

Rochester Institute of Technology

**RIT Scholar Works**

---

Theses

---

1988

## Performance analysis of text-oriented printing using PostScript

Thomas Kowalczyk

Follow this and additional works at: <https://scholarworks.rit.edu/theses>

---

### Recommended Citation

Kowalczyk, Thomas, "Performance analysis of text-oriented printing using PostScript" (1988). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact [ritscholarworks@rit.edu](mailto:ritscholarworks@rit.edu).

Rochester Institute of Technology  
School of Computer Science and Technology

Performance Analysis of  
Text-Oriented Printing  
Using POSTSCRIPT®

by  
Thomas L. Kowalczyk

A thesis, submitted to  
The Faculty of the School of Computer Science and Technology,  
in partial fulfillment of the requirements for the degree of  
Master of Science in Computer Science

Approved by:

Guy Johnson

---

Professor and Chairman of  
Applied Computer Studies

Dr. Vishwas G. Abhyankar

---

instructor

Frank R. Hubbell

---

instructor

October 10, 1988

Thesis Title:

Performance Analysis of  
Text-Oriented Printing  
Using POSTSCRIPT®

I, Thomas L. Kowalczyk, hereby grant permission to the Wallace Memorial Library, of R.I.T., to reproduce my thesis in whole or in part under the following conditions:

1. I am contacted each time a reproduction is made. I can be reached at the following address:

82 Brush Hollow Road  
Rochester, New York 14626

Phone # (716) 225-8569

2. Any reproduction will not be for commercial use or profit.

Date: October 10, 1988

Apple, Appletalk, LaserWriter, and Macintosh are registered trademarks of Apple Computer, Inc.

Fluent Laser Fonts and Galileo Roman are trademarks of CasadyWare Inc.

Helvetica, Palatino, and Times are registered trademarks of Linotype Co.

ITC Avant Garde, ITC Bookman, ITC Garamond, ITC Zapf Chancery, and ITC Zapf Dingbats are registered trademarks of International Typeface Corporation

POSTSCRIPT is a registered trademark of Adobe Systems Incorporated.

Interpress is a registered trademark of Xerox Corp.

## Abstract

POSTSCRIPT® is a page description language which is used to transmit printing information from a host computer (i.e. Apple Macintosh) to a printer (i.e. Apple LaserWriter Plus). It has the ability to describe pages consisting of text, vector graphics, and scanned bit-map images. Printing text is the area of concentration for this thesis. Specifically several variables that affect the printing speed of a common POSTSCRIPT® printer, the Apple LaserWriter Plus, are looked at when printing text in a variety of fonts, sizes, and orientations. The variables that affect printer performance include:

- use of outline vs. bit-map fonts;
- the outline font rasterization process;
- the use of pre-cached bit-map fonts;
- background outline font rasterization;
- arbitrary scaling and rotation;
- downloading host-resident fonts;
- Adobe and Third Party host-resident downloadable fonts vs. printer-resident fonts;
- Appletalk vs. RS-232 communications interfaces;
- use of the POSTSCRIPT® show, **ashow**, and **widthshow** instructions;
- targeting the POSTSCRIPT® instructions at a particular engine resolution;
- print engine overhead.

A sequence of POSTSCRIPT® files were transmitted to the Apple LaserWriter Plus printer. The experiments were carefully constructed to exercise each of the variables listed above. Performance measurements were carefully recorded and analyzed. Where applicable, improvements were proposed to improve printer performance.

# Table of Contents

	Page
<b>Permission Statement</b>	<b>i</b>
<b>Trademarks</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Table of Contents</b>	<b>iv</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Background	<b>1</b>
1.2 "PDL" Defined	<b>1</b>
1.3 History of Interpress® and POSTSCRIPT®	<b>2</b>
1.4 A Comparison of Outline vs. Bitmap Fonts	<b>4</b>
1.5 Text, Vector Graphics, and Bitmap Images	<b>8</b>
1.5.1 Text	<b>8</b>
1.5.2 Bitmap Images	<b>9</b>
1.5.3 Vector Graphics	<b>10</b>
<b>2. Problem Description</b>	<b>13</b>
2.1 Variables that Affect Performance	<b>14</b>
<i>problem description and expected results</i>	
2.1.1 Font Formats: Outlines vs. Bitmaps	<b>14</b>
2.1.1.1 Outline Font Rasterization and Caching	<b>15</b>
2.1.1.2 Pre-Cached Bit-Maps	<b>15</b>
2.1.1.3 Background Outline Font Rasterization	<b>16</b>
2.1.1.4 Effect of Scaling and Rotation	<b>17</b>
2.1.1.5 Variations of Font Style	<b>19</b>

	<b>Page</b>
2.1.2 Downloaded Fonts	20
2.1.2.1 Downloaded Bitmaps vs. Printer Resident (Cached) Bitmaps	21
2.1.2.2 Adobe Downloaded Outlines vs. Printer Resident (Internal) Outlines	22
2.1.2.3 Adobe Downloaded Outlines vs. Third Party Downloaded Outlines	23
2.1.2.4 Appletalk vs. RS-232	25
2.1.3 Methods of Printing Strings	26
2.1.4 Resolution "Targeting" vs. Total Resolution Independence	27
2.1.5 Effect of Varied Page Complexity on POSTSCRIPT® Processing Time	28
2.1.5.1 Effect on the Inter-Page Time Delay Program sec	28
2.1.5.2 Effect on the Compilation and Rasterization of Text Pages	29
2.2 Performance Measurement and Calculation	30
2.2.1 Communications	31
2.2.2 Interpretation (Compilation) and Rasterization	35
2.2.3 Print Engine Paper Ejection	44
2.3 Programming Tasks	45
2.3.1 Extraction of Font Widths from Apple Laser Writer Plus®	45
2.3.2 Batch Composition Program	47
2.3.3 Downloadable POSTSCRIPT® Bitmap Font Program	49
2.3.4 Installing Downloadable POSTSCRIPT® Outline Fonts	54
2.4 Assumed RIP Architectural Model	55

	<b>Page</b>
<b>3. Analysis of Experimental Results</b>	<b>59</b>
3.1 Font Formats: Outlines vs. Bitmaps	<b>59</b>
3.1.1 Real Time Outline Font Rasterization and Caching	<b>59</b>
3.1.1.A Experimental Results	
3.1.1.B Analysis	
3.1.1.C Proposed Improvements	
3.1.2 Pre-Cached Bit-Maps	<b>65</b>
3.1.2.A Experimental Results	
3.1.2.B Analysis	
3.1.2.C Proposed Improvements	
3.1.3 Background Outline Font Rasterization	<b>67</b>
3.1.3.A Experimental Results	
3.1.3.B Analysis	
3.1.3.C Proposed Improvements	
3.1.4 Effect of Scaling and Rotation	<b>71</b>
3.1.4.A Experimental Results	
3.1.4.B Analysis	
3.1.4.C Proposed Improvements	
3.1.5 Variations of Font Style	<b>78</b>
3.1.5.A Experimental Results	
3.1.5.B Analysis	
3.1.5.C Proposed Improvements	
3.2 Downloaded Fonts	<b>81</b>
3.2.1 Downloaded Bitmaps vs. Printer Resident (Cached) Bitmaps	<b>81</b>
3.2.1.A Experimental Results	
3.2.1.B Analysis	
3.2.1.C Proposed Improvements	
3.2.2 Adobe Downloaded Outlines vs. Printer Resident (Internal) Outlines	<b>86</b>
3.2.2.A Experimental Results	
3.2.2.B Analysis	
3.2.2.C Proposed Improvements	

	<b>Page</b>
3.2.3 Adobe Downloaded Outlines vs. Third Party Downloaded Outlines	<b>91</b>
3.2.3.A Experimental Results	
3.2.3.B Analysis	
3.2.3.C Proposed Improvements	
3.2.4 Appletalk vs. RS-232	<b>97</b>
3.2.4.A Experimental Results	
3.2.4.B Analysis	
3.2.4.C Proposed Improvements	
3.3 Methods of Printing Strings	<b>102</b>
3.3.A Experimental Results	
3.3.B Analysis	
3.3.C Proposed Improvements	
3.4 Resolution "Targeting" vs. Total Resolution Independence	<b>111</b>
3.4.A Experimental Results	
3.4.B Analysis	
3.4.C Proposed Improvements	
3.5 Effect of Varied Page Complexity on POSTSCRIPT® Processing Time	<b>115</b>
3.5.1 Effect on the Inter-Page Time Delay Program sec	<b>115</b>
3.8.1.A Experimental Results	
3.8.1.B Analysis	
3.8.1.C Proposed Improvements	
3.5.2 Effect on the Compilation and Rasterization of Text Pages	<b>118</b>
3.8.2.A Experimental Results	
3.8.2.B Analysis	
3.8.2.C Proposed Improvements	
<b>4. Conclusions</b>	<b>123</b>
4.1 Scan Converting Outline Fonts vs. Caching Bitmap Fonts	<b>123</b>
4.2 Scan Converting Font Outlines with "Quality Hints" Applied	<b>124</b>
4.3 Font Style Complexity	<b>124</b>
4.4 Downloading Fonts	<b>125</b>
4.5 Methods of Printing Strings	<b>125</b>
4.6 Resolution Targeting	<b>126</b>

4.7 Printer Induced Delays	126
5. Glossary	127
6. References	131
7. Bibliography	133

**Appendices**

**1. Equipment and Software Tools**

1.1 POSTSCRIPT® Evaluation Equipment Configuration

**2. Support Programs**

2.1 POSTSCRIPT® Font Widths

2.1.1 Font Extraction Program

2.1.2 Printed Font Width Tables from the Apple Laser Writer Plus ®

2.1.3 Times-Roman Font Width Table (ASCII File)

2.2 Generation of POSTSCRIPT® Text Pages

2.2.1 Batch Composition Pseudo-Code

2.2.2 Input Data "Content" File

2.2.3 Input Markup "Style" File

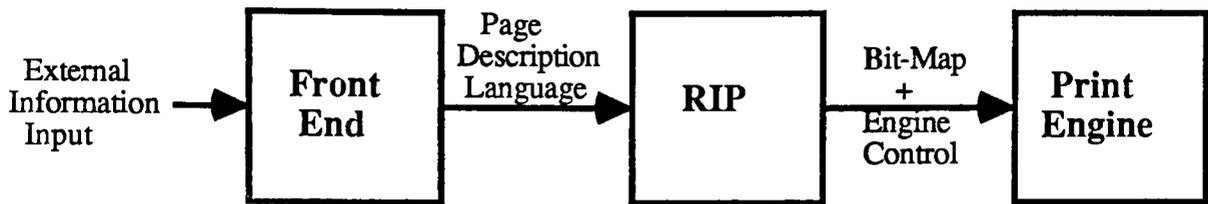
2.2.4 Text POSTSCRIPT® File

2.2.5 Printed Text and Timing Pages from the Apple Laser Writer Plus ®

# 1. Introduction

## 1.1 Background

An electronic publishing system is made up of three main parts: the front-end, the RIP, and the print engine. Figure 1.1 is a block diagram of a typical electronic publishing system.



## Electronic Publishing System Block Diagram

figure 1.1

The front-end provides interfaces to the user and to other computer systems. The user interface handles input of text, graphics, and bit-map (scanned) images, and provides the merging and page layout of these pieces in either a batch or interactive (WYSIWYG) mode. The computer interface provides a communications interface with other information input systems (ie. word processors, illustration stations). This information is filtered by the front-end to yield the internal page format of the electronic publishing system. Ultimately, the output of the front-end is the "page description language" (PDL) file which is sent to the "raster image processor" (RIP). The PDL file describes the marks on the printed page.

The Raster Image Processor (RIP) inputs a PDL file for a page or a set of pages generated by the front-end. The RIP then generates the corresponding bit-map image(s) and sends it to the print engine, usually in a raster form. The print engine prints the bit-map page.

The Page Description Language (PDL) is the interface between the "soft image" world of the front-end and the "hard-copy" world of the RIP and the print engine. One particular PDL is quickly becoming an industry defacto standard: POSTSCRIPT®.

## 1.2 "PDL" Defined

PDL stands for Page Description Language. The term PDL applies to the set of printing commands that describe the printing of text and usually vector graphics on a printed page. The location on a page is specified and a "mark" is placed at that location. That mark can be a dot, a character, a string of characters, a line segment, a curve segment, or a bit-map. The PDL has no knowledge of a "line of text" or a "carriage return". Subsequently, conventional line printers are not considered to use PDL's. Determining where the line of text begins and ends is the responsibility of the front end application. A typical sequence of PDL (pseudo)code to print out a simple string of text goes as follows:

Move cursor to the (x,y) coordinate of the page.

