketright	135

Graphic	Description	Code Name	Octal Code
^	ASCII circumflex (large & hangs from top of zero)	asciicircum	136
_	Underscore, underline	underscore	137
`	Grave accent	grave	140
a-z	Lower-case alphabet	a-z	141-172
{	Left brace	braceleft	173
|	Vertical line or bar	bar	174
}	Right brace	braceright	175
~	ASCII tilde (large & centered like math operator)	asciitilde	176
Ä	Capital A with dieresis accent	Adieresis	200
Å	Capital A with ring accent	Aring	201
Ç	Capital C with cedilla accent	Ccedilla	202
É	Capital E with acute accent	Eacute	203
Ñ	Capital N with tilde accent	Ntilde	204
Ö	Capital O with dieresis accent	Odieresis	205
Ü	Capital U with dieresis accent	Udieresis	206
á	Lower-case a with acute accent	aacute	207
à	Lower-case a with grave accent	agrave	210
â	Lower-case a with circumflex accent	acircumflex	211
ä	Lower-case a with dieresis accent	adieresis	212
ã	Lower-case a with tilde accent	atilde	213
å	Lower-case a with ring accent	aring	214
ç	Lower-case c with cedilla accent	ccedilla	215
é	Lower-case e with acute accent	eacute	216
è	Lower-case e with grave accent	egrave	217
ê	Lower-case e with circumflex accent	ecircumflex	220
ë	Lower-case e with dieresis accent	edieresis	221
í	Lower-case i with acute accent	iacute	222
ì	Lower-case i with grave accent	igrave	223

Graphic	Description	Code Name	Octal Code
î	Lower-case i with circumflex accent	icircumflex	224
ï	Lower-case i with dieresis accent	idieresis	225
ñ	Lower-case n with tilde accent	ntilde	226
ó	Lower-case o with acute accent	oacute	227
ò	Lower-case o with grave accent	ograve	230
ô	Lower-case o with circumflex accent	ocircumflex	231
ö	Lower-case o with dieresis accent	odieresis	232
õ	Lower-case o with tilde accent	otilde	233
ú	Lower-case u with acute accent	uacute	234
ù	Lower-case u with grave accent	ugrave	235
û	Lower-case u with circumflex accent	ucircumflex	236
ü	Lower-case u with dieresis accent	udieresis	237
†	Dagger	dagger	240
°	Degree (hangs from cap height)	degree	241
¢	Cent sign	cent	242
£	Pound sterling	sterling	243
§	Section mark	section	244
•	Bullet (larger than dot)	bullet	245
¶	Paragraph mark, pilcrow	paragraph	246
ß	German double s	germandbls	247
®	Registered	registerserif	250
©	Copyright	copyrightserif	251
™	Trademark	trademarkserif	252
´	Acute accent	acute	253
¨	Dieresis or umlaut accent	dieresis	254
≠	Not equal	notequal	255
Æ	Capital AE diphthong	AE	256
Ø	Capital O with slash	Oslash	257

Graphic	Description	Code Name	Octal Code
∞	Infinity	infinity	260
±	Plus or minus	plusminus	261
≤	Less than or equal	lessequal	262
≥	Greater than or equal	greaterequal	263
¥	Yen	yen	264
µ	Lower-case mu, micron	mu	265
∂	Partial differential	partialdiff	266
Σ	Summation (larger than capital sigma)	summation	267
∏	Product (larger than capital pi)	product	270
π	Lower-case pi	pi	271
∫	Integral	integral	272
ª	Feminine ordinal indicator	ordfeminine	273
º	Masculine ordinal indicator	ordmasculine	274
Ω	Capital omega, ohm	Omega	275
æ	Lower-case ae diphthong	ae	276
ø	Lower-case o with slash	oslash	277
¿	Inverted question mark	questiondown	300
¡	Inverted exclamation mark	exclamdown	301
¬	Negation, logical NOT	logicalnot	302
√	Long division, square root, radical	radical	303
f	Florin, function (mathematical)	florin	304
≈	Nearly or approximately equal	approxequal	305
Δ	Capital delta	Delta	306
«	Left double angle quote, left guillemet	guillemotleft	307
»	Right double angle quote, right guillemet	guillemotright	310
...	Ellipsis, 3-dot leader	ellipsis	311
	Space, blank	space	312
Á	Capital A with acute accent	Aacute	313

Graphic	Description	Code Name	Octal Code
Ã	Capital A with tilde accent	Atilde	314
Õ	Capital O with tilde accent	Otilde	315
Œ	Capital OE diphthong	OE	316
œ	Lower-case oe diphthong	oe	317
–	En dash (medium dash)	endash	320
—	Em dash (long dash)	emdash	321
“	Left double quote	quotedblleft	322
”	Right double quote	quotedblright	323
‘	Left single quote	quoteleft	324
’	Right single quote, apostrophe	quoteright	325
÷	Division	divide	326
◊	Lozenge, subtotal, diamond	lozenge	327
ÿ	Lower-case y with dieresis accent	ydieresis	330
Ÿ	Capital Y with dieresis accent	Ydieresis	331
/	Diagonal fraction bar (shallower than slash)	fraction	332
¤	General currency symbol	currency	333
‹	Left single angle quote, left single guillemet	guilsinglleft	334
›	Right single angle quote, right single guillemet	guilsinglright	335
fi	fi ligature	fi	336
fl	fl ligature	fl	337
‡	Double dagger	daggerdbl	340
·	Period centered vertically, dot	periodcentered	341
,	Left single quote (on base line)	quotesinglbase	342
„	Left double quote (on base line)	quotedblbase	343
‰	Per mill, per thousand	perthousand	344
Â	Capital A with circumflex accent	Acircumflex	345
Ê	Capital E with circumflex accent	Ecircumflex	346
À	Capital A with grave accent	Agrave	347

Graphic	Description	Code Name	Octal Code
Ë	Capital E with dieresis accent	Edieresis	350
È	Capital E with grave accent	Egrave	351
Í	Capital I with acute accent	Iacute	352
Î	Capital I with circumflex accent	Icircumflex	353
Ï	Capital I with dieresis accent	Idieresis	354
Ì	Capital I with grave accent	Igrave	355
Ó	Capital O with acute accent	Oacute	356
Ô	Capital O with circumflex accent	Ocircumflex	357
	Apple logo	apple	360
Ò	Capital O with grave accent	Ograve	361
Ú	Capital U with acute accent	Uacute	362
Û	Capital U with circumflex accent	Ucircumflex	363
Ù	Capital U with grave accent	Ugrave	364
ı	Dotless lower-case i	dotlessi	365
ˆ	Circumflex accent	circumflex	366
˜	Tilde accent	tilde	367
¯	Macron accent	macron	370
˘	Breve accent	breve	371
˙	Dot accent (above)	dotaccent	372
˚	Ring accent	ring	373
¸	Cedilla accent	cedilla	374
˝	Hungarian umlaut or double acute accent	hungarumlaut	375
˛	Ogonek accent, nasalization sign	ogonek	376
ˇ	Caron or hacek accent	caron	377

Apple Symbol Encoding

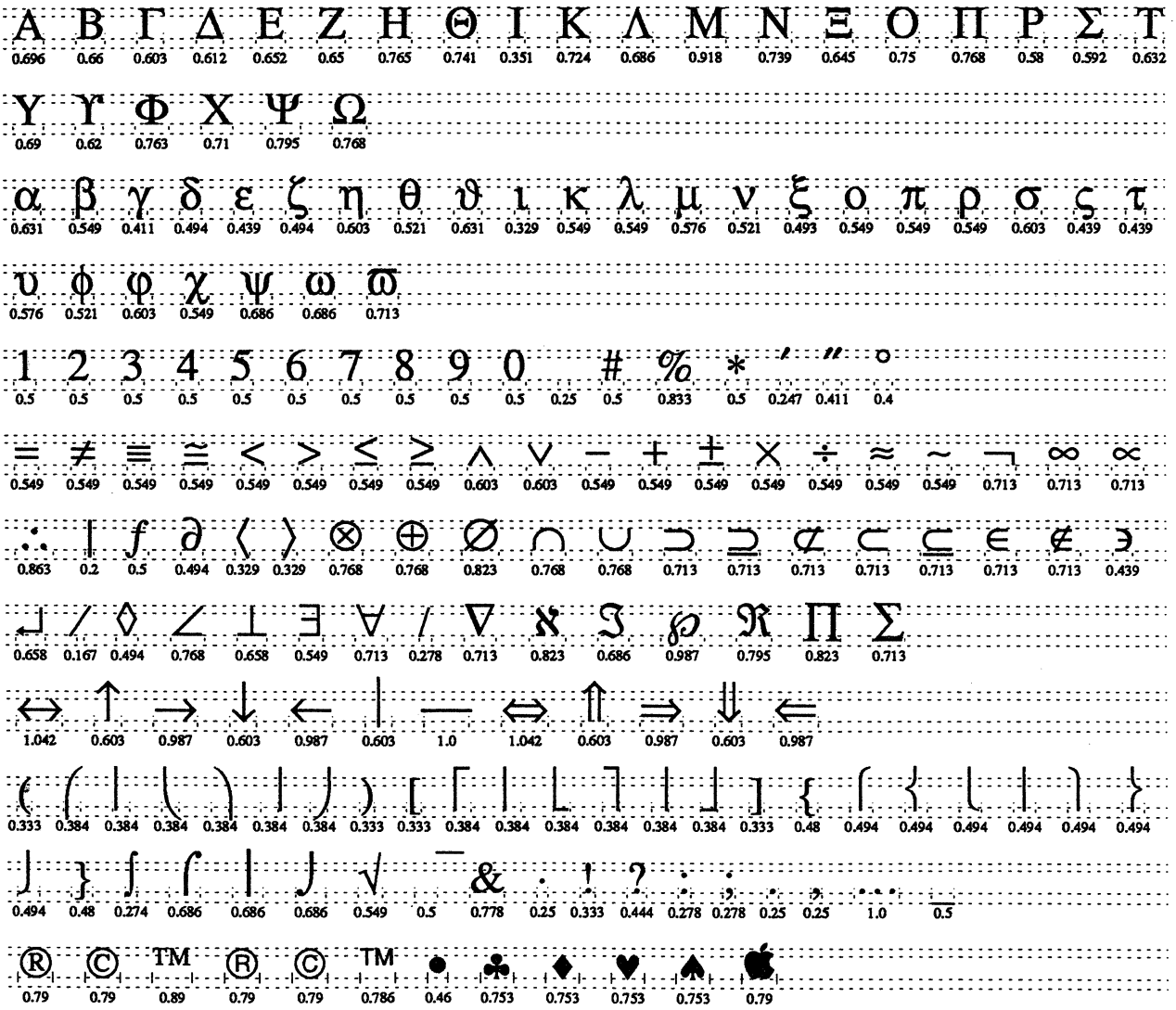
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0			[1]	0	≅	Π	[4]	π				°	ℵ	∠	◊	
1			!	1	Α	Θ	α	θ			ϒ	±	ℑ	∇	〈	〉
2			∀	2	Β	Ρ	β	ρ			′	″	ℜ	®	®	∫
3			#	3	Χ	Σ	χ	σ			≤	≥	℘	©	©	⌠
4			∃	4	Δ	Τ	δ	τ			⁄ [6]	×	⊗	™	™	⎮
5			%	5	Ε	Υ	ε	υ			∞	∝	⊕	∏	∑	⌡
6			&	6	Φ	ς	φ	ϖ			ƒ	∂	∅	√	⎛	⎞
7			∍	7	Γ	Ω	γ	ω			♣	•	∩	·	⎜	⎜
8			(	8	Η	Ξ	η	ξ			♦	÷	∪	¬	⎝	⎠
9			)	9	Ι	Ψ	ι	ψ			♥	≠	⊃	∧	⎡	⎤
10			* [2]	:	ϑ	Ζ	φ	ζ			♠	≡	⊇	∨	⎜	⎜
11			+	;	Κ	[	κ	{			↔	≈	⊄	⇔	⎣	⎦
12			,	<	Λ	∴	λ	|			←	…	⊂	⇐	⎡	⎤
13			− [3]	=	Μ	]	μ	}			↑	⎪ [7]	⊆	⇑	⎧	⎫
14			.	>	Ν	⊥	ν	~ [5]			→	— [8]	∈	⇒	⎩	⎭
15			/	?	Ο	_	ο				↓	↵	∉	⇓	⎪ [9]	

Control character

[1] *Space or blank.*
[2] *Asterisk at height of math operator.*
[3] *Minus.*
[4] *Extension for radical.*
[5] *Approximately equal to, similar to.*
[6] *Fraction (shallower than slash).*
[7] *Extension for upward/downward arrow.*
[8] *Extension for leftward/rightward arrow.*
[9] *Extension for left/right brace.*

Apple Symbol Widths

1-point widths



Apple Symbol Set

Graphic	Description	Code Name	Octal Code
	Space, blank	space	40
!	Exclamation mark, screamer	exclam	41
∀	Universal quantifier, "For every"	universal	42
#	Number sign, pound, hash mark	numbersign	43
∃	Existential quantifier, "There exists"	existential	44
%	Percent	percent	45
&	Ampersand	ampersand	46
∍	Such that	suchthat	47
(	Left parenthesis	parenleft	50
)	Right parenthesis	parenright	51
*	Asterisk (centered like math operator)	asteriskmath	52
+	Plus	plus	53
,	Comma	comma	54
−	Minus	minus	55
.	Period	period	56
/	Slash, slant, solidus, oblique, stroke	slash	57
0-9	Lining figures	zero-nine	60-71
:	Colon	colon	72
;	Semicolon	semicolon	73
<	Less than	less	74
=	Equal	equal	75
>	Greater than	greater	76
?	Question mark, query	question	77
≅	Congruent (same shape and same size)	congruent	100
A	Capital alpha	Alpha	101
B	Capital beta	Beta	102
X	Capital chi	Chi	103
Δ	Capital delta	Delta	104

Graphic	Description	Code Name	Octal Code
E	Capital epsilon	Epsilon	105
Φ	Capital phi	Phi	106
Γ	Capital gamma	Gamma	107
H	Capital eta	Eta	110
I	Capital iota	Iota	111
ϑ	Alternate lower-case theta	theta1	112
K	Capital kappa	Kappa	113
Λ	Capital lambda	Lambda	114
M	Capital mu	Mu	115
N	Capital nu	Nu	116
O	Capital omicron	Omicron	117
Π	Capital pi	Pi	120
Θ	Capital theta	Theta	121
P	Capital rho	Rho	122
Σ	Capital sigma	Sigma	123
T	Capital tau	Tau	124
Y	Capital upsilon	Upsilon	125
ς	Alternate lower-case sigma	sigma1	126
Ω	Capital omega, ohm	Omega	127
Ξ	Capital xi	Xi	130
Ψ	Capital psi	Psi	131
Z	Capital zeta	Zeta	132
[	Left bracket	bracketleft	133
∴	Hence, therefore	therefore	134
]	Right bracket	bracketright	135
⊥	Perpendicular	perpendicular	136
_	Underscore, underline	underscore	137
	Extension for radical	radicalex	140

Graphic	Description	Code Name	Octal Code
α	Lower-case alpha	alpha	141
β	Lower-case beta	beta	142
χ	Lower-case chi	chi	143
δ	Lower-case delta	delta	144
ε	Lower-case epsilon	epsilon	145
φ	Lower-case phi	phi	146
γ	Lower-case gamma	gamma	147
η	Lower-case eta	eta	150
ι	Lower-case iota	iota	151
φ	Alternate lower-case phi	phi1	152
κ	Lower-case kappa	kappa	153
λ	Lower-case lambda	lambda	154
μ	Lower-case mu, micron	mu	155
ν	Lower-case nu	nu	156
ο	Lower-case omicron	omicron	157
π	Lower-case pi	pi	160
θ	Lower-case theta	theta	161
ρ	Lower-case rho	rho	162
σ	Lower-case sigma	sigma	163
τ	Lower-case tau	tau	164
υ	Lower-case upsilon	upsilon	165
ϖ	Alternate lower-case omega	omega1	166
ω	Lower-case omega	omega	167
ξ	Lower-case xi	xi	170
ψ	Lower-case psi	psi	171
ζ	Lower-case zeta	zeta	172
{	Left brace	braceleft	173
|	Vertical line or bar	bar	174

Graphic	Description	Code Name	Octal Code
}	Right brace	braceright	175
~	Approximately equal, similar, difference	similar	176
ϒ	Alternate upsilon	Upsilon1	241
′	Foot, minute, prime	minute	242
≤	Less than or equal	lessequal	243
⁄	Diagonal fraction bar (shallower than slash)	fraction	244
∞	Infinity	infinity	245
f	Florin, function (mathematical)	florin	246
♣	Club	club	247
♦	Diamond	diamond	250
♥	Heart	heart	251
♠	Spade	spade	252
↔	Left-and-right arrow	arrowboth	253
←	Leftward arrow	arrowleft	254
↑	Upward arrow	arrowup	255
→	Rightward arrow	arrowright	256
↓	Downward arrow	arrowdown	257
°	Degree (hangs from cap height)	degree	260
±	Plus or minus	plusminus	261
″	Inch, second, double prime	second	262
≥	Greater than or equal	greaterequal	263
×	Multiplication	multiply	264
∝	Varies directly as, proportional	proportional	265
∂	Partial differential	partialdiff	266
•	Bullet (larger than dot)	bullet	267
÷	Division	divide	270
≠	Not equal	notequal	271
≡	Equivalent, identical with, congruent	equivalence	272

Graphic	Description	Code Name	Octal Code
≈	Nearly or approximately equal	approxequal	273
...	Ellipsis, 3-dot leader	ellipsis	274
|	Extension for upward/downward arrow	arrowvertex	275
—	Extension for leftward/rightward arrow	arrowhorizex	276
↵	Carriage return and line feed	carriagereturn	277
ℵ	Aleph, transfinite cardinal number	aleph	300
ℑ	Fraktur I, imaginary number	Ifraktur	301
ℜ	Fraktur R, real number	Rfraktur	302
℘	Script P, Weierstrass elliptic function	weierstrass	303
⊗	Set (vector) multiplication	circlemultiply	304
⊕	Set (vector) summation	circleplus	305
∅	Null set, empty set	emptyset	306
∩	Intersection, product	intersection	307
∪	Union, sum, join	union	310
⊃	Contains as proper sub-class, implies	propersuperset	311
⊇	Contains as sub-class	reflexsuperset	312
⊄	Not contained as proper sub-class within	notsubset	313
⊂	Contained as proper sub-class within	propersubset	314
⊆	Contained as sub-class within	reflexsubset	315
∈	Member or element of a set	element	316
∉	Not a member or element of a set	notelement	317
∠	Angle	angle	320
∇	Gradient, divergence, curl	gradient	321
®	Registered (serif version)	registerserif	322
©	Copyright (serif version)	copyrightserif	323
™	Trademark (serif version)	trademarkserif	324
∏	Product (larger than capital pi)	product	325
√	Long division, square root, radical	radical	326

Graphic	Description	Code Name	Octal Code
·	Dot (centered like math operator)	dotmath	327
¬	Negation, logical NOT	logicalnot	330
∧	Logical AND	logicaland	331
∨	Logical OR	logicalor	332
⇔	Double left-and-right arrow	arrowdblboth	333
⇐	Double leftward arrow	arrowdblleft	334
⇑	Double upward arrow	arrowdblup	335
⇒	Double rightward arrow	arrowdblright	336
⇓	Double downward arrow	arrowdbldown	337
◊	Lozenge, subtotal, diamond	lozenge	340
〈	Left angle bracket	angleleft	341
®	Registered (sans-serif version)	registersans	342
©	Copyright (sans-serif version)	copyrightsans	343
™	Trademark (sans-serif version)	trademarksans	344
Σ	Summation (larger than capital sigma)	summation	345
⎛	Extensible left parenthesis: top	parenlefttp	346
⎜	Extension for left parenthesis	parenleftex	347
⎝	Extensible left parenthesis: bottom	parenleftbt	350
⎡	Extensible left bracket: top (ceiling)	bracketlefttp	351
⎢	Extension for left bracket	bracketleftex	352
⎣	Extensible left bracket: bottom (floor)	bracketleftbt	353
⎧	Extensible left brace: top	bracelefttp	354
⎨	Extensible left brace: middle	braceleftmid	355
⎩	Extensible left brace: bottom	braceleftbt	356
⎪	Extension for left/right brace	braceex	357
	Apple logo	apple	360
〉	Right angle bracket	angleright	361
∫	Integral	integral	362

Graphic	Description	Code Name	Octal Code
⌠	Extensible integral: top	integraltp	363
⎮	Extension for integral	integralex	364
⌡	Extensible integral: bottom	integralbt	365
⎞	Extensible right parenthesis: top	parenrighttp	366
⎟	Extension for right parenthesis	parenrightex	367
⎠	Extensible right parenthesis: bottom	parenrightbt	370
⎤	Extensible right bracket: top (ceiling)	bracketrighttp	371
⎥	Extension for right bracket	bracketrightex	372
⎦	Extensible right bracket: bottom (floor)	bracketrightbt	373
⎫	Extensible right brace: top	bracerighttp	374
⎬	Extensible right brace: middle	bracerightmid	375
⎭	Extensible right brace: bottom	bracerightbt	376

Apple Updates

Revision 2

Additions	Apple Symbol: width table.
Corrections	Apple QuickDraw Encoding and Character Set: Á's octal position is 313, and À's octal position is 347.

Appendix D

The Advanced Users Supplement

Apple LaserWriter™
Advanced User's Supplement

First Edition
January 1985

Adobe Systems Incorporated

Adobe Systems Incorporated
1870 Embarcadero Road, Suite 100
Palo Alto, California 94303

Apple LaserWriter Advanced User's Supplement
13 January 1985
Copyright © 1985 Adobe Systems, Inc.
All Rights Reserved

POSTSCRIPT is a trademark of Adobe Systems, Inc.

LaserWriter and AppleTalk are trademarks of Apple Computer, Inc.

Times and Helvetica® are trademarks of Allied Corporation.

Scribe is a registered trademark of UNILOGIC, Ltd.

The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems, Inc. or Apple Computer, Inc. Adobe Systems and Apple assume no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

Table of Contents

1. Introduction 1
2. Basic operation 2
   - 2.1. The server 2
   - 2.2. Switches and lights 3
   - 2.3. Modes of operation 4
3. Communication 5
   - 3.1. AppleTalk 5
   - 3.2. Serial I/O 6
   - 3.3. Communication dynamics 9
   - 3.4. Status queries and spontaneous messages 9
4. Details of server operation 11
   - 4.1. Power-on test page 11
   - 4.2. Page types 11
   - 4.3. Manual feed 12
   - 4.4. Timeouts 12
   - 4.5. Interactive mode operation 13
   - 4.6. Diablo 630 emulation 14
5. System parameters 17
   - 5.1. Changing persistent parameters 17
   - 5.2. Persistent parameters 18
   - 5.3. Idle-time font scan conversion 24
   - 5.4. Volatile parameters 25
6. Known problems 28

Introduction

This is a supplement to the POSTSCRIPT Language Manual giving detailed information about use of the Apple LaserWriter laser printer. There are several documents relevant to operation and programming of the LaserWriter; the three most important are:

- *Apple LaserWriter User's Manual* – describes how to set up a LaserWriter in a single standard configuration, namely as a network printer connected to AppleTalk and accessed from Macintosh and other AppleTalk hosts running various applications. This is strictly an operations guide; it gives no information about programming the LaserWriter or communicating with it in any way other than via AppleTalk.

- *PostScript Language Manual* – describes the programming language that is used to tell the LaserWriter what to print and how to print it. The LaserWriter is just one of many printers that can be programmed using POSTSCRIPT. This manual limits itself to describing features of the language that are available on all POSTSCRIPT printers.

- *Apple LaserWriter Advanced User's Supplement* (this document) – describes all the operating modes and special capabilities of the LaserWriter, and documents the additions to the POSTSCRIPT language that are used to control them.

Who needs to read this supplement? First, users wishing to access a LaserWriter from any computer other than a Macintosh will at least need to know something about how the LaserWriter operates and communicates (sections 2 and 3). Second, programmers wishing to develop applications making use of the LaserWriter's special capabilities will need to know about the POSTSCRIPT extensions for accessing those capabilities.

This document does not repeat basic operating information given in the *LaserWriter User's Manual*, such as how to load paper or change toner cartridges; nor does it describe the basics of the POSTSCRIPT language. For these you should refer to the other two documents.

Basic operation

This section gives an overview of the operation and use of the LaserWriter. It assumes that you understand the basic operating procedures given in the *LaserWriter User's Manual.*

In the following descriptions, we assume that all user-adjustable options are set to their standard default values. Later sections describe the machine's operation in more detail and document how to change the options.

2.1. The server

The principal function of the LaserWriter is to execute POSTSCRIPT programs sent to it from another computer. POSTSCRIPT is a programming language for describing the appearance of text, graphics, and images on printed pages. POSTSCRIPT programs may be composed by human users; but more typically they are generated by application programs running on other computers. Sending a POSTSCRIPT program to a LaserWriter usually causes it to produce one or more printed pages.

In normal operation, the LaserWriter cycles endlessly through the following sequence of steps. First, it sets up a clean initial execution environment (virtual memory) for a user's POSTSCRIPT program, which we will refer to as a "job". Then it obtains that job over some communication channel (either AppleTalk or serial I/O) and interprets it on the fly. When end-of-file is encountered or an error occurs, the LaserWriter cleans up after the user's job and restores the virtual memory to its initial state in preparation for the next job.

Thus, the LaserWriter's main role is as a *server* for execution of POSTSCRIPT programs sent to it by applications running on other computers. Ordinarily, each such program is executed solely for its side-effect, namely the generation of printed pages. However, under suitable conditions, a program may change some permanent parameters in the LaserWriter itself, or may perform some computation whose results are sent back over the communication channel rather than causing hardcopy to be produced.

Because the LaserWriter is a general-purpose computer, it can be programmed with the capability to *emulate* other printers. That is, it can be connected in place of some other printer and produce correct hardcopy results. The LaserWriter has a built-in emulator for the Diablo 630 printer, which is widely supported by personal computer application programs.

2.2. Switches and lights

The LaserWriter has two switches and four lights visible on the outside of the machine. Aside from the power switch, there is a four-position switch that, in combination with some parameters previously established, controls the mode of operation and the communication discipline.

The switch positions are labelled "1200", "9600", "Special", and "AppleTalk". These positions are assigned the following meanings, the details of which are described in later sections:

1200 POSTSCRIPT batch mode operation; serial (RS232/422) communication via either of the two connectors (see section 3.2), at 1200 baud, with parity ignored.

9600 POSTSCRIPT batch mode operation; serial communication using parameters established previously. The default parameters are 9600 baud, parity ignored. Since these parameters can be set under software control, the "9600" switch position may select a baud rate different from 9600.

Special Diablo 630 emulation mode; serial communication using parameters established previously. The default parameters are 9600 baud, parity ignored.

AppleTalk POSTSCRIPT batch mode operation; AppleTalk communication.

Changing the switch setting has immediate effect; if a job is in progress, it is aborted by execution of a POSTSCRIPT **interrupt**.

The lights are intended to provide a simple visual indication of what the LaserWriter is doing; more detailed information is available by querying the software, as will be described later. The lights on the front panel are Ready (green), Empty Paper Tray (yellow), and Paper Jam (red); additionally, there is a light on the rear of the machine labelled "Test". The lights are used in combination to indicate various states of operation.

Ready If the green light is on continuously, the machine is completely idle and awaiting the next user job to be executed. If green is flashing and the other lights are off continuously, the printer is warming up (this should take no longer than two minutes). If green is alternating with a single quick flash of the yellow light, the machine is busy executing a user job (or, immediately after power-on, computing the test page to be printed). If green is alternating with two quick flashes of the yellow light, the machine is in the midst of executing a user job but is suspended waiting for more I/O over the serial or AppleTalk connection presently being used; usually this indicates that it is waiting for the host machine to send it more text to be interpreted.

Empty Paper Tray
If the yellow light is on continuously, the paper tray is either

empty or absent, or the printer is in manual feed mode waiting for a sheet of paper to be inserted. Quick flashes of yellow are described above under green.

Jam
: If the red light is on, a sheet of paper has failed to feed from the paper tray or has jammed in the printer. The jam may be cleared by releasing the top of the printer and removing the paper; it is not necessary to turn power off while doing this.

Printer failure
: If the green, yellow, and red lights are all off, the printer mechanism or electronics have suffered a failure requiring manual intervention and possibly a service call.

Digital logic failure
: If the "Test" light on the rear of the machine remains on (either flashing or continuously) more than one second after power-on, a failure has occurred in the digital electronics and the machine is inoperable. If the light is flashing, an error message is being repeatedly transmitted over the 25-pin serial connector at 1200 baud. This indication may occur in combination with any of the other three lights.