Select a font, size, and orientation; make it the "current" font.

Print the string "abc"; letters are automatically advanced using proportional spacing.

Some of the more recent PDL's, like POSTSCRIPT®, offer fully scalable and rotatable outline fonts. The older PDL's, like IMPRESS, provide a limited number of discrete font sizes and the four quadrant orientations. The PDL is the "assembly language" of electronic printing.

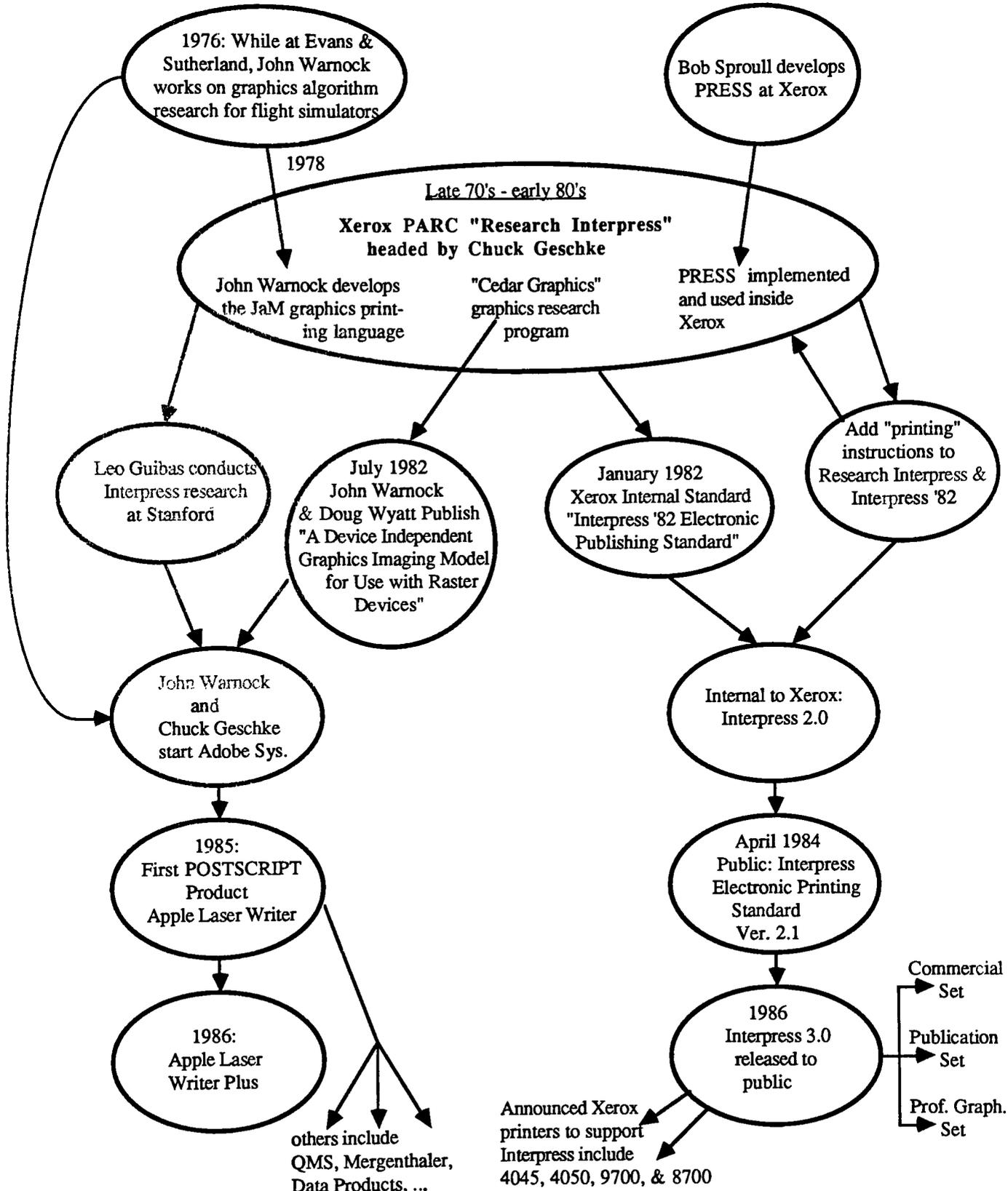
### 1.3 History of Interpress® and POSTSCRIPT® (1)

Figure 1.3 shows how Interpress® and POSTSCRIPT® began. Bob Sproull first developed a printing language called "PRESS" while at Xerox. Independently, John Warnock developed his own graphics/printing language. After coming to Xerox John Warnock put together a similar printing language he called JaM. A third concurrent activity at Xerox Palo Alto Research Center (PARC) was the "Cedar Graphics" graphics research program. "Research Interpress" was based on all three of these activities.

Xerox allowed some of their work to become public. Xerox permitted publication of a paper based on the work conducted in the "Cedar Graphics" research program. It was written by John Warnock and Doug Wyatt and was called "A Device Independent Graphics Imaging Model for Use with Raster Devices". Secondly, Xerox allowed Leo Guibas, a Stanford professor, to use some of the Interpress® -related research work as course material for a course at Stanford. John Warnock used the public information disclosed by Xerox in addition to the work that he did prior to coming to Xerox as a basis for starting his own company, Adobe Systems, Inc., and continued the development of a language which he called POSTSCRIPT®. The Apple Laser Writer was the first commercial product to use POSTSCRIPT®. The definition of POSTSCRIPT® is public. Adobe's product is their implementation of POSTSCRIPT®, which they will port to a hardware RIP that either Adobe or Adobe's customer designs.

On the Interpress® side, a subset of Research Interpress was chosen which became the Xerox internal standard language that was called the "Interpress '82™ Electronic Publishing Standard". Printing instructions were added to Interpress '82 to become another Xerox internal version of Interpress®, version 2.0. Xerox made version 2.1 public in April of 1984. In 1986, Xerox released Interpress® version 3.0 which contained three sets. The "Commercial Set" handles basic text and trivial graphics; the "Publication Set" adds bit-map fonts, vector graphics, and bit-map printing; the "Professional Graphics Set" provides high functionality, like scaled and rotated outline fonts and bit-map image processing. The "Publication Set" is a medium functionality language similar to "imPress", a language developed and sold by Imagen Corp. The "Professional Graphics Set" is a high functionality page description language similar to POSTSCRIPT®.

Much of this information was taken from reference #1.  
(Reference list may be found in section 6.)



## History of Interpress and POSTSCRIPT®

figure 1.3

## 1.4 A Comparison of Outline vs. Bitmap Fonts

Before POSTSCRIPT® was introduced most printers could only use bitmap fonts. One of the key capabilities of POSTSCRIPT® that certainly set Adobe Systems, Inc. ahead of the competition was the extensive use of outline fonts. This is in addition to the less used bitmap font functionality. Outline fonts have many advantages over bitmaps. Four of the more important advantages are outlined below.

### 1. Resolution Independence

Since a font outline is not tied to a specific resolution output device, the same outline can easily be used to drive any resolution output device.

For bitmap fonts, entirely different bitmaps are required for printers with different resolutions. For example, a 300 dpi font will be 75 % size when printed on a 400 dpi printer, or 150 % size on a 200 dpi printer.

### 2. Device Independence

Different printers have different processes that lay black marks on a page. In some cases even if two printers of the same resolution print the exact same bitmap (font) data, the output can appear significantly different. In the case of electrophotographic printers a light (laser or LED) exposes an area of electrophotosensitive material. The process is set up such that the exposed area either picks up toner, in a "write black" system, or does not pick up toner, in a "write white" system. If the same bitmap is printed on two printers with different polarity processes the print produced with "write white" tends to be lighter than the print produced with "write black". Inconsistencies across printers can be compensated for in the outline font rendering algorithm, so that the same basic outline can be used to print on different process printers.

In contrast, to make bitmap fonts look the same on printers with very different processes, different font bitmaps are needed.

### 3. Better selection of high quality rotated and scaled fonts

A single font outline can be scaled to any size and rotated to any angle. In addition it can be "obliqued" or tilted or reflected in either or both axes. In short, any outline font can be transformed in a way that can be described by any linear transformation.

By comparison bitmap fonts are applicable for only one size and orientation (and resolution and process). A 12 point font contains an entirely different set of bitmaps than a 10 point font of the same font style. The next section (4) discusses what this means relative to font storage space. Another method of printing bitmaps is to use one or several bitmaps and scale from the nearest bitmap font available. Although this method saves storage space and communications download time the quality of the font is severely degraded.

### 4. More memory space efficient.

An outline font is specified at one size, typically on a 1,000 by 1,000 unit grid for POSTSCRIPT®. The outline description for each character of the font is specified relative to discrete points on this grid. These outlines are made up of straight line segments and bezier curve segments. Characters in this font are then fully scalable and rotatable for any size, resolution, and orientation. An outline font takes approximately 30 KBytes of memory space. For example the size of the Adobe POSTSCRIPT® *Stone Serif* downloadable font is 34,260 bytes (\*) when stored on the host Mac II computer. Once the font is downloaded it is packed inside the Apple LaserWriter Plus® in a slightly more compact manner and occupies 28 KBytes of RAM \*\*.

\* measured with the *Get Info* utility of the Apple Macintosh® *Finder*

\*\* measured with the *Printer Font Directory* function of the *Font Downloader* program

By contrast a bitmap font only describes a typeface rendered at a single size, resolution, and orientation. If a 10 point and a 12 point font are needed (say at a given resolution and orientation) then two different files must be created and stored for the best quality. One size can be scaled from the other (see #3 above), but this yields very poor results.

The size of a bitmap font varies as the size of the font and the resolution varies. For example take a 12 point font with 96 printable characters at a 300 dot per inch resolution. The largest possible bounding box of a 12 point font is 12 points wide by 12 point high. This can be calculated to be 50 dots:

$$\frac{12 \text{ points}}{72 \text{ points per inch}} * 300 \text{ dots per inch} = 50 \text{ dots}$$

So the 12 point by 12 point box when translated into dots is a 50 dot by 50 dot character bounding box. This represents 2500 dots, each of which can be represented by one bit, assuming the dot is bitonal (i.e. Black or White). 2500 bits can be stored as 313 bytes per character. On average the characters only occupy about one third of this space, so it takes approximately 104 bytes to store each character's bitmap alone. For 96 characters this turns out to be about 10 Kbytes to store the bitmaps of a full 96 character set. Additional font information which can take several KBytes also needs to be included along with the character bitmaps. An example of this kind of information is the character metrics of a font (i.e. width, height, kerning info). If this information takes 4 KBytes, then the hypothetical font described above will need 14 KBytes to store a single size of one font. There are other factors which influence the size of bitmap fonts. Depending on the combination of the size and resolution of a bitmap font, the effect of scaling a font will have varied results. Three (size-resolution) "regions" are looked at in more detail below:

#### Low (Size-Resolution) "Region"

This region can be active under three conditions:

1. small to medium point sizes are selected (e.g. 10, 12, 14);
2. (screen) resolutions of approximately 75 dots per inch are used;
3. the method of storing the bitmaps uses "Byte" or "Word" padding.

At these size-resolution combinations the average characters all fit within one byte (of width). The 12 point bounding box is approximately 12 dots by 12 dots at this resolution. If the width of the average character is one half of this, the character is 6 dots wide. With "byte padding" the bitmap of this character would be stored as a column of bytes with the rightmost two bits of each byte padded with zeroes. Using the same assumptions a 10 point font would have an average character width of approximately 5 dots; at 14 point, 7 dots. In this region, as fonts are scaled the size of storage needed scales linearly, not geometrically. This is because only the height of the font changes (more rows) but the width does not (most characters fit in the one byte).

Note that even though the bitmaps scale linearly in this region, the space required to store the constant metric information often outweighs the amount of storage space needed for the variable bitmap data.

No specific example is provided to demonstrate this effect.

## Medium (Size-Resolution) "Region"

This region can be active under either of the following two conditions:

1. No byte padding (i.e. one character's bitmap runs into the next);

*or*

2. Byte padding is used with

- small to medium point sizes (e.g. 10, 12, 14);
- (printer) resolutions of approximately 300 dots per inch.

In the first case, if no byte padding is used both the width and height change as the font size changes. The size therefore scales quadratically (as the square).

In the second case, if the size-resolution combination is large enough so the overhead incurred with byte padding becomes negligible then, once again, the font size scales quadratically. As discussed earlier in this section the 12 point bounding box is 50 dots by 50 dots at this resolution, so the average character is about 25 dots wide. With this size-resolution combination a much smaller percentage of the information is byte padding, so as the size of a font increases the space needed to store it increases in two directions. The formula that better explains this phenomena is:

$$y = x^2 + b$$

Note that the space required to store the constant metric information, specified as **b** above, is quite substantial.

Macintosh® bitmap fonts are stored in memory with each character butting up to the next character all having a constant height. Byte padding is not used. To show that the (variable) bitmap portion of the screen font scales geometrically with font size, the 12 point and 24 point Times Roman® screen fonts were examined using ResEDIT (a Macintosh® resource editor).

<u>Point Size</u>	<u>Total Size*</u>	<u>Bitmap Font Size**</u>	<u>Metrics, etc.**</u>
12	7,570 bytes	1,752 bytes	5,818 bytes
24	12,730 bytes	6,912 bytes	5,818 bytes

\* Measured with the Font/DA Mover utility program.

\*\* Measured with ResEDIT.

Macintosh® Times Roman® Screen Font

**Figure 1.4.1**

$$b = 5,818$$

$$x = (\text{point size} / 12) * \sqrt{1,752} = 41.9 \text{ for the Times Roman® screen font}$$

It is clearly shown in the figure 1.4.1 that the Bitmap Font Size scales quadratically with the point size. The size of the bitmap portion of the 24 point font is almost exactly four times that of the 12 point font (within 1.4% error due to rounding). See references 29 and 30 for more information on the Macintosh® sreen font formats.

## High (Size-Resolution) "Region"

As the (size \* resolution) product gets much larger the basic effect on the bitmap storage needed continues to be rise exponentially with the size of the font. However, when the bitmaps get much larger, they are generally compressed. A typical compression scheme is one called "run-length encoding". The simple one dimensional version of this compression scheme simply stores the number of white and black pixel runs for each scan line of font bitmap data. More sophisticated schemes allow compression in two dimensions. The obvious advantage to using data compression is that it reduces the amount of data storage space needed. The disadvantage is that more time (and or hardware assist) is needed to process the data on both ends: compression and de-compression.

"Compressed characters consume much less space .. than full pixel arrays (by factors of up to 40), but require more computation to reconstitute when you need them. Reconstituting a compressed character is still substantially faster than re-executing the original character description." In the Apple LaserWriter IINTX® bitmaps of fonts that are larger than 20 points in size are stored in compressed format. (*all Ref. 2*)

## 1.5 Text, Vector Graphics, and Bitmap Images

The POSTSCRIPT® page description language has instructions that allows the user to draw text, vector graphics, and (scanned) bitmap images on a page. This thesis concentrates on text and does not address the vector graphics and bitmap imaging areas in any detail. However, many real-life applications, like electronic publishing, regularly include these types of images. POSTSCRIPT® is very rich in vector graphics and bit-map imaging functions. It would be misleading to not even mention them in a paper such as this one. Therefore, to present a more complete picture of POSTSCRIPT®, all three will be discussed briefly here.

### 1.5.1 Text

Text is generally the simplest, most efficient, and most used "mark" that is put on the page. Once a font is defined with a string like `/Times-Roman findfont 12 scalefont setfont` most of the POSTSCRIPT® program consists of the actual strings of characters. In simple line printers the only overhead is the carriage return - line feed characters delimiting lines of text. This can be two ACSII characters per each 100 character (or so) line of text, or 2 % overhead. With POSTSCRIPT® the overhead can vary widely, due to POSTSCRIPT®'s diverse functionality. But, for a simple line printer emulation, the overhead of textual POSTSCRIPT® can be made with a few simple assumptions:

1. a single positioning command, like `72 720 moveto`, will be used per line of text;
2. the strings of text will be encapsulated in parentheses, and followed by the show command;  
`( this is a string of text ) show`
3. 100 characters per average line;
4. the number of lines are sufficiently large such that the overhead incurred by the instructions selecting the font can be ignored;
5. additionally one carriage return - line feed pair per line is used.

With these assumptions there are 22 characters per 100 characters of output text, or 22% overhead. A very dense page of text, containing 4,000 characters, can be described by approximately 4,900 characters of POSTSCRIPT® code.

Since POSTSCRIPT® is a full programming language, a simple program that emulates the carriage return - line feed function can be defined in the beginning of the file and accessed by calling that function with a single character (i.e. the name of the function) after each string of text is specified. Using this method the overhead can approach 3 %, ignoring initialization overhead, and assuming one matching pair of parentheses and one character (`c`, below) which calls the carriage return - line feed function:

initialization includes:

- specifying the font;
- defining the `c` (carriage return - line feed) procedure;
- positioning the first line of text.

`(first string of characters )c(second string of characters )c(third string of char )`

Generally, as the complexity of the text increases, though, so does the overhead of describing that text. Much more is said about text throughout this thesis.

## 1.5.2 Bitmap Images

On the other end of the spectrum of efficiency and speed are bitmap images. If the entire page were defined by a bitmap image that matches the resolution of the Apple Laser Writer Plus print engine (300 dots per inch), it would take over 8 Mbits or 1 MByte of binary data. That's 90,000 bits per square inch! The overhead of POSTSCRIPT® for images is much more severe than for text. A single binary byte of image data is described by two ASCII characters that specify two hexadecimal numbers. So 1 MByte of binary data is described by 2 MBytes of ASCII-encoded hex in POSTSCRIPT®, or 100% overhead. Fortunately one does not necessarily need to specify that much data to fill the entire page with a bitmap image. This is because POSTSCRIPT®:

- is resolution independent. The input image does not have to match the output 300 dpi resolution. The POSTSCRIPT® **image** operator automatically adjusts the input to output mapping for printers with two different resolutions (say 300 dpi and 600 dpi). The input can be presented to the RIP in any resolution.
- allows scaling of all images (whether they are text, graphics, or bitmaps). An image scanned from a 1 inch by 1 inch image can be scaled to any size on the output. In fact any linear transformation can be specified which can:
  - scale in x and y by the same amount (1 inch by 1 inch scales to 7 inches by 7 inches);
  - anamorphically scale in x and y (1 inch by 1 inch scales to 3 inches by 7 inches);
  - "tilt" the image (rectangular image becomes an image fitting a parallelogram);
  - rotate the image.

Images can also be scanned as "contone" images (i.e. more than 1 bit per pixel; typically 8 bits per pixel for monochrome). Since the output image cannot print grey dots matching the grey areas of input image, a mapping of grey values to black and white printable dots must be made. This mapping is called *screening*. Variables that can be set with POSTSCRIPT® are:

- frequency \* of the screen. This is the number of "halftone dots", or "super-pixels", are printed per inch on the printer. Each of these halftone dots represent a grey value (or set of grey values) from the input contone image.
- angle \* of the screen. This specifies the number of degrees by which the halftone screen is to be rotated with respect to the device coordinate system.
- spot function \* of the halftone dot. This specifies how the halftone dot is to be "grown" as the input grey area changes from white (no black pixels turned on) to black (all black pixels turned on). Specifically, a procedure is specified which describes the order in which pixels within a halftone cell are whitened to produce any desired shade of grey. (*setscreen, reference 3*)
- input to output transfer function. A procedure can be specified which maps the input grey values to the printer grey values, which is a fraction of all pixels that are to be whitened. (*settransfer, reference 4*)

The two greatest performance bottlenecks when dealing with images with POSTSCRIPT® are:

1. accessing the image. Images tend to be large. If a communications line is used, it simply takes a long time to download the image. POSTSCRIPT®'s method of encoding the image with ASCII-encoded hex, described on the previous page, slows down this process by a factor of two. For example a 256 KByte binary image, which when encoded this way becomes 512 KBytes, takes almost 9 minutes to download using a conventional RS-232 serial interface line running at 9600 baud. Even when the image is resident inside the RIP, say on a disk, once again due to the size of the image it still takes a relatively long time to retrieve the image from disk, and to use the disk as a backup scratch pad when processing the image.

This bottleneck can be diminished by using faster communications interfaces, faster disks, and more RAM. Of course, along with these RIP enhancements usually comes a higher cost.

2. processing the image. The previous page summarizes the image processing functions that can be applied to bitmap images. As in all of POSTSCRIPT®, the functions vary from very simple to very complex. Applying a particular image processing algorithm to an image takes an amount of time that is loosely proportional to the "complexity" of the image processing function selected and, you guessed it, the size of the image.

This bottleneck can be reduced by:

- once again, more RAM and/or faster disc;
- faster general purpose CPU;
- computer architecture that better suits the image processing algorithms that are commonly used. This can include special purpose hardware.

### 1.5.3 Vector Graphics

Somewhere in between text and bitmap images lies vector graphics in speed and efficiency. A single straight line that traverses the height and width of a page can be described in just a few POSTSCRIPT® instructions:

```
0 0 moveto           % set cursor to lower left corner of page
612 792 lineto       % draw line to upper right corner, coordinates are in
stroke              % "points" or 1/72 inch
```

Now the Raster Image Processor needs to fill all of the pixels in page memory that correspond to the pixels that make up the black line. Other settings that effect the rendering of (connected) lines are:

- width of the line, settable with the `setlinewidth` operator;
- type of line join, which can be miter, round, or bevel; settable with the `setlinejoin` operator;
- type of line cap, which can be butt, round, or projected square; settable with the `setlinecap` operator.

Instead of the simple sequence of instructions shown above, a path consisting of many shorter line segments, all connected together, which define the same resultant line could also be specified in POSTSCRIPT®. Even though the same exact bits are turned one, this problem is significantly more difficult for at least two reasons:

1. more POSTSCRIPT® code needs to be read and interpreted;
2. each small line segment needs not only to be drawn, but connected to the next line segment in such a manner to make it appear as one continuous line.

Other graphical shapes can be specified:

- circular arc segments (*arc*, *arcn*, *reference 5*);
- Bezier cubic spline segments (*curveto*, *reference 6*);

in addition to stroking (drawing) a line or curved segment, an area that is described by a sequence of these segments can be filled with black or some halftone pattern corresponding to the selected grey value.  
(*stroke*, *strokepath*, *reference 7*; *fill*, *ref. 8*; *eofill*, *ref. 9*; *setgray*, *ref. 10*)

In addition to drawing or filling the paths that are defined, these same paths can be used to specify a clipping boundary through which other text, vector graphics, or bitmap images are clipped before being rendered in page memory and subsequently on the printed page. (*clip*, *reference 11*)

Once again POSTSCRIPT® gives the user almost limitless flexibility for printing vector graphics. Both the amount of POSTSCRIPT® code as well as the rendering complexity can vary widely. I have no doubt that similar papers to this one could be written with concentration on performance aspects of vector graphics or bitmap imaging instead of text. Hopefully, this brief section has provided the reader with some appreciation for the number and magnitude of issues when dealing with POSTSCRIPT® vector graphics and bitmap images.



## 2. Problem Description

There are many different ways of describing a page of text in POSTSCRIPT®. How one describes a page can significantly influence the overall performance of the printing system. The variables to be investigated, along with the expected outcome of the experiments, are described in sections 2.1.1 through 2.1.8. The expected results are based on assumptions of the RIP architectural model described in section 2.4. Sections 2.1.1 through 2.1.5 match with sections 3.1 through 3.5. The latter sections deal with the specific experimental procedures used, the results that are produced, analysis of these results, and proposals, if any, on what changes could be made to improve performance.

For each variable in question, a set of benchmark pages are generated to demonstrate the significance of the variable. Many of the benchmark pages are included in sections 3.1 through 3.5. Due to space constraints, benchmark pages that are similar to ones included, but different in some small way, are described, but not included as inserts.

Section 3.2 specifies the methods used in calculating and measuring times taken for the different phases of the RIPing process, namely communications and the combined compilation and rasterization time. The programs written to support the experiments performed for this thesis are explained in section 3.3 with backup information provided in Appendix 2.

Once again, section 3 records the experimentally measured times, analyzes these times, and suggests improvements to the RIP architecture and/or to the POSTSCRIPT® language itself that might yield better performance.

Section 4 presents overall conclusions using all of the information in section 3 to comment on the general trends that were observed.

## 2.1 Variables that Affect Performance

There are many different ways that a POSTSCRIPT® printer can be instructed in how to print text on a page. Some variables which affect system performance are listed in the following sections.