2.3. Modes of operation

There are three basic modes of server operation: *batch*, *interactive*, and *emulation*. The four-position switch selects among these modes in combination with various communication options.[1]

In batch mode, a job consists of the execution of a single file containing a POSTSCRIPT program. When end-of-file is reached or the POSTSCRIPT program terminates, the job is finished. The only data transmitted from the LaserWriter to the host is that generated explicitly by the POSTSCRIPT **print** operator or by errors; in particular, the server provides no echoing, editing, or other user amenities. Batch mode is the normal way of operating the LaserWriter as a printing device for another computer.

In interactive mode, a job consists of an arbitrarily long dialogue in which the user issues a POSTSCRIPT command and the server generates a response and prompts for the next command. The state of POSTSCRIPT's virtual memory persists until the job is ended by explicit user request. During user type-in, the server echoes characters and allows some minimal editing functions. Interactive mode is the means by which a user may interact with the LaserWriter from a terminal connected directly to it. This is useful for experimenting with POSTSCRIPT and for using the LaserWriter as a general-purpose computer. More information about interactive mode operation is presented in section 4.5.

In emulation mode, the server emulates the operation of some other printer, usually a Diablo 630. In this mode, the LaserWriter does not inter-

[1]More precisely, the switch selects between batch and emulation modes with various communication options. Interactive mode is invoked by a procedure given in section 4.5.

pret the incoming data as a POSTSCRIPT program, but instead treats it as text and control codes understood by the printer being emulated. Complete information about Diablo emulation mode may be found in section 4.6.

Communication

The LaserWriter's connection with the outside world is via either AppleTalk or a point-to-point serial (RS232/422) link. The setting of the four-position switch determines the choice of communication discipline, as described previously.

In the following paragraphs, the connection is referred to as the "communication channel" (or just "channel"). The computer at the other end of the channel is referred to as the "host". The host uses the channel to send the LaserWriter POSTSCRIPT programs to execute or data on which to operate. (Alternatively, the device at the other end of the channel may be a terminal operated directly by a human user.)

It is important to understand that this channel is bidirectional. As well as reading programs and data from the channel, the LaserWriter may send output to the channel, either by explicit request of the program being executed (e.g., the POSTSCRIPT **print** operator) or by some spontaneous event such as an error. In this context, you should remember that **print** results in sending characters to the host computer or terminal, and has nothing to do with causing printed pages to emerge from the LaserWriter.

3.1. AppleTalk

Before connecting a LaserWriter to an AppleTalk network, it is important that you first turn the machine off and then set the server mode switch to "AppleTalk". Never operate a LaserWriter connected to AppleTalk with the switch set to any but the "AppleTalk" position. Failure to heed this precaution may leave the machine in an inoperable state or even bring down the entire network.

Connecting a LaserWriter requires that you use an AppleTalk connector box with a 9-pin plug, the same as is used with a Macintosh. A connector box with a 25-pin plug will not work, even though the LaserWriter does have a 25-pin socket.

While the LaserWriter is attached to AppleTalk, it listens for a connection request from another AppleTalk host. The server then executes a job using that connection as its source. Any error messages or other output produced by **print** are sent back to the host over the same connection. Data is carried transparently in both directions; that is, there are no character codes reserved for AppleTalk communication functions.

The AppleTalk protocols define an end-of-file indication. When the POSTSCRIPT interpreter reaches end-of-file, the LaserWriter sends a matching end-of-file indication back to the host, terminates the current job, and starts a new one. Thus, more than one job may be sent over the same AppleTalk connection. The host is permitted to close the connection any time after sending its end-of-file indication.

While the LaserWriter is busy with one connection, any further connection requests are refused. This causes the requesting hosts to queue up and wait for the server to become free. The next request chosen is the one that has waited the longest.

AppleTalk communication with the LaserWriter is accomplished by means of the Printer Access Protocol, which makes use of the Apple Transaction Protocol, Datagram Delivery Protocol, and Name Binding Protocol. These protocols are published separately by Apple.

A LaserWriter is identified by a three-part name constructed according to the Name Binding Protocol. The first or *object* part is the printer's individual name, which is initially "LaserWriter" but may be set to any other value by means of the **setprintername** operator described in section 5. The second or *type* part is always "LaserWriter", and the third or *zone* part is unspecified.

It is possible to connect more than one LaserWriter to the same AppleTalk network. If an additional machine has the same name as an existing one, it will automatically choose a new name, such as "LaserWriter1" or "LaserWriter2", in order to resolve the conflict.

3.2. Serial I/O

The LaserWriter has two serial channels, one wired to a 9-pin (RS422) connector and the other to a 25-pin (RS232) connector, either of which can be used for conventional asynchronous serial communication. (The 9-pin connector is also used for connecting to AppleTalk; but serial and AppleTalk communication are incompatible and will never occur at the same time.)

The signal pin assignments for the 9-pin (RS422) connector are:

1, 3	Signal Ground
4	Transmit Data +
5	Transmit Data –
8	Receive Data +
9	Receive Data –

This is compatible with the Macintosh. It is possible to connect a LaserWriter directly to a Macintosh using an Apple Modem cable and to communicate with it using MacTerminal.

The assignments for the 25-pin (RS232) connector are:

2 Transmit Data
3 Receive Data
4 Request To Send (optional; needed only if host requires it)
7 Signal Ground
20 Data Terminal Ready (optional; needed only if host requires it)

The other signals are not used. Technically, the LaserWriter has a "DTE" type of RS232 interface. This means it can be connected directly to a host computer or a modem, with no signal reversals required. Connecting to a terminal requires interposing a "null modem", which at a minimum involves reversing the Transmit Data and Receive Data signals.

When the LaserWriter is in any of the serial I/O modes, it uses one of the two channels to send and receive serial data encoded in ASCII.[2] Certain character codes serve special purposes, such as control-D to mark end-of-file. The server performs a job by reading and executing a POSTSCRIPT program from the serial channel; when the end-of-file character is received and the program terminates, the server sends an end-of-file character, ends the job, and starts a new one.

At the beginning of a job, both channels are enabled with independent baud rate and parity. The first channel to receive a character is the one chosen for execution of the next job. (The other channel is not disabled; if characters start to arrive on it, they are buffered and that channel is selected when the current job is finished.)

The details of the serial communication are determined by three parameters: channel, baud rate, and parity. These parameters may be changed by invoking the **statusdict** operators **setsccbatch** and **setsccinteractive**, described in section 5. Serial communication is asynchronous, start-stop, with 8 data bits per character (of which the high-order bit may or may not be used for parity), one start bit, and two stop bits.

The 9- and 25-pin connectors are designated in POSTSCRIPT by the integers 9 and 25. The baud rate is given as an integer, such as 1200 or 9600. The maximum baud rate supported by the software is 9600. The parity is specified by an integer in the range 0 to 3, as follows:

[2]ASCII is the American Standard Code for Information Interchange, a widely-used convention for encoding characters as binary numbers.

0 Ignore: the high-order bit of each 8-bit character received is ignored, and the high-order bit of each character transmitted is zero.

1 Odd: the high-order bit of each 8-bit character received is checked for odd parity (a POSTSCRIPT **ioerror** occurs if it is incorrect), and each character transmitted has odd parity.

2 Even: like odd, but for even parity.

3 None: all 8 bits of each character are treated as data, and no checking is performed.

As described in section 2.2, switch setting "1200" establishes communication with standard parameters (1200 baud, parity ignored); and switch settings "9600" and "Special" use parameters established previously. If no such parameters have been established, the defaults for the latter two switch positions are 9600 baud, parity ignored. If you are attempting to make contact with a LaserWriter for the first time and you don't know how the parameters might have been set by the previous user, you should start with the "1200" setting. A particular user or installation will likely want to establish different standard parameters. The facilities for adjusting these and other parameters are described in section 5.

The serial communication protocol is quite minimal. There are several character codes reserved for communication functions and not passed through to POSTSCRIPT:

Control-C	interrupt (causes POSTSCRIPT **interrupt** operator to be executed)
Control-D	end-of-file
Control-Q	(XON) start output
Control-S	(XOFF) stop output
Control-T	status query (see section 3.4)
Return	end-of-line
Line-feed	end-of-line (but ignored if it immediately follows Return)

As may be inferred, the server makes use of XON/XOFF flow control and expects the other party to do likewise. For batch mode operation, this form of flow control is required; in particular, use of the RS232 Data Terminal Ready (DTR) signal for flow control is not supported. Failure to conform to XON/XOFF flow control will result in occurrence of **ioerror** while transferring files longer than about 5000 characters.

There is no way to "quote" the reserved characters (to pass them through as data to POSTSCRIPT); nor is there any way to transmit characters in the "high ASCII" range (128 to 255) when the high-order bit is being ignored or used for parity. Thus, the serial link is not a fully transparent channel. However, this causes no difficulty in normal use since the POSTSCRIPT language consists entirely of printable characters. The lan-

guage itself provides means for encoding arbitrary characters in strings (the "*\nnn*" escape sequence). True binary data, such as images and encrypted programs, are transmitted in hexadecimal.

Serial data sent from the LaserWriter during execution of a job is followed by an end-of-file character sent when the job terminates. This enables the application program running on the host computer to synchronize with the server (if desired) and to correlate a given batch of output with the job that generated it. Note that there is no necessity for the application program to wait for one job to finish before beginning to send the input for the next job.

3.3. Communication dynamics

It is important to keep in mind that data transmitted by the LaserWriter, whether generated by the POSTSCRIPT program being executed or by some spontaneous event such as an error, is logically asynchronous with respect to the data received. In particular, this means that the host computer must be prepared to consume data generated by the LaserWriter while waiting to send more data to the LaserWriter. If this is not done, the LaserWriter and the host may each end up waiting for the other to consume some data, and a deadlock will result.

Data generated by POSTSCRIPT operators such as **print** is typically not sent immediately but is buffered until a **flush** is executed. (A **flush** is generated automatically by end-of-job and, in interactive mode, by each prompt for user type-in.) It is important that a POSTSCRIPT program execute a **flush** whenever it is required that data be sent immediately, such as when the host must wait for data from the LaserWriter before it can proceed. Failure to issue a needed **flush** can also result in a deadlock.

3.4. Status queries and spontaneous messages

At any time, it is possible to query the LaserWriter about what it is doing. Response to this status query is asynchronous with respect to normal job execution; that is, it is generated immediately regardless of what has gone on before or how much input data has been buffered. This facility is intended primarily to enable spooler programs to keep track of the activities of LaserWriters under their control.

The status query mechanism works differently depending on whether AppleTalk or serial communication is in use; but the syntax and semantics of the response are the same in either case.

In the case of AppleTalk, a request to open a connection to a busy LaserWriter yields a rejection packet whose data consists of a status message. There is also a separate status request packet that yields the same information. The path over which the status response packet travels is logically separate from the one through which the server is receiving its current job.

In the case of serial communication, receipt of a control-T character

from either channel elicits a one-line status message over the same channel. This channel need not be the one through which the server is receiving its current job. The message is bracketed by the text sequences "%%[" and "]%%" so as to enable host software to extract it from ordinary data generated by the job being executed.

The status message has a standardized syntax that is intended to be machine-readable. It consists of one or more "key: value" pairs, separated by semicolons. For example:

```
%%[job: Fred's Memo; status: busy; source: serial 9]%%
```

The possible values and meanings of the various fields are as follows:

job — the name of the job, as stored in the **jobname** entry in **statusdict**. This field is omitted if no job name has been set. (**statusdict** is described in section 5.)

status — idle (no job in progress), busy (executing user's POSTSCRIPT program), waiting (I/O wait in mid-job), printing (paper in motion), PrinterError: reason (e.g., paper out or jam), initializing (during startup), printing test page.

source — serial 9, serial 25, AppleTalk. This is the source of the job that the server is currently executing. This field is omitted if the server is idle.

All messages generated spontaneously by the server (as opposed to those generated by a user's POSTSCRIPT program) conform to the same syntax as status messages.[3] These are:

%%[Error: *error;* **OffendingCommand:** *operator*]%%
Error detected by the POSTSCRIPT interpreter (see *PostScript Language Manual*).

%%[PrinterError: *reason*]%%
Problem involving the printer mechanism (paper out, no paper tray, jam, cover open, etc.) A printer error can occur only when the machine is actually trying to print a page; in most cases, the server then waits for the condition to be corrected and proceeds automatically.

%%[Flushing: rest of job (to EOF) will be ignored]%%
Due to a previous error or other abort (e.g., **stop** or control-C), the remainder of the current job is being discarded. Further input is ignored until the next end-of-file indication is received.

%%[exitserver: permanent state may be changed]%%
See section 5.

[3]Note, however, that these messages are sent as ordinary data through the communication channel. Consequently, they are always bracketed with "%%[" and "]%%" whether the channel is serial or AppleTalk.

Details of server operation

Much of the behavior of the LaserWriter is subject to change by the user. There is a collection of operators and other parameters in a special POSTSCRIPT dictionary called **statusdict**. These are mentioned in the paragraphs below; but complete documentation is deferred to section 5.

4.1. Power-on test page

When the LaserWriter is turned on, it attempts to print a test page containing various simple text and graphics. The test page is not printed if the **dostartpage** parameter in **statusdict** has been set to false or if the printer takes more than three minutes to warm up. The normal startup time is about 50 seconds if the test page is printed and about 25 seconds if it is not.

Certain information about the current communication parameters is encoded in the two graph examples in the middle of the page. The number of tick marks along the bottom of the line graph corresponds to the current switch setting, as follows: no ticks = "1200", one tick = "9600", two ticks = "Special", three ticks = "AppleTalk". The communication parameters selected by the current switch setting (if not AppleTalk) are shown in the bar graph. The height of the two bars indicates the baud rates for the 9 and 25-pin connectors. The color of the bars indicates the parity settings: dark gray is ignore parity; medium gray (same as apple) is odd parity; light gray is even parity; white is no parity.

At the top of the page is the printer's AppleTalk name (returned by **printername**). At the bottom is the total number of pages that have been printed since the machine was built. There is a border that is intended to appear exactly one-half inch from the edges of a standard letter-size (8.5 by 11 inch) page; also, the left border of the apple illustration and the bottom border of the bar graph illustration intersect at the exact center of the paper. The printer alignment can be adjusted if necessary by invoking the **setmargins** operator in **statusdict**.

4.2. Page types

The imageable region of the page is subject to both hardware limits (the physical page size) and software constraints (the amount of memory available for the full page frame buffer). Space is traded off between the frame buffer and POSTSCRIPT's virtual memory (VM). The built-in LaserWriter software supports three standard "page types":

letter an imageable region of 8.0 by 10.5 inches, centered on an 8.5 by 11 inch page (that is, with 0.25 inch borders on all sides).

note an imageable region of 7.6 by 10.1 inches, centered on an 8.5 by 11 inch page. This page type is of interest to jobs that require unusually large amounts of VM for execution.

legal an imageable region of 7.0 by 12.5 inches, centered on an 8.5 by 14 inch page.

For all page types, the point (0, 0) in default user coordinate space is the lower left corner of the entire page, not of the imageable region; that is, the origin lies some distance outside the lower left corner of the imageable region.

At the beginning of each job, the software detects whether a letter or legal size paper tray is installed and sets the default page type automatically. If a legal size paper tray is present, page type **legal** is used; otherwise either **letter** or **note** is used according to the **pagetype** parameter previously established (the default is **letter**). A user's job can override the default page type by explicit execution of the operator **letter**, **note**, or **legal**.

4.3. Manual feed

It is possible to feed individual sheets of paper manually. If a job sets **manualfeed** to **true** in **statusdict**, the printer does not take paper from the paper tray during subsequent **showpage** operations. Instead, for each page printed, the yellow light comes on and the printer waits for a sheet of paper to be inserted into the slot in the right-hand side of the machine (opposite the paper exit slot). If no paper is inserted within **manualfeedtimeout** seconds, a **timeout** error occurs and the job is aborted.

4.4. Timeouts

There is a timeout facility for limiting the amount of time the server will remain in various states. There are three timeouts of interest: the *job* timeout, the *manual feed* timeout, and the *wait* timeout. At the beginning of a job, these timeouts are set to default values (initially 0, 60, and 30 seconds respectively), but a user program can set the timeouts for that job to other values if desired. The operators for controlling timeouts are located in **statusdict** and are described in section 5.

The manual feed timeout was described above. The job timeout, if nonzero, limits the total amount of time the job will execute; this is to protect the server from being tied up by user programs that run for an unexpectedly long time (or forever). The program itself can rejuvenate the timer any number of times during the job if that is desirable.

The wait timeout, if nonzero, limits the time the server will wait to receive additional input for a job that is in progress; this is to protect the

server from being tied up indefinitely by a host that crashes or is disconnected in the midst of sending a file to the server.

If a nonzero job or wait timeout has been set and it expires, the POSTSCRIPT operator **timeout** is executed from **errordict**. With the standard definition of **timeout**, this causes a batch job to terminate. The timeout facility is not ordinarily used when the server is in interactive mode.

4.5. Interactive mode operation

As was mentioned earlier, it is possible for a human user to interact directly with the LaserWriter from a terminal. To facilitate this, the LaserWriter has an interactive mode of operation that provides some simple user amenities.

A terminal with a standard RS232 interface may be connected directly to the LaserWriter, usually via its 25-pin connector. When making this connection, it is generally necessary to use a "null modem" or "modem eliminator" that reverses the Transmit Data and Receive Data signals. In place of a terminal, it is possible to use a personal computer running terminal emulation software. For example, a Macintosh running MacTerminal can be connected to the LaserWriter's 9-pin connector using an Apple Modem cable.

There are two ways to put the LaserWriter into interactive mode. The first is to select one of the batch mode switch positions ("1200" or "9600"), make sure the attached terminal is set to the correct baud rate and parity, and invoke the POSTSCRIPT procedure **executive**. That is, type "executive" followed by return or new-line. (Since the server in batch mode, the characters you type are not echoed back to you.) Once you do this, a POSTSCRIPT herald and prompt should appear:

```
PostScript(tm) version 23.0
Copyright (c) 1984 Adobe Systems Incorporated.
PS>
```

Each time the LaserWriter prints the "PS>" prompt, it is waiting for you to type in a POSTSCRIPT statement followed by return or new-line. It then executes that statement and prints another "PS>" prompt. While you are typing, the LaserWriter echoes the characters you type back to your terminal (so you can see them). Additionally, you can use the following special characters while typing:

Backspace	(control-H) erases one character.
Delete	(rubout) same as backspace.
Control-U	erases the current line.
Control-R	re-displays the current line.
Control-C	aborts the entire statement and starts over.

Interactive mode continues until you type control-D (the serial end-of-file character), execute a POSTSCRIPT **quit** command, or change the switch setting.

The other way to put the LaserWriter into interactive mode is to redefine the meaning of the "Special" switch position so that selecting it invokes interactive instead of emulation mode. For information about this, see the description of **eescratch** parameter 58 in section 5.2.

4.6. Diablo 630 emulation

The LaserWriter is capable of printing text intended for the Diablo 630 daisy wheel printer, which is a product of Diablo Systems, Inc. (a Xerox company). In Diablo emulation mode, the LaserWriter accepts documents with Diablo 630 formatting commands and produces hardcopy output. This capability is intended mainly for use in printing simple text files that are not in POSTSCRIPT form, and for processing output from software packages that do not directly support POSTSCRIPT.

If the system parameters have not been changed from their default state, all that is necessary to invoke the Diablo emulator is to set the server mode switch to the "Special" position and connect one of the LaserWriter's serial ports to the host's RS232 interface. Text to be printed may then be sent at 9600 baud with any parity.

Most of the information about serial communication in section 3.2 also applies in the case of Diablo emulation. However, the special meanings of control characters such as control-C, control-D, etc., are disabled; instead, all characters are treated according to the Diablo 630 protocol. The LaserWriter still sends XON and XOFF characters to control the flow of data from the host. Note that not all print drivers in microcomputer operating systems support the XON/XOFF protocol (e.g., DOS 2.0), and it may be necessary to obtain a separate software package to support this protocol. There are several available.

All the parameter settings that can be changed with Diablo commands are initialized as they are in the Diablo. For information about these commands, refer to the Diablo 630 documentation.

There are other parameters that in the Diablo require setting hardware switches or changing print wheels; in the LaserWriter these are system parameters that may be adjusted as described in section 5. The complete set of persistent parameters pertaining to Diablo emulation is given in the following table. To change them, refer to section 5.

Parameter	*Initial setting*
pitch	10
font	Courier
font for bold	Courier-Bold
auto-linefeed	off

The Diablo emulator supports all the standard LaserWriter typefaces.

server from being tied up indefinitely by a host that crashes or is disconnected in the midst of sending a file to the server.

If a nonzero job or wait timeout has been set and it expires, the POSTSCRIPT operator **timeout** is executed from **errordict**. With the standard definition of **timeout**, this causes a batch job to terminate. The timeout facility is not ordinarily used when the server is in interactive mode.

4.5. Interactive mode operation

As was mentioned earlier, it is possible for a human user to interact directly with the LaserWriter from a terminal. To facilitate this, the LaserWriter has an interactive mode of operation that provides some simple user amenities.

A terminal with a standard RS232 interface may be connected directly to the LaserWriter, usually via its 25-pin connector. When making this connection, it is generally necessary to use a "null modem" or "modem eliminator" that reverses the Transmit Data and Receive Data signals. In place of a terminal, it is possible to use a personal computer running terminal emulation software. For example, a Macintosh running MacTerminal can be connected to the LaserWriter's 9-pin connector using an Apple Modem cable.

There are two ways to put the LaserWriter into interactive mode. The first is to select one of the batch mode switch positions ("1200" or "9600"), make sure the attached terminal is set to the correct baud rate and parity, and invoke the POSTSCRIPT procedure **executive**. That is, type "executive" followed by return or new-line. (Since the server in batch mode, the characters you type are not echoed back to you.) Once you do this, a POSTSCRIPT herald and prompt should appear:

```
PostScript(tm) version 23.0
Copyright (c) 1984 Adobe Systems Incorporated.
PS>
```

Each time the LaserWriter prints the "PS>" prompt, it is waiting for you to type in a POSTSCRIPT statement followed by return or new-line. It then executes that statement and prints another "PS>" prompt. While you are typing, the LaserWriter echoes the characters you type back to your terminal (so you can see them). Additionally, you can use the following special characters while typing:

Backspace	(control-H) erases one character.
Delete	(rubout) same as backspace.
Control-U	erases the current line.
Control-R	re-displays the current line.
Control-C	aborts the entire statement and starts over.

Interactive mode continues until you type control-D (the serial end-of-file character), execute a POSTSCRIPT **quit** command, or change the switch setting.

The other way to put the LaserWriter into interactive mode is to redefine the meaning of the "Special" switch position so that selecting it invokes interactive instead of emulation mode. For information about this, see the description of **eescratch** parameter 58 in section 5.2.

4.6. Diablo 630 emulation

The LaserWriter is capable of printing text intended for the Diablo 630 daisy wheel printer, which is a product of Diablo Systems, Inc. (a Xerox company). In Diablo emulation mode, the LaserWriter accepts documents with Diablo 630 formatting commands and produces hardcopy output. This capability is intended mainly for use in printing simple text files that are not in POSTSCRIPT form, and for processing output from software packages that do not directly support POSTSCRIPT.

If the system parameters have not been changed from their default state, all that is necessary to invoke the Diablo emulator is to set the server mode switch to the "Special" position and connect one of the LaserWriter's serial ports to the host's RS232 interface. Text to be printed may then be sent at 9600 baud with any parity.

Most of the information about serial communication in section 3.2 also applies in the case of Diablo emulation. However, the special meanings of control characters such as control-C, control-D, etc., are disabled; instead, all characters are treated according to the Diablo 630 protocol. The LaserWriter still sends XON and XOFF characters to control the flow of data from the host. Note that not all print drivers in microcomputer operating systems support the XON/XOFF protocol (e.g., DOS 2.0), and it may be necessary to obtain a separate software package to support this protocol. There are several available.

All the parameter settings that can be changed with Diablo commands are initialized as they are in the Diablo. For information about these commands, refer to the Diablo 630 documentation.

There are other parameters that in the Diablo require setting hardware switches or changing print wheels; in the LaserWriter these are system parameters that may be adjusted as described in section 5. The complete set of persistent parameters pertaining to Diablo emulation is given in the following table. To change them, refer to section 5.

Parameter	*Initial setting*
pitch	10
font	Courier
font for bold	Courier-Bold
auto-linefeed	off

The Diablo emulator supports all the standard LaserWriter typefaces.

The default font is Courier, which is the fixed-pitch font most commonly used on daisy-wheel printers, and which is most likely to give correct results for typical microcomputer application programs. Note that the regular and bold fonts are specified separately. Thus, one could use Courier for regular printing and Courier-Oblique for bold; then the "bold" text would print as italic instead.

The LaserWriter emulates the Diablo as closely as possible; however, there are some differences of which you should be aware:

- The LaserWriter does not have any way to detect that the end of a document has been reached other than by noticing that data has stopped arriving. All Diablo printer settings (margins, tabs, spacing, etc.) remain in effect for about 30 seconds (or whatever the default wait timeout is set to) after the last page is processed. Then a Diablo "reset" operation is automatically performed to restore all settings to standard values; i.e., the margins are cleared, spacing is put back to standard, and tab settings and any special word processing modes are cleared.

- The LaserWriter actually prints a page when it either reaches the bottom of the page or receives a form-feed (control-L) character. If the last page of a document is not full and does not have a form-feed at the end, it will not be printed immediately. Instead, it will be printed when the LaserWriter resets approximately 30 seconds later, or as part of the next document (at the top of the first page). When documents are being printed in close succession, care should be taken to ensure that each one has a final form-feed so that they do not get run together.

- Some word processors produce "bold" by double striking a character. That will not appear as bold in the LaserWriter. Only the bold produced by issuing the proper Diablo command sequence (escape-O) will result in bold characters.

- Times-Roman and Helvetica are narrow fonts that may look squeezed if no adjustment of page width is made by the word processor. Very few word processing programs are capable of producing correctly formatted output using proportionally spaced fonts such as these.

- The emulator uses exact positioning on the paper. Output from a word processor that has attempted to compensate for slippage on vertical movement may appear slightly uneven.

The following Diablo 630 commands are not supported by the LaserWriter:

- print suppression
- HY-Plot
- extended character set
- the ability to download information for print wheels, including program mode

- the ability to override printwheel spacing (for proportional spacing), although the offset for proportional spacing can be changed
- page lengths other than 11 inches
- paper feeder control
- hammer energy control
- remote diagnostic
- backward printing control (note, however, that "reverse printing" is supported)

If you are an IBM-PC user, you may wish to issue the following commands to set up serial port 1 for communication with the LaserWriter. These commands set the baud rate to 9600 and map printer output to the serial port:

```
MODE COM1:9600,n,8,1
MODE LPT1:=COM1:
```

This by itself is not sufficient to support XON/XOFF flow control. Some applications may handle this protocol themselves; otherwise a different printer driver should be installed to avoid communication problems while printing large documents.

System parameters

The LaserWriter has a fairly extensive collection of parameters that control its behavior. Some of these parameters are stored in non-volatile memory (EEROM), so they persist even when the machine is turned off. Other parameters are volatile, and generally remain in effect only through the execution of a single job. This section documents both types of parameters.

To change system parameters requires that you send the LaserWriter a POSTSCRIPT program containing the necessary commands. To write such a program requires that you understand at least the fundamentals of the POSTSCRIPT language, for which you must refer to the *PostScript Language Manual*.

All system parameters are accessed via a special dictionary called **statusdict**, which is separate from **systemdict** and **userdict** and is not ordinarily on POSTSCRIPT's dictionary stack. The easiest way to gain access to **statusdict** is to execute **statusdict begin**, which pushes **statusdict** onto the dictionary stack. Some parameters are read and written by invoking operators defined in **statusdict**, while other parameters are accessed as ordinary data values (integers, booleans, strings, etc.)

5.1. Changing persistent parameters

Ordinarily, the server brackets each job with **save** and **restore** so that changes made to the virtual memory (VM) by the job do not persist into the next job. To make permanent changes (e.g., to install additional fonts), it is necessary to escape from the normal server and execute a job that is not bracketed by **save** and **restore**. This is also necessary in order to execute any of the **statusdict** operators that change the persistent (non-volatile) parameters.

The ability to make permanent changes is controlled by a password. Some LaserWriters are used in a shared environment in which it is undesirable for individual users to make changes to a server's persistent state. In such cases, only a system administrator should be permitted to make such changes. But in the case of a LaserWriter dedicated to a single user or a small group of cooperative users, the users should be permitted to make changes freely.