### 2.1.1. Font Formats: Outline vs. Bit-Maps

#### Problem Description

Rasterizing a font outline takes much more time than copying (BLT- Block Logical Transfer) a font already in bitmap form to the page buffer memory. There is a tendency to use only a small number of different fonts (at a certain size and orientation) in any given written work. In fact it is considered "good style" to limit the number of fonts and sizes to just a few within any document. Furthermore, when using any given font, most of the time the writer uses characters of the alphabet that have been used before. For example if the letter "a" is used 20 times on a single page in the same font it needs only to be rendered once and is repeated 19 times. In this case the letter "a" is repeated 95% of the time. Repeat use of fonts and of characters within a font are the two primary reasons why all POSTSCRIPT® printers to date have a font cache which stores the scan converted fonts for later use in the faster bitmap form. "Printing a character that is already in the font cache is typically a thousand times faster than scan converting it from the character description in the font" (*reference 12*). The time consuming process of outline font rasterization can be done at various times, influencing the printer's performance. This section looks at three of these particular times:

1. inside a POSTSCRIPT® program, when a new non-cached character is specified to be printed;
2. once at the "factory" where the bit-maps are permanently stored in non-volatile memory, typically some type of ROM;
3. during idle time. When the RIP is not busy generating pages it can perform the time consuming font outline rasterization task.

These three are looked at in more detail in sections 2.1.1.1 through 2.1.1.3, respectively. The corresponding experimental procedures, results, and analysis can be found in sections 3.1.1 through 3.1.3.

Section 2.1.1.4 and its complement section 3.1.4 deal with the performance effects of scaling and rotating outline fonts. Since detailed performance measurements will be done in section 3.1.4, the print job for this experiment contains no repeat characters. This is the only way to isolate the scan conversion process from the bit BLTing (moving bit-maps) from bit-map cache. More on this in section 3.1.4.

### 2.1.1.1 Real Time Outline Font Rasterization and Caching

During the course of a typical POSTSCRIPT® file, fonts resident in outline form are specified to be scaled, (possibly) rotated, and printed. When this occurs the first print using these fonts tends to be slow due to the time consuming task of in-line (or real time) scan conversion of the outline fonts into the faster bit-map form.

#### Expected Results

The first page after a power up sequence printing a font that has not been stored in bit-map form is expected to be slow. Once this first page has been printed, all of the fonts on that page will have been stored in the RIP's bit-map font cache.

Subsequent pages are expected to print at a much faster rate, assuming the printer is kept powered-up.

### 2.1.1.2 Pre-Cached Bit-Maps

If the time consuming process of outline font rasterization is done before the font is requested in the POSTSCRIPT® program, substantial performance gains can be realized. This section deals with the case where the outline font rasterization is done once "at the factory" and the resultant bit-maps of a select few fonts are stored in non-volatile bit-map font memory, such as EPROM in the case of the Apple Laser-Writer Plus.

#### Expected Results

Pages that print fonts that have been pre-scan converted and permanently stored in ROM are expected to yield equivalent performance whether the page is the first one after power up or not. The time consuming outline scan conversion has been done once, at the factory. The RIP need not be bogged down with this task.

### 2.1.1.3 Background Outline Rasterization

While a page printer is in the powering-up state, the user is typically forced to wait up to several minutes before the first page can be printed. In laser or LED electrophotographic printers much of this delay is due to the time it takes for the fuser heating element to rise up to its required temperature. During a portion of this time the RIP usually goes through an initialization and diagnostic checking phase. Once this is done, the RIP must wait until the print engine is ready to accept a print job from the RIP. This is one example of *idle time*. Another example of idle time is the time in between jobs. If the time consuming process of outline font rasterization is done before the font is requested in the POSTSCRIPT® program, say during this idle time, substantial performance gains can be realized.

#### Expected Results

Several fonts in the Apple LaserWriter Plus™ are scan converted from their original outline form to bit-map form and stored in the RIP's font cache memory during idle time. The POSTSCRIPT® Language Reference Manual lists several fonts and sizes at a zero degree rotation (i.e. portrait orientation) that are automatically scan converted during idle time (*reference 13*). They include all of the alphanumerics and common punctuation of the 10 and 14 point Times-Roman® and Helvetica® fonts, and also the lower case letters of the 10, 12, and 14 point Times-Bold® and Helvetica-Bold® fonts. In addition, an extension to the POSTSCRIPT® page description language has been specified for the Apple LaserWriter Plus™ to allow the POSTSCRIPT® programmer to specify which fonts he or she would like to be scan converted during the printer's idle time. The `setidlefonts` operator allows the selection of one of thirteen fonts with a different x and y point size selectable in tenths of a point increments, and an angle selectable in increments of five degrees (*reference 26*).

Print times for pages that use the 10 point Helvetica® and the 14 point Times-Roman® fonts will be looked at. The POSTSCRIPT® files containing references to these fonts will be sent at varying times after the power-up sequence is begun. This will be done by simply turning the printer off and then back on. It is expected that if a job using an idle time scan converted font is sent immediately to the printer following a power up sequence, it will take longer than if some time is waited before this is done. The experiments are reported in section 3.1.3.

#### 2.1.1.4 Effect of Scaling and Rotation

Section 2.1.1.4 and its complement section 3.1.4 deal with the performance effects of scaling and rotating outline fonts. Page 91 of the POSTSCRIPT® Language Reference Manual states the the "built-in POSTSCRIPT (outline) fonts are usually 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 **scalefont** or **makefont** operator, the new matrix is concatenated with the FontMatrix to yield a transformed font." In other words the FontMatrix maps a font in the 1000 unit space to a one unit space scaling the x and y font units by 1/1000 with the following matrix:

$$\begin{bmatrix} 0.001 & 0 & 0 \\ 0 & 0.001 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix scales both x and y coordinates by 1/1000. See the definition of **CTM** in the Glossary (section 5) for more information on the variables of a given matrix.

Since the default unit in POSTSCRIPT® is the point, this means that if any given font is not scaled by some means, it will print out at a 1 point size. Therefore all usable fonts that are scan converted from outline are initially scaled. The time consuming matrix multiplication step must always be done. The second part of the scan conversion process involves the rasterization, or rendering, of the transformed (scaled, etc.) font outline.

Note that, in addition to the concatenation of the **FontMatrix** with the transform matrices generated by the **scalefont** and **makefont** operators, the fonts are also affected by the **Current Transformation Matrix** or the **CTM**. Other general POSTSCRIPT® operators that modify the **CTM** are the **translate**, **rotate**, and **scale** operators. So, there are several ways to scale and rotate fonts. As already mentioned, executing the **scalefont** and **makefont** operators can scale fonts. In addition, the POSTSCRIPT® **scale** operator, which affects the **CTM**, also applies to fonts. Fonts can be rotated in two ways: by specifying the correct matrix variables using the **makefont** operator; or by executing the **rotate** command. Note that the **makefont** operator provides the user with a much more powerful tool than just the scaling and rotating functions. Since the user specifies a matrix, any linear transformation can be specified. This is how fonts are obliqued (ie. tilted) for example.

There is one more function that should be mentioned: that is, applying "intelligence" to the outlines. It is commonly known that Adobe Systems, Inc., the implementors of POSTSCRIPT® in the Apple LaserWriter Plus®, applies proprietary "hints" to their font outlines when rendering them. This is done to yield better looking characters at the 300 dot per inch resolution of the Apple LaserWriter Plus®. This procedure is certainly done at the 0 degree (portrait) orientation and is probably done at other quadrant orientations - 90 degrees (landscape), 180 degrees, and 270 degrees. The performance impact of this factor is not known.

This section compares the performance of scaling and rotating a printer resident POSTSCRIPT® outline font versus the same font that has already been scan converted and stored in the RIP's bit-map font cache memory. The font that has been selected for these experiments is the Times-Roman® font; two point sizes will be 11 points and 22 points; three angles of rotation will be 0, 70 and 90 degrees.

### Expected Results

As discussed above, there are two steps in rendering an outline font:

- applying the scaling and rotation to the current **FontMatrix**. This needs to be done for any scale factor or rotation angle. Some multiplies are easier, however, and should execute faster. Of the three angles specified, 0 and 90 degrees should be faster than 70 degrees.
- scan converting the transformed outline. In going from an 11 to a 22 point font, twice the number of scan lines need to be accessed; four times the number of pixels need to be written to.

### Scaling Outline Fonts:

If scan converting the transformed outline font is the dominant factor, the 22 point font should take two to four times longer than the 11 point font. If however applying the transformation to the font outline is the dominant factor, then the performance should be closer to equal (i.e. independent of point size). Not knowing which of these is dominant, the expectation relative to font size variation is as follows:

As the size of the font increases, the time needed to render that font is expected to increase at a rate in between one and the square of the size difference. In the case of 11 and 22 point size fonts, the 11 point font is expected to perform from 1 to 4 times faster than the 22 point font. Note that the effect of incorporating font "intelligence" into this was not considered.

### Font Outline Scan Conversion vs. Pre-Cached Bit-Map Fonts:

Rendering an outline font is expected to take much longer than pulling the same font, already in bit-map form, from the font cache. "Printing a character that is already in the font cache is typically a thousand times faster than scan converting it from the character description in the font." (*reference 12*) Based on this information it is expected that the measured performance of pre-cached fonts should be approximately 1000 times faster than the measured performance of outline fonts. The overhead of a base (i.e. blank) page must be considered in this analysis. See section 3.2.4 for more details on this experiment.

### Rotating Outline Fonts:

As for the effect of angle on performance, it is expected that 0 degrees will yield the best performance because of two reasons:

- it is assumed that the POSTSCRIPT® interpreter is probably optimized for this highly used case;
- at 0 degrees, the cos and sin values yield 1 and 0 respectively. Assuming this information is used to decrease the number of calculations, an increase in performance should be realized.

The performance of the 90 degree rotated case should follow with the 70 degree rotation case yielding the poorest performance. On the other hand, the 70 degree rotation case could, instead, yield the fastest results if font intelligence step turned out to be a dominant (and slow) process.

## 2.1.1.5 Variations of Font Style

On page 20 of the Fluent Laser Fonts™ User's Guide it states that "complex fonts ... will take longer to print because of the intricate curves and shapes". Three printer-resident fonts with different levels of "complexity" will be used:

1. Helvetica® was chosen for its simple style. There are no serifs and many of the characters, like H, A, and V use straight line segments exclusively. This information can be stored in a very compact outline form and is expected to scan convert faster than the two other, more complex fonts described in 2 and 3 below. The text of this paragraph is set in Helvetica® at 12 point.
2. Times Roman® was chosen for its medium complexity style. On the one hand Times Roman® is a complex font since it has serifs on many of the characters, like "i", "m", and "T". On the other hand many outline segments of the characters of the Times Roman® font are made up of straight line segments orthogonal to the device coordinate system. The page that uses the Times Roman® font is expected to yield better performance than the same page using the simpler Helvetica® font shown in (1) above, but worse performance than the more complex Zapf Chancery® Medium Italic font show in (3) below. The text of this paragraph is set in Times Roman® at 12 point.
3. *Zapf Chancery® was chosen for its complex style. Whereas the fonts shown in (1) and (2) above used many straight line segments, the Zapf Chancery® Medium Italic font uses many more curved segments, which are inherently more complex. The performance of a printed page using this font is expected to yield the worst performance when compared to (1) and (2). The text of this paragraph is set in Zapf Chancery® Medium Italic at 12 point.*

Note that when "performance" is mentioned above, only the scan conversion process is intended to be used as a comparison. Once the font is in the bitmap font cache all three fonts are expected to perform equally well. If there is a small difference in speed when printing from the font cache, the font that is the smallest (i.e. Zapf Chancery®) should be the fastest, with the font that is the largest (i.e. Helvetica®) should be the slowest. This is because the time to BitBLT a font character from the bitmap font cache is assumed to be directly proportional to the size that character occupies in memory. Note that although each of the fonts shown in (1), (2) and (3) are set at 12 points, they each appear to be a different size.

### Expected Results

The following relative scan conversion rates are expected:

- Helvetica®: fastest due to the simple style;
- Times Roman®: intermediate;
- Zapf Chancery®: slowest due to the complex font style.

Once the font has been scan converted and placed into the bitmap font cache, all fonts are expected to print equally fast.

## 2.1.2. Resident vs. Downloaded Fonts

### Problem Description

Fonts can be either resident in the printer (i.e. RIP) or be downloaded from the host computer (host-resident). Furthermore, fonts can be in either bit-map or outline format. This section concentrates on observation of the performance differences between downloadable vs. printer-resident fonts. To simplify the task and to get more meaningful results outlines are compared with outlines, and bit-maps with bit-maps.

Downloading fonts can significantly impact system performance. The key parameter that effects performance is communication time (i.e. time to download the font). For outline fonts this may be about 30 KBytes per typeface at any size. For bit-map fonts the amount of data varies with the size of the font (see section 1.4). A full bit-map representing each character at each size is needed.

Fonts can be in one of the following formats:

#### Bit-map fonts

- A. Downloadable bit-map fonts;
- B. Printer-resident bit-map fonts.