The system administrator password is a POSTSCRIPT integer. The default value is zero; but it can be changed to any other value by means of the operator **setpassword** in **statusdict**.

To exit from the normal server, it is necessary to execute the POSTSCRIPT command:

```
password serverdict begin exitserver
```

where *password* is the system administrator password. If the password is incorrect, an error results. If it is correct, the message

```
%%[ exitserver: permanent state may be changed ]%%
```

appears, and the remainder of the current job is permitted to make permanent changes.[4] This may be done in either batch or interactive mode.

The POSTSCRIPT program executed between a successful **exitserver** and the next end-of-file is permitted to invoke the **statusdict** operators that change persistent parameters. Additionally, all changes made by that program to the state of POSTSCRIPT's VM, such as creating new objects, storing values into dictionaries, etc., persist until power-off; the modified VM appears as the initial state of all subsequent jobs.

While executing a job outside the normal server **save/restore**, the system is not protected from harmful changes to the environment that could cause it to malfunction. (This is to permit the server software itself to be patched, should that become necessary.) Also, VM consumed by that job remains in use indefinitely; there is no way to reclaim it other than by turning the machine off and on.

5.2. Persistent parameters

The **statusdict** operators for accessing persistent parameters are described in this section. The volatile parameters are dealt with in section 5.4.

In order to invoke any of the operators that change persistent parameters it is first necessary to escape from the normal server environment, as described in the previous section (otherwise an **invalidaccess** error will result.)

♦ **pagecount**

– **pagecount** `int`

returns (i.e., pushes onto the operand stack) the number of pages that have been printed since the machine was built. (There is no way to reset this value.)

Errors: stackoverflow.

[4]Actually, a new job is started, but without the usual end-of-file indication on the communication channel. That is, **exitserver** performs an implicit **restore**, clears the operand and dictionary stacks, etc. Consequently, you must *not* issue an **end** to match the **serverdict begin**.

♦ setprintername

```
string  setprintername  –
```

takes a string of 31 or fewer characters from the operand stack and remembers it. This string is printed on the test page at power-on time, and also defines the name used to identify this LaserWriter on AppleTalk.

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ printername

```
string  printername  substring
```

takes a string, stores into it the printer name string previously saved, and returns a string object designating the substring actually used (default: LaserWriter).

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ setsccbatch

```
channel baud parity  setsccbatch  –
```

takes three integers designating channel (9 or 25), baud rate, and parity (see section 3.2). These determine how serial communication is to be performed on that channel during subsequent batch jobs when the switch is in the "9600" position. Note that these parameters may be set independently for each of the two channels. The new baud rate and parity do not take effect until the end of the current job. Setting the baud rate to zero disables the channel; but disabling both channels is not permitted.[5]

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ sccbatch

```
channel  sccbatch  baud parity
```

takes a channel number (9 or 25) and returns the batch baud rate and parity previously set for that channel (default: 9600 0).

Errors: rangecheck, stackoverflow, stackunderflow, typecheck.

[5]SCC stands for Serial Communications Controller, which is the device that operates the two I/O connectors.

♦ setsccinteractive

`channel baud parity` **`setsccinteractive`** `–`

same as **setsccbatch**, but sets serial communication parameters to be used when the switch is in the "Special" position (which selects either interactive or emulation mode operation with adjustable communication parameters).

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ sccinteractive

`channel` **`sccinteractive`** `baud parity`

takes a channel number (9 or 25) and returns the "Special" (interactive or emulation) baud rate and parity previously set for that channel (default: 300 0).

Errors: rangecheck, stackoverflow, stackunderflow, typecheck.

♦ setdostartpage

`boolean` **`setdostartpage`** `–`

takes a boolean that determines whether a test page is printed upon subsequent power-on.

Errors: invalidaccess, stackunderflow, typecheck.

♦ dostartpage

`–` **`dostartpage`** `boolean`

returns the start page parameter previously set (default: **true**).

Errors: stackoverflow.

♦ setmargins

```
top left  setmargins  –
```

takes two integer parameters and adds them to the top and left page margins respectively, treating them in units of pixels in device coordinate space. This is intended only for use at installation time to align the imageable area on the page. (The left margin parameter is quantized in units of 16 pixels, so it may not be possible to align the imageable area closer than 8 pixels to the true center of the page. Also, the printer hardware imposes margins that cause the image to be clipped if it is moved too close to the edge of the paper; unfortunately, the hardware-imposed margins are not symmetrical about the center of the paper.)

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ margins

```
–  margins  top left
```

returns the two margin parameters previously set (default: 0 0).

Errors: stackoverflow.

♦ setpagetype

```
integer  setpagetype  –
```

takes an integer from the stack specifying the page type to be used when the letter-size paper tray is installed (see section 4.2). The values defined at present are 0 for **letter** and 1 for **note**.

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ pagetype

```
–  pagetype  integer
```

returns the page type parameter previously set (default: 0).

Errors: stackoverflow.

♦ setdefaulttimeouts

```
job manualfeed wait  setdefaulttimeouts  –
```

takes three non-negative integers and sets the default values of **jobtimeout**, **manualfeedtimeout**, and **waittimeout** respectively (see section 4.4).

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ defaulttimeouts

– **defaulttimeouts** job manualfeed wait

returns the default job, manual feed, and wait timeouts previously set (default: 0 60 30).

Errors: stackoverflow.

♦ setpassword

old new **setpassword** success

sets the system administrator password, controlling the ability to escape from the protected server and make persistent environmental changes (see section 5.1). **setpassword** takes two integers from the stack: the old password and the new password; and it returns **true** if it was successful and **false** if unsuccessful.

Errors: stackunderflow, typecheck.

♦ checkpassword

integer **checkpassword** boolean

takes an integer from the stack and returns **true** if that is equal to the password last set by **setpassword**. If they are not equal, **checkpassword** delays one second before returning **false**. If **setpassword** has never been called, the correct password is zero.

Errors: stackunderflow, typecheck.

♦ setidlefonts

mark font sx sy rot nchars ... **setidlefonts** –

expects the operand stack to contain up to 150 integers in the range 0 to 255, delimited by a **mark** immediately below them. Removes the **mark** and the integers and remembers them. The integers specify fonts to be scan-converted during idle time, as is described in section 5.3.

Errors: invalidaccess, rangecheck, typecheck, unmatchedmark.

♦ idlefonts

– **idlefonts** mark font sx sy rot nchars ...

pushes a **mark** followed by the integers last passed to **setidlefonts** (default: just the **mark** followed by no integers at all).

Errors: stackoverflow.

♦ **seteescratch**

`index value` **`seteescratch`** `–`

takes an index in the range 0 to 63 and a value in the range 0 to 255, and writes the value into an array in the EEROM reserved for scratch use. This is intended for storing persistent information not envisioned in the original design of the LaserWriter. Several of these have already been used; they are described later in this section.

Errors: invalidaccess, rangecheck, stackunderflow, typecheck.

♦ **eescratch**

`index` **`eescratch`** `value`

takes an index and returns the EEROM scratch value previously set (default: 0).

Errors: rangecheck, stackunderflow, typecheck.

♦ **pagestackorder**

`–` **`pagestackorder`** `boolean`

returns false if the first page printed faces the back of the second page; true if the first page faces away from the second (for the current LaserWriter product this is always false, meaning that pages end up stacked in reverse order).

Errors: stackoverflow.

Several capabilities have been added to the LaserWriter since the standard set of persistent parameters (just described) was established, most notably the Diablo 630 emulator and the sharing of the "Special" switch position between interactive and emulation modes. New persistent parameters that control these capabilities have been assigned using the cells accessed by **eescratch** and **seteescratch**. In the next major revision of the LaserWriter software, these parameters will be assigned names of their own.

The following **eescratch** locations have been assigned. Thus, for example, to change the meaning of the "Special" switch position from Diablo emulation to POSTSCRIPT interactive mode, issue the command:

```
58 1 seteescratch
```

The default value of every **eescratch** cell is zero.

58 selects the function of the "Special" switch setting: 0 means Diablo 630 emulation mode; 1 means POSTSCRIPT interactive mode; and other values are reserved for future capabilities.

59 the value 1 enables the Diablo auto-linefeed feature.

60 selects the Diablo pitch (number of characters per inch). Reasonable values are 10, 12, and 15; 0 selects 10.

61 selects the "bold" font used for Diablo emulation. This is a font number taken from the table given in section 5.3, except that if the number is 0 (selecting Courier) then 1 (selecting Courier-Bold) is used instead. (To actually select Courier as the "bold" font, use some illegal font number such as 255.)

62 selects the "normal" font used for Diablo emulation. This is a font number taken from the table given in section 5.3. The default value 0 selects Courier.

63 has an internal use which is not documented.

The EEROM in which the persistent parameters are stored can be written only a limited number of times before wearing out. Each location in the EEROM is capable of approximately 10,000 writes. For this reason, the EEROM is used only for parameters that are expected to change infrequently. (The copy count is an exception; it is implemented in such a way that the wear is distributed over a large number of locations.)

At power-on time, the contents of the EEROM are checked for consistency, and an entry named **eerom** in **statusdict** is used to report the result. Normally, **eerom** contains **true**. If an inconsistency is detected, **eerom** is redefined to be a 512-character POSTSCRIPT string into which are read the entire contents of the EEROM; then the page count is set to zero and all parameters are reset to default values. If the EEROM fails altogether, **eerom** is set to **false** and the software shifts to a simulation of the EEROM parameters in RAM; all the operations for setting and reading parameters continue to work, but the values no longer survive across power-off.

5.3. Idle-time font scan conversion

While the server is waiting for a job to begin (i.e., before the first character has been received from the source specified by the switch), it utilizes the available time to scan-convert and cache a standard selection of characters in commonly-used fonts and point sizes. If a subsequent document uses those characters, the document will be processed faster than it otherwise would be.

The characters scan-converted during idle time are listed below. The character sets marked with an asterisk are pre-scanned and permanently resident in ROM.

- Courier 10* point, full ASCII set (intended for program listings and other "line printer" applications)
- Times-Roman and Helvetica 10, 12*, and 14 point, alphanumerics and common punctuation
- Times-Bold and Helvetica-Bold 10, 12, and 14 point, lower-case letters only

The standard selection of fonts to be scan converted during idle time may be overridden (except for the ones stored in ROM) by use of the **setidlefonts** operator in **statusdict**. Each font to be scan converted is specified by a group of five integers:

*font sx**10 *sy**10 *rot*/5 *nchars*

where *font* is a font number taken from the table below; *sx* and *sy* are the scale factors for x and y; *rot* is the rotation in degrees (applied after scaling); and *nchars* is the number of characters to be converted. The font numbers are:

0	Courier	7	Times-BoldItalic
1	Courier-Bold	8	Helvetica
2	Courier-Oblique	9	Helvetica-Bold
3	Courier-BoldOblique	10	Helvetica-Oblique
4	Times-Roman	11	Helvetica-BoldOblique
5	Times-Bold	12	Symbol
6	Times-Italic		

The characters converted are the first *nchars* characters of the following string, which contains 94 in all:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789.,;?:-()'"!+[]$%&*/_=@#`{}<>^~|\
```

For example, the following group of numbers would cause conversion of all lower- and upper-case alphabetic characters of Helvetica-Bold in a 12-point size, narrowed by the ratio 10/12, and rotated by 90 degrees:

```
9  100  120  18  52
```

The complete set of fonts to be scan converted is specified as a sequence of integers, interpreted in groups of five as just described. If the sequence is empty, the standard fonts are converted.

5.4. Volatile parameters

statusdict also contains several operators with immediate effects that do not persist from one job to the next. There are no restrictions on changing these parameters.

♦ setjobtimeout

`integer` **`setjobtimeout`** `–`

takes a non-negative integer and sets the timeout for the current job in seconds; zero means "never time out". (The job timeout is initially set to the default job timeout returned by **defaulttimeouts** for batch jobs and to zero for interactive jobs.)

Errors: rangecheck, stackunderflow, typecheck.

♦ jobtimeout

`–` **`jobtimeout`** `integer`

returns the amount of time remaining before the job timeout occurs; zero means the job will never time out.

Errors: stackoverflow.

The remaining **statusdict** entries are not operators but rather are ordinary data values such as booleans, integers, and strings. They may be read and written in the usual way by POSTSCRIPT dictionary operators such as **get** and **put**. In general, changes to these entries persist only until the end of the current job.

♦ manualfeedtimeout

`manualfeedtimeout` `integer`

the manual feed timeout currently in effect (default: the default manual feed timeout returned by **defaulttimeouts**).

♦ waittimeout

`waittimeout` `integer`

the wait timeout currently in effect (default: the default wait timeout returned by **defaulttimeouts** for batch jobs and zero for interactive jobs).

♦ manualfeed

`manualfeed` `boolean`

a boolean that controls whether paper is to be fed manually (**true**) or from the paper tray (**false**) (default: **false**).

♦ prefeed

prefeed boolean

a boolean that controls pre-feeding of paper from the paper tray. If **prefeed** is **true**, the next sheet of paper is fed immediately upon completion of each **showpage** or **copypage**; if **false**, the next sheet is not fed until the next **showpage** or **copypage**. To maximize throughput, the user's program should set **prefeed** to **true** at the beginning of a job and to **false** immediately before printing the last page of the job. This feature should be used only by programs that know that successive **showpage** operations will be done with a minimum of computation (a few seconds), such as when printing simple text documents using only pre-cached fonts. Misuse of this feature can cause the printer mechanism and the laser to be left running for long periods of time, resulting in premature wear (default: **false**).

♦ jobname

jobname string

a string specifying the name of the current job. If set by the user's program, this name will appear as part of status responses generated during the remainder of that job (default: **null**).

♦ product

product string

a string object which is the name of the laser printer product (LaserWriter). The rare program that needs to know what type of printer it is running on should check this string. Also, this string defines the *zone* portion of the printer's AppleTalk name.

Errors: stackoverflow.

♦ revision

revision integer

an integer which is the current revision level of the machine-dependent software. (Note that the **version** operator in **systemdict** returns the version number of the machine-independent portion of POSTSCRIPT.)

Errors: stackoverflow.

There are several additional **statusdict** entries that are not documented. They have to do with the operation of the server and are not intended for execution by user programs.

There is a new convention, established since the most recent revision of

the *PostScript Language Manual*, for specifying the number of copies to be printed by each execution of the **showpage** operator. This convention is obeyed by most POSTSCRIPT printers, including the LaserWriter.

Each time **showpage** is executed, the name **#copies** is looked up in the context of the current dictionary stack. The resulting value should be a non-negative integer specifying the number of copies of each page to be printed. At the beginning of every job, **#copies** is defined to be 1 in **userdict**.

Known problems

The following bugs are known to exist in the initial release of the LaserWriter software (POSTSCRIPT version 23.0). These bugs will be present in the product until the next complete ROM revision. Most of the problems are relatively obscure. Fortunately, it is possible either to avoid or to work around the problems that affect the LaserWriter's function. You should not worry about the bugs that affect only its performance.

- During serial input, if the input buffer becomes full and the LaserWriter sends XOFF to stop transmission from the host, it occasionally fails to send XON to restart transmission. Assuming the wait timeout is enabled (as it ordinarily is), this causes the job to time out and abort, which resets the erroneous buffer full indication and sends an XON. (The existence of this bug makes it inadvisable to operate the LaserWriter with the wait timeout disabled, since communication with the host could become hung up indefinitely. This bug occurs only under unusual circumstances that are difficult to describe; it is timing- and data-dependent.)

- If characters are positioned by adjusting the translation component of **currentmatrix** rather than by adjusting **currentpoint** as is ordinarily done, character positioning may be as much as one pixel off, leading to ragged base lines.

- The font cache may work at less than full efficiency due to the presence of certain characters that are inappropriately locked in the cache. The performance effects of this bug are slight.

- Redefining any of the built-in fonts or fonts derived from them causes the font cache to malfunction under certain complex and hard-to-describe circumstances. The effect is that some characters are displayed in the wrong font or point size or both. To avoid this bug, do not define a new font with the same name as any of the built-in fonts.

- It is possible to copy an existing font dictionary and then add a **Metrics** entry to the copy in order to create a new font with different spacings for the characters. Unfortunately, this does not always work correctly, because the cache fails to distinguish between characters belonging to the new and old fonts. To work around this bug, it is necessary also to change some *other* entry in the new dictionary. The simplest way to do this is to change the **FontBBox** entry to be a *new* array which is a copy of the **FontBBox** array from the original font.

- More than two levels of recursion in character building may cause the LaserWriter to crash. That is, it's OK to have a user-defined font whose **BuildChar** procedure in turn does a **show** using a built-in font. But it is unsafe for that user-defined font to be invoked from the **BuildChar** procedure of yet another user-defined font.

- The path created by **strokepath**, or the path created by **charpath** for a stroked font (e.g., Courier), may not be completely suitable for subsequent clipping or filling if round end-caps or joins are used. Portions of the round end-caps or joins may incorrectly be found to be "outside" the path thus created.

- Images built in strips may contain seams between the strips under certain circumstances.

- If the procedure passed to **image** or **imagemask** fails to return a string as it is supposed to, the LaserWriter software may cease to function correctly until the machine is next turned off and on.

- If a **gsave** is done and a new transfer function is established by **settransfer**, the subsequent **grestore** may not properly restore the old transfer function. This bug occurs randomly with low probability. (This bug does not affect the restoration of the default transfer function, which is done at the beginning of each job and during any explicit invocation of **letter**, **legal**, or **note**.)

- If manual feed is invoked too quickly after printing a previous page using normal feed (from the paper tray), the printer mechanism ignores the request to use manual feed. To avoid this problem, when switching from normal to manual feed be sure at least 5 seconds elapse before issuing the next **showpage**. If necessary, the delay may be inserted artificially by executing the statement:

```
usertime 5000 add
{dup usertime lt {pop exit} if} loop
```

- Exhausting POSTSCRIPT's VM sometimes causes the LaserWriter to crash and restart rather than simply abort the current job as it should.

- If a job uses the **note** page type and then fills the VM close to overflowing, a subsequent attempt by the same job to set any page type causes the LaserWriter to crash and restart.

- The Diablo 630 emulator may fail to produce any output if it is sent two or more successive single-page documents that do not end with form feed characters.

- The automatic recovery from total failure of the EEROM device, described at the end of section 5.2, does not work properly. If the EEROM fails, the LaserWriter will not start up after power-on.

Appendix E

The Apple Talk Printer Access Protocol

This section contains:

1. The AppleTalk Printer Access Protocol Specification.

2. The source of an example application that calls the Printer Access Protocol directly. The application chosen is the application on the Programming and Debugging Aids disk entitled "Downloading Program". For details on the operation of this program, see appendix F in the section entitled " Instructions for Spooling, Editing and Downloading a Postscript file from a Macintosh application." (Note that the program was compiled under the name PSDump and was subsequently renamed Downloading Program.)

AppleTalk Printer Access Protocol

by

Gursharan S. Sidhu and Alan B. Oppenheimer,
Apple Computer Inc.,
September 4, 1984.
(revised February 15, 1985)

Scope

This document describes in detail the Printer Access Protocol used to access Apple's laser print server (LaserWriter) over AppleTalk.

I. Introduction

The AppleTalk Printer Access Protocol (PAP) allows workstations on AppleTalk (e.g. a Macintosh) to communicate with the laser print server (LaserWriter). The protocol has been designed to be frugal in its use of workstation memory and cable bandwidth. From the architectural point of view, PAP is a client of the name binding (NBP) and transaction protocols (ATP).

PAP is a connection-oriented protocol. A PAP client in a workstation issues a PAPOpen call which initiates a connection-establishment dialogue with the server (the client specifies the server by its complete name). PAP calls NBP to obtain the address of the server's listener socket from the server's name.

Once a connection has been opened to the server, the PAP client at either end of the connection can receive data from the other end by issuing PAPRead calls, and write data to the other end through PAPWrite calls. PAP uses ATP transactions (in exactly-once mode) to transfer the data.

When the data transfer has been completed a PAPClose call is issued by the PAP client in the workstation to close the connection.

At any time, the PAP client in the workstation can issue a PAPStatus call to find out the status of the server.

There are several calls for use only in the server. The first of these is the SLInit call. This is issued by the server when it is first started up, after it has completed its internal initialization and is ready to accept print jobs from workstations. The SLInit call opens a service listener socket in the server (PAP does this by calling ATP to open a responding socket), and causes the server's name to be installed in the server's names table (PAP does this by issuing a call to NBP).

A second call is used by the server's PAP client to indicate to the server PAP's connection arbitration code that the server is ready to accept a connection. This would be done just after the SLInit call or after each printing job has been completed and the server is idle (and ready to accept another connection). This GetNextJob call primes the connection arbitration code to accept a connection establishment request over the AppleTalk.

Two calls PAPRegName and PAPRemName can be used by the PAP client in the server to respectively, register and remove (deregister) the server's name. This might be necessary for giving the server more than one name, or to change the server's name (e.g. at server setup time).

One of the situations that PAP must deal with is the well-known case of half open connections. Such a connection is said to exist when one of the connection ends "dies" (or terminates the connection without informing the other end). Half open connections must be detected and torn down/closed. For this purpose, PAP maintains a connection timeout (at each end). Furthermore, each end of an open connection must send "tickling" packets to the other end on a periodic basis. The purpose of these packets is to inform the other end that the sender's end is open and "alive". The receipt of any packet on a connection resets the connection timer at the receiving end. If the connection timer expires (i.e., no packets have been received since the timer was last reset) then the decision is made that the other end is dead and the connection end is torn down.

The rest of this document contains a detailed discussion of the PAP protocol, its interaction with (use of) NBP and ATP, and the details of the PAP client interface at the workstation and server ends.

II. The Protocol

The basic model of the server is that it processes jobs from workstations, one job at a time. While a workstation is being served (i.e., its job is being processed by the server) requests for service from other stations are not accepted (they are informed that the server is busy). At this time, a connection is said to be open between the workstation being served and the server. When the server is done with a particular job, the connection is closed and the server becomes idle. It can now accept requests for service from other work stations.

Clearly, at any time, the server can have at most one open connection. Also, in a sense there is a one-to-one correspondence between connections and jobs processed by the server.

When a server is first started, it goes through its internal initialization and then the server control software issues an SLInit call to its PAP code. This causes the PAP to call ATP to open a service listener (SL) socket (this is an ATP responding socket). Then, PAP calls NBP to register the server's name(s) and bind them to the SL socket. An ATPGetRequest call is then issued by PAP on this socket (so that the server can respond to PAPOpen or PAPStatus request packets). But the server is still not ready to accept a job/connection.

After the SLInit call completes, the PAP client in the server issues a GetNextJob call to indicate that the server is ready to accept jobs. The server is now in the IDLE state and is ready to accept jobs/connections.

Connection Establishment (Opening) Phase:

A connection is a logical relationship between two PAP code entities (one in the workstation and the other in the server). Data can be exchanged by two PAP clients only after a connection has been established/opened. Since PAP uses ATP to transfer data, the two communicating PAPs must in the connection establishment phase discover the address of the ATP responding socket of the other connection end. Also, the amount of data that can be transferred in an ATP transaction is of a maximum size equal to the available receive buffers at the end issuing the read requests. This maximum size (called the "flow quantum") is sent by each end to the other in the connection establishment phase.

Connection establishment is initiated by PAP clients in the workstations by issuing a PAPOpen call. Such a client provides as a call parameter the complete name of the server. The PAP code obtains the complete internet address of the server's SL socket by issuing an NBP Lookup call. It opens an ATP responding socket Rw, generates an 8-bit connection identifier ConnID and then sends a transaction request (TReq), with PAP-type OpenConn, to the server's SL socket. This packet contains the ConnID, the address of socket Rw, the flow quantum for the workstation, and a wait time used by the server for arbitration (discussed later). All packets related to this connection (sent by either end) must contain this connection identifier.

When an ATP TReq of PAP-type OpenConn is received at the server's SL socket, PAP executes a connection acceptance algorithm. If the server is BUSY (i.e. it is processing a job), then its PAP responds to the "OpenConn" with an ATP response of PAP-type OpenConnReply indicating "Server busy".

If however, the server is idle, then upon receiving an OpenConn (the first one since the server went into the idle state), its PAP goes into an arbitration (ARB) state for a fixed amount of time (approximately two seconds). In the ARB state, the PAP receives all incoming "OpenConns" and tries to find the one corresponding to the work station that has been waiting for a connection for the longest amount of time. The idea is to implement a fairness scheme that accepts the request generated by this station over those from more recent entrants to the contest.

The time, in seconds, for which a station has been waiting (call it the WaitTime) is sent with the OpenConn. When the first OpenConn is received since the server went IDLE, the WaitTime value from that request is loaded into a variable called OldestReq. During the ARB interval, whenever an OpenConn request is received, its WaitTime is compared with OldestReq. If WaitTime <= OldestReq then the just received call has waited less time than a previously received one. In this case, PAP responds to the just received request with an OpenConnReply indicating "Server busy". If WaitTime > OldestReq then the just received call has waited longer than the previously received (and pending) one. In this case, PAP saves the just-received WaitTime in OldestReq. Now, the just received OpenConn request(and now the oldest waiting one, so far) is kept pending until the end of the ARB interval, or until an older request arrives. At the end of the ARB interval, PAP opens an ATP responding socket Rs and sends an ATP response of PAP-type OpenConnReply indicating "Connection accepted" to the selected (and pending) request. This carries the ConnID received in the "OpenConn", the address of socket Rs and the flow quantum of the server end [The flow quantum value for the server end is set by the SLInit call issued when the server is initialized. It is currently 8 for the LaserWriter]. The connection is now open, and that workstation's job is being processed. The server is now in the BUSY state.

At the workstation end, if a response of PAP-type OpenConnReply is received indicating that the server is busy, then that end's PAP waits some time (approx 2 seconds) and issues another connection opening transaction. Each time it repeats this process it updates a "wait time" -- the time in seconds that it has been trying to open the connection. The current value of this wait time is sent with each OpenConn. Each of these OpenConn ATP transaction requests is issued with a retry count of 5 and retry interval (approx 2 seconds). If the server is dead, or in its 6-second imaging-loop, it will be unable to respond; then the transaction will terminate without receiving a reply at all. The workstation's PAP updates the wait time and tries again.

Data Transfer Phase:

Once a connection has been opened PAP's data transfer phase is started. In this phase, PAP has two functions: to actually transfer data over the connection, and to detect and tear down half-open connections. The detection of half-open connections is done by maintaining a connection timer (of the order of 2 minutes) at each end of the

connection. This timer is started as soon as the connection is opened. Whenever a packet of any sort is received from the other end of the connection, the timer is reset. If the timer expires (clearly, without receiving a packet from the other end) the connection is torn down. The presumption is made that the other end has "died" or has closed its connection.

For this to work properly, it is important that even though no data is being exchanged on the connection, PAP exchanges control packets to signal that the connection ends are alive. This process is referred to as "tickling" and the control packets are called "tickling packets". As soon as a connection is established, each end starts an ATP transaction with PAP type Tickle. This transaction, known as the "Tickle" transaction has a retry count of infinity (ATP retry value of 255 is used to signify infinite retries) and a retry time interval of half the connection timeout. Tickle packets are sent to the other end's ATP responding socket (i.e. Rs or Rw). The receiver of such a packet must reset its connection timer but must not send a transaction response. These "Tickle" transactions are cancelled by each end when the connection is closed.

The basic data transfer model used by PAP is "read-driven". By this we mean that either end sends transaction requests to read data from the other end (another way of saying this is that each end issues an ATP transaction request to the other end asking it to send data). When the PAP client at either end of the connection wishes to read data from the other end, it issues a PAPRead call. This call provides PAP with a read buffer (of size equal to this end's flow quantum) into which the data is to be read. As a consequence of this call, PAP calls ATP to send an ATP transaction request with PAP-type SendData, and the ATP bitmap reflecting the size of the call's read buffer. This transaction is issued with a retry count of "infinite" (i.e. 255) and a retry time interval (15 seconds). To prevent duplicate delivery of data to PAP's clients, all these ATP transactions for the transfer of data use ATP's exactly-once mode.

The receipt of an ATP TReq packet with PAP-type SendData implies that there is a pending PAPRead at the other end. This "send credit" can be remembered by the PAP code, and used to service any pending or future PAPWrite calls issued by its client.

When a PAP client (at either end) issues a PAPWrite call, PAP examines its internal data structures to see if it has received a "send credit". If it has, then it takes the data from the PAPWrite and sends it in ATP Response packets with PAP-type Data (the EOM-bit is set in the last of these ATP response packets). If no send credit has been received, then PAP queues the PAPWrite call and awaits a "send credit" from the other end (i.e. the receipt of an ATP request of PAP-type SendData from the other end). The amount of data to be sent in a PAPWrite call cannot exceed the flow quantum of the other end (PAPWrite calls that violate this restriction return immediatedly with an error message).

When a PAP client issues the last PAPWrite call for a particular job, it must ask PAP to send an End-of-File (EOF) indication with that call's data. The EOF indication is delivered to the PAP client at the other end (as part of the received information for a PAPRead call); this indication notifies the client that the other end is through sending data on this connection. Note that for this purpose the client can issue a PAPWrite call with no data to be sent; in this case, just an EOF indication is conveyed to the client at the other end.

Connection Termination (Closing) Phase:

When the PAP client at either end issues a PAPClose call, PAP closes the connection. Typically, after the workstation's PAP client has completed sending all data to the server and received an EOF in return, it will issue the PAPClose call. An ATP transaction request is sent to the other end with PAP-type "CloseConn". An end receiving a "CloseConn" should immediately send back, as a courtesy, an ATP transaction response of PAP-type "CloseConn Reply". To close a connection's end, it is important to cancel any pending ATP transactions issued by that end, including the "tickle" transaction. Note that the end receiving the "CloseConn" might not necessarily cancel these pending transactions immediately, as it will probably be at interrupt level.

At the server end (see Figure 1), the receipt of the CloseConn will cause the connection to be torn down, but the server will continue in the BUSY state until it actually finishes the processing of the data pertaining to the job. When this is completed, the PAP client in the server issues a GetNextJob call. This call puts the server back in the IDLE state, and it now is ready to entertain requests for connection establishment.

PAP Packet formats:

PAP uses both NBP and ATP. The use of NBP is strictly for the purpose of registering the server's SL socket and, given a server's name, for determining the address of its SL socket.

However, packets sent by ATP in response to PAP calls include a PAP header. This is built using the user bytes of the ATP header, and in some cases by sending four or more bytes of PAP header in the data part of the ATP packet.

In all cases, the first of the ATP user bytes is the ConnID, and the second the PAP-type of the packet. [Permissible values of PAP-type are: OpenConn, OpenConnReply, SendData, Data, CloseConn, CloseConnReply, Tickle, SendStatus, and StatusReply]. For packets of PAP-type equal to Data, the third ATP user byte is the EOF indication.

Figure 2 illustrates the PAP header for the different types of PAP packets.

PAPType Values:

The permissible PAP Type field values are:

OpenConn = 1
OpenConnReply = 2
SendData = 3
Data = 4
Tickle = 5
CloseConn = 6
CloseConnReply = 7
SendStatus = 8
StatusReply = 9

III. The Client-PAP Interface

In this section we take each PAP call and list the parameters the client must pass and the significant interface-level aspects of each call. These calls, while intended to document a generic client-to-PAP interface, in fact also correspond exactly to those provided by the Macintosh implementation of PAP.

(i) PAPOpen:

This call is issued by a PAP client in a workstation to start/open a connection to a specified server. In a Pascal-like form the call is:

```
FUNCTION PAPOpen (VAR RefNum: INTEGER;
                  PrinterName: Ptr;
                  FlowQuantum: INTEGER;
                  StatusBuff: Ptr;
                  VAR CompState: INTEGER ): INTEGER;
```

where:

`RefNum`	is the connection reference number returned after the connection has been opened;
`PrinterName`	is a pointer to the entity name (see definition of entity name below) of the print server to which the connection is to be opened;
`FlowQuantum`	is an integer specifying the flow quantum equal to the number of 512 byte buffers (e.g. if `FlowQuantum` = N, then the flow quantum = 512*N bytes; the LaserWriter uses N = 8);
`StatusBuff`	is a pointer to the buffer structure in which the printer status is returned to the caller during the opening process (the details of this data structure are given below);
`CompState`	is an integer that can be monitored by the caller for call completion and error reporting.

PAPOpen is executed asynchronously. As soon as control returns to the caller, if the function's returned value equals `NoErr`, then the caller can monitor for call completion by examining the variable `CompState`. While the call is executing, this variable will have a value greater than zero. When the call has completed, it will take on a value of zero (no error) or a negative value which is an error code.

An entity name, as defined in the "Calling the AppleTalk Manager from Assembly Language" section of the AppleTalk Manager chapter in Inside MacIntosh, consists of: the object name length byte, the object name, the type length byte, the type, the zone length byte and the zone.

The structure of a `StatusRec` to which `StatusBuff` is a pointer is given by the Pascal type declaration:

```
TYPE StatusRec  = PACKED RECORD
                    SystemStuff: LongInt;{PAP internal use}
                    StatusStr: STR255  {status string}
                  END;
```

It is important that the caller clear `StatusStr` before making the call. While the call is being processed the caller can monitor StatusStr in which PAP will continuously insert the status information being returned from the server in PAP OpenConnReply packets. The PAP client in the workstation might wish to display this string in an appropriate fashion in order to provide appropriate feedback to the user.

(ii) PAPClose:

This call is issued by the PAP client in a workstation to close the connection specified by its reference number. In a Pascal-like form the call is:

```
FUNCTION PAPClose (RefNum: INTEGER ): INTEGER;
```

where:

`RefNum`	is the connection reference number.

This call is executed synchronously. This call cancels any pending PAPRead and PAPWrite calls for the indicated connection.

(iii) PAPRead:

This call is issued by the PAP client at either end to read data from the other end over the connection specified by the reference number. In a Pascal-like form the call is:

```
FUNCTION PAPRead (RefNum: INTEGER;
                  ReadBuff: Ptr;
                  VAR DataSize: INTEGER;
                  VAR EOF: INTEGER;
                  VAR CompState: INTEGER ): INTEGER;
```

where:

`RefNum`	is the connection reference number;
`ReadBuff`	is a pointer to the buffer into which the data is to be read;
`DataSize`	is an integer in which the number of bytes of data read into the buffer is returned when the call completes;
`EOF`	is an integer in which the end-of-file indication received from the other end is returned to the caller (a non-zero value indicates an end of file; otherwise a value of 0 is returned);
`CompState`	is an integer that can be monitored by the caller for call completion and error reporting.

It is important to note that PAP assumes that the buffer to which `ReadBuff` points is of

size equal to (no smaller than) the flow quantum specified in the PAPOpen call.

This call is executed asynchronously. As soon as control returns to the caller, if the function's returned value equals `NoErr`, then the caller can monitor for call completion by examining the variable `CompState` . While the call is executing, this variable will have a value greater than zero. When the call has completed, it will take on a value of zero (no error) or a negative value which is an error code.

When the call has completed without error, then the variable `DataSize` is equal to the number of bytes of data received into the buffer. When the call completes with error, the value of this variable has no significance and is unpredictable.

(iv) PAPWrite:

This call is issued by the PAP client at either end to write data to the other end over the connection specified by the reference number. In a Pascal-like form the call is:

```
FUNCTION PAPWrite (RefNum: INTEGER;
                   DataBuff: Ptr;
                   DataSize: INTEGER;
                   EOF: INTEGER;
                   VAR CompState: INTEGER ): INTEGER;
```

where:

`RefNum`	is the connection reference number;
`DataBuff`	is a pointer to the data to be written;
`DataSize`	is equal to the number of bytes of data to be written;
`EOF`	is the end-of-file indication to be sent to the other end (a non-zero value indicates end of file; otherwise a value of zero should be sent);
`CompState`	is an integer that can be monitored by the caller for call completion and error reporting.

It is important to note that if the data size is bigger than the flow quantum of the other end (value received during the connection establishment phase) the call will return with an error.

This call is executed asynchronously. As soon as control returns to the caller, if the function's returned value equals `NoErr`, then the caller can monitor for call completion by examining `CompState` . While the call is executing, this variable will be greater than zero. When the call has completed, it will have a value of zero (no error) or a negative value which is an error code.

(v) PAPStatus:

This call is used by a PAP client in the workstation to determine the currrent status of the print server. It can be used at any time (i.e. whether a connection has been opened by the PAP client to the server or not). It is executed synchronously, and upon completion returns a string with the status message sent by the print server. In a Pascal-like form the call is:

```
FUNCTION PAPStatus (PrinterName: Ptr;
                    StatusBuff: Ptr): INTEGER;
```

where:

PrinterName	is a pointer to the entity name (see PAPOpen) of the print server whose status is to be determined;
StatusBuff	points to a structure of type StatusRec (see PAPOpen).

In addition to these calls, the server uses the following four calls:

(vi) PAPRegName:

This call is used by the server only. It registers a name (as the entity name for the print server) on the server's listening socket.

```
FUNCTION PAPRegName (PrinterName: Ptr): INTEGER;
```

where:

PrinterName	points to a structure of type Entity Name.

(vii) PAPRemName:

This call is used by the server only. It deregisters a name from the server's listening socket.

```
FUNCTION PAPRemName (PrinterName: Ptr): INTEGER;
```

where:

PrinterName	points to a structure of type Entity Name.

(viii) SLInit:

This call is issued by the PAP client in the server to perform several functions: to open a service listening socket and to register the server's name on this service listening socket. In a Pascal-like form the call is:

```
FUNCTION SLInit (PrinterName: Ptr;
                 FlowQuantum: INTEGER): INTEGER;
```

where:

PrinterName	is the name of the print server;
FlowQuantum	is an integer specifying the flow quantum equal to the number of 512 byte buffers (e.g. if FlowQuantum = N, then the flow quantum = 512*N bytes; the LaserWriter uses N = 8).

SLInit is executed synchronously.

(ix) GetNextJob:

This call is issued by the PAP client in the server either just after the SLInit call or when it has finished processing a job. It closes any open connection, and then puts the server in the IDLE state and ready to open another connection. In a Pascal-like form the call is:

```
FUNCTION GetNextJob (VAR RefNum,
                     CompState: INTEGER ): INTEGER;
```

where:

RefNum	is a variable in which a reference number is returned when a connection has been opened.

(x) PAPUnload:

This call can be used at either end, the work station or the server, to cause the PAP data structures to be unloaded and currently open connection(s) to be closed. It could be used on the server, if for instance, the communication mode switch is moved from AppleTalk to RS-232 etc. In the work station, the client would use this before exiting to the Finder. In a Pascal-like form the call is:

```
FUNCTION PAPUnload: INTEGER.
```

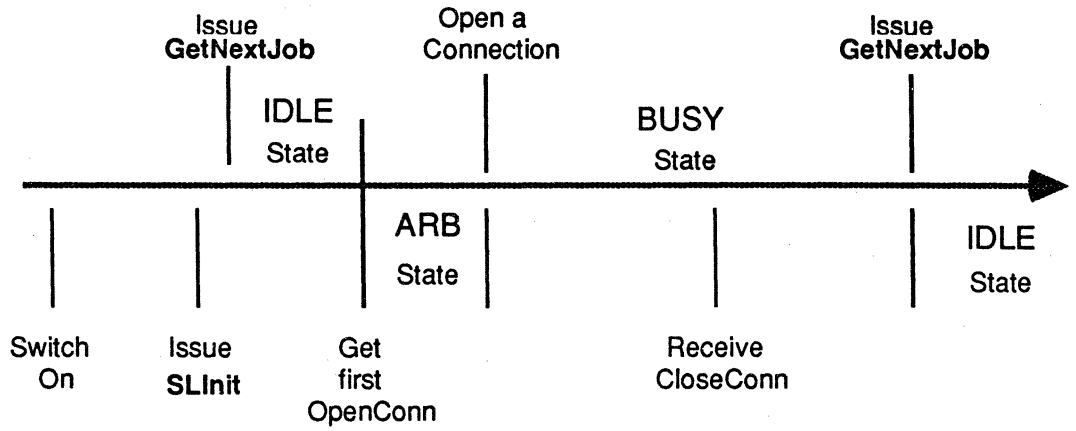


Figure 1: Server State Diagram

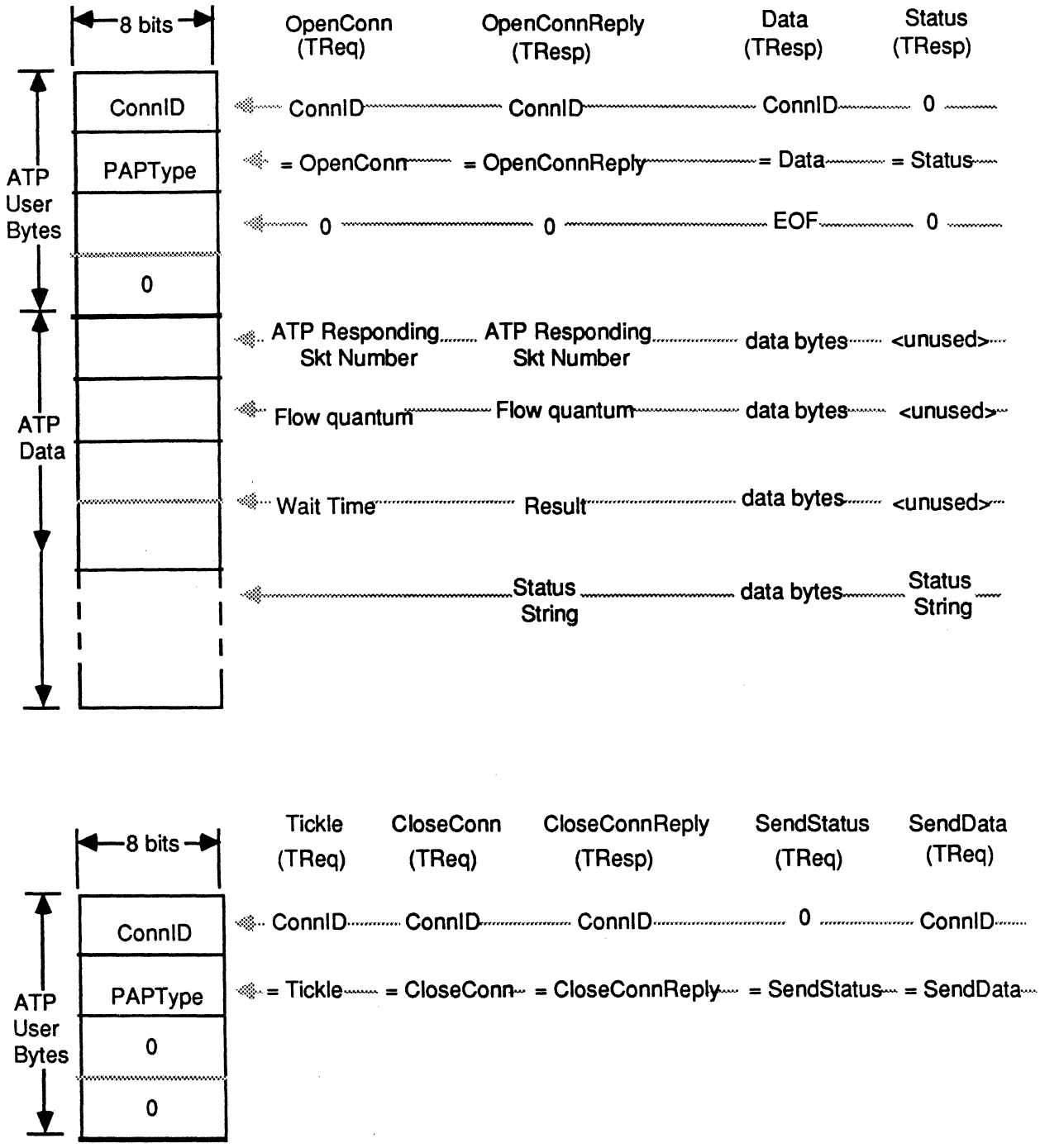


Figure 2. PAP packet format

AppleTalk Printer Access Protocol Example Program

The following is an example of a program that calls the AppleTalk Printer Access Protocol. It is the Downloading Program that is available on the Programming and Debugging Aids disk. Note that it was compiled under the name PSDump and then renamed to be the Downloading Program.

```
PROGRAM PSDump;

{ Alan B. Oppenheimer
  November 9, 1984  V0.1A
  January 17, 1985  V1.0A  Update for release ??!!
  February 5, 1985  V1.0B  Re-link for PAP 1.2C
  COPYRIGHT (C) 1984, 1985 APPLE COMPUTER INC. }

{-------------------------------------------------------------------------------

     Program to read a postscript text file and dump it to PAP

-------------------------------------------------------------------------------}

{$ASM+}
{$M+}
{$U-}
  USES     {$U Obj/QuickDraw    } QuickDraw,
           {$U Obj/OSIntf       } OSIntf,
           {$U Obj/ToolIntf     } ToolIntf,
           {$U Obj/PackIntf     } PackIntf;


  CONST
           lastMenu = 3;        { number of menus }
           appleMenu = 1;       { menu ID for desk accessory menu }
           FileMenu = 256;      { menu ID for File menu }
           EditMenu = 257;      { menu ID for Edit menu }

  VAR
           myMenus:  ARRAY [1..lastMenu] OF MenuHandle;
           code, i : INTEGER;
           temp, doneFlag:  BOOLEAN;
           myEvent:  EventRecord;
           RdBuffPtr, WrtBuffPtr: Ptr;
           ServerName, WrtStr: STR255;
           hCurs,wCurs:  CursHandle;
           iBeam,watch:  Cursor;
           whichWindow: WindowPtr;

           {debug : TEXT;lf:  CHAR;}

  { *** PAP Workstation Procedures *** }

  FUNCTION PAPOpen ( VAR RefNum : INTEGER; PrinterName : Ptr;
                     FlowQuantum : INTEGER;
                     StatusBuff : Ptr; VAR Compstate : INTEGER) : INTEGER; EXTERNAL;

  FUNCTION PAPStatus ( PrinterName : Ptr; StatusBuff : Ptr ) : INTEGER; EXTERNAL;

  FUNCTION PAPRead ( RefNum : INTEGER; rxBufPtr : Ptr;
                     VAR rxBufLen : INTEGER; VAR EOFByte : INTEGER;
                     VAR CompState : INTEGER) : INTEGER; EXTERNAL;

  FUNCTION PAPWrite ( RefNum : INTEGER; txBufPtr : Ptr;
                      txBufLen : INTEGER; EOFByte : INTEGER;
                      VAR CompState : INTEGER) : INTEGER; EXTERNAL;

  FUNCTION PAPClose ( RefNum : INTEGER ) : INTEGER; EXTERNAL;

  FUNCTION PAPUnload : INTEGER; EXTERNAL;

  { *** End PAP Procedures *** }

  PROCEDURE SetUpMenus;
    { Once-only initialization for menus }
    VAR i:  INTEGER;
    BEGIN
      InitMenus;    { initialize Menu Manager }
      myMenus[1] := GetMenu(appleMenu);
      myMenus[1]^^.menudata[1] := CHR(Applesymbol);
      AddResMenu(myMenus[1],'DRVR');     { desk accessories }
      myMenus[2] := GetMenu(fileMenu);
```

```
    myMenus[3] := GetMenu(editMenu);
    FOR i:=1 TO lastMenu DO
      InsertMenu(myMenus[i],0);
    DrawMenuBar;
  END;    { of SetUpMenus }

PROCEDURE BeSFE;

  CONST ReadSize = 4096;

  TYPE
      ProtoBuf = PACKED ARRAY [1..ReadSize] OF CHAR;

  VAR Res, FRes, RCount, InEOFByte, OutEOFByte : INTEGER;
      ourPRefNum,ourFRefNum : INTEGER;
      PAPWrBuff : ProtoBuf;
      ReadBuf : PACKED ARRAY [1..512] OF Char;
      ourStatBuf,ourVolName : STR255;
      OpenCompState, RdCompState, WrtCompState : INTEGER;
      junk,FCount : LONGINT;
      ourPtr : Ptr;
      openStuff : SFReply;
      ourPoint : Point;
      ourSFTypeList : SFTypeList;

  PROCEDURE IssueReads;
      BEGIN
        IF RdCompState <= 0 THEN
            Res := PAPRead (ourPRefNum,@ReadBuf,RCount,InEOFByte,RdCompState);
      END;

  BEGIN { of BeSFE }

    OutEOFByte := 0; InEOFByte := 0;
    ourPoint.H := 50;    ourPoint.V := 50;

    ourSFTypeList[0] := 'TEXT';
    SFGetFile (ourPoint,ourStatBuf,NIL,1,ourSFTypeList,NIL,openStuff);
    IF NOT openStuff.good THEN EXIT(BeSFE);     { User hit cancel }

    Res := PAPOpen (ourPRefNum,@ServerName,1,@ourStatBuf,OpenCompState);
    IF Res < 0 THEN EXIT(BeSFE);       { Exit on error }
    REPEAT UNTIL OpenCompState <= 0;

    Res := FSOpen (openStuff.fName,openStuff.vRefNum,ourFRefNum);
    WrtCompState := 0;
    Res := PAPRead (ourPRefNum,@ReadBuf,RCount,InEOFByte,RdCompState);

    REPEAT

      FCount := ReadSize;

      REPEAT IssueReads UNTIL WrtCompState <= 0;

      FRes := FSRead (ourFRefNum,FCount,@PAPWrBuf);
      IF FRes <> 0 THEN OutEOFByte := 1;
      Res := PAPWrite (ourPRefNum,@PAPWrBuf,FCount,OutEOFByte,WrtCompState);

    UNTIL FRes <> 0;

    REPEAT IssueReads UNTIL (WrtCompState <= 0) AND (InEOFByte > 0);

    Res := PAPClose (ourPRefNum);
    Res := FSClose (ourFRefNum);

  END;  { of BeSFE }

PROCEDURE DoCommand(mResult: LongInt);
  VAR name:  STR255;
      arefNum,j, theMenu, theItem : INTEGER;
  BEGIN { of DoCommand }
    theMenu := HiWord(mResult); theItem := LoWord(mResult);
    CASE theMenu OF
```

```
        appleMenu:
          BEGIN
            GetItem(myMenus[1],theItem,name);
            arefNum := OpenDeskAcc(name)
          END;

        FileMenu:
          BEGIN
            CASE theItem OF
              1: EXIT(PSDump);  { exit the program}
              2: BEGIN {Start Command}
                   BeSFE;
                 END;
            END {of CASE theItem};
          END;

        END;    { of menu case }
      HiliteMenu(0);
    END;  { of DoCommand }


BEGIN    { main program }
  InitGraf(@thePort);
  InitFonts;
  FlushEvents(everyEvent,0);
  InitWindows;
  SetUpMenus;

  SetCursor(arrow);

  { Set up the server name data structure }
  ServerName := 'LaserWriter:LaserWriter@*';
  ServerName[0] := CHR(11);  {change string length to length of object string}
  ServerName[12] := CHR(11); {change colon to length of type string}
  ServerName[24] := CHR(0);  {change @ to length of zone part of name}

  doneFlag := FALSE;
  REPEAT
    SystemTask;
      temp := GetNextEvent(everyEvent,myEvent);
      CASE myEvent.what OF
      mouseDown:
        BEGIN
          code := FindWindow(myEvent.where,whichWindow);  {returns whichWindow}
          CASE code OF

            inMenuBar: DoCommand(MenuSelect(myEvent.where));

            inSysWindow: SystemClick(myEvent,whichWindow);

          END     { of code case }
        END;    { of mouseDown }

      keyDown,autoKey: ;

      updateEvt:
        BEGIN
          whichWindow := POINTER(myEvent.message);
          BeginUpdate(whichWindow);
          EndUpdate(whichWindow);
        END;   { of updateEvt }

      END;     { of event case }
  UNTIL doneFlag;