#### Adobe Bezier outline fonts

- C. Resident in the Apple Laser Writer Plus printer (RIP and print engine combination);
- D. Host-resident, downloadable fonts in Adobe encoded Bezier format;

#### Third party outline fonts

- E. Downloadable fonts in Bezier format described with generic POSTSCRIPT® instructions;

All five of these formats are used to print simple text and are compared on a performance basis in this section.

Three comparisons are made:

- A vs. B Resident vs. downloaded bit-map fonts.  
See sections 2.1.2.1 and 3.2.1.
- C vs. D Resident vs downloaded Adobe outline fonts.  
See sections 2.1.2.2 and 3.2.2.
- D vs. E Adobe vs. third party downloaded outline fonts.  
See sections 2.1.2.3 and 3.2.3.

Two bitmap fonts are used for this experiment:

### A. Downloadable bit-map font

A hand-coded POSTSCRIPT® bitmap font similar to the one described in section 2.3.3 is downloaded to the Apple Laser Writer Plus. For this experiment only ten 12-point characters were defined at 300 dots per inch. These ten characters are then printed. The download time as well as the printing times are both measured

### B. Printer-resident bitmap font.

The Apple Laser Writer Plus has several pre-scan converted printer-resident bit-map fonts (*reference 14*). The 12 point Helvetica® font is selected for this experiment. The time to print the same ten character string is measured and compared to the time taken with the downloadable bitmap font of (A).

## Expected Results

The total time to print the downloadable font includes not only printing the ten character string but, even more significantly, also includes the time to download the bitmap font itself. A 12 point font is assumed at a printer resolution of 300 dots per inch. If a full character set were specified, complete with upper and lower case letters, then the average size of a single character bitmap would be approximately 170 bytes, which is 50% of the 50 pixel by 50 pixel character bounding box (see section 1.4). Since the ten characters chosen for this experiment are all lower case letters it is assumed that, on average, the size of these ten character bitmaps are 25% smaller than the average bitmap across the entire character set. This brings the 170 bytes average per all characters down to 127 bytes average per lower case character. Therefore, the size of the combined bitmaps of the ten characters is 1,270 bytes. Since POSTSCRIPT® describes all bitmaps with ASCII encoded hex, the bitmap data is expanded by a factor of two (i.e. a hex value of "A7" is coded as two sequential ASCII characters: "A" and "7"). When this is taken into account the ASCII encoded bitmaps for the ten lower case characters take up 2,540 bytes. Examining the POSTSCRIPT® bitmap font shown in section 2.3.3, it appears that the font overhead excluding the font metrics is about 1 Kbyte. The metrics information overhead is about 40 bytes per character or 400 bytes per the ten character set. The anticipated downloaded font size is:

$$\begin{array}{r} 2,540 \text{ bytes: bitmaps for ten characters} \\ 400 \text{ bytes: metric information for ten characters} \\ + \underline{1,000 \text{ bytes: additional font overhead}} \\ 3,940 \text{ bytes: expected size of ten character bitmap font} \end{array}$$

If the RS-232 serial communications interface is used at 9600 baud, then the expected download time for the 3,940 byte bitmap font is 4.10 seconds. For this experiment Appletalk is used instead of RS-232, so a faster download time is expected. Note that the raw data transmission speed of Appletalk is 220 Kbits/sec, although the actual throughput is a fraction of this mainly due to protocol overhead. See sections 2.1.2.4 and 3.2.4 for more information on this topic.

Once the bitmap font is downloaded it is expected that the downloaded font will perform as well as the printer-resident bitmap font. The overall time to print the downloadable font will be significantly longer due to the font download time.

## 2.1.2.2 Adobe Downloaded Outlines vs. Printer Resident (Internal) Outlines

### Problem Description

The Apple LaserWriter Plus printer has 35 resident outline fonts (see section 2.4 for a list of resident fonts). In addition, a user can purchase additional fonts for use with Macintosh application programs. These fonts are supplied on Macintosh compatible micro-floppy discs and include several (usually four: regular, *italic*, **bold**, and *bold italic*) styles of a single font family. One printer outline font and several (about five different sizes) bitmap screen fonts is provided per each style. The bitmap screen fonts are used by application programs to provide the desired WYSIWYG effect. The printer outline font needs to be downloaded to the printer before the application program can send the POSTSCRIPT® program to the printer that references that particular font. This can be done either explicitly, with a program which downloads the outline printer font, or implicitly, by the application program needing the font. In the latter case the application interrogates the printer to determine if the font is already loaded into the printer. If it is, the print program is simply downloaded. If it is not loaded, then the font is downloaded right before the print program.

Section 3.2.2 compares a particular printer resident font, Times-Roman® to another Adobe outline font, the downloadable Stone Serif font. They are compared on the basis of scan conversion speed, BitBLT from cache to page buffer speed, and download time.

### Expected Results

Whether the font originates from the host or from an internal RIP font data structure (i.e. in ROM), the task of generating a bit-map from outline is essentially the same in either case, especially if both fonts were supplied by the same font vendor who, in this case, is Adobe Systems, Inc. So, the task of scan converting the outlines and the subsequent task of moving the cached bitmaps to the page buffer in the two cases are expected to yield similar results.

The font styles of the downloaded Adobe font, called *Stone Serif*, was chosen to be similar in complexity to Times-Roman®, the internal font it is being compared to. This was done to minimize the effect that varying the complexity of a font style has on performance. Section 2.1.1.5 and its complement, section 3.1.5 deal with this subject in detail.

The one area where a large difference in system performance is expected is in the area of font download time. Resident fonts are held in ROM so there is no download time. Downloaded outline fonts are approximately 30 to 40 KBytes in size. If an RS-232 serial interface were used running at 9600 baud, the download time would be about 35 seconds. Since Appletalk is used this time should be less, but still significant.

### Problem Description

As mentioned in the previous section a user can purchase downloadable fonts for use with Macintosh application programs. These fonts are supplied on Macintosh compatible micro-floppy discs and include several (usually four: regular, *italic*, **bold**, and ***bold italic***) styles of a single font family. One printer outline font and several (about five different sizes) bitmap screen fonts is provided per each style. The bitmap screen fonts are used by application programs to provide the desired WYSIWYG effect. The printer outline font needs to be downloaded to the printer before the application program can send the POSTSCRIPT® program to the printer that references that particular font. This can be done either explicitly, with a program which downloads the outline printer font, or implicitly, by the application program needing the font. In the latter case the application program interrogates the printer to determine if the font is already loaded into the printer. If it is, the print program is simply downloaded. If the font is not loaded, then it is downloaded right before the print program.

Downloadable fonts can be purchased from Adobe, the creators of POSTSCRIPT® and POSTSCRIPT® fonts. They can also be purchased from other third party vendors. There are several differences:

1. Adobe fonts are encrypted.

Third party fonts directly follow the convention set in the POSTSCRIPT® Language Reference Manual with no encryption (*reference 15*).

2. Adobe fonts have "intelligence". That is, there is a proprietary Adobe algorithm which reads the encrypted outline font with "intelligence hints" and scan converts the outline producing a high quality font at the 300 dpi printer resolution.

Third party font outlines are treated as generic graphic shapes when scaled. They do not have any "intelligence". The quality produced on a 300 dpi device is noticeably poorer at typical sizes (i.e. 10 or 12 point). Compare figures 3.2.3.A.1 & 2 (no intelligence) to figures 3.1.1.A.1 & 2 (with intelligence).

3. Adobe fonts generally originate from well known font houses like Linotype and ITC and are aimed at providing virtually all printing and publishing needs.

Third party fonts generally are new designs with an unconventional artistic flare. Note, however, that the third party font selected for this experiment, the CasadyWare Galileo Roman font, was chosen for its similarity (in style and complexity) to the Adobe Stone Serif font.

4. Adobe fonts are typically more expensive, costing in the \$100 to \$300. As an example the list price of the Adobe Stone Serif font used in this section was \$275.

Third party fonts are typically less expensive, costing in the \$50 range. As an example the list price of the CasadyWare Galileo Roman™ font used in this section was \$45.

Section 3.2.3 compares a two downloaded fonts, the Adobe Stone Serif and the CasadyWare Galileo Roman™ font. They are compared on the basis of scan conversion speed, BitBLT from cache to page buffer speed, and download time.

### Expected Results

The task of scan converting the Adobe "intelligent" outline is inherently more difficult than scan converting a non-intelligent outline (i.e. treating characters like graphic shapes). Both tasks need to convert the outlines into bitmaps but the Adobe conversion has one more step, the step that incorporates the intelligence. This implies that the Adobe downloaded font (with intelligence) should scan convert slower than the CasadyWare downloaded font (without intelligence).

Once the bitmap has been generated from outline and stored in the font cache the task of printing the bitmaps is essentially the same in either case. So, the task of moving the cached bitmaps to the page buffer in the two cases are expected to yield similar results.

Both the Stone Serif and Galileo Roman™ fonts are downloadable POSTSCRIPT® outline fonts. It is assumed that their sizes are similar and subsequently their download times should also be similar.

The font styles of the downloaded Adobe font, called *Stone Serif*, was chosen to be similar in complexity to CasadyWare Galileo Roman™ font it is being compared to. This was done to minimize the effect that varying the complexity of a font style has on performance. Section 2.1.1.5 and its complement, section 3.1.5 deal with this subject in detail.

Probably the largest factor is the one that is least predictable. That is the variability introduced when comparing two different font vendors. Even if the original outlines were exactly the same, how these outlines are converted into a POSTSCRIPT® program may have significant effects on how fast they scan convert.

## 2.1.2.4 Appletalk vs. RS-232 Comparison

Both the Appletalk and RS-232 communications interfaces were used throughout this paper. RS-232 was used more extensively due to the greater control this interface allows:

- ability to capture the real-time conversations between the host computer and the printer with a protocol analyzer;
- the predictable overhead (i.e. one start bit and one stop bit);
- the point to point nature of the interface.

Appletalk, on the other hand, was used primarily to allow the download of Adobe and third party (CasadyWare) downloadable fonts. Sections 2.1.2.1-3 and their complementary sections 3.2.1-3 used Appletalk. It is more difficult to separate out the protocol overhead and interference from other computer systems sharing the Appletalk network. Nevertheless, Appletalk is looked at in this section and section 3.2.4 and compared with RS-232. The printing times are examined as well as the communications download throughput.

### Expected Results

The printing times are expected to be the same. The communications interface should not influence the time that a page prints once it is downloaded.

The raw data rate of the RS-232 communications interface is 9600 bits / second. It is shown in section 2.2.1 that the communications overhead is two bits per eight bit character (i.e. the start and stop bits). Therefore it is expected that the efficiency of the RS-232 interface will be 80% of the rated bit rate of 9600, or 7,680 bits per second.

The raw data of Appletalk is 230.4 KBits per second. It is expected that the effective throughput speed will be much less than the raw data rate, but how much is not easily predictable. It is expected, though, that larger files will transfer at a slower effective data rate than small files, due to the packet size limitations of Appletalk. Again how significant this factor is, is not predictable.

### 2.1.3. Methods of Printing Strings

Strings of characters can be placed on a given page in a number of ways. Each method outlined in this section will be timed and printed in section 3.3. All of the fonts used will be in resident outline format.

The following methods are described in general terms. The exact experimental procedure used is described in section 3.3.

#### A. Simple unjustified text (i.e. "ragged right")

x y moveto	string starts at specified (x,y) point
( string of characters ) show	character string is proportionally spaced

#### B. Justified text with the front end application providing the extra incremental spacing to be applied between adjacent characters, using the **ashow** command.

x y moveto	string starts at specified (x,y) point
xinc 0 ( string of characters ) ashow	characters are spaced proportionally with xinc added to relative x increment to every character; note that "0" is the y increment.

#### C. Justified text with the front end application providing the extra incremental spacing to be applied between adjacent words, using the **widthshow** command.

x y moveto	string starts at specified (x,y) point
xinc 0 32 ( string of characters ) widthshow	characters are spaced proportionally with xinc added to relative x increment to every character with an ASCII value of "32" (i.e. a space); note that "0" is the y increment.

#### D. Justified text with a downloaded POSTSCRIPT® program that calculates the excess space and distributes it between all adjacent characters within the line. The routine to justify a string of text is repeatedly called once per line. This routine provides the following functions:

- calculates the width of line of text;
- calculates the extra space needed to fill the line;
- calculates the incremental space that must be added to each character to achieve proper justification (note: number of characters in line is passed to this procedure);
- prints the line of text using the **ashow** command, equally distributing the extra space in between adjacent characters within the string.

- E. This method is the same as that used in (D) on the previous page except the downloaded justification program has one additional feature: it calculates the number of characters in the string.
- F. This method is the same as that used in (E) above except the downloaded justification program also handles the carriage return - line feed functions.