END.
```

Appendix F

Programming and Debugging Aids

Programming and Debugging Aids

Inside LaserWriter contains various files and programs of interest to developers of applications for the Apple LaserWriter and other POSTSCRIPT printers. Two disks are supplied. The first disk, entitled "Screen Fonts", contains a complete set of screen fonts and a copy of Font Mover. The second disk, entitled "Programming and Debugging Aids" contains the following folders:

Error Handler

Font Metrics

Cookbook Examples

Downloading Program

Screen Fonts Disk

This disk contains a complete set of printer screen fonts, including the roman, bold, italic and bold italic faces. Since the standard LaserWriter software derives all screen fonts and character widths from the Roman face by using the same width for italic and by adding one extra pixel per character for bold, there is an error between the screen widths and the printer widths, particularly for bold characters. If you want to write an application that computes line breaks exactly and that does line layouts exactly, you may want to take advantage of these more accurate screen fonts. Use Font Mover to move these fonts into the system file on your startup disk. To do this:

1. Transfer the font file that you want and Font Mover to your startup disk.
2. Open up Font Mover by double clicking on the font file.
3. Copy the fonts that you want by selecting "Copy" from the Font Mover menu.
4. Select Quit.

Error Handler

When the LaserWriter detects an error in a file that it has received, it normally sends an error message to the workstation from which the file originated. Unfortunately, the current Macintosh printing software has no way to display or otherwise record such an error. If you are developing POSTSCRIPT applications, you may wish first to send the contents of the Error Handler document to the LaserWriter (by means of the Downloading Program. See the downloading instructions in the section entitlied " Instructions for spooling, editing and downloading a Postscript file from a Macintosh application.", in this appendix). This causes the LaserWriter to report any subsequent errors by printing them on a hardcopy page in addition to sending them back to the workstation. (Note that you can install the Error Handler only if the LaserWriter's system administrator password has not been changed from its standard value of zero.)

Font Metrics

In order to produce the most accurate and pleasing printed results, application programs require information about the sizes of individual printed characters. While this information can be obtained by querying the printer directly, it is often more advantageous to obtain it from a separate file. For each of the 13 LaserWriter fonts, the Font Metrics folder includes an Adobe Font Metrics file that gives details about all the characters in the font. The format of an Adobe Font Metrics (.afm) file is described in the POSTSCRIPT Language Manual. These Font Metrics files are in memory based MacWrite format. It you read them with Disk Based MacWrite, MacWrite will automatically convert them to the Disk Based MacWrite format. If you do this, there is no way to convert them back again, so if you need them in memory based MacWrite format, make a copy of them first.

Cookbook Examples

See the Postscript Cookbook section of this document.

Downloading Program

See the next section, entitled " Instructions for spooling, editing and downloading a Postscript file from a Macintosh application."

Instructions for Spooling, Editing and Downloading a Postscript file from a Macintosh application.

In normal use, the Macintosh Print Manager sends a Postscript file to the printer each time that it wants to print a document on LaserWriter. It is possible to spool that file to disk and look at the result for debugging purposes. It is also possible to edit the spooled file and then transmit it to the printer. This feature allows advanced users to add features to Macintosh documents that are not supported by the current Macintosh Print Manager (translation and rotation on the page or rotated text for example). The syntax for this spooled Macintosh file will be defined in the final release of Inside LaserWriter, but it is not currently available.

To Spool a Postscript File.

When a Macintosh application is run to print a document on LaserWriter, the Postscript output that is generated can be spooled to a disk file instead of sending it to the printer by using the following procedure:

1. Select "Print" fron the application's file menu in the usual way.
2. When the print dialog appears, click OK in the usual way.
3. Immediately press the Control and F keys simultaneously and hold them down until the message "Creating Postscript file" appears on the top of the screen.
4. Quit from the application.
5. A file labelled "Postscript" should be on the desktop. (If there was a previous file labelled "Postscript" on the desktop, it will have been destroyed.)
6. Rename the file as something other than Postscript if you want to keep it so that it is not destroyed the next time that you spool a Postscript file.

To Edit the Postscript File.

The Postscript file created above can be edited using Mac Write as follows:

1. Open the file labelled "Postscript" with MacWrite available on one of the disk drives.
2. A message saying "should a Carriage return signify a new paragraph or a line break?" will appear. Select "Paragraphs".
3. A message saying "This document is being converted and will open as Untitled" will appear. Select "OK". (This is because the file generated is a Text Only file, not a full MacWrite file.)
4. The Postscript file will appear as a MacWrite document. This file is in Postscript using the QuickDraw translation routines predefined in the Apple Printer Initialization File (LaserPrep) that is downloaded to the printer whenever the printer is power on. The syntax of this file is defined in Appendix J of Inside LaserWriter.
5. Edit the Postscript file in the usual way.
6. Save the Postscript file by selecting "Save As" from the file menu.
7. Select "Text Only" from the save dialog.
8. Save the document in the usual way. When a dialog comes saying "Should a Carriage Return be put at the end of each line or only between paragraphs?", select "Paragraphs".

To Download the Postscript File and run it.
(usually resulting in printing)

The Postscript file created above can be run on Laserwriter as follows:

1. Open the application on the diagnostic disk entitled "Downloading Program".
2. Select "Start" from the file menu.
3. Select the Postscript file that you want to transmit to the printer and select "Open". The Downloading Program will load the Postscript file and transmit it to the LaserWriter named "LaserWriter". If your LaserWriter is not named "LaserWriter" you will have to use the application "Namer" that comes on the Installation disk to rename the printer back to the name "LaserWriter".
4. When the wristwatch disappears from the screen, select "Quit" from the File menu to get back to the desktop.

Appendix G

Example of the things that you can do with LaserWriter

Introducing

LaserWriter . . .

Apple's breakthrough in visual communication

THE WATERMILL
RESTAURANT

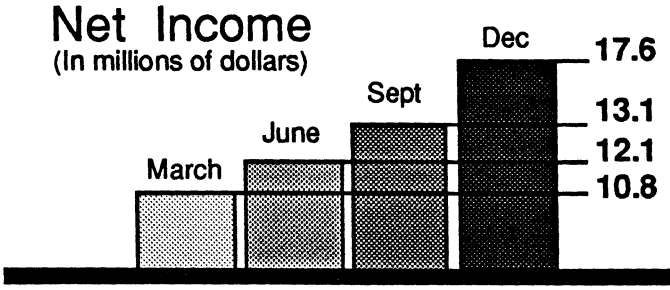
Financial Report

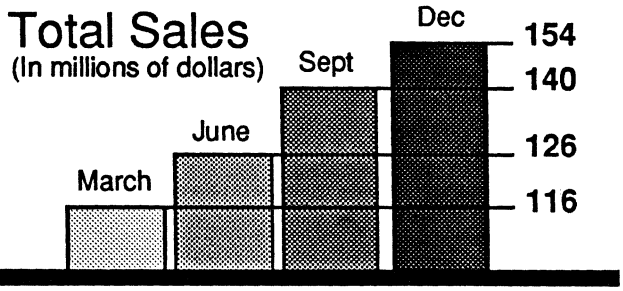
The Watermill Restaurants, Inc.

1984 Year in Review

	March 31	June 30	Sept. 30	Dec. 31
TOTAL RESTAURANT SALES	$115,600	$125,790	$139,723	$153,660
COSTS AND EXPENSES				
Cost of Sales	61,460	65,035	71,994	76,140
Operating, G &A (see Note 1)	32,722	36,400	40,542	42,890
Interest (long-term)	251	226	185	96
	94,433	101,661	112,721	119,126
Income before Federal Taxes	21,167	24,129	27,002	34,534
Provision for Federal Income Taxes	10,374	12,003	13,902	16,976
NET INCOME	$10,793	$12,126	$13,100	$17,558
NET INCOME PER SHARE	$1.08	$1.20	$1.31	$1.76
CASH DIVIDENDS	$0.20	$0.20	$0.20	$0.20

(Dollars in thousands, except per share amounts.)



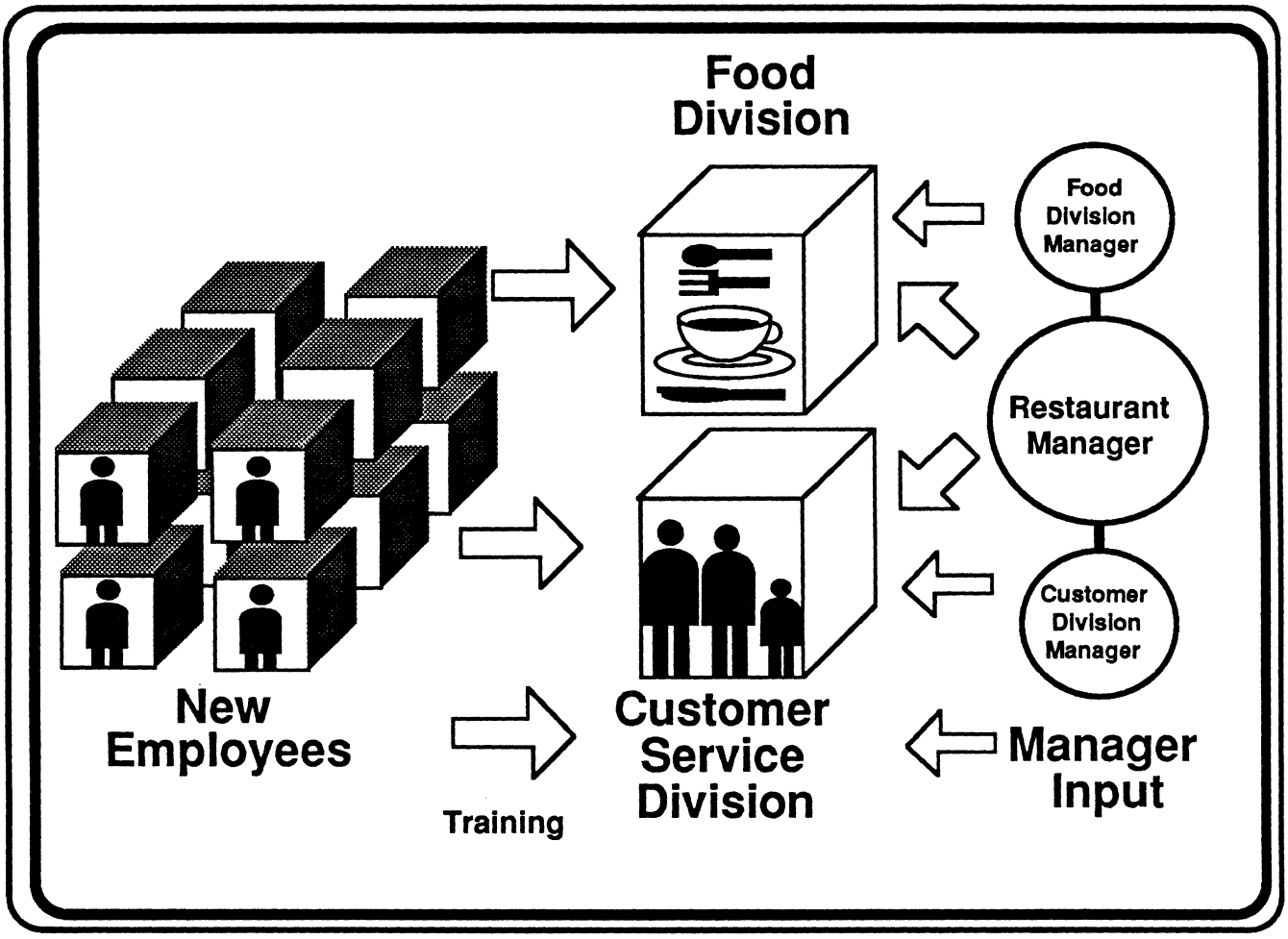


Note 1. Six restaurants owned by others, including certain directors and officers of the Company, are managed by the Company under contracts entered into in fiscal year 1972. As consideration for managing the restaurants, the Company receives 35% of the restaurants' net operating income as defined in the agreements. The Company compensates the restaurant managers out of its management fees.

THE WATERMILL
RESTAURANT

EMPLOYEE TRAINING

- Food and Customer Services are responsible for training new employees.
- Restaurant managers oversee training and review progress frequently with division managers.
- Newly trained employees are placed in the field for a six-week trial period.



THE WATERMILL News

The Monthly Newsletter of THE WATERMILL Restaurants, Inc.

FEBRUARY 12, 1985 VOLUME XIII NUMBER 45

Grand Opening of New WATERMILL in Rolling Hills, West Virginia

Rolling Hills, West Virginia will soon be the proud host to a new WATERMILL Restaurant, opening in March. A print ad campaign, offering a free coupon good for one glass of wine or a slice of our famous Chocolate Toffee Pie, will run for two weeks in local newspapers prior to the opening.

Take a valentine to lunch or dinner!

Welcome to our new look!

After market testing in 35 WATERMILL sites across the country, the new WATERMILL Restaurant logo (see above) was unanimously approved yesterday by the Board of Directors. It will be implemented next quarter. Watch your mail for phase-out requirements and start planning your inventory levels now. Kudos to Kristee Kreitman of our design staff for the snappy new look!

Sign up now for Spring courses in pastry making

Recent customer surveys confirmed our guess that demand for delicate French dessert pastries is increasing by leaps and bounds. In response to the demand, French chef Jean-Pierre Dubonnet will be teaching pastry courses in the regional sites during the month of April. All master chefs are encouraged to attend. Registration forms are available from Personnel.

35 New Managers Complete WATERMILL Training Course

Corporate Training Division is pleased to announce the arrival of 35 new managers for the East Coast region. The sixteen men and nineteen women, ranging in age from 25 to 38, completed their coursework at Corporate Headquarters with flying colors--the best scores overall for an incoming class in the company's history. Please welcome them aboard! They will be honored with diplomas and full fanfare at the annual Spring Banquet.

Special Wine Discounts Now Available

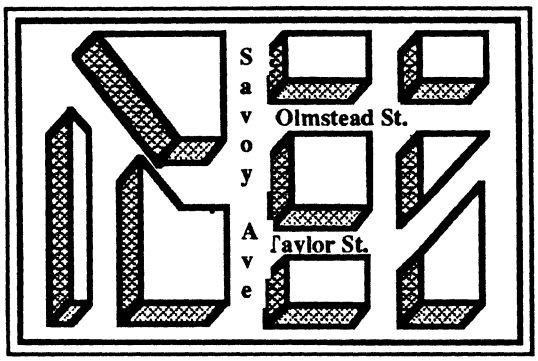
After successful contract negotiations last month with Sutter Home, Stag's Leap, and Trefethen, a new per case discount with those California wineries will go into effect May 1, 1985. Wine stewards should look for a special March mailing outlining the wines included in the discount program.

THE WATERMILL Restaurants-- Now Celebrating 25 Years of Fine Dining

(See feature article, page 3)

Published monthly by THE WATERMILL Publishing Group, Cambridge, MA 02142. Story ideas and letters should be submitted to Lynnea Johnson and Elizabeth Yerxa, Managing Editors, THE WATERMILL Publishing Group, P.O. Box 269, Cambridge, MA 02142.

Cut Here



Our Newest
Watermill Restaurant
is located at 101 Savoy Ave.

The Watermill Restaurant is located between Olmstead St. and Taylor St. on Savoy Ave. Plenty of Free Parking. Open 11am-12pm Mon. thru Sun.

THE WATERMILL
RESTAURANT

First Class Mail

ANNOUNCING THE OPENING OF
THE WATERMILL RESTAURANT
AT 101 SAVOY AVE.

G·R·A·N·D O·P·E·N·I·N·G

The Watermill Restaurants, Inc.

125 West Broadway
Personnel, Suite 300
Cambridge, Ma. 02142

PERSONNEL REQUISITION

REQUISITION NO.

EMPLOYMENT SPECIALIST

JOB TITLE		DATE NEEDED
DEPARTMENT NAME/NUMBER	JOB LOCATION	
SHIFT ☐ DAYS ☐ SWING ☐ GRAVEYARD	SALARY RANGE	☐ EXEMPT ☐ NON-EXEMPT
☐ PERMANENT ☐ TEMPORARY (DURATION)		PAY GRADE
☐ ADDITION TO HEAD COUNT ☐ REPLACEMENT (NO ADDITION TO HEAD COUNT)		NAME OF EMPLOYEE REPLACED
CAUSE OF REPLACEMENT		
TO WHOM WILL EMPLOYEE REPORT?		WHO WILL CONDUCT INTERVIEW

QUALIFICATIONS

EDUCATION

EXPERIENCE

DESCRIPTION OF JOB (BE SPECIFIC)

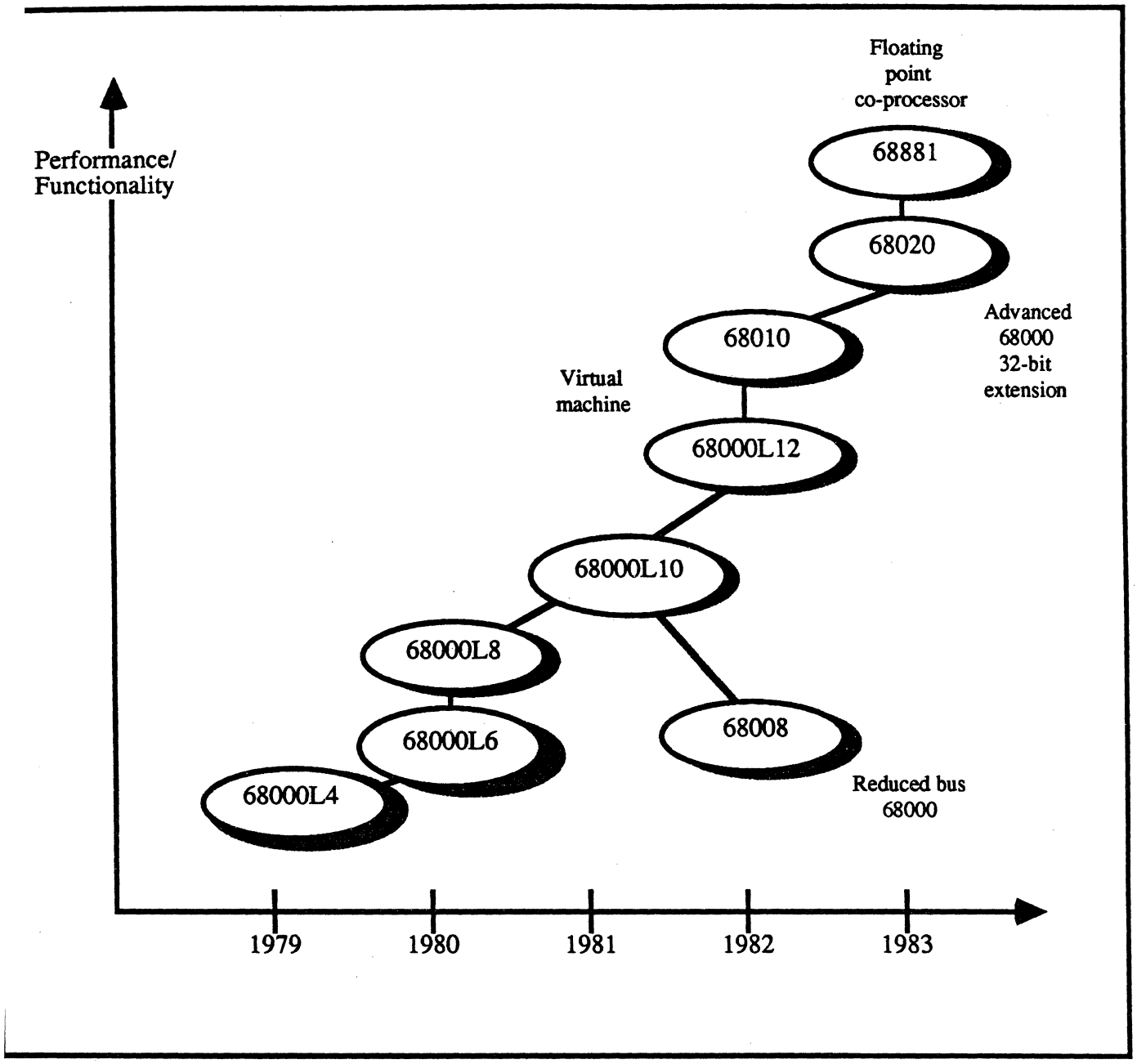
APPROVALS

INITIATOR	DATE	DIRECTOR	DATE	PRESIDENT	DATE
SUPERVISOR/MANAGER	DATE	VICE PRESIDENT	DATE	PERSONNEL	DATE

NEW HIRE

NAME	START DATE	SALARY

Form Number 001-0013-00 3/81

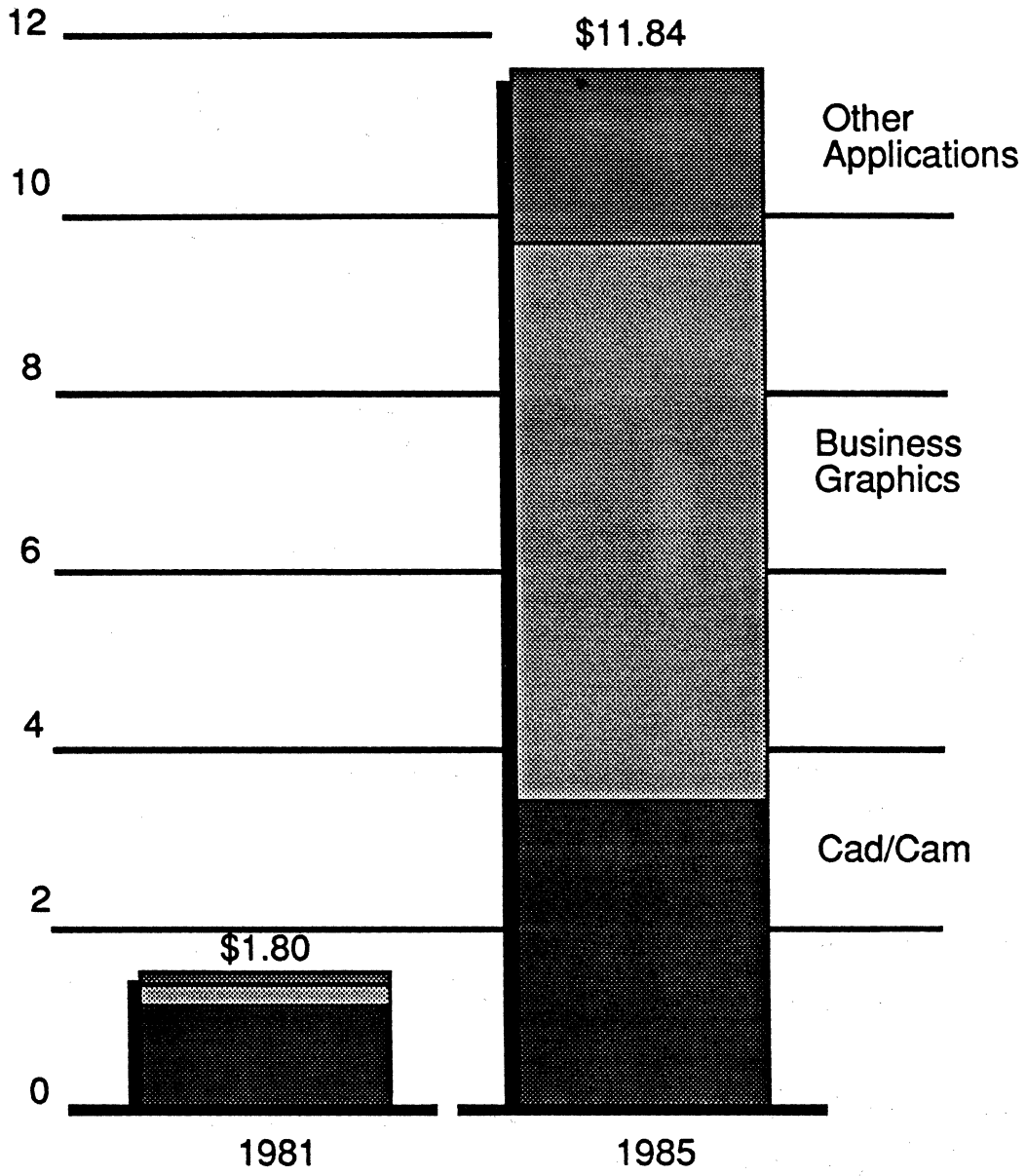


Genealogy of the MC68000 processor family. The first chip on the market was a 4-MHz version, denotated by the L4 at the end of the model number. Its siblings are either faster or have different functional features.

High-end, 16-bit microcomputers gravitate toward MC68000 chips

Introduced in late 1979, the Motorola MC68000 has cut a wide swath through the field of 16-bit microprocessors, emerging as the designers' favorite chip for high-end micro- computer systems. Although it still costs considerably more than its two main competitors - the Intel 8086 and the Zilog Z8000 - the MC 68000's 32-bit internal architecture, speed and sheer elegance, far surpass all of the competition.

The expected explosion of computerized graphics in the office...



* Billions of Dollars

source: The Yankee Group

1984 Republican Convention

Last month in Decker Communications Report, we featured lessons learned from speakers at the Democratic Convention. The lessons were different in Dallas. For the most part it was as if the media masters behind the scenes managed the speakers to serve as a colorless setting from which President Reagan could sparkle like a diamond. In contrast to the Cuomo's, Jacksons, et. al., there were few standouts. But there were lessons.

Katherine Ortega: Confidence was lacking

As a keynote speaker, she was out of her league. You had to feel sorry for her. Here was a good example of why we shouldn't thrust people into positions before they're ready -- or better yet -- why people should always be ready for the positions that are thrust upon them. Ortega did not have the confidence -- the personal impact -- to carry her words. If she had speaking experience, had she built up her confidence over the years, here speech would have had a much different result without changing her words.

As one commentator said, "she seemed like someone giving a valedictory address," That doesn't make for memorability. Where we'll definitely hear from Mario Cuomo again, we'll not see much of Katherine Ortega except for her signature on our money.

Jeanne Kirkpartick: Tough and believable

On the other hand, she was a pleasant surprise -- a tough speech from a tough lady. She was strong, straightforward and forceful.

What makes her so believable is a good low voice that is not affected but reeks of credibility. She had very strong eye communication with the crowd -- better than almost all the convention speakers. (Surprisingly, neither Democrats and Republicans knew how to use the teleprompters well, obviously a non-partisan issue.) Her phrasing, timing, and pausing were excellent, reminding one of Barbara Jordan, another great speaker.

Baker, Kemp, Dole: Sincerity but no fire

This trio of potential 1988 campaign frontrunners seem to have blended together -- adequate enough, sincere, earnest, but lacking the fire in the belly. And in the case of Robert Dole, his final "thank you" was tossed off as if he had to get off the stage quick, and maybe he did. His wife was next.

Elisabeth Dole: Bright but cliched

She was good -- yet we'd give her mixed reviews. Introduced by her husband as having laryngitis, she did super with the physical affliction, and made no excuses or weak asides about it. She's obviously a strong and bright speaker -- with good voice, bearing and posture, and eye communication. Why then resort to old cliches, as in her opening, "Thank you for that great introduction, Bob. You gave it just like I wrote it."

George Bush: Obvious second fiddle

In preparation for "The Great Communicator," there was the acceptance speech of good old reliable George Bush. He is professional, he is polished and he is persuasive. Yet he is no burning Bush -- here we miss that fire of enthusiasm and excitement. Perhaps, like Ferraro, he consciously or unconsciously holds back because he's in second position. If so, he is a lesson to all of us whom hold ourselves back, whether from a perceived lesser position or from just a lack of confidence. Our energy and enthusiasm will suffer for it.

STATECO ADMINISTRATIVE SERVICES
MEMBER CLAIM FORM

INSTRUCTIONS:

1. Complete one Member Claim Form for each patient.
2. Attach an itemized bill containing patient's name, provider of service's IRS # name and address, type date and amount charged for each supply or service for each member claim.

MAIL THIS FORM WHEN COMPLETED T

Stateco Adminstrative Services
P.O. Box 28367
San Jose, Ca. 95159
Attn: Claims Dept.

PATIENT'S NAME _____ Date of Birth Mo. Day Yr SEX ☐ MALE ☐ FEMALE RELATIONSHIP TO EMPLOYEE: ☐ SELF ☐ SPOUSE ☐ CHILD ☐ OTHER

LAST FIRST MIDDLE

OCCUPATION _____ EMPLOYER _____

COVERED BY MEDICARE? ☐ YES ☐ NO IF YES, EFFECTIVE DATE (HOSP) PART A Mo. Day Yr (MED) PART B Mo. Day Yr

GROUP NO.	COVERAGE CODE	AREA CODE	PHONE NUMBER.

EMPLOYEE SOCIAL SECURITY NUMBER _____

EMPLOYEE NAME _____
LAST FIRST MIDDLE

ADDRESS _____

CITY STATE ZIP

NAME OF EMPLOYER

PROVIDER NAME _____
Physician, Laboratory, Pharmacy, Clinic, etc...

ADDRESS _____

CITY _____ STATE _____ ZIP _____

DATE OF 1ST SERVICE Mo. Day Yr

ILLNESS	ACCIDENT	WORK RELATED	PREGNANCY RELATED
☐ YES ☐ NO	☐ YES ☐ NO	☐ YES ☐ NO	☐ YES ☐ NO

KIND OF ILLNESS _____ DATE OF ONSET Mo. Day Yr

DATE OF ACCIDENT Mo. Day Yr

HOW ACCIDENT OCCURRED _____

WHAT INJURIES WERE SUSTAINED _____

DOES PATIENT HAVE OTHER HEALTH INSURANCE? ☐ YES ☐ NO

POLICY HOLDER'S NAME _____ POLICY NUMBER _____
LAST FIRST MIDDLE

INSURANCE COMPANY NAME _____
STREET CITY STATE

I CERTIFY THAT THE INFORMATION ON THIS CLAIM FORM IS TRUE AND CORRECT TO THE BEST OF MY KNOWLEDGE . I AUTHORIZE THE RELEASE OF ANY MEDICAL INFORMATION NECESSARY TO PROCESS THIS CLAIM FOR THE DURATION MARKED ABOVE.

_____ Mo. Day Yr
MEMBER'S SIGNATURE (PARENT'S SIGNATURE IF PATIENT IS A MINOR)

So that Stateco can promptly review your claim for benefits, please review the form and the instructions to insure it has been completed correctly.

Appendix H

Using the Macintosh Print Manager

Appendix H - Using the Macintosh Printing Manager

This appendix contains information specific to the LaserWriter and the use of the Macintosh Printing Manager and Printer Driver. For more details on printing from the Macintosh, refer to the Printing section of the Inside Macintosh Reference Manual. Also included is an example of an application that uses each of the print manager calls.

Printing Overview

The Printing Manager was designed in such a way that the application programmer could write printing code that was printer independent. This independence was achieved by:

1. Allowing the application to interface to the Printing Manager through specific calls which remain the same even though the internal Printing Manager code changes with each particular printer.

2. Allowing the user to interface directly to the Printing Manager to set the desired printing configuration through the use of standard printing dialogs.

The figure below shows the flow of control for printing on the Macintosh.

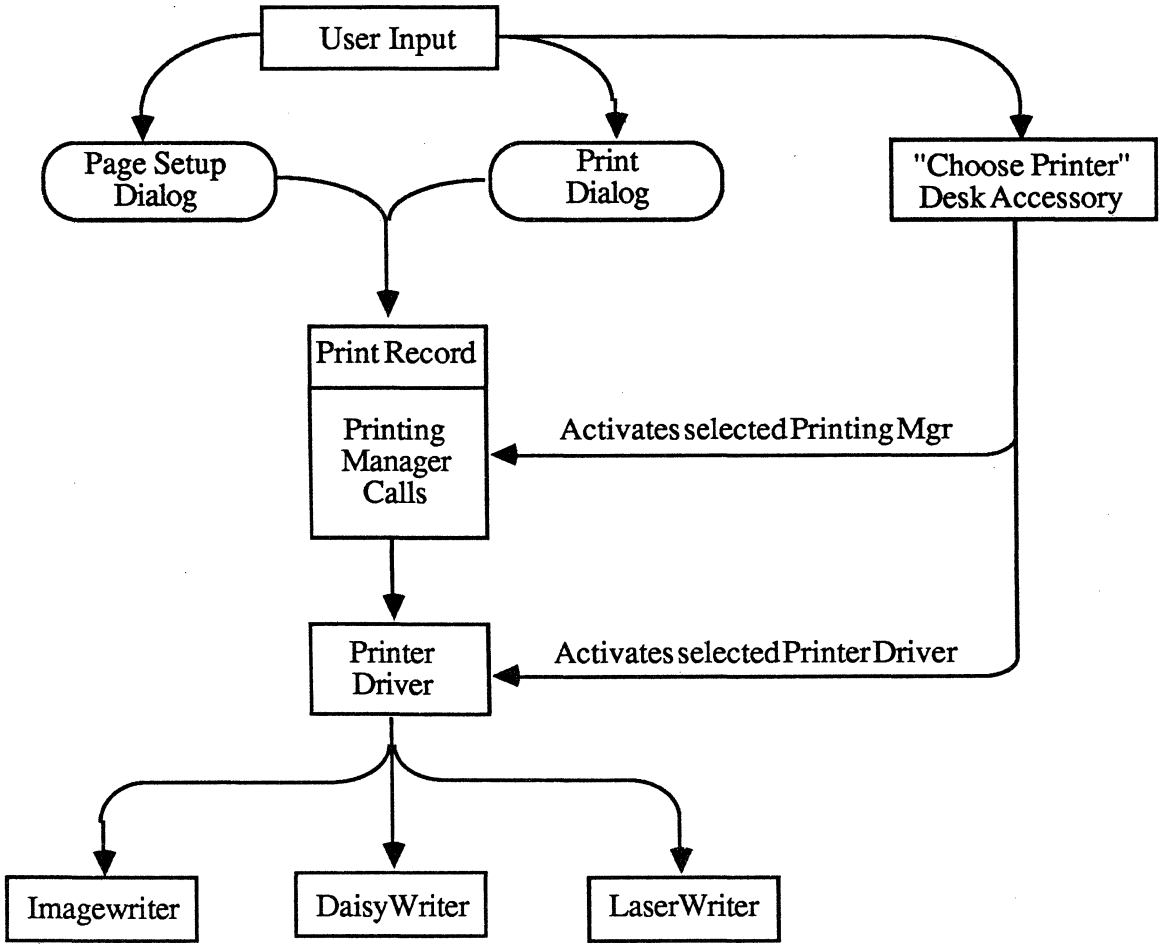


Figure 1. Printing Overview

Below is quick overview of the various items in the above figure.

Page Setup Dialog
This is the Printing Managers standard dialog for setting the page size, orientation, etc of a print job. It can only be used if the Printing Manager has been opened(see below). It will be displayed by a call to PrStlDialog(...);

Print Dialog
This is the Printing Managers standard dialog for setting the print quality, number of copies, pages to print, etc. of a print job. Like the Page Setup Dialog, it can also only be used if the Printing Manager has been opened. It will be displayed by a call to PrJobDialog(...); When the user clicks OK in this dialog, the application should jump to its routine that uses the Printing Manager routines for creating the printout .

Choose Printer Desk Accessory
This is a desk accessory that will allow the user to switch between available printers without having to leave the application. If the application works with the printable page size, it will have to handle these printer changes and the resulting page size change.

Print Record
Information that describes the printing job (ie. style, quality, copies, page dimensions, etc) is stored in the Print Record. The Printing Manager will use this data to create the proper print image. **Note that the application should not change any of the data in this record.**

Printing Manager
The Printing Managers internal code is dependent on the printer that is currently selected, but the routines the Application calls will always remain constant.

The standard Printing Manager routines are:

To open and setup the Printing Manager:

PrOpen;	Prepares Printing Manager for use.
PrintDefault(...);	Fills the Print Record with default values.
PrValidate(...);	Checks compatability of Print Record and the Printing Manager.

To create a print image or draft print:

PrOpenDoc(...);	Initializes a printing port for creating the print image.
PrOpenPage(...);	Starts a new page and sets up to receive the print image.
PrClosePage(...);	Ends the page that PrPageOpen'ed.
PrCloseDoc(...);	Closes the printing port terminates the print imaging process.

To print an image that has been spooled:

PrPicFile(...);	Prints the print image if it was spooled to disk.

To detect or set error conditions during printing:

PrError;	Used to check for possible printing errors.
PrSetError;	Used to cancel the printing process.

To close the Printing Manager:

PrClose; Closes the Printing Manager, but not the Printer Driver.

Printer Driver

As with the Printing Manager, the Printer Drivers code is dependent on the printer that is currently selected, but the routines the application calls will remain constant.

The standard Printer Driver calls are:

To open or close the driver.

PrDrvrOpen;
PrDrvrClose;

To make the driver purgeable or not purgeable.

PrPurge;
PrNoPurge;

To get the software version number and Device Control Entry.

PrDrvrVers;
PrDrvrDCE;

To perfrom a variety of operations: Control printer, print bitmaps, stream text, etc.

PrCtlCall(iWhichCtl, lParam1, lParam2, lParam3);

In summary, as shown in the figure above:

1. The User, using the Page Setup Dialog, controls the parameters which determine:
   - Paper size
   - Orientation of printed view
   - Reduction or enlargement of the view

2. The User, using the Print Dialog, controls the parameters which determine:
   - Number of copies
   - Pages to print
   - Source of paper (ie manual feed, form feed, etc.)

3. The User, using the Choose Printer desk accessory, selects the printer on which they wish to print on.

4. The Application, on the other hand, just has to be concerned with:
   - Calling PrJobDialog(...) when the user selects "Print" from the menu list.
   - Calling PrStlDialog(...) when the user selects menu item "Page Setup" .
   - Invoking standard Printing Mgr calls when the user decides to print.

Using the Printing Manager

The Printing Manager is the high level interface to the Printer Driver. It is composed of code particular to the currently selected printer. You can access the Printing Manager calls by linking your object code to PrLink.obj. In your USES section you should include the line {$U Obj/MacPrint } MacPrint which is the applications inerface to the Printing Manager and the Printer Driver.

Depending on the specific printer, the Printing Manager has different printing capabilities. For instance;

When printing to the Imagewriter it can print an image in High Resolution, Standard Resolution, or Draft (text only). It will spool the image in High or Standard Resolution modes and send the text directly to the printer in Draft mode.

When printing to the LaserWriter, it sends the image directly to the printer. Spooling is not an option as when printing to the Imagewriter. Also, it will print the image in only one mode - High Resolution.

Below, a discussion of when to use the Printing Manager calls is presented. For more information on the various calls, refer to the Printing section of Inside Macintosh.

Opening and Closing the Printing Manager

Use PrOpen to open the Printing Manager and PrClose to close it. PrOpen will open the Printing Manager's resource file and and allow you to use all the Printing Manager calls. It also opens the Printer Driver. If you are short on memory, bracket every Printing Manager call with PrOpen and PrClose.
Example:

```
PrOpen;
TrueFalse := PrStlDialog(PrintRecord);
PrClose;
```

This will allow you to use the dialogs and other calls without having the resource file continually open along with its associated overhead. Note that normally you would call PrOpen during the initialization of the application and PrClose during termination.

Printing Manager Dialogs

There are two Printing Manager dialogs, a style dialog and a job dialog. PrStlDialog is the dialog which is used to determine the print style (ie. paper size and paper orientation, etc.) that is to be used. PrJobDialog is the dialog that is used to determine the printing quality (if relevant), number of pages to print, etc. The dialogs are customized specifically for each printer, so you should always use the standard Printing Manager dialog calls to get the appropriate dialog. You should call the dialogs from the "File" menu. When the user selects "Page Setup" call PrStlDialog and when they select "Print" call the PrJobDialog and if it returns true, jump to your routine containing the printing loop (see below). If you are going to have a menu item such as "Print One" where the user can print without going through the dialogs, be sure to call PrValidate before entering the printing loop.

Note: You should never second guess what the Printing Manager is going to do with the variables in the Print Record. If you set the Print Record variables directly from your own dialog, chances are you will eventually have problems as the Printing Manager code gets revised and changed. So don't use your own dialogs to set the Print Record variables. This is especially true when printing to the laser printer.

The Printing Loop

The printing loop consists of the following Printing Manager calls that actually image your text or graphics to the printer. Depending on you particular drawing procedure and the way you image pages, the actual printing code will vary, but the basic printing loop will always follow the format below.

```
PrintPort := PrDocOpen(PrintRecHdl,nil,nil);        {opens the printing grafport }
For count := 1 to numCopies do
Begin
  PrOpenPage(PrintPort, nil);                       {starts a new page}
  CallmyDrawingProc;
  PrClosePage(PrintPort);                           {ends the current page}
End;
PrDocClose(PrintPort);                              {ends the print job}
```

Printing a Spooled File

With the Imagewriter, when the user printed in High or Standard resoultion, a print file containing the print image was spooled to disk first and then later printed. This was done for memory consideration on the 128K Macintosh. The printing loop above would create the spooled file and then the application would call PrPicFile to actually print the file after it had allocated the memory for the imaging process (sometimes as much as 20K). Because the user might have selected Draft printing which does not create a spooled file the application would call PrPicFile with the following IF statement:

```
IF PrintRecHdl^^.PrJob.bJDocLoop = bSpoolLoop THEN
PrPicFile(PrintRecHdl, nil,nil,nil,Status);
```

With the LaserWriter, the print file is currently not spooled. All printing occurs in the Draft mode so PrPicFile is never utilized. It should however be included in your printing code in case it does spool at some future date.

Checking for Printing Aborts and Errors

You can check for printing errors by making a call to PrError. It will return either a printer specific error or an OS error. Your applicatiuon can then take appropriate action.

You can monitor whether the user cancels the printing process with your own dialog by installing a pointer to the monitoring procedure in the PrintRecords idleProc variable: prInfoPT.pIdleProc. When you detect that the user has selected the cancel button, call PrSetError(iPrAbort). This will flag the Printing Manager to terminate the print job so it will close all the files and end properly. Do not force printing termination yourself.

Optimizing For The LaserWriter

Below is some information that will help you optimize your code for the LaserWriter.

How to determine which printer is currently selected.
The printers are designated as:

Imagewriter = 1
DaisyWriter = 2
LaserWriter = 3

Call PrValidate to insure you have the current Print Record. After this call, byte $947 in low memory will contain the negative of one of the three constants above. This value corresponds to the current printer. The default value is -1which stands for the Imagewriter.

Using QuickDraw with the LaserWriter

- Only SrcCopy transfer mode is supported, the other 15 are not.
- The grafverb "invert" is not supported.
- Regions are not supported, try to simulate them with polygons.
- Clip regions should be limited to rectangles.
- Rotated or Scaled bit images will not print correctly
- There is a small error in character widths between screen and printer fonts, so don't rely on them being exactly the same. Only the end points will be accurate. If you are in left, right or center justify mode, only those points will be accurate.

Memory Considerations
When you print on the LaserWriter, you will only be able to print in Draft mode except that the quality will be high as opposed to low quality on the Imagewriter. This means that you will not be spooling and therefore your data and printing code will have to be resident in memory at the same time. In terms of memory requirements, you will need around 15 to 20K just for the Printer Driver, AppleTalk, etc. every time you print.

Printable Paper Area
There is a 0.45 inch border that surrounds the printable area of the paper. Note that this is different from the print area that was available when using the Imagewriter. The value of the printable rectangle is stored in the Print Record in the variable prInfoPT.rPage.

Speed Considerations

- Try to avoid using any of the QuickDraw Erase calls (ie. EraseRect, EraseOval, etc.). It takes a lot of time to handle the erase function because every bit (90,000 bits/sq.in.) has to be cleared. Erasing is generally unnecessary because the paper does not need to be erased the way the screen does.
- Printing patterns takes a long time, since the pattern bitmap has to be built. The patterns of Black, White, and all the Grays have been optimized for the LaserWriter. If you use a different pattern, it will work but just take a little longer to print. Also,

- Try to aviod changing fonts frequently. Font characters are stored as general mathematical functions and it takes 0.5 seconds to build the bit image of a character the first time it is used. For the fastest possible printing, use the fonts that have their bit image built in to the ROM (Courier 10, Times 12, Helvetica 12) and the fonts whose bit image is built whenever the printer is idle (Times and Helevtica 10, 14 and Times and Helvetica Bold 10, 12, 14). See Appendix D (the Advanced Users Suppliment) for more details.
- When clipping strings, make sure that the clipping region/rect is greater than the bounding box of the text. The reason is that a clipped character will need to be rebuilt and this takes time. So beware especially of ascenders and decenders.

When to validate the Print Record

You validate the Print Record by calling PrValidate(...); You should call it when the application starts up and whenever you interface with the Print Record (like when you get the printable page size). The dialogs PrStlDialog(...) and PrJobDialog(...) will call PrValidate(...) when they are called.

Spool-A-Page, Print-A-Page

Many applications when printing on the Imagewriter, because of disk space limitations, spooled a page and then printed a page. As noted above in Memory Considerations, there is not any spooling when printing to the LaserWriter. Inorder to optimize for the LaserWriter though, you will probably want to have two sets of printing loops. One where you spool a page and then print it (for the Imagewriter) and the other where you would just print without any consideration for spooling (for the LaserWriter). Since you can tell which printer is currently selected (see above), you will be able to correctly switch between the two methods. Note that the majority of applications will not have to know what printer they are currently printing on.

Zero Width QuickDraw Objects that are Filled

QuickDraw objects that enclose zero pixels and are not framed but filled, will not print on the Imagewriter nor show up on the screen, but they are real and will be printed on the LaserWriter.

pPageFrame inPrOpenPage

This parameter was originally intended to be for scaling the QuickDraw picture of the given page which was contained in the spooled file. When printing to the ImageWriter, this parameter works fine. When printing on the LaserWriter, this parameter is ignored and does not have any effect on the print output.

Canceling, Pausing the Printing Process

If you install a procedure for handling the users requests to cancel printing, with the option of pausing the printing process, beware of timeout problems when printing to the LaserWriter. Communication between the Macintosh and the LaserWriter have to be maintained, so if you have a pause option and do not let communication continue a no-response error will be generated and the Printing Manager will abort the print process. This will probably not make your user very happy. The solution is to check if you are printing to the LaserWriter, if so disable the pause option. If printing to the Imagewriter, enable this option.

Using the Printer Driver

The Printer Driver is the device driver that communicates with the currently selected printer through the Printer or Modem Port. For each printer, there will be a different Printer Driver. If you are going to print using only the Printer Driver calls, link your code with PrScreen.obj. This will give you full access to the Printer Driver calls. Note that if you link with PrLink.obj, you will not have access to PrDrvrNoPurge, PrDrvrPugre, PrDrvrVers, and PrDrvrDCE.

Communicating with the Printer Driver through the Device Manager

You can communicate with the Printer Driver through the standard Device Manager calls: OpenDriver, CloseDriver, Control, and Status. Its driver name and reference number are available as predefined constants:

```
CONST sPrDrvr    = '.Print';  {Printer Driver name}
      iPrDrvrRef = -3;        {Printer Driver reference number}
```

If you want to communicate with the Printer Driver in this manner, read the Device Manager section of Inside Macintosh for more details.

Communicating with the Printer Driver through the Standard Printing Manager Calls

You can communicate with the Printer Driver through standard Printing Manager routines that allow you to:

1. Open or close the Printer Driver.
2. Reset Printer and control its characteristics.
3. Make the Printer Driver purgeable or unpurgeable.
4. Obtain information about its software version number.
5. Obtain information about its Device Control Entry.
6. Print a bit map on the printer.
7. Stream text to the printer.
8. Send "blind IO" (uninterpreted ASCII characters) to the printer.
9. Send special PostScript characters and commands.

These capabilities are described in detail below:

Opening and Closing the Printer Driver

PROCEDURE PrDrvrOpen;	Opens the driver, you will still need to reset the printer to actually print anything.
PROCEDURE PrDrvrClose;	Closes the driver. PrClose of the Printing Manager will not close the driver.

Making the Printer Driver Purgeable or Unpurgeable

PROCEDURE PrNoPurge;	Prevents driver from being purged from the heap.
PROCEDURE PrPurge;	Allows the driver to be purged (this is the default).

Printer Driver Software Version Number and Device Control Entry

```
FUNCTION PrDrvrDCE: Handle;     Returns handle to the drivers DCE.
FUNCTION PrDrvrVers: INTEGER;   Returns the drivers software version number.
                                Currently it is 2.
```

NOTE: The rest of this section will deal with the standard Printer Driver Control Call and the operations you can perform with it. The form of the call is as follows:

```
PrCtlCall(iWhichCtl: INTEGER; lParam1,lParam2,lParam3: LONGINT);
```

The parameter iWhichCtl designates the operation to be performed and lParam1,2,3 depend on the operation that is being performed.

Printer Control Calls

The printer controls, iWhichCtl = iPrDevCtl, have been expanded to match those of the Printing Manager. The ReSet and PageEnd controls have been renamed DocOpen and PageClose, and two new controls have been added - PageOpen and DocClose. Below are some control constants which have been predefined for you followed by an expalaination of the new control operations.

```
CONST lPrRest      = $00010000; {resets printer, same as DocIOpen}
      lPrPageEnd   = $00020000; {same as PageClose}
      lPrLineFeed  = $00030000; {send only carriage return}
      lPrLFSixth   = $0003FFFF; {space down 1/6 inch}
```

All the Control Calls will take the form:

```
PrCtlCall(iDevCtl, lParam1 0, 0);
```

where lParam1is composed of two integers, **ihigh** and **iLow.** lParam2 and lParam3 should always be set to 0 on all control calls.

DocOpen

You will need to call this once before you transmit data. It opens the printer and prepares the driver to transmit data.

```
iHigh := 1;                          {It is a DocOpen operation}
iLow := #;                           {number of copies to make of each page}

lParam1 := $00010002;
PrCtlCall(iPrDevCtl, lParam1,0,0);     {Open the printer & set #copies to 2}
```

PageOpen
This will initialize PostScrip and the driver buffers for a new page

```
iHigh := 4;                          {specify it is a PageOpen operation}
iLow  := 0;                          {does nothing}

lParam1 := $00040000;
PrCtlCAll(iPrDevCtl, lParam1,0,0);   {Start a new page}
```

Line Feed/Downward Spacing
Linefeeding through the Printer Driver is quite versitile. There are three methods. You can space a specific amount, just send a carriage return (no feed), or space 1/6 or 1/8 of an inch. The particular method is designated in the iLow integer of lParam1. The conventionis that negative numbers signify 1/6th of an inch linefeed, zero signifies a carriage return, and a positive number will cause the printer to space down 1/72th times the value. The high integer contains the value of 3 to designate the Linefeed operation.

```
iHigh := 3;                          { specify the Linefeed operation}
iLOw := +#, 0, -#;                   { specify the amount of linefeed}

Example:
lParam1 := $00030000;                Low word = 0, only  send carriage return.
lParam1 := $0003FFFF;                Low word = -1, space down 1/6 of an inch.
lParam1 := $00030020;                Low word = +20, space down 20/72 of an inch.
lParam1 := $00030001;                Low word = +1, space down 1/72 of an inch.

lParam1 := $0003FFFF;
PrCtlCAll(iPrDevCtl, lParam1,0 ,0);  {space down 1/6th of an inch}
```

PageClose
This operation flushes the buffers and sends a signal to the LaserWriter to print the page. The number of copies printed is determined by the value specified in the low integer of lParam1 during the DocOpen call.

```
lParam1 := $00020000;
PrCtlCAll(iPrDevCtl, lParam1,0 ,0);  {close the page & print iCopies of it}
```

DocClose
This operation closes the printer connection and releases the driver buffers.

```
lParam1 := $00050000;
PrCtlCAll(iPrDevCtl, lParam1,0 ,0);  {close the driver}
```

BitMap Printing
This capability allows you to send a bitmap directly to the printer. The call is:

```
PrCtlCall(iPrBitsCtl, lParam1, lParam2, lParam3);
```

where:

lParam1 = pointer to the QuickDraw bitmap.
lParam2 = pointer to the bounds rectangle of the bitmap.
lParam3 = 0 for non-square pixels, 1 square pixels.

An example of this call used to print the screen would be:

PrCtlCall(iprBitsCtl, Ord(@ScreenBits), Ord(@ScreenBIts.bounds), lPaintBits);
where lPaintsBits is a constant equal to 1.

Note that when printing to the LaserWriter, lParam3 should always be equal to 1. This designates that the bitmap will be printed out with square pixels, which is the only way the LaserWriter can print.

Text Streaming

The basic IO capability has been expanded to include six different variations of the standard text streaming function. The six variations are designated by the lParam3 parameter. Below is a table listing the variants. Note that if the selected printer is the Imagewriter, only lPAram3 = 0 will be recognized.

Operation	**lParam1**	**lParam2**	**lParam3**
ShowBuf	BufPtr	numBytes	0
StdBuf	BufPtr	-/+ numBytes	1
HexBuf	BufPtr	Hi=Offset; Lo=-/+numBytes	2
Fill	FillByte	Hi=numlines; Lo=numBytes	3
PrintF	FmtStrPtr	ArgPtr	4
PrintR	Hi=ResId;Lo=Index	ArgPtr	5

ShowBuf (lParam3 =0)

This operation takes a pointer to a text buffer in lParam1 and the length of that buffer in lparam2. This call will stream text to the LaserWriter by imbedding the text in a PostScript [...] show call. On the ImageWriter, it will just stream text to the printer.

StdBuf (lParam3 = 1)

This operation is simply a "raw IO" call that sends the data to the printer without the PostScript Show call as in ShowBuf. It takes a pointer to the data buffer in lParam1 and the length of the buffer in lParam2. If numBytes is negative, the data is treated as PostScript "text". This means that parentheses and the backslash characters are preceeded by the PostScript backslash escape character and characters >127 are sent as octal. If numBytes is positive the data is sent out without modification.

HexBuf (lParam3 = 2)

This operation sends data as ASCII Hex data. It takes a pointer to the data buffer in lParam1. The High integer of lParam2 contains a shift count which specifies the number of bits to skip over, providing a bit bit addressing capability. The Low integer specifies the number of bytes in the data buffer. If the number of bytes is specified as a negative number, the data is inverted before sending it to the printer.

Fill (lParam3 = 3)

This operation is used for sending the same byte of information to the printer many times. lParam1 contains the byte to be sent in the low byte of the low integer. The number of times this byte will be written per line -ibytes- is specified in the low integer of lParam2. The number of lines to be written -iLines- is specified by the value in the high integer of lParam2. Each line is seperated by a carriage return. Thus, the total amount of data is iLines X iBytes

PrintF (lParam3 = 4)
This is a formatting operation, similar to the PrintF in the 'C' language. Lparam1 points to a string with imbedded commands. LParam2 points to the data used by these commands.

PrintR (lParam3 = 5)
This is the same as PrintF except that the format string is in a string indexed resource. The high integer of lParam1 is the resource ID and the low integer is the index into that resource. The resource type is POST= STR#.

Command	***Args***	***Meaning***
^i	integer	The argument is converted to decimal ascii.
^c	char	The argument is a PostScript character. Parentheses and backslashes are preceeded by an escape character, and characters>127 are sent as octal.
^b	ptr, integer	The arguments compose a buffer and are treated exactly as in the StdBuf control call.
^h	ptr,2 integers	The arguments compose a buffer and are treated exactly as in the HexBuf control call.
^n	-	A NewLine is sent.
^r	long,2integers	The long is a resource type, the integers a resource id/index pair. If the index is zero, the resource is treated as a string resource. If it is positive, it is treated as a string index resource. If it is negative, its size is determined by GetHandleSize.
^b	integer	The argument is treated as a boolean.
^^	-	A single ^ character is sent.
^R	long,2 integers	Just as ^r but the data immediately follows the ^R in the format string itself.

Example

The following is an example of an application program that uses all of the Print Manager calls. The application itself can be found on the disk entitled "Programming and Debugging Aids" under the name "Example Application".

```
{$X-}   {Turn off stack expansion. This is a Lisa concept, not needed on Mac}
{$U-}   {Turn off the Lisa Libraries. This is required by the WorkShop}
{$R-}   {Turn off range checking}

Program LaserPrinting;

(*
 --   Jeffery J. Bradford,  Macintosh Technical Support, Jan 1985
 --
 -- This is a printing example which demonstrates how to print using
 -- the Printing Manager. To use the calls of the Printing Manager
 -- link with obj/PrLink.obj.
 --
 -- This program was written to test out printing cases for the LaserWriter.
 -- If you want to use it to test your own stuff, add the procedure and
 -- call it from the menu list. (see how the program works - its simple).
 -- To print just put your procedure into the Case statement in the Print loop.
 --
 -- The printer dialogs are in a separate menu so you can set up the
 -- format any way you want and then choose Printing Operation from
 -- another menu. Also, be sure to select the desired font, style, and
 -- text size before selecting the print menu item.
 --
 -- If you follow the steps below, your code should print on the Imagewriter
 -- as well as the LaserWriter without any problem.
 --
 --         0. Link with obj/prLink.obj.
 --         0. include {$U Obj/MacPrint  } MacPrint;  in the USES statment.
 --
 --         1. PrOpen       to open the Printing Mgr resource file.
 --         2. PrintDefault to set the initial default settings
 --         2a PrValidate   to set the initial default settings also
 --
 --    now you are ready to print:
 --         3. PrOpenDoc    to open the printing grafport.
 --         4. PrOpenPage   to setup a new page up for printing.
 --         5. Draw into printer port whatever you want printed.
 --         6. PrClosePage  to finish the current page print
 --         7. PrCloseDoc   to close and dealocate the printing grafport.
 --
 --    now you are finished printing
 --         8. PrClose      to close the Printing Mgr resource file
 --
 --
 *)

USES
    {$U Obj/Memtypes  } MemTypes,
    {$U Obj/QuickDraw } QuickDraw,
    {$U Obj/OSIntf    } OSIntf,
    {$U Obj/ToolIntf  } ToolIntf,
    {$U Obj/PackIntf  } PackIntf,
    {$U Obj/MacPrint  } MacPrint;

CONST
    Bit7      = 7;
{menu stuff}
    AppleMenu = 256;
    PrintMenu = 257;
    FontMenu  = 258;
    StyleMenu = 259;
    PrDlogMenu= 260;
    PrDrvrMenu= 261;
    PicScrMenu= 262;

{print tests for Pr Mgr only}
    PrDrawPicture = 1;
    PrMakeQDCalls = 2;
    PrFramePage   = 3;
    PrFrameText   = 4;
    PrUseTextBox  = 5;
    PrBitMap      = 6;

{devices}
    theScreen  = 0;
    theImageW  =-1;
    theDaisyW  =-2;
    theLaserW  =-3;

TYPE
    IconData = Array[0..95] of integer;
```

```
    GetStuff = Packed Record
       Case Integer of
        0: (a0: Integer);
        1: (b1,b0: SignedByte);
       End;

VAR
{bit map stuff}
    icons: Array[0..5] of IconData; {store 6 icons in here}
    whichIcon:    integer;          {holds icon ID number}
    QDPicture:    PicHandle;        {handle to the QD Picture}

{global program stuff}
    Finished:      Boolean;    {used to terminate the program}

{font stuff}
    CurntFontID:   Integer;    {holds the currently selected text font}
    CurntStyleID:  Style;      {holds the currently selected text style}
    CurntSizeID:   Integer;    {holds the currently selected text size}
    PrevFontChked: Integer;    {holds the previously slected font}

{printer stuff}
    PrRecordHdl:   THPrint;    {handle to the print record}
    PrPortStorage: TPrPort;    {storage for the printer grafport}
    PrintPort:     TPPrPort;   {pointer to the printers grafport}
    DefaltPage:    Rect;       {holds the currently selected printer page size}
    CurPrTest:     Integer;    {holds the value to the current drawing routine}
    PrDlgPtr:      DialogPtr;  {pointer to the cancel/pause dialog}
    PrStopDlgRec:DialogRecord;{record for the cance/pause dialog}

{window stuff}
    DragArea,                  {holds the area where window can be dragged in}
    GrowArea,                  {holds the area to which a window's size can change}
    Screen:     Rect;          {holds the screen dimensions}
    aWindow:    WindowPtr;     {pointer to text window}
    WRec:       WindowRecord;  {storage for text window record}
    errRect:    Rect;          {rect for displaying printer errors}

{-------------------------------------------------------------------------------
                    end of global variable definition
-------------------------------------------------------------------------------}

{The following procedures contain printing code to: Print text, print graphics,}
{print a bitmap, print the screen, and test out weird things developers do}

{-------------------------------------------------------------------------------}
PROCEDURE FramePage     (Where: integer);              FORWARD;
PROCEDURE PrintBitMap   (Where: integer);              FORWARD;
PROCEDURE MakeQDCalls   (where:integer);               FORWARD;
PROCEDURE ShowAllQDCalls(Where:integer);               FORWARD;
PROCEDURE ShowQDPic     (Where:integer);               FORWARD;
PROCEDURE UseTextBox    (Where: Integer);              FORWARD;
PROCEDURE FrameText     (Where: Integer);              FORWARD;
PROCEDURE PutPicScrap;                                 FORWARD;
PROCEDURE PrDrBitMap;                                  FORWARD;
PROCEDURE PrDrScr_wEvtCtl;                             FORWARD;
PROCEDURE PrDrScrBitMap;                               FORWARD;
PROCEDURE PrDrStreamText;                              FORWARD;

{-------------------------------------------------------------------------------}

PROCEDURE SetPrDialog(Printer: Integer);
Var IType: Integer;
    IHdl:  Handle;
    IRect: Rect;
Begin
  PrDlgPtr := GetNewDialog(257, @PrStopDlgRec, Pointer(-1));

{disable the continue item to start with}
  GetDItem(PrDLgPtr, 3, Itype, IHdl, IRect);    {get the item}
  HiliteControl(ControlHandle(IHdl), 255);      {disable it}

{if its the laser disable the pause item}
  If Printer = theLaserW then
  begin
    GetDItem(PrDLgPtr, 2, Itype, IHdl, IRect);    {get the item}
    HiliteControl(ControlHandle(IHdl), 255);      {disable it}
  end;

  DrawDialog(PrDlgPtr);
```

```
End;

{-------------------------------------------------------------------------------}

PROCEDURE ChkForCanceOrPause;
Var ProcessIt: Boolean;
    itemHit:   Integer;
    itemHdl:   Handle;
    itemRect:  Rect;
    Event:     EventRecord;
    DlgPtr:    DialogPtr;