### Expected Results

Each case uses more capabilities of the POSTSCRIPT® page description language. It is expected that as cases A through F are timed and printed, the times will progressively get larger (i.e. page printing times get slower), indicating that the simpler any program is, the faster it executes.

#### 2.1.4. Resolution "Targeting" vs. Total Resolution Independence

POSTSCRIPT® is a resolution independent language. Its default coordinate system is the point, which Adobe defines to be exactly 1/72 of one inch. If one knows the actual resolution of the printer to be R dots per inch, then the Current Transformation Matrix (CTM) can be set up to scale all coordinate references by R / 72. Then if only integer references are made, these references will map directly to the engine pixels. In other words, the CTM which maps user coordinates (now in R units per inch) to device coordinates (in R pixels per inch) is the unity matrix. If the POSTSCRIPT® software has the intelligence to recognize this situation, a possible performance increase can be achieved. The "ragged right" (case A) page described in the previous section will be printed and compared to a similar page with all of the coordinates given in integers representing pixels. The following POSTSCRIPT® commands will be issued:

```
72 300 div dup scale           change CTM to unity matrix

xn1 yn1 (string of text) show
xn1 yn2 (string of text) show
:
xn1 ynm (string of text) show
```

Note that only if the actual print engine resolution is 300 dpi, and if the POSTSCRIPT® imaging software is "smart" enough to recognize this case, can the potential performance gain be realized. If the print engine resolution were, say 400 dpi, the CTM would not be the unity matrix and the output would be generated looking the same. That is why this is termed resolution "targeting" rather than resolution "dependence".

### Expected Results

The case that uses an integer coordinate system that corresponds to the print engine resolution should execute faster; it is not known how much faster.

## 2.1.5 Effect of Varied Page Complexity on POSTSCRIPT® Processing Time

### 2.1.5.1 Effect on the Inter-Page Time Delay Program sec

The sec program, shown below, is a delay program which takes as its input an integer and a token (i.e. variable). It has been empirically designed to execute in  $n$  seconds, where  $n$  is the integer passed to it. The final def command loads the measured sec execution time into the variable "passed" to to the sec routine.

```
/sec                                % one second delay under no load
{ usertime /T exch def              % keep track of time actually taken
  {
    1228 {                            % loop that executes in 1 second under a
      373.737 737.373 mul pop        % "no load" condition
    }repeat
  } repeat
usertime T sub def
} def

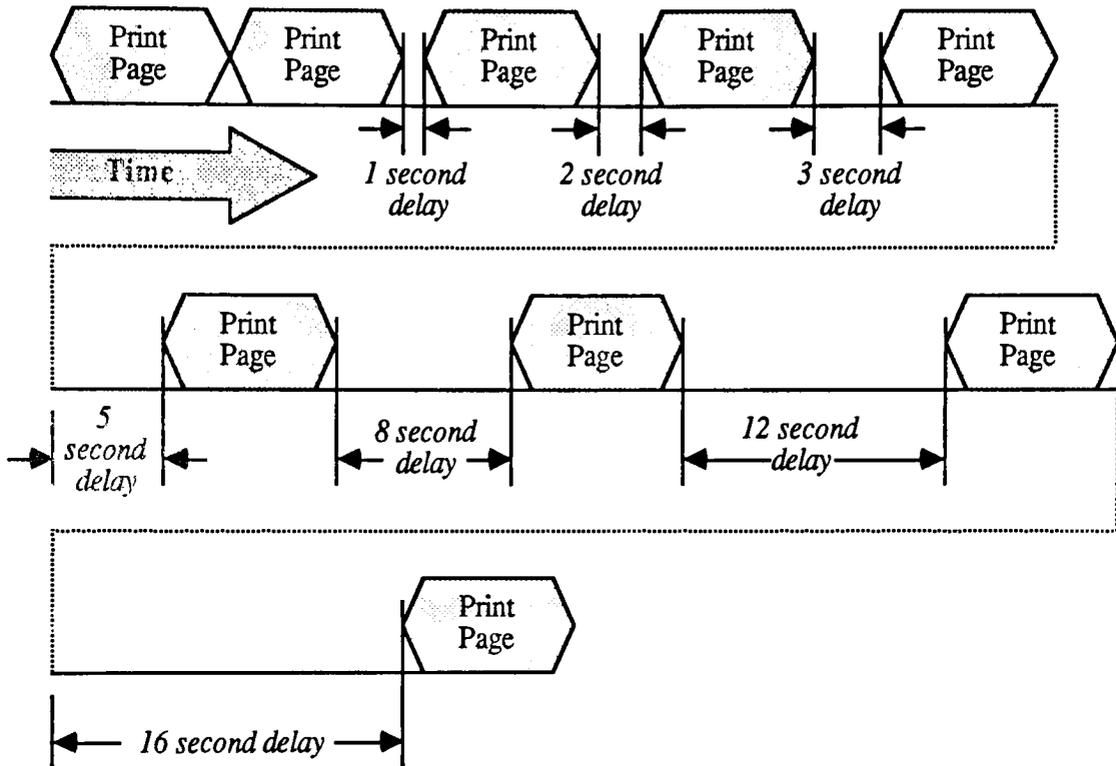
/W7 7 sec                            % "W7" is the variable which will contain
                                     % the measured execution time of sec
                                     % "7" is the target number of seconds
```

#### Expected Results

It is reasonable to expect a program to execute in a constant amount of time independent of the operations that are executed outside this program. This would normally be the expected results. However, during the course of running other experiments it was observed that the measured time to execute sec varied as the contents, complexity, and position of text on the printed page varied. Unless this phenomenon were not blindly stumbled upon, there would be no reason to suspect this to occur. Section 3.5.1 looks into this strange behavior and attempts to explain why it happens.

### 2.1.5.2 Effect on the Compilation and Rasterization of Text Pages

The sec program, described in the previous section, is used to induce delays in between the printing of consecutive pages. In this experiment the target delays are 0, 1, 2, 3, 5, 8, 12, and 16 seconds in duration. The nine pages printed are all the same page that is individually interpreted (or compiled) and rasterized each time. Figure 2.1.5.2.1 shows how these delays are interleaved with the task of printing these nine pages.



Sequencing of Printing Pages

Figure 2.1.5.2.1

#### Expected Results

As written in the previous section, it is reasonable to expect a program to execute in a constant amount of time independent of the operations that are executed outside this program. The last section dealt with the delay program sec. This section deals with the program that times and prints a page of text. In either case, this would normally be the expected results. However, during the course of running other experiments it was observed that the measured time to execute the printing of the pages varied as the delay time between pages varied. Once again, unless this phenomenon were not blindly stumbled upon, there would be no reason to suspect this to occur. Section 3.5.2 looks into this strange behavior and attempts to explain why it happens.

## 2.2 Measurement Techniques

The POSTSCRIPT® text files, generated by the Text Batch Processing Program outlined in section 2.3.2 and appendix 2.2.1, generates a file with the format shown below. Note that Appendix 2.2.4 contains a sample POSTSCRIPT® file that has been generated by the Text Batch Processing Program. References to appendix 2.2.4 are made to show the actual lines of code that correspond to the three parts of the generated file: Header, Body, and Trailer.

**Header** Two lines of POSTSCRIPT® code that saves the current time in a local variable and sets up the text file that follows to be a callable procedure enabling timing measurements to be taken.

See Appendix 2.2.4, page 1, (the first two lines):

```
/Tsd usertime def  
/Page{
```

**Body** A sequence of POSTSCRIPT® instructions which selects fonts, positions the cursor at new-line positions, and writes strings of text.

See Appendix 2.2.4, page 1 (starting with the third line) to page 2 (up to and including the last show command):

```
/Times-Roman findfont 9.0 scalefont setfont  
:  
:  
(effectively transfers the copy image to the paper.) show
```

**Trailer** POSTSCRIPT® code that calls the Body a number of times, with different delays between calls, and keeps track of the Body download time, the page rasterization times and delay times; a trailer page with all of the timing information is also printed.

See Appendix 2.2.4, page 2 (starting with the first usertime instruction) to the end of page 6:

```
usertime /T0 exch def           % record time before showpage  
:  
:  
showpage
```

The POSTSCRIPT® **usertime** instruction is used extensively in recording times. When the **usertime** instruction is executed an integer representing the current time, in milliseconds, is placed on top of the stack. This is actually the time from power-up to the current time. To measure the time it takes to download the Body of POSTSCRIPT® code, or to execute the Body (generate the corresponding page bit-map), a second time is taken. The first time is subtracted from the second time to derive elapsed time. This procedure is repeated a number of times. These times are kept in variables until all of the pages are printed. Then a final trailer page is printed (example shown on the second page of Appendix 2.2.5) showing the times, in milliseconds, and graphing the page generation times relative to the programatically induced delay times between the pages. Many different times are extracted in this way and are discussed more fully in sections 2.2.1 and 2.2.2. Figures 2.2.1, 2.2.2.1, and 2.2.2.2 illustrate how the times are used and are based on the example text POSTSCRIPT® page shown in Appendix 2.2.4. Key instructions are extracted from the example in Appendix 2.2.4 and highlighted in these figures to explain the method used to measure the various times.

Please refer to figure 2.2.1.1, *Download Time Measurement*, for the following explanation. An initial time is loaded into the variable **Tsd**, start download time. The **Page** procedure contains the sequence of commands that specify the printed text on the page. However, the first time the POSTSCRIPT® interpreter "sees" this sequence of instructions, previously called the *Body*, it does not execute them. Later, in the *Trailer* portion of the POSTSCRIPT® file, the **Page** procedure is called and subsequently executed. The first time through the *Body* simply passes through the interpreter until it "sees" the } **def** sequence, which tells the interpreter that the procedure *Body*, named **Page**, is done. Immediately following download of the *Page Body* another time is loaded into the variable **Ted**. The download time, **Tdl**, is derived by subtracting the start download time, **Tsd**, from the end download time, **Ted**.

To verify the validity of this method the times generated in this manner were compared to times generated by two other methods. A protocol analyzer was used to take two measurements: the actual download time and the number of characters that were downloaded. A Hewlette Packard 4953A Protocol Analyzer was connected in a (passive) monitor mode in between the host computer, which emits the POSTSCRIPT® program to the printer, and the Apple Laserwriter Plus printer itself. The HP4953A was programmed to measure the time to download the **Page** portion of four increasingly more difficult pages. The first page did not print anything at all, but merely ejected pages. The second page printed the A through C sections of text shown in Appendix 2.2.5 in one font style, size, and orientation. The third page included printing of sections A through F; the fourth page, A through I. In all cases the trailer page reporting the measured download time using the POSTSCRIPT® **usertime** operator was also printed. The minimum and maximum recorded times (from five runs) are reported in column A of the table shown in figure 2.2.1.2.

To measure the download times with the HP5953A Protocol Analyzer the triggers were set such that one of the protocol analyzer's internal timers was started at the beginning of the downloaded character stream being measured, and was stopped at the end of this same character stream. Measurements were taken on all five runs. The minimum and maximum times recorded with the protocol analyzer are reported in column B of the table shown in figure 2.2.1.2. In all four cases the maximum error is less than one percent, ranging from a maximum error of 0.02 % to 0.90 %.

The HP4953A Protocol Analyzer captures the entire character string that is transmitted. It is temporarily stored in the protocol analyzer's internal memory buffer and then displayed on the protocol analyzer's CRT screen. Once on the screen, the relevant downloaded character stream is easily counted. The number of downloaded characters for all four jobs are shown in column C of figure 2.2.1.2. Job 1 downloaded 213 characters, ... job 4 downloaded 4,749 characters. To calculate the download time several transmission parameters must be known: # bits/character, # bits / second, # start bits, and # stop bits. This information was gotten by reading and interpreting the switch settings on the serial interface board used to drive the printer.