Begin
  ProcessIt := GetNextEvent(EveryEvent, Event);
  If IsDialogEvent(Event) then
    If DialogSelect(Event, DlgPtr, ItemHit) then
      Case itemHit of
        1: PrSetError(iPrAbort);
        2: begin end; {pause enable continue disable pause go into repeat loop}
        3: begin end; {continue and enable pause}
      End;
End;

{-------------------------------------------------------------------------------}

{Following is the Procedure for printing with the Printing Manager}


PROCEDURE PrintIt(PrintWhat: Integer);
Var
    numCopies:   Integer;      {holds the number of copies the user wants}
    Count:       Integer;      {used to count number of copies}
    TempPort:    GrafPtr;      {holds the current port while printport is used}
    Status:      TPrStatus;    {record for status while spool printing occors}
    dummy:       boolean;      {just a dummy boolean for function assignment}
    thePrinter:  integer;      {ID of the type of printer}

Begin

{get the current port & save it}
    GetPort(TempPort);

{get the type of printer we are printing to}
    thePrinter:= GetStuff(                                  {get the high byte}
                          Get Printer type from Low Memory byte at $947 (See documentation)
{set our idleproc to handle aborts & pauses; Setup the Dialog also}
    PrRecordHdl^^.prJob.pIdleProc := @ChkForCancelOrPause;
    SetPrDialog(thePrinter);

{open up the printer port, port is set automaticly}
    PrintPort := PrOpenDoc(PrRecordHdl, @PrPortStorage, Nil);

{loop on the number of copies}
    numCopies := PrRecordHdl^^.prJob.iCopies;
    For count := 1 to numCopies do
    begin
       PrOpenPage(PrintPort, Nil);    {Nil= do not scale the drawing}

         Case CurPrTest of
           PrDrawPicture: ShowQDPic      (thePrinter);— Bug: don't build picture within
           PrMakeQDCalls: ShowAllQDCalls(thePrinter);   OpenPage & ClosePage
           PrFramePage:   FramePage      (thePrinter);
           PrFrameText:   FrameText      (thePrinter);
           PrUseTextBox:  UseTextBox     (thePrinter);
           PrBitMap:      PrintBitMap    (thePrinter);
         End;

       PrClosePage(PrintPort);
    end;

    PrCloseDoc(PrintPort);    {close PrGrafport}
    SetPort(TempPort);        {Reset the port}

{If spooling was selected, print the file now}
    If PrRecordHdl^^.PrJob.bJDocLoop = bSpoolLoop
    then PrPicFile(PrRecordHdl,@PrPortStorage, NIL, NIL, Status);

{get rid of Cancel dialog}
   CloseDialog(PrDlgPtr);
End;
```

```
{-------------------------------------------------------------------------------}

{AAA}
{The procedures below print directly to the Driver}


PROCEDURE PrDrBitMap;
{This procedure prints directly to the Pr Driver, PrClose & PrOpen are}
{here only to test the Driver without Pr Manager interference}
Var
    srcBits    : BitMap;
    srcRect    : Rect;

Begin
  PRCLOSE;   {Only calls below needed, if going to directly to PrDriver }

  srcBits.baseAddr:=@icons[0];            {set start address for icon data}
  srcBits.rowBytes:=6;                   {set 6 as # of bytes per row}
  SetRect(srcBits.bounds,0,0,48,32);     {48 X 32 pixels = 6 X 4 bytes}

  PrDRvrOpen;  {not needed if PrOpen has been called}
  PrCtlCall(iPrDevCtl, lPrReset, 0, 0);
  PrCtlCall(iPrBitsCtl, Ord(@srcBits), Ord(@SrcBits.bounds), 1);
  PrDrvrClose;

  PROPEN;  {open up the Printing Manager again}
End;

{-------------------------------------------------------------------------------}

PROCEDURE PrDrScr_wEvtCtl;
{This procedure prints directly to the Pr Driver, PrClose & PrOpen are}
{here only to test the Driver without Pr Manager interference}
Begin
  PRCLOSE;   {Only calls below needed, if going to directly to PrDriver }

  PrDRvrOpen;  {not needed if PrOpen has been called}
  PrCtlCall(iPrDevCtl, lPrReset, 0, 0);
  PrCtlCall(iPrEvtCtl, lPrEvtAll, 0, 0);
  PrDrvrClose;

  PROPEN;  {open up the Printing Manager again}
End;

{-------------------------------------------------------------------------------}

PROCEDURE PrDrScrBitMap;
{This procedure prints directly to the Pr Driver, PrClose & PrOpen are}
{here only to test the Driver without Pr Manager interference}

Begin
  PRCLOSE;   {Only calls below needed, if going to directly to PrDriver }

  PrDRvrOpen;
  PrCtlCall(iPrDevCtl, lPrReset, 0, 0);
  PrCtlCall(iPrBitsCtl, Ord(@ScreenBits), Ord(@ScreenBits.bounds), 1);
  PrDrvrClose;

  PROPEN;  {open up the Printing Manager again}
End;

{-------------------------------------------------------------------------------}

PROCEDURE PrDrStreamText;
{This procedure prints directly to the Pr Driver, PrClose & PrOpen are}
{here only to test the Driver without Pr Manager interference}

Var TxT: Str255;
    len: Integer;
    lParam1: LongInt;

Begin
  PRCLOSE;   {Only calls below needed, if going to directly to PrDriver }

  TextFont(CurntFontID);         {test changing the font}
  TextFace(CurntStyleID);        {test changing the style}
  TextSize(CurntSizeID);         {test changing the size}

  Txt := 'This is text streaming to the LaserWriter';
  Len := Length(Txt);
  lParam1 := $0003FFFF;
```

```
  PrDrvrOpen;
  PrCtlCall(iPrDevCtl, lPrReset, 0, 0);

  PrCtlCall(iPrIOCtl, Ord(@Txt), LongInt(Len), 0);
  PrCtlCall(iPrDevCtl, lParam1, 0,0);

  PrCtlCall(iPrIOCtl, Ord(@Txt), LongInt(Len), 0);
  PrCtlCall(iPrDevCtl, lParam1, 0,0);

  PrCtlCall(iPrIOCtl, Ord(@Txt), LongInt(Len), 0);
  PrCtlCall(iPrDevCtl, lParam1, 0,0);

  PrCtlCall(iPrIOCtl, Ord(@Txt), LongInt(Len), 0);
  PrCtlCall(iPrDevCtl, lParam1, 0,0);

  PrCtlCall(iPrDevCtl, lPrPageEnd, 0, 0);
  PrDrvrClose;

  PROPEN;  {open up the Printing Manager again}
End;

{-------------------------------------------------------------------------------}

{BBB}
{the procedures below are used to draw into the Print Managers port}


PROCEDURE InitDisplayArea(Where:integer; Var DisplayArea: Rect);
Begin
  If where = theScreen
  then begin
    DisplayArea := aWindow^.portRect;
    SetPort(aWindow);                          {to be sure}
    eraseRect(DisplayArea);
  end
  else DisplayArea := PrRecordHdl^^.prInfoPT.rPage;
End;

{-------------------------------------------------------------------------------}

PROCEDURE FramePage(Where: integer);
{This procedure will frame the windoiw or printable page.}
Var
    DisplayArea: Rect;
    TempPort:    GrafPtr;      {holds the current port while printport is used}
    halflen:     integer;      {used for centering the text}
    Starth:      integer;      {horizontal position of centered text}
    Startv:      integer;      {vertical position of centered text}
    dummy:       boolean;      {just a dummy boolean for function assignment}

Begin
  InitDisplayArea(Where, DisplayArea);

{frame the display area}
  Pensize(3,3);
  FrameRect(DisplayArea);
  pensize(1,1);

{place some centered text in frame, first set the text params}
  TextFont(CurntFontID);                    {set the printers port font}
  TextFace(CurntStyleID);                   {set the printers port style}
  TextSize(CurntSizeID);                    {set the printers port size}

{find the center}
  starth := (DisplayArea.right - DisplayArea.left) div 2;
  Halflen := StringWidth('The printable area is enclosed by this frame') Div 2;
  starth := starth - halflen;
  startv := (DisplayArea.bottom - DisplayArea.top) div 2;

{move to position & draw}
  MoveTo(starth, startv);
  DrawString('The printable area is enclosed by this frame');

End;

{-------------------------------------------------------------------------------}

PROCEDURE PrintBitMap(where: integer);
{This prints a bit map in the rPage area.}
Var
```

```
    DisplayArea: Rect;
    srcBits:     BitMap;
    srcRect:     Rect;
    dummy:       boolean;
Begin
  InitDisplayArea(Where, DisplayArea);

{set the bit map up}
  srcBits.baseAddr:=@icons[0];          {set start address for Lisa icon}
  srcBits.rowBytes:=6;                  {set 6 as # of bytes per row}
  SetRect(srcBits.bounds,0,0,48,32);    {48 X 32 pixels = 6 X 4 bytes}
  srcRect:=srcBits.bounds;              {set the source bounding rect}

{show it}
  If where = theScreen then
  CopyBits(srcBits,thePort^.portBits,srcRect,DisplayArea,srcCopy,Nil) {fill scr}
  else
  CopyBits(srcBits,thePort^.portBits,srcRect,DefaltPage,srcCopy,Nil); {full page}

End;

{-------------------------------------------------------------------------}

PROCEDURE UseTextBox(Where: Integer);
Var
    DisplayArea: Rect;
    Count:       Integer;      {used as a counter}
    TextPage:    Rect;         {destRect for the text}
    TextPtr:     Ptr;          {pointer to the actual text}
    TextLength:  integer;      {length of the text}
    TextJustify: integer;      {justification for the text}
    ViewRect:    Rect;         {rect for viewing text}
    DestRect:    Rect;         {rect for storing text}
    TextHandle:  TEHandle;     {handle to text record}
    TextString:  StringHandle; {store string from resources}

Begin
  InitDisplayArea(Where, DisplayArea);

{first setup the text in the TE record and draw it to the screen}
  ViewRect := DisplayArea;                  {set the display rect}
  DestRect := DisplayArea;
  InSetRect(DestRect,0,4);                  {make the destRect smaller}

  TextHandle := TENew(DestRect,ViewRect);   {get a new record}
  TextHandle^^.txFont := CurntFontID;       {set font for display}
  TextHandle^^.txFace := CurntStyleID;      {set style for displaying the text}
  TextHandle^^.txSize := CurntSizeID;       {set size for displaying the text}

  TextString := GetString(256);             {get the test string from resources}

  HLock(Handle(TextString));                {lock string down}
  HLock(Handle(TextHandle));                {lock text handle down}
  Hlock(Handle(TextHandle^^.hText));        {lock the char handle down}

  For count := 1 to 5 do                    {insert it 5 times}
  begin
    TESetSelect(0,0,TextHandle);            {set the place to insert at begining}
    TEInsert(pointer(ord4(TextString^)+1),  {point to the first character}
                  length(TextString^^),     {get the length of the string}
                           TextHandle);     {pass the string to TextHandle}
  end;

  TECalText(TextHandle);                    {just to be sure everything is OK}

  TextPtr := TextHandle^^.hText^;           {get pointer to the text, its locked}
  TextLength := TextHandle^^.TELength;      {get the length of  the text}
  TextJustify:= 0;                          {set the text justification}

  TextBox(TextPtr, TextLength, DisplayArea, TextJustify);   {draw the text}

  HUnlock(Handle(TextHandle^^.hText));       {unlock the char handle
  HUnLock(Handle(TextHandle));               {unlock the text handle}
  HUnLock(Handle(TextString));               {unlock the string handle}

  TEDispose(TextHandle);
End;

{-------------------------------------------------------------------------}

PROCEDURE FrameText(Where: Integer);
```

```
Var Txt:         Str255;
    len:         integer;
    i:           integer;
    DisplayArea: Rect;
    Frame:       Rect;
    Start:       Point;
    fInfo:       FontInfo;
Begin
  InitDisplayArea(Where, DisplayArea);

{use current settings}
  TextFont(CurntFontID);                    {set the font}
  TextFace(CurntStyleID);                   {set the style}
  TextSize(CurntSizeID);                    {set the size}

{always start the text at this point}
  Start.v := 50;
  Start.h := 50;

{get the string dimensions}
  GetFontInfo(fInfo);                                {using current font}
  Frame.right := StringWidth('Have I been framed correctly- jg') + Start.h;
  Frame.left  := Start.h;
  Frame.bottom:= Start.v + fInfo.descent;
  Frame.top   := Start.v - fInfo.ascent;

{now draw the stuff}
  InSetRect(Frame, -1, -1);   {move it out one pixel}
  FrameRect(Frame);
  Moveto(Start.h, Start.v);
  DrawString('Have I been framed correctly- jg');

End;

{-------------------------------------------------------------------------------}

PROCEDURE BuildQDPicture(where:integer);
Var
    OriginalRect: Rect;
    SaveClip:     RgnHandle;

Begin
  SetRect(OriginalRect,0,0,719,363);        {this rect holds the initial Pic}
  SaveClip := NewRgn;                       {get a Rgn to store the clip}
  GetClip(SaveClip);                        {save the current clip region}
  ClipRect(OriginalRect);                   {set the clip to the drawing area}

  QDPicture := OpenPicture(OriginalRect);   {start the picture}
  Pensize(3,3);
  FrameRect(OriginalRect);                  {frame it}
  PenSize(1,1);
  MakeQDCalls(where);                       {draw the QD calls}
  ClosePicture;                             {close it}

  SetClip(SaveClip);                        {reset the clip }
  DisposeRgn(SaveClip);                     {get rid of new clip}
End;

{-------------------------------------------------------------------------------}

PROCEDURE ShowQDPic(Where:integer);
Var DisplayArea: Rect;
Begin
  InitDisplayArea(Where, DisplayArea);
  BuildQDPicture(where);
  HLock(Handle(QDPicture));
  If Where = theScreen
  then DrawPicture(QDPicture, DisplayArea)
  else DrawPicture(QDPicture, DefaltPage);
  HunLock(Handle(QDPicture));
  KillPicture(QDPicture);
End;

{-------------------------------------------------------------------------------}

PROCEDURE ShowAllQDCalls(Where:integer);
Var DisplayArea: Rect;
Begin
  InitDisplayArea(Where, DisplayArea);
  MakeQDCalls(Where);
End;
```

```
{------------------------------------------------------------------------------}

PROCEDURE DrawIcon(whichIcon,h,v: integer);
{This procedure draws an icon at location h, v}
Var
    srcBits           : BitMap;
    srcRect, dstRect  : Rect;

Begin
  srcBits.baseAddr:=@icons[whichIcon];   {set start address for icon data}
  srcBits.rowBytes:=6;                   {set 6 as # of bytes per row}
  SetRect(srcBits.bounds,0,0,48,32);     {48 X 32 pixels = 6 X 4 bytes}
  srcRect:=srcBits.bounds;               {set the source bounding rect}
  dstRect:=srcRect;                      {make the destination rect the same}
  OffsetRect(dstRect,h,v);               {offset from other icons}

  CopyBits(srcBits,thePort^.portBits,srcRect,dstRect,srcOr,Nil);
End;

{------------------------------------------------------------------------------}

PROCEDURE MakeQDCalls(where:integer);
VAR i: INTEGER;
    tempRect,
    OriginalRect : Rect;
    myPoly       : PolyHandle;
    myRgn        : RgnHandle;
    myPattern    : Pattern;

BEGIN

  {SetRect(OriginalRect,0,0,719,363);  this rect holds the initial Pic}

  { draw two horizontal lines across the top }
  MoveTo(0,18);
  LineTo(719,18);
  MoveTo(0,20);
  LineTo(719,20);

  { draw divider lines }
  MoveTo(0,134);
  LineTo(719,134);
  MoveTo(0,248);
  LineTo(719,248);
  MoveTo(240,21);
  LineTo(240,363);
  MoveTo(480,21);
  LineTo(480,363);

  { draw title }
  TextFont(CurntFontID);    {set the font to currently selected one}
  MoveTo(210,14);
  DrawString('Look what you can draw with QuickDraw');


  {---------  draw text samples --------- }

  MoveTo(80,34);  DrawString('Text');

  TextFace([bold]);
  MoveTo(70,55);  DrawString('Bold');

  TextFace([italic]);
  MoveTo(70,70); DrawString('Italic');

  TextFace([underline]);
  MoveTo(70,85); DrawString('Underline');

  TextFace([outline]);
  MoveTo(70,100); DrawString('Outline');

  TextFace([shadow]);
  MoveTo(70,115); DrawString('Shadow');

  TextFace([]);   { restore to normal }


  { --------- draw line samples --------- }

  MoveTo(330,34);  DrawString('Lines');
```

```
MoveTo(280,25);  Line(160,40);

PenSize(3,2);
MoveTo(280,35);  Line(160,40);

PenSize(6,4);
MoveTo(280,46);  Line(160,40);

PenSize(12,8);
PenPat(gray);
MoveTo(280,61); Line(160,40);

PenSize(15,10);
StuffHex(@myPattern,'8040200002040800'); {create a new pattern}
PenPat(myPattern);                       {set as the new pen pattern}
MoveTo(280,80); Line(160,40);
PenNormal;

{ --------- draw rectangle samples --------- }

MoveTo(560,34);  DrawString('Rectangles');

SetRect(tempRect,510,40,570,70);
FrameRect(tempRect);

OffsetRect(tempRect,25,15);
PenSize(3,2);
EraseRect(tempRect); {this is so the top rect will not show thru the next one}
FrameRect(tempRect);

OffsetRect(tempRect,25,15);
PaintRect(tempRect);   {this rect is painted so we do not have to erase area}

OffsetRect(tempRect,25,15);
PenNormal;
FillRect(tempRect,gray);
FrameRect(tempRect);

OffsetRect(tempRect,25,15);
FillRect(tempRect,myPattern);
FrameRect(tempRect);

{ --------- draw roundRect samples --------- }

MoveTo(70,148);  DrawString('RoundRects');

SetRect(tempRect,30,150,90,180);
FrameRoundRect(tempRect,30,20);

OffsetRect(tempRect,25,15);
PenSize(3,2);
EraseRoundRect(tempRect,30,20);
FrameRoundRect(tempRect,30,20);

OffsetRect(tempRect,25,15);
PaintRoundRect(tempRect,30,20);

OffsetRect(tempRect,25,15);
PenNormal;
FillRoundRect(tempRect,30,20,gray);
FrameRoundRect(tempRect,30,20);

OffsetRect(tempRect,25,15);
FillRoundRect(tempRect,30,20,myPattern);
FrameRoundRect(tempRect,30,20);

{ --------- draw bitmap samples --------- }

MoveTo(320,148);  DrawString('BitMaps');

DrawIcon(0,266,156);
DrawIcon(1,336,156);
DrawIcon(2,406,156);
DrawIcon(3,266,196);
DrawIcon(4,336,196);
DrawIcon(5,406,196);

{ --------- draw ARC samples --------- }

MoveTo(570,148);  DrawString('Arcs');
```

```
SetRect(tempRect,520,153,655,243);
FillArc(tempRect,135,65,dkGray);
FillArc(tempRect,200,130,myPattern);
FillArc(tempRect,330,75,gray);
FrameArc(tempRect,135,270);
OffsetRect(tempRect,20,0);
PaintArc(tempRect,45,90);

{ --------- draw polygon samples --------- }

MoveTo(80,262);  DrawString('Polygons');

myPoly:=OpenPoly;      {capture QD calls that make up the polygon}
  MoveTo(30,290);
  LineTo(30,280);
  LineTo(50,265);
  LineTo(90,265);
  LineTo(80,280);
  LineTo(95,290);
  LineTo(30,290);
ClosePoly;             { end of definition of the polygon}

FramePoly(myPoly);     {now use it just like you would a rectangle or etc.}

OffsetPoly(myPoly,25,15);
PenSize(3,2);
ErasePoly(myPoly);
FramePoly(myPoly);

OffsetPoly(myPoly,25,15);
PaintPoly(myPoly);

OffsetPoly(myPoly,25,15);
PenNormal;
FillPoly(myPoly,gray);
FramePoly(myPoly);

OffsetPoly(myPoly,25,15);
FillPoly(myPoly,myPattern);
FramePoly(myPoly);

KillPoly(myPoly);

{ --------- demonstrate regions --------- }

MoveTo(320,262);  DrawString('Regions');

If where <> theLaserW
then
  begin

    myRgn:=NewRgn;         {allocate space of a new region}
    OpenRgn;               {start saving region defintion calls}

      ShowPen;  {OpenRgn calls HidePen so if drawing to screen call ShowPen }
                {if creating a picture delete this call}

      SetRect(tempRect,260,270,460,350);
      FrameRoundRect(tempRect,24,16);     {rounded corner rectangle}

      MoveTo(275,335);  { define triangular hole }
      LineTo(325,285);
      LineTo(375,335);
      LineTo(275,335);

      SetRect(tempRect,365,277,445,325);    { oval hole }
      FrameOval(tempRect);

      HidePen;    {this call would balance the ShowPen call set above}
    CloseRgn(myRgn);         { end of definition of the region}
    PaintRgn(myRgn);         {show the region with black pattern}
    DisposeRgn(myRgn);       {dont need it any more so throw it away}
  end

else
  begin
    MoveTo(270,300);  DrawString('Dont use regions');
    Moveto(275,320);  DrawString('on LaserPrinter');
  end;
```

```
  { --------- draw oval samples --------- }

  MoveTo(580,262);  DrawString('Ovals');

  SetRect(tempRect,510,264,570,294);
  FrameOval(tempRect);

  OffsetRect(tempRect,25,15);
  PenSize(3,2);
  EraseOval(tempRect);
  FrameOval(tempRect);

  OffsetRect(tempRect,25,15);
  PaintOval(tempRect);

  OffsetRect(tempRect,25,15);
  PenNormal;
  FillOval(tempRect,gray);
  FrameOval(tempRect);

  OffsetRect(tempRect,25,15);
  FillOval(tempRect,myPattern);
  FrameOval(tempRect);

END; {QDCalls}

{-------------------------------------------------------------------------------}

PROCEDURE ChkOnOffItem(MenuHdl:MenuHandle; item, fst, lst:Integer);
Var i: integer;
Begin
  For i := fst to lst do
    If item = i
    then CheckItem(MenuHdl, i, TRUE)        {check it on in menu}
    else CheckItem(MenuHdl, i, FALSE);      {check it off in menu}
End;

{-------------------------------------------------------------------------------}

PROCEDURE ProcessMenu_in(CodeWord:longint; fromMenu:Boolean);
Var
  Menu_No,                      {menu number that was selected}
  Item_No:    integer;          {item in menu that was selected}
  NameHolder: Str255;           {name holder for desk accessory or font}
  MenuHdl:    MenuHandle;       {handle to the menu}
  dummy:      boolean;
  LDummy:     LongInt;

Begin
  If CodeWord <> 0 then {go ahead and process the command}
  begin
    Menu_No := HiWord(CodeWord);
    Item_No := LoWord(CodeWord);

    Case Menu_No of

     AppleMenu: begin
                  GetItem(GetMenu(AppleMenu), Item_No, NameHolder);
                  If OpenDeskAcc(NameHolder) = 0
                  then begin {put up a dialog saying it cannot open it} end;
                end;

    PrDlogMenu: begin
                   Case Item_No of
                     1: begin
                          dummy := PrStlDialog(PrRecordHdl);
                        end;
                     2: begin
                          If PrJobDialog(PrRecordHdl)
                          then PrintIt(CurPrTest);
                        end;
                    {3: line divider}
                     4:Finished := true;       {terminate the program}
                   End;
                 end;

      PrintMenu: Begin
                  MenuHdl := GetMenu(PrintMenu);     {menu handle for PrTests}
                  Case Item_No of
                   1: begin
                        CurPrTest := PrFramePage;
```

```
            ChkOnOffItem(MenuHdl, 1, 1, 6);
            FramePage(theScreen);
          end;

        2: begin
            CurPrTest := PrFrameText;
            ChkOnOffItem(MenuHdl, 2, 1, 6);
            FrameText(theScreen);
          end;

        3: begin
            CurPrTest := PrMakeQDCalls;
            ChkOnOffItem(MenuHdl, 3, 1, 6);
            ShowAllQDCalls(theScreen);
          end;

        4: begin
            CurPrTest := PrDrawPicture;
            ChkOnOffItem(MenuHdl, 4, 1, 6);
            ShowQDPic(theScreen);
          end;

        5: begin
            CurPrTest := PrUseTextBox;
            ChkOnOffItem(MenuHdl, 5, 1, 6);
            UseTextBox(theScreen);
          end;

        6: begin
            CurPrTest := PrBitMap;
            ChkOnOffItem(MenuHdl, 6, 1, 6);
            PrintBitMap(theScreen);
          end;
      End;
    End;

FontMenu: begin
      MenuHdl := GetMenu(FontMenu);              {menu handle for fonts}
      CheckItem(MenuHdl, PrevFontChked, False); {uncheck the prev.one}
      GetItem(MenuHdl, Item_No, NameHolder);     {get new font name}
      PrevFontChked := Item_No;                  {save the new font No}
      GetFNum(NameHolder, CurntFontID);          {get the font ID}
      CheckItem(MenuHdl, Item_No, True);         {check it off in menu}
    end;

StyleMenu: begin
      MenuHdl := GetMenu(StyleMenu);           {menu handle for style}
      Case Item_No of
        1:begin
            CurntStyleID := [];       {plain}
            ChkOnOffItem(MenuHdl, 1, 1, 6);
          end;
        2:begin
            CurntStyleID := CurntStyleID + [Bold];
            CheckItem(MenuHdl, 2, True);       {check it off in menu}
            CheckItem(MenuHdl, 1, False);      {uncheck it in menu}
          end;
        3:begin
            CurntStyleID := CurntStyleID + [Italic];
            CheckItem(MenuHdl, 3, True);      {check it off in menu}
            CheckItem(MenuHdl, 1, False);      {uncheck it in menu}
          end;
        4:begin
            CurntStyleID := CurntStyleID + [underline];
            CheckItem(MenuHdl, 4, True);      {check it off in menu}
            CheckItem(MenuHdl, 1, False);      {uncheck it in menu}
          end;
        5:begin
            CurntStyleID := CurntStyleID + [outline];
            CheckItem(MenuHdl, 5, True);      {check it off in menu}
            CheckItem(MenuHdl, 1, False);      {uncheck it in menu}
          end;
        6:begin
            CurntStyleID := CurntStyleID + [shadow];
            CheckItem(MenuHdl, 6, True);       {check it off in menu}
            CheckItem(MenuHdl, 1, False);      {uncheck it in menu}
          end;

        {7: line divider}

        8:begin  {9 point}
```

```
                        CurntSizeID := 9;
                        ChkOnOffItem(MenuHdl, 8, 8, 13);
                      end;
                     9:begin  {10 point}
                        CurntSizeID := 10;
                        ChkOnOffItem(MenuHdl, 9, 8, 13);
                      end;
                    10:begin  {12 point}
                        CurntSizeID := 12;
                        ChkOnOffItem(MenuHdl, 10, 8, 13);
                      end;
                    11:begin  {14 point}
                        CurntSizeID := 14;
                        ChkOnOffItem(MenuHdl, 11, 8, 13);
                      end;
                    12:begin  {18 point}
                        CurntSizeID := 18;
                        ChkOnOffItem(MenuHdl, 12, 8, 13);
                      end;
                    13:begin  {24 point}
                        CurntSizeID := 24;
                        ChkOnOffItem(MenuHdl, 13, 8, 13);
                      end;

                  End;
                end;

      PrDrvrMenu:begin
                   Case Item_No of
                     1: PrDrBitMap;
                     2: PrDrScr_wEvtCtl;
                     3: PrDrScrBitMap;
                     4: PrDrStreamText;
                     5: begin end;
                   End;
                 end;

      PicScrMenu:begin
                   If Item_No = 1 then PutPicScrap;
                 end;

    End;{case of Menu_No}

    HiliteMenu(0);                {unhilite after processing menu}
  end; {the If codeword <> 0}
End; {of ProcessMenu_in procedure}

{-------------------------------------------------------------------------------}

PROCEDURE DealwthMouseDowns(Event:EventRecord);
Var Location: integer;
    WindowPointedTo: WindowPtr;
    MouseLoc:Point;
    WindoLoc:integer;
Begin
  MouseLoc := Event.Where;
  WindoLoc := FindWindow(MouseLoc, WindowPointedTo);
  Case WindoLoc of

     inMenuBar: ProcessMenu_in(MenuSelect(MouseLoc), True);

     inSysWindow: SystemClick(Event,WindowPointedTo);

     inContent: begin end;
              (*If WindowPointedTo <> FrontWindow
                then SelectWindow(WindowPointedTo)
                else begin {do something} end;*)

     inGrow   : begin end;
              (*If WindowPointedTo <> FrontWindow
                then SelectWindow(WindowPointedTo)
                else ReSizeWindow(WindowPointedTo,MouseLoc,GrowArea);*)

     inDrag   :DragWindow(WindowPointedTo,MouseLoc,DragArea);

     inGoAway :If TrackGoAway(WindowPointedTo,MouseLoc)
               then
                 begin
                   CloseWindow(WindowPointedTo);
                   Finished := true;
                 end;
```

```
  End{ of case};
End;

{-------------------------------------------------------------------------------}

PROCEDURE DealwthKeyDowns(Event:EventRecord);
Var Character:char;
Begin
   Character:= CHR(Event.message MOD 256);

  If BitTst(@Event.modifier,Bit7)
   then
     begin  {key board command}
       ProcessMenu_in(MenuKey(Character), False);
     end
   else
     begin  {regular keyboard entry}
       {TEKey(Character,TextHandle);}
       {Scrolltext}
     end;
End;

{-------------------------------------------------------------------------------}

PROCEDURE DealwthActivates(Event: EventRecord);
Var EventMsgWindow:WindowPtr;
Begin
   EventMsgWindow := WindowPtr(Event.message);
   {DrawGrowIcon(EventMsgWindow);}

   If Odd(Event.modifiers) {then the window is becoming active}
   then
     begin
       SetPort(EventMsgWindow);
       {and activate whatever else you need}
     end
   else
     begin
       {deactivate whatever you need}
     end;
End;

{-------------------------------------------------------------------------------}

PROCEDURE DealwthUpdates(Event:EventRecord);
Var EventMsgWindow,
          TempPort: WindowPtr;
Begin
   EventMsgWindow := WindowPtr(Event.message);
   GetPort(TempPort);                {Save the current port}

   SetPort    (EventMsgWindow);      {set the port to one in Evt.msg}
   BeginUpDate(EventMsgWindow);
     EraseRect(EventMsgWindow^.portRect);
    { WhichPrinter;                                Proc to ID the printer}
     {DrawGrowIcon(EventMsgWindow);}
   EndUpDate  (EventMsgWindow);
   SetPort    (TempPort);             {restore to the previous port}
End;

{-------------------------------------------------------------------------------}

PROCEDURE MainEventLoop;
Var Event:EventRecord;
    ProcessIt: Boolean;
Begin
  Repeat
    SystemTask;             {so you can support Desk Accessories}

    ProcessIt := GetNextEvent(EveryEvent,Event);
    If ProcessIt{is true} then {we'll ProcessIt}
          Case Event.what of

            mouseDown   : DealwthMouseDowns(Event);
            KeyDown     : DealwthKeyDowns  (Event);
            ActivateEvt: DealwthActivates (Event);
            UpDateEvt   : DealwthUpdates   (Event);

          End;{of Case}
  Until Finished; {terminate the program}
```

```
End;

{-------------------------------------------------------------------------------}

PROCEDURE InitIcons;
{ Manually stuff some icons.  Normally we would read them from a file }
BEGIN
{each line contains 48 HEX #s which fill 12 consecutive words up to 96}

  { Lisa }
  StuffHex(@icons[0, 0],'00000000000000000000000000000000000001FFFFFFFC');
  StuffHex(@icons[0,12],'006000000006018000000000B0600000000130FFFFFFFFFA3');
  StuffHex(@icons[0,24],'18000000004311FFFFF00023120000080F231200000BF923');
  StuffHex(@icons[0,36],'120000080F23120000080023120000080023120000080F23');
  StuffHex(@icons[0,48],'1200000BF923120000080F23120000080023111FFFFF00023');
  StuffHex(@icons[0,60],'08000000004307FFFFFFFFA30100000000260FFFFFFFFE2C');
  StuffHex(@icons[0,72],'18000000013832AAAAA8A9F0655555515380C2AAAA82A580');
  StuffHex(@icons[0,84],'800000000980FFFFFFFFF30080000000001600FFFFFFFFC00');

  { Printer }
  StuffHex(@icons[1, 0],'00000000000000000000000000000000000000000000000000');
  StuffHex(@icons[1,12],'00000000000000007FFFFF0000008000028000011151444 0');
  StuffHex(@icons[1,24],'000200000840000445451040000400001 7C00004A5151000');
  StuffHex(@icons[1,36],'000400001000000 4A5451000000400017FE00F4A5151003');
  StuffHex(@icons[1,48],'018400000 13870327FFFFF10F06400000021B0CFFFFFFFC37');
  StuffHex(@icons[1,60],'18000000006B3000000000D77FFFFFFFFFABC00000000356');
  StuffHex(@icons[1,72],'8000000001AC87F000000158841000CCC1B087F000CCC160');
  StuffHex(@icons[1,84],'8000000001C0C000000003807FFFFFFFFF0007800001E000');

  { Trash Can }
  StuffHex(@icons[2, 0],'000001FC000000000E06000000003003000000000C0918000');
  StuffHex(@icons[2,12],'000138498000000026C4980000004C0930000000861260000');
  StuffHex(@icons[2,24],'00100 64FE000003119983000002 0E6301800002418E00800');
  StuffHex(@icons[2,36],'0033E3801C0000180E002C00000FF801CC0000047FFE0C00');
  StuffHex(@icons[2,48],'000500004C000005259A4C000005250A4C00000525FA4C00');
  StuffHex(@icons[2,60],'000524024C00000524924C006 0524924C0090E524924C7C');
  StuffHex(@icons[2,72],'932524924C82A44524924D01C88524924CF10C4524924C09');
  StuffHex(@icons[2,84],'0784249258E700030492331 00000E000E40800001FFFC3F0');

  { tray }
  StuffHex(@icons[3, 0],'00000000000000000000000000000000000000000000000000');
  StuffHex(@icons[3,12],'00000000000000000000000000000000000007FFFFFFF0');
  StuffHex(@icons[3,24],'000E00000018001A0000038003600000078006A000000D8');
  StuffHex(@icons[3,36],'00D7FFFFFFB801AC000003580358000006B807FC000FFD58');
  StuffHex(@icons[3,48],'040600180AB80403FFF00D58040000000AB80400000000D58');
  StuffHex(@icons[3,60],'040000000AB807FFFFFFFD5806AC00000AB8055800000D58');
  StuffHex(@icons[3,72],'06B000000AB807FC000FFD70040600180AE00403FFF00DC0');
  StuffHex(@icons[3,84],'040000000B80040000000F00040000000E0007FFFFFFFC00');

  { File Cabinet }
  StuffHex(@icons[4, 0],'0007FFFFFC00000800000C00001000001C00002000003400');
  StuffHex(@icons[4,12],'004000006C0000FFFFFFD40000800000AC0000BFFFFED400');
  StuffHex(@icons[4,24],'00A00002AC0000A07F02D40000A04102AC0000A07F02D400');
  StuffHex(@icons[4,36],'00A00002AC0000A08082D40000A0FF82AC0000A00002D400');
  StuffHex(@icons[4,48],'00A00002AC0000BFFFFED40000800000AC0000BFFFFED400');
  StuffHex(@icons[4,60],'00A00002AC0000A07F02D40000A04102AC0000A07F02D400');
  StuffHex(@icons[4,72],'00A00002AC0000A08082D40000A0FF82AC0000A00002D800');
  StuffHex(@icons[4,84],'00A00002B00000BFFFFEE00000800000C00000FFFFFF8000');

  { drawer }
  StuffHex(@icons[5, 0],'00000000000000000000000000000000000000000000000000');
  StuffHex(@icons[5,12],'00000000000000000000000000000000000000000000000000');
  StuffHex(@icons[5,24],'00000000000000000000000000000000000000000000000000');
  StuffHex(@icons[5,36],'00000000000000000000000000000000000001FFFFFF0');
  StuffHex(@icons[5,48],'000038000030000068000070000 0D80000D0003FFFFFF1B0');
  StuffHex(@icons[5,60],'00200000135000200000 16B000201FE01D50002010201AB0');
  StuffHex(@icons[5,72],'00201FE01560002000001AC0002000001580002020101B00');
  StuffHex(@icons[5,84],'00203FF01600002000001C0000200000180000 3FFFFF000');

END;

{-------------------------------------------------------------------------------}

PROCEDURE PutPicScrap;
Var err:     LongInt;
    PicRect: Rect;
    PicHdl:  PicHandle;
    PicLen:  LongInt;
Begin
  PicRect := DefaltPage;
  PicRect.bottom := PicRect.bottom Div 2;
```

```
  PicRect.right := PicRect.right Div 2;

  BuildQDPicture(theScreen);

  PicHdl := OpenPicture(PicRect);
  DrawPicture(QDPicture, PicRect);
  ClosePicture;
  PicLen := PicHdl^^.PicSize;

  HLock(Handle(PicHdl));
  err := ZeroScrap;
  err := PutScrap(PicLen, 'PICT', Pointer(PicHdl^));
  HUnLock(Handle(PicHdl));
  KillPicture(QDPicture);
  KillPicture(PicHdl);
End;

{-------------------------------------------------------------------------}

PROCEDURE InitThings;
Begin
  InitGraf(@thePort);             {create a graphport to the screen}
  InitFonts;                      {startup the fonts manager}
  InitWindows;                    {startup the window manager}
  InitMenus;                      {startup the menu manager}
  InitDialogs(Nil);               {startup the dialog manager}
  TEInit;                         {startup the text edit manager}
  InitCursor;                     {make the cursor an arrow}
  FlushEvents(everyEvent,0);      {clear events from previous program}
  InitIcons;                      {procedure to init 5 icons}

  Screen := ScreenBits.Bounds;    {set the size of the screen}
  Finished := False;              {set program terminator to false}
End;

{-------------------------------------------------------------------------}

PROCEDURE SetupLimits;
Begin
  SetRect(DragArea,Screen.left+4,Screen.top+24,Screen.right-4,Screen.bottom-4);
  SetRect(GrowArea,Screen.left,Screen.top+24,Screen.right,Screen.bottom);
End;

{-------------------------------------------------------------------------}

PROCEDURE SetupMenus;
Var
  MenuTopic: MenuHandle;
Begin
  MenuTopic := GetMenu(AppleMenu);  {get the apple desk accessories menu}
  AddResMenu(MenuTopic,'DRVR');     {adds all names into item list}
  InsertMenu(MenuTopic,0);          {put in list held by menu manager}

  MenuTopic := GetMenu(PrDlogMenu);
  InsertMenu(MenuTopic,0);

  MenuTopic := GetMenu(PrintMenu);
  InsertMenu(MenuTopic,0);

  MenuTopic := GetMenu(PrDrvrMenu);
  InsertMenu(MenuTopic,0);

  MenuTopic := GetMenu(FontMenu);
  AddResMenu(MenuTopic,'FONT');
  InsertMenu(MenuTopic,0);

  MenuTopic := GetMenu(StyleMenu);
  InsertMenu(MenuTopic,0);

  MenuTopic := GetMenu(PicScrMenu);
  InsertMenu(MenuTopic,0);

  DrawMenuBar;            {all done so show the menu bar}
End;

{-------------------------------------------------------------------------}

PROCEDURE SetupAWindow;
Type LowMPtr =   ^Integer;

Var  MenuHdl:     MenuHandle;
```

```
    DefaultFont: LowMPtr;
    NameHolder:  STR255;
    FoundIt:     Boolean;
    Item_No:     Integer;
    NumItems:    Integer;
    FontID:      Integer;

Begin
  aWindow := GetNewWindow(257, @WRec, Pointer(-1));

{setup a rect in the window for reporting errors}
  errRect := aWindow^.portRect;
  errRect.top := errRect.bottom - 20;

{check off the default font, LaserWriter will set default to Helvetica}
  DefaultFont := LowMPtr($0984);            {set low memory adderess}
  MenuHdl := GetMenu(FontMenu);             {menu handle for fonts}
  NumItems := CountMItems(MenuHdl);         {number of fonts in menu}
  FoundIt := False;
  Item_No := 1;
  Repeat
    GetItem(MenuHdl, Item_No, NameHolder);    {get new font name}
    GetFNum(NameHolder, FontID);              {get the font ID}
    If FontID = DefaultFont^ then
    begin
      PrevFontChked := Item_No;                 {save the new font No}
      CheckItem(MenuHdl, Item_No, True);        {check it off in menu}
      FoundIt := true;
    end;
    Item_No := Item_No + 1;
  Until (Item_No > NumItems) or FoundIt;

{check off the font style}
  MenuHdl := GetMenu(StyleMenu);             {menu handle for style}
  CheckItem(MenuHdl, 1, True);              {check the plain style}

{check off the size}
  CheckItem(MenuHdl, 10, True);              {check the 12 point}

{set the global guys}
  CurntFontID := FontID;    {the default font}
  CurntStyleID := [];       {plain}
  CurntSizeID := 12;        {size 12}

End;

{-------------------------------------------------------------------------------}

PROCEDURE SetupPrPort;
Var dummy: boolean;
Begin
  PrRecordHdl := THPrint(NewHandle(SizeOf(TPrint)));   {Make space for the record}
  PrOpen;                                              {open up ptr resource file}
  PrintDefault(PrRecordHdl);                           {fill rec w/default params}
  DefaltPage := PrRecordHdl^^.prInfoPT.rPage;          {default printer page size}
End;

{-------------------------------------------------------------------------------}

PROCEDURE SetUpThings;
Begin
  SetupLimits;
  SetupMenus;
  SetupPrPort;
  SetupAWindow;
End;

{-------------------------------------------------------------------------------}

BEGIN
  InitThings;
  SetUpThings;
  MainEventLoop;


{clean up, probably should be in a closing procedure}
  PrClose;
END.
```

Pr Mgr TestR

Saturday, January 23, 1904 2:06:10 AM

```
*  Input Resource File for Printing example
*

BPrint.Rsrc

Type PMGR = STR
  ,0
LASR  Version 1.0 July 31, 1984

Type BNDL
  ,128
  PMGR 0
  2
  ICN# 1
  0  128
  FREF 1
  0  128

Type FREF
  ,128
  APPL 0

Type ICN#
  ,128
  2
00000000
00000000
00000000
00000000
000FFFF8
00100010
0026FF20
00400040
009F6880
070001FF
097F7203
130003E5
2500024B
4FFFFF95
80000029
FFFFFFD1
80000061
FFFFFFC1
80000041
90000E45
8000004A
80000E50
80000050
8FFFFE50
88000260
F80003C0
00000000
00000000
00000000
00000000
00000000
00000000
*MASK FOLLOWS
00000000
00000000
00000000
00000000
000FFFF8
001FFFF0
003FFFE0
007FFFC0
00FFFF80
07FFFFFF
0FFFFFFF
1FFFFFFF
3FFFFFFF
7FFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFA
FFFFFFF0
FFFFFFF0
FFFFFFF0
F80003E0
F80003C0
00000000
00000000
00000000
00000000
00000000
00000000

* apple menu, 14 is apple symbol in HEX
Type MENU
  ,256
  \14

Type MENU
  ,260
  Print Dialogs
  Page Setup Dialog /S
  Print Dialog /P
  (-
  Quit /Q
```

```
* file menu
Type MENU
  ,257
  Print Mgr
  Frame Page
  Frame Text
  All QD Calls
  QD Picture
  Use TextBox
  Print Bitmap

*Print driver test menu items
Type MENU
  ,261
  Print Driver
  Bit Map
  Screen wEvt
  Screen wBits
  Stream Text

* make a scrap of the QDPicture
Type MENU
  ,262
  Copy QDPic
  Put in ClipBoard

* edit menu
Type MENU
  ,258
  Font

* Font menu
Type MENU
  ,259
  Style
  Plain
  Bold <B
  Italic <I
  Underline <U
  Outline <O
  Shadow <S
  (-
  9  Point
  10 Point
  12 Point
  14 Point
  18 Point
  24 Point

* text window definition
Type WIND
  ,257
  Printer Display
  40 40 330 400
  Visible GoAway
  0
  0

*
* Dialog to cancel or pause printing
* it is used as a modeless dialog
Type DLOG
  ,257
  130 125 180 375
  Visible 1 NoGoAway 0
  256

* Item list for the print cancel dialog
Type DITL
  ,256
  3
  BtnItem  Enabled
  15 10 35 80
Cancel

  BtnItem  Enabled
  15 90 35 160
Pause

  BtnItem  Enabled
  15 170 35 240
Continue

* this is the text string
Type STR
  ,256(4)
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
Type CODE
  BPrintL,0
```

Appendix I

Using MacTerminal to talk directly to the Postscript computer in LaserWriter.

Using MacTerminal to Talk Directly to the "PostScript Computer"

[This guide contains a short series of exercises for those who would like to interact with the PostScript Interpreter directly, using MacTerminal as an ascii terminal interface. Interested parties should read the previous document, "Apple LaserWriter Advanced User's Supplement" for in depth information on printer operation.]

1. Start out with the printer OFF, nothing in any of the connector ports and the selector switch on the back of the Laserwriter in the1200 baud position.

2. Cable the Macintosh to the LaserWriter using either a 9 pin to 9 pin cable or a 9 pin to 25 pin cable (a standard ImageWriter cable will work). Connect one end of the cable to the appropriate port on the printer (9 or 25 pin) and the other end to the Macintosh's 9 pin MODEM port.

3. Turn the printer ON. Several things will happen. First, the green light will blink. This is the printer's normal warm-up indicator. Next, the yellow light will blink. This is the printer's signal that data is being processed. In this case the printer is processing the start-up page.

4. Start up a disk with MacTerminal on it. For these examples you should also have MacWrite handy.

5. Check to see that the following settings are correct in the "Settings" menu:

Terminal:	VT100; ANSI; Underline; U.S.; 80 column; On-Line; Auto Repeat; Auto Wraparound
Compatablity:	Baud Rate = 1200; 8 bits, Parity = none; Handshake = none; connection = another computer; connection port = modem
File Transfer:	Transfer Method = text

6. Now press Control T (The Control key is the one next to the space bar that has a symbol on it that looks like a snowflake). This causes printer status to be displayed on the screen. It should display something that looks like:

%% [status: waiting; source serial 9] %%

7. Press Control D (this will stablize the printer). If you now press Control T again you should get

%%[status: idle]%%

8. At this point we are ready to 'talk' to PostScript. To get into Interactive PostScript mode type the word **executive** followed by a carriage return. (This will <u>not</u> echo on your terminal).

9. The following should print out:

```
PostScript (tm) Version 23.0
Copyright (c) 1984 Adobe Systems Incorporated
PS>
```

10. Hitting more carriage returns (**cr**'s) will get you more PostScript prompts' (**PS>**)

11. You are now 'talking' directly to the PostScript interpreter. For more information on the PostScript language see the PostScript Language Reference Manual.

12. Test to see if the connection is working by typing:

```
showpage  (cr)
```

    This should eject a blank peice of paper.

13. Now try printing something simple. Type the following characters followed by a carriage return in response to the PostScript prompt (**PS>**).

```
/Times-BoldItalic findfont
72 scalefont setfont
100  100 moveto
30 rotate
(Put your name here) show
showpage
```

    Note that you do not get a prompt back right away after the **show** statement. This is because the Postscript interpreter is busy creating a scaled and rotated font.

    This exercise should have created a page with your name printed at an angle.

14. To get into Batch mode type CONTROL D. Batch allows you to stream many lines of PostScript to the printer at once.This can be done from the Interactive mode as well. MacTerminal echos all lines of text back to the screen in Interactive mode. In non-interactive mode (batch) the file just gets shipped to the printer and executed.To test these modes out follow these steps:

15. Create a PostScript file using MacWrite as a text editor. Start-up MacWrite and enter the following code in. Make sure you save it using the **SAVE AS** function. MacWrite defaults to writing files out in MacWrite format, but make sure you change this to **TEXT** format before saving the file. (Identing the following code is not important.)

```
2 2 scale
/Times-BoldItalic find font 27 scalefont setfont
/rays
 {0  1.5  179
  {gsave
    rotate
     0  0 moveto 108  0 lineto
    stroke
    grestore
        } for
 } def
125   200 translate
.25  setlinewidth
newpath
0  0 moveto
(StarLines)   true
charpath clip
newpath
54  -15  translate
rays
showpage
```

16. Write the above text (in Text format) to your disk as "PSTEST".

17. Go back to MacTerminal (make sure your settings are correct, see 2a in this document). Under the File menu click **Send File.** A sub-menu screen will appear. Click on the file we have just created, PSTEST, and then click **Open.** This proceedure should send the entire file to the printer. If you are still in interactive mode you will see the ascii playback on the screen, if you "Control D'd" (batch mode) you will not. This file may take a little while to execute but, it's worth it. In about two or three minutes the page should print.

18. You can now experiment with the switch settings. Turn the printer Off and switch it to 9600 baud. Set the communication settings in MacTerminal to 9600 and try these exercises again.

19. Normal Macintosh applications that are supported by the LaserWriter Print Manager use the AppleTalk connector. This should **not** be used with any other communication link to the printer (ie, **don't** hook-up AppleTalk and RS232 at the same time). The AppleTalk cable hooks up to the 9 pin port on the printer and the PRINTER port on the back of the Mac. For further information on this connection refer to the appropriate documents on AppleTalk (For further information make sure you read "Apple LaserWriter Advanced User's Supplement")

Appendix J

Postscript File Structuring Conventions

POSTSCRIPT™
File Structuring Conventions

First Edition
January 1985

Adobe Systems Incorporated

Adobe Systems Incorporated
1870 Embarcadero Road, Suite 100
Palo Alto, California 94303

POSTSCRIPT File Structuring Conventions
9 January 1985
Copyright © 1985 Adobe Systems, Inc.
All Rights Reserved

POSTSCRIPT is a trademark of Adobe Systems, Inc.

Times and Helvetica® are trademarks of Allied Corporation.
UNIX is a trademark of AT&T Bell Laboratories.
Scribe is a registered trademark of UNILOGIC, Ltd.

The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems, Inc. Adobe Systems assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

The POSTSCRIPT language standard as described in the POSTSCRIPT Language Manual is a specification of the rules by which POSTSCRIPT operators and operands are combined into valid POSTSCRIPT programs. Those rules say nothing about the overall structure of a POSTSCRIPT program, or about how a POSTSCRIPT program can interact with the operating system, or how POSTSCRIPT files are actually handled by printers.

Since the language standard is silent about the overall structure of POSTSCRIPT files, the structure of a file is specified by conventions rather than rules. A POSTSCRIPT file that obeys the structuring conventions is called *conforming*; a POSTSCRIPT file that does not obey the structuring conventions is called *nonconforming*. The structuring conventions have no effect on the execution of a file or on the image produced when it executes: if a POSTSCRIPT interpreter is presented with a nonconforming file, the execution result will be exactly the same as an equivalent conforming file.

However, some operating systems might not be able to recognize that a file is a POSTSCRIPT program unless that file conforms to the structuring conventions. Furthermore, some of the processing that is done on POSTSCRIPT files, such as editing them, moving pages around, or combining small documents into large ones, is much easier to perform on conforming files.

POSTSCRIPT document structure is represented by means of "comments". The syntax of POSTSCRIPT comments is described in the POSTSCRIPT language manual in Section 2.5. Comments in a POSTSCRIPT file can contain any text at all; they are not processed by the interpreter. However, if those comments match certain patterns, they are said to follow the structuring *comment convention*. Various programs that operate on POSTSCRIPT files, other than the interpreter, look for comments that obey these conventions and use them to assist processing.

The kinds of processing that are facilitated with the comment conventions of a conforming document include:

- Managing downloaded fonts and facilitating the "closure" of documents.
- Selecting subsets of the pages of a document to be printed, or changing the order in which the pages will be printed.
- Enabling the proper positioning of other POSTSCRIPT programs (e.g., for illustrations to be incorporated by document preparation and composition systems).

Note that compliance with these conventions is not an all-or-nothing situation. Applications need not supply all of the entries described here. Simple applications on small processors may only be able to specify basic header elements, while larger applications might implement the complete

specification. A POSTSCRIPT file is called *minimally conforming* if it obeys the conventions flagged below with a dagger†.

The first line of every POSTSCRIPT file should begin with the characters "`%!`". This marks the file as a POSTSCRIPT file. Some operating systems such as UNIX have a scheme whereby the first 16 bits of a file are a "magic number" that identifies the file type to the operating system kernel. The "`%!`" serves also as a 16-bit "magic number" for these systems that operate this way.

The remainder of the first line of a conforming file is the version identifier, identifying the version number of the structuring convention that the file obeys. The version described in the document you are now reading is version "PS-Adobe-1.0". A file is taken to be minimally conforming if the version identifier begins with the characters "`PS-Adobe-`". In other words, a file is minimally conforming if its first 11 characters are "`%!PS-Adobe-`".

Following the magic number/version line, are some *header comments* which have meaning for the document as a whole. Each header comment is on a line by itself; it begins with the characters "`%%`" and ends with a newline character. A few of these header fields can be deferred to the end of the document, if an application program does not have the ability to generate them in the header. The header comments are:

`%%Title: document-title`
: The title of the document, POSTSCRIPT program, or file name.

`%%DocumentFonts: font1 font2 ...`
: where *font1*, *font2*, etc. are the POSTSCRIPT font names of fonts used by the document. A conforming file can also specify "`(atend)`" instead of the font list, indicating that the real `DocumentFonts` specification is in the last few lines of the file. A utility program might wish to verify that these fonts are resident on a specified printer, and/or download them ahead of this job.

`%%Creator: character-string`
: The person or program (or both) that created this POSTSCRIPT file. This may be different from the person printing the file (see the `For` comment below).

`%%CreationDate: character-string`
: The date and time this POSTSCRIPT file was created. The date string may have any form.

`%%Pages: ##`
: The number of pages present in this document. If the number is not known, a question mark (?) or a blank

field is specified.[1] If the document generates *no* pages, but, for example, is meant to be an included illustration, the number should be zero (0). The specification "`(atend)`" is allowed.

`%%BoundingBox: llx lly urx ury`
The bounding box (in the default POSTSCRIPT coordinate system) of the printed marks specified by this file. This may be used by composition programs for placing included illustrations. If the file is a multipage document, these numbers are of less utility, but should be unioned over all pages. Here again, the specification "`(atend)`" is allowed.

`%%For: character-string`
The intended recipient of this document. If no `For` field is present, the printer software will normally assume that the file is for its `Creator`.

For those fields specified as "`(atend)`", the true values must be given in the last 5 nonblank lines of the conforming POSTSCRIPT file. Note that utility programs may need to add, change or duplicate the information in the header comments. They should do so by *prepending* information to the previous header. The first occurrence of a header comment item is used; all but the first are ignored. The comment-header section ends at the first occurrence of a line that does not begin with "*%!*" or "*%%*" or by a line with the comment `%%EndComments`.

The conforming comments after the comment-header section are used to signal the boundaries of the various parts of a POSTSCRIPT print file. They are used by utilities that wish to reverse the page order (for collated stacking) or to collect and print a subset of the pages of a document.

`%%EndProlog` signals the end of the prolog section of the document. (In reversing the pages of a document, the prolog should still come first.)

`%%Page: label cardinal`
signals the beginning of a page "body"; where *label* is the page number in the document's internal numbering scheme (e.g., vii, 10-34, etc.) and *cardinal* is the absolute page number in terms of pages printed (from 1 through N for an N-page document). If the number in either scheme is not known, a question mark (?)

[1]Note that static analysis of an arbitrary POSTSCRIPT file (e.g., counting the occurrences of **showpage**) is not sufficient to determine how many pages it will print.

takes its place. A utility program that collects and prints selected pages may take page number specifications in either form; e.g., "print pages vii and ix" or "print the first ten pages and the last ten pages".

`%%PageFonts: font1 font2 ...`
can come directly after the `Page` marker. Like `DocumentFonts`, `PageFonts` lists the fonts needed by a particular page body. In large or complex documents that have a high probability of being segmented, the `PageFonts` entry provides a finer degree of detail for utility programs to use. Pages without a `PageFonts` entry are assumed to need all of the fonts listed in `DocumentFonts`.

`%%Trailer` signals the beginning of the trailer section of the document, after the end of the last page body. The trailer continues to the end-of-file. This section should always come last, no matter what page re-ordering takes place. Trailer sections may do final cleanup of a document's state (e.g., a `restore`).

A short, skeletal example of a POSTSCRIPT file structured in the above manner follows.

```
%!PS-Adobe-1.0
%%Creator: Anthony Abstract
%%Title: Tropic of Calculus
%%CreationDate: Fri Aug  9 11:33:03 1974
%%Pages: (atend)
%%DocumentFonts: Times-Roman Times-Italic Times-Bold
%%Dimensions: 0 0 612 792
%%EndComments
... document prolog goes here ...
%%EndProlog
%%Page: 0 1
... this might be the title page ...
%%Page: 1 2
... the first text page of the document ...
%%Page: 2 3
... the last page of the document ...
%%Trailer
... document trailer goes here ...
%%Pages: 3
```