**/Tsd** usertime def  ..... "Tsd" is the start download time

```

/Page {
  /Times-Roman findfont 9.0 scalefont setfont

  72.00 709.20 moveto
  (A. IMAGE LOOP. The KODAK EKTAPRINT
  ... of film that is capable of being electrically)show
  :
  :
  72.00 48.00 moveto
  (... transfers the copy image to the paper. )show

  usertime /T0 exch def

  showpage
} def

```

The first time the PostScript® file is downloaded the "Page" routine is not executed, so the primary factors that influences Tdl (download time) are the size of the "Page" procedure and the speed of the communication link.

usertime **/Ted** exch def  ..... "Ted" is the end download time

Page

usertime Ted sub /T1B exch def

T0 Ted sub /T1A exch def

**Ted Tsd sub /Tdl exch def**

Download Time (Tdl) =  
End Download Time (Ted) -  
Start Download Time (Tsd)

## Download Time Measurement

Figure 2.2.1.1

		Measured using HP4953A Protocol Anal. ---Reference Data---				
Job	A: Measured with internal PostScript® "usertime" timer min                      max	B: Download time was measured directly min                      max	C: - # Char's is meas. - Time is calculated (9600 Bd.; 1 start; 1 stop bit)	% error (max) A vs. B	% error (max) A vs. C	
1	220                      222 <i>msec.</i>	221                      222 <i>msec.</i>	213 characters 222 milliseconds	0.90 %	0.90 %	
2	2.142                      2.144 <i>sec.</i>	↔ 2.145 ↔ <i>sec.</i>	2,060 characters 2.146 seconds	0.14 %	0.19 %	
3	3.684                      3.686 <i>sec.</i>	↔ 3.681 ↔ <i>sec.</i>	3,534 characters 3.681 seconds	0.14 %	0.14 %	
4	4.956                      4.958 <i>sec.</i>	↔ 4.947 ↔ <i>sec.</i>	4,749 characters 4.947 seconds	0.02 %	0.02 %	

Communication Download Time Measurement

Figure 2.2.1.2

The MTI-850/1650B multiple terminal interface is a 16 port serial controller internal to the Sun Microsystem 3/280 computer. Port 7 of this controller was used to download the POSTSCRIPT® jobs for all of the experiments run for this thesis. The communication port was set up as follows:

- baud rate = 9600 bits/second
- character length - 8 bits
- 1 start bit
- 1 stop bit
- No parity

Calculating the download time is now a simple arithmetic calculation. To demonstrate this, job 4 is used below:

Total number of bits per transmitted character  
= 1 start bit + 8 information bits + 1 stop bit  
= 10 bits / transmitted character

$$\text{Download time (job 4)} = \frac{4,747 \text{ char} * 10 \text{ bits/char}}{9600 \text{ bits / sec}} = 4.947 \text{ seconds}$$

The calculated times for jobs 1 through 4 are shown in column C of figure 2.2.1.2 just below the # character entries. Once again the results are excellent. In all four cases the maximum error is less than one percent, ranging from a maximum error of 0.02 % to 0.90 %.

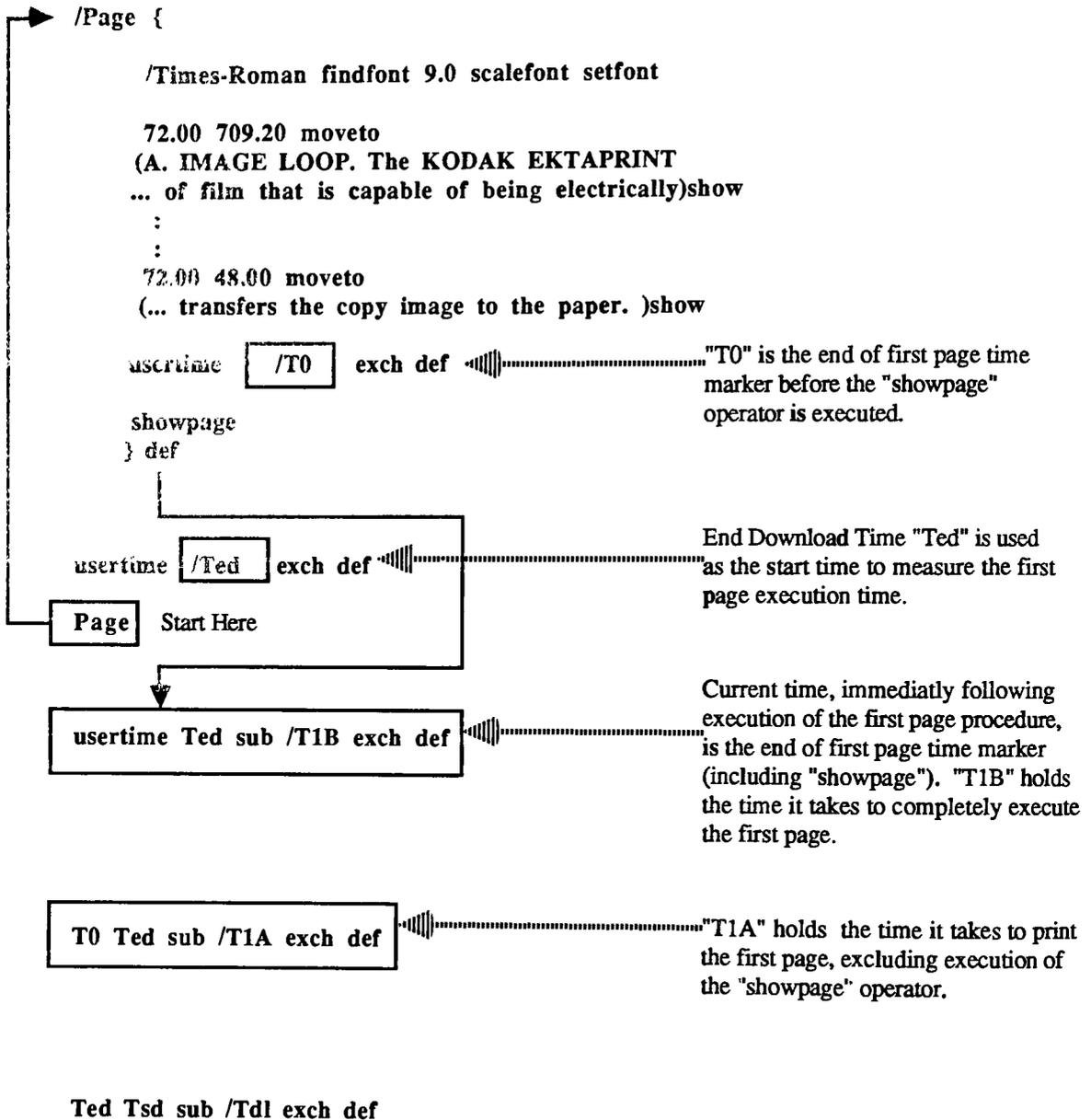
## 2.2.2 Interpretation (Compilation) and Rasterization

Once the *Body* is downloaded, it is then called and subsequently executed. There are two parts to this task. The first is interpreting the string of ASCII bytes to generate the POSTSCRIPT® tokens. It is assumed that these tokens are then executed in an interpretive nature, immediately generating the raster page bit-map. However, in some RIPs an intermediate "display list" is generated which better fits the hardware capabilities of the RIP. The combined process of tokenizing the ASCII string and generating this "display list" is commonly called "compilation". This process is much like the process that is done by a "C" or "ForTran" language compiler.

The second part, called "rasterization", is the process of generating the raster image from either the list of tokens, in a single interpretive manner, or from a pre-compiled display list. Since there is no way to externally separate the interpretation or compilation step from the rasterization step, they are treated atomically throughout this paper.

Figure 2.2.2.1, *Timing of First Printed Page*, shows how the timing of the first page is done. After the *Body* has been downloaded, the current time is stored in the variable */Ted*. This is the same variable that was used in measuring the *Body* download time (see section 2.2.1). The *Page* call causes the *Page* procedure to be executed. All of the commands in the *Body* are inside the *Page* procedure. They consist of selection of fonts, positioning of new lines, and printing of strings of text. The last POSTSCRIPT® instruction in the *Page* procedure is *showpage*, which causes the page to be printed and the page buffer to be erased in preparation for the next page. Two times are taken immediately before and after this *showpage* instruction. The time that is of primary interest is the one following complete execution of the *Page* procedure. Upon completion of this procedure, program control returns to the instruction following the *Page* call. This instruction gets the current time, using *usertime*, subtracts the time value that was recorded prior to the *Page* call, in */Ted*, and stores the difference in a new variable */T1B*. */T1B* now contains the time taken to fully generate the first page as specified in the *Page* procedure. A second time, stored in */T1A*, is the same time as in */T1B*, less the time taken to execute the *showpage* operator.

/Tsd usertime def



## Timing of First Printed Page

Figure 2.2.2.1

Figure 2.2.2.2, *Timing of Pages 2 Through 9* , shows how the timing of the following eight pages is done. First two supporting POSTSCRIPT® procedures must be described:

**/sec** This procedure is passed a literal (eg. */W2*) and an integer (eg. *2*). The integer acts as the outer loop variable. The inner loop, which multiplies two real numbers 1,228 times, executed under a no-load condition has been measured to yield a one second delay. The "no-load" condition was simply the **sec** procedure with calls to it along with the "Trailer" page reporting these times. No printing of text strings, other than the "Trailer" page itself, was executed for this measurement. Also, delay times corresponding to the integers passed to it produced the expected linear result. That is "2 sec" yielded a two second delay, "3 sec" yielded a three second delay, ...

The literal preceding the **sec** call (eg. */W2*) is loaded with the actual measured time of the induced delay.

**/Pr** This procedure is also passed a literal (eg. */T4*) that is loaded with the time taken to execute the **Page** procedure. Inside the **Pr** procedure times are taken before and after **Page** is called. The difference of these times are loaded into the literal.

Using the **sec** and **Pr** procedures a total of nine pages are printed in the following manner:

Print page 1	method shown in figure 2.2.2.1
no delay Print page 2	using <b>Pr</b> shown in figure 2.2.2.2
1 second delay Print page 3	using <b>sec</b> shown in figure 2.2.2.2
2 second delay Print page 4	:
3 second delay Print page 5	:
5 second delay Print page 6	
8 second delay Print page 7	
12 second delay Print page 8	
16 second delay Print page 9	

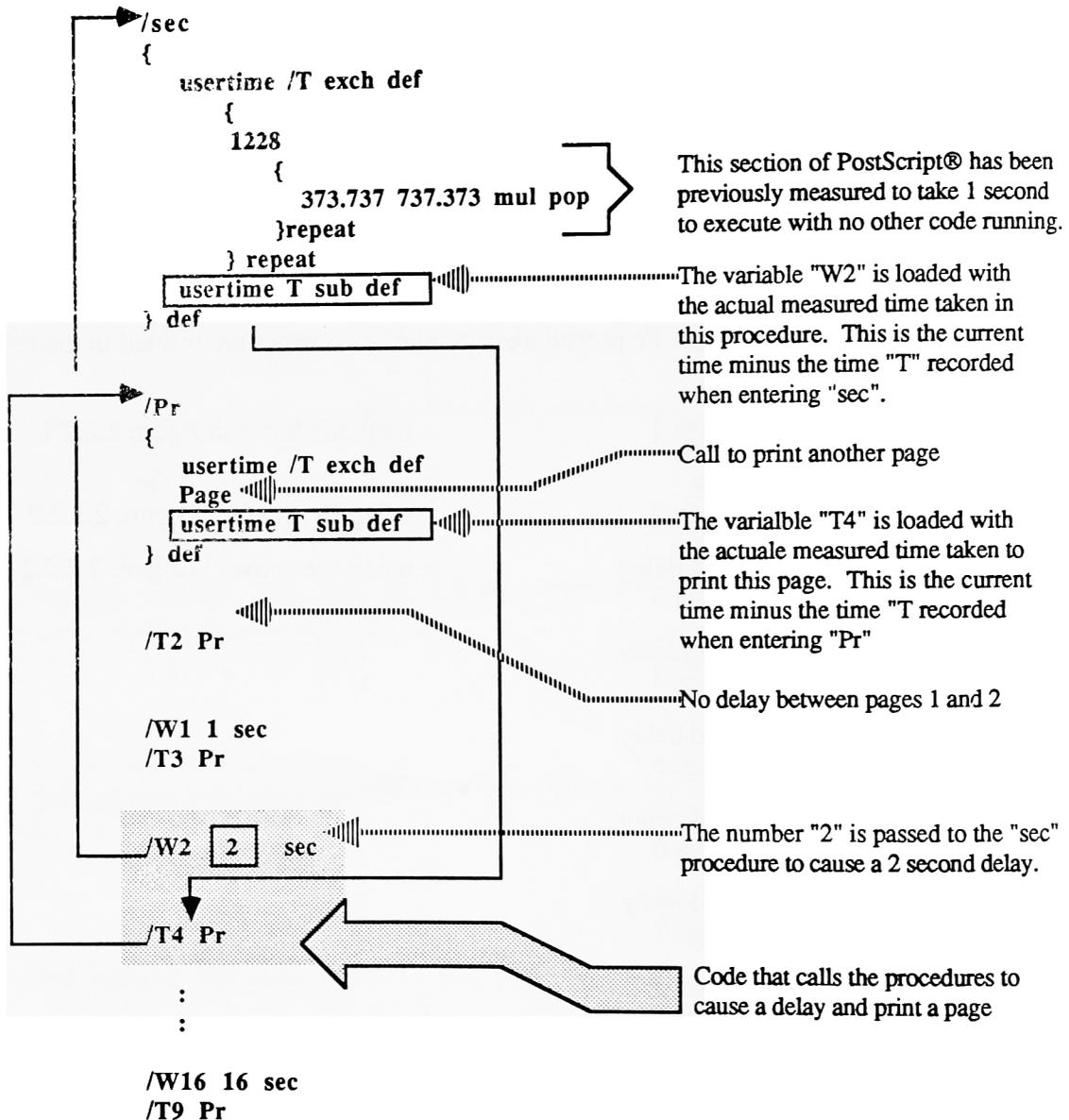
All delays and measured page times are reported on the trailer page (see page two of Appendix 3.2.5). Additionally, the (delay before a page, page generation) time pairs are graphed showing the effect delays between page has on page rendering times.

```

/Page
{
:
:
:
} def

:
Page Calling to print first page
:

```



## Timing of Pages 2 Through 9

Figure 2.2.2.2

To test the repeatability of measurements the method just described was modified slightly. Instead of changing the delay times from page to page, all delays were set to be equal. Each job was executed eight times with the delay set to 0 for the 1st run, 1 second for the 2nd run, 2 seconds for the 3rd run, ..., and 16 seconds for the 8th run.

During each of the eight runs per job, nine pages plus the trailer page were printed.

Print page 1

N second delay  
Print page 2

N second delay  
Print page 3

N second delay  
Print page 4

N second delay  
Print page 5

N second delay  
Print page 6

N second delay  
Print page 7

N second delay  
Print page 8

N second delay  
Print page 9

Print trailer with timing information

*N = 0 (no delay instructions are executed), 1, 2, 3, 5, 8, 12, and 16.*

The minimum and maximum times of pages 2 through 9 are shown in figure 2.2.2.5. Minimum and maximum times of all eight delays per run are also recorded there. Three jobs were run:

Job 1 prints a blank page.

Job 2 prints a simple text page and shown in figure 2.2.2.3. The page had been printed before this experiment was done so that all of the fonts were already in bit-map form (see sections 2.1.2 and 3.2 for more on this).

Job 3 prints a vector graphics page and is shown in figure 2.2.2.4.

**A. IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

**B. PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

**C. EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

**D. AUXILIARY ERASE.** Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

**E. DEVELOPER STATION ASSEMBLY.** The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

**F. SCAVENGER ROLLER.** Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

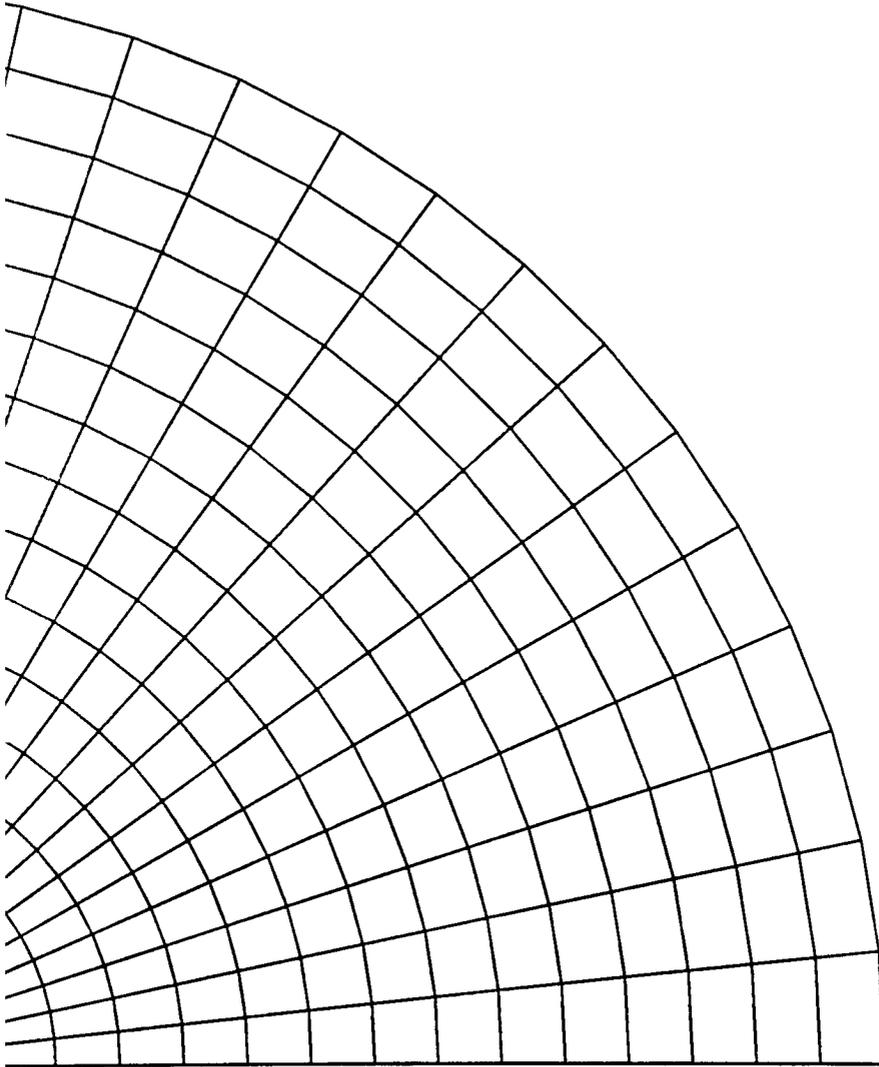
**G. POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

**H. REGISTRATION.** While the developed electrostatic image moves around the CORE, a sheet of copy paper is advanced to the REGISTRATION ASSEMBLY (not shown in Figure 1). At precisely the right time, the copy paper is directed into contact with the IMAGE LOOP and its developed image. This aligns the copy paper and the image on the IMAGE LOOP.

**I. TRANSFER CHARGER.** The IMAGE LOOP and copy paper now pass under the TRANSFER CHARGER, which produces a negative charge on the paper surface to attract the positive charged developer toner. This effectively transfers the copy image to the paper.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Simple Text Page



Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Vector Graphics Page

Figure 2.2.2.4

page 41

### Repeatability Experiment Results:

All of the repeatability results are shown in figure 2.2.2.5.

The measured delay times were within 2 milliseconds per run. If one assumes the accuracy of the of the `usertime` command to be 1 millisecond, as the POSTSCRIPT® Language Reference Manual indicates, the best results that can be expected is exactly 2 milliseconds (i.e. +/- 1 millisecond). However, as figure 2.2.2.5 as well as figure 2.2.1.2 shows, all differences are even numbers (in milliseconds). This suggests the accuracy may actually be limited to 2 milliseconds. In any event, this experiment shows excellent repeatability results for the delay time measurements. There is still one more interesting (puzzling?) point to note; that is, how the delay times change depending on the complexity of the job. This is not the expected result, since one would normally expect the same instructions executed on the same processor to take the same amount of time. This point is further explored in sections 2.1.5 and 3.5.

The measured page times were all within 0.46 % per run, and most were much better. The worst case is job 2 with a delay time of 16 seconds: a 28 millisecond variation was recorded which translates to a 0.46% error. The vector graphics job (#3) seems to yield more stable results than the text job (#2) or even the blank page job (#1). Even though this experiment shows a small amount of variation from run to run, the results of less than 1/2 of 1 percent indicates that this method of measurement is adequate for timing of the experiments run for this thesis.

Input Delay Time	Job 1 <i>Blank page</i>			Job 2 <i>Full page - simple text</i>			Job 3 <i>vector graphic</i>		
	<u>min</u>	<u>max</u>	<u>% diff *</u>	<u>min</u>	<u>max</u>	<u>% diff *</u>	<u>min</u>	<u>max</u>	<u>% diff *</u>
	← Measured DELAY →								
0									
1	1006	1008	-	1714	1716	-	1712	1714	-
2	←2006→		-	3536	3538	-	←3536→		-
3	3004	3006	-	5156	5158	-	4904	4906	-
5	5004	5006	-	7154	7156		6904	6906	
8	8002	8004	-	←10154→		-	←9904→		-
12	←12000→			←14152→			←13900→		-
16	←15998→		-	18148	18150	-	17900		-
	← Measured PAGE TIME →								
0	7362	7368	.08%	7364	7366	-	←10124→		
1	6354	6358	.06%	5646	5650	.07%	← 9410→		-
2	5356	5360	.07%	5088	5096	.16%	8588	8592	.05%
3	4356	4358	-	4470	4476	.13%	8218	8220	-
5	3512	3526	.40%	4466	4482	.36%	8218	8220	-
8	← 3518→		-	6008	6014	.10%	← 8218→		-
12	5050	5060	.20%	6062	6074	.20%	8214	8218	.05%
16	5104	5114	.20%	6054	6082	.46%	8214	8218	.05%

### Results of Repeatability Experiment

Figure 2.2.2.5

\* Note: if time difference is 2 msec. or less, "% diff" is not shown

### 2.2.3 Print Engine Paper Ejection

Once the RIP has completed generating the bit-map page, the print engine main drive motor is turned on to start the actual page printing process. The Canon LBP-CX print engine page rate is 8 pages per minute. The 8.5" x 11" page is printed lengthwise. That is the RIP writes each scan line in the 8.5" direction. The Canon LBP-CX print engine employs a real-time serial interface to facilitate the transfer of the full page bit-map from the RIP to the print engine. To transfer an 8.5" x 11" page at a 300 dot per inch resolution at a rate of 8 pages per minute, the "video" data interface must sustain a bit rate of approximately 1.1 MBits/sec. This number is calculated as follows:

$$\begin{aligned}\text{Bits / page} &\sim (8.5 \text{ inches} * 300 \text{ dots / inch}) * (11 \text{ inches} * 300 \text{ dots / inch}) \\ &= 8,415,000 \text{ bits / page}\end{aligned}$$

$$\begin{aligned}\text{Ave. data rate} &\sim (8,415,000 \text{ bits/page}) * (8 \text{ pages/minute}) / (60 \text{ seconds/minute}) \\ &= 1.122 \text{ MBits / sec.}\end{aligned}$$

These calculated numbers are only estimates and will differ from the actual specified data rate for following reasons:

1. The writeable scan width is not exactly 8.5 inches, nor is the page length exactly 11 inches (note that European A4 paper size is supported which is 8.3" x 11.7").
2. Most RIPs do not print on the outer 0.1 inch border reducing the amount of page bit-map storage required.
3. There are inefficiencies in the transfer of data between scan lines and between pages.

It is assumed that the RIP has hardware to serialize a 16 bit word and transmit this serial stream to the print engine. It is also assumed that no DMA (Direct Memory Access) hardware exists in the RIP. This means the RIP CPU must transfer a word from bit-map page memory to the parallel-to-serial converter word buffer approximately every 14 microseconds. This number is calculated as follows:

$$\text{Ave. data rate} \sim 1.1 \text{ Mbits / sec (from above)}$$

$$\begin{aligned}\text{Time per word} &\sim (16 \text{ bits/word}) / (1.1 \text{ Mbits/sec}) \\ &\sim 14 \text{ microseconds/word}\end{aligned}$$

At a 10 MHz CPU clock rate this turns out to be 140 clock cycles or every 14 microseconds. It is assumed the transfer takes ~ 14 clock cycles or 1.4 microseconds (estimate). A memory to memory word move takes 12 clock cycles if the source (page bit-map) address is kept in an address register that is post-incremented ( $A_n$ ), and if the destination (memory-mapped buffer) address is kept in a second address register ( $A_n$ ) (*reference 27*). Two additional clock cycles are added assuming read and write memory that requires a nominal one wait-state, which brings the total to 14 clock cycles. This means that approximately 10% of the CPU is dedicated to performing the bit-map to print engine writing function.

It is not the intent of this thesis to further evaluate the print engine specific performance, but the RIP performance. Exact numbers to drive the print engine can only be calculated or measured by examining the exact implementation. This is impossible due to the fundamental rule followed throughout all of the experiments run for this thesis: only external examination of the Apple Laser Writer Plus is allowed (i.e. no reverse-engineering).

It was observed that the page ejection time for a single page was approximately 15 seconds for the simplest POSTSCRIPT® page. Once the printer main drive motor was in motion, however, subsequent pages could eject at a continuous 8 page per minute rate. Since only 10% of the CPU time is taken up by the data transfer to the print engine, 90% is free for communications and generation of the next page. Timing the interaction of the RIP with the print engine is further investigated in sections 3.5.1 and 3.5.2.

## **2.3 Programming Tasks**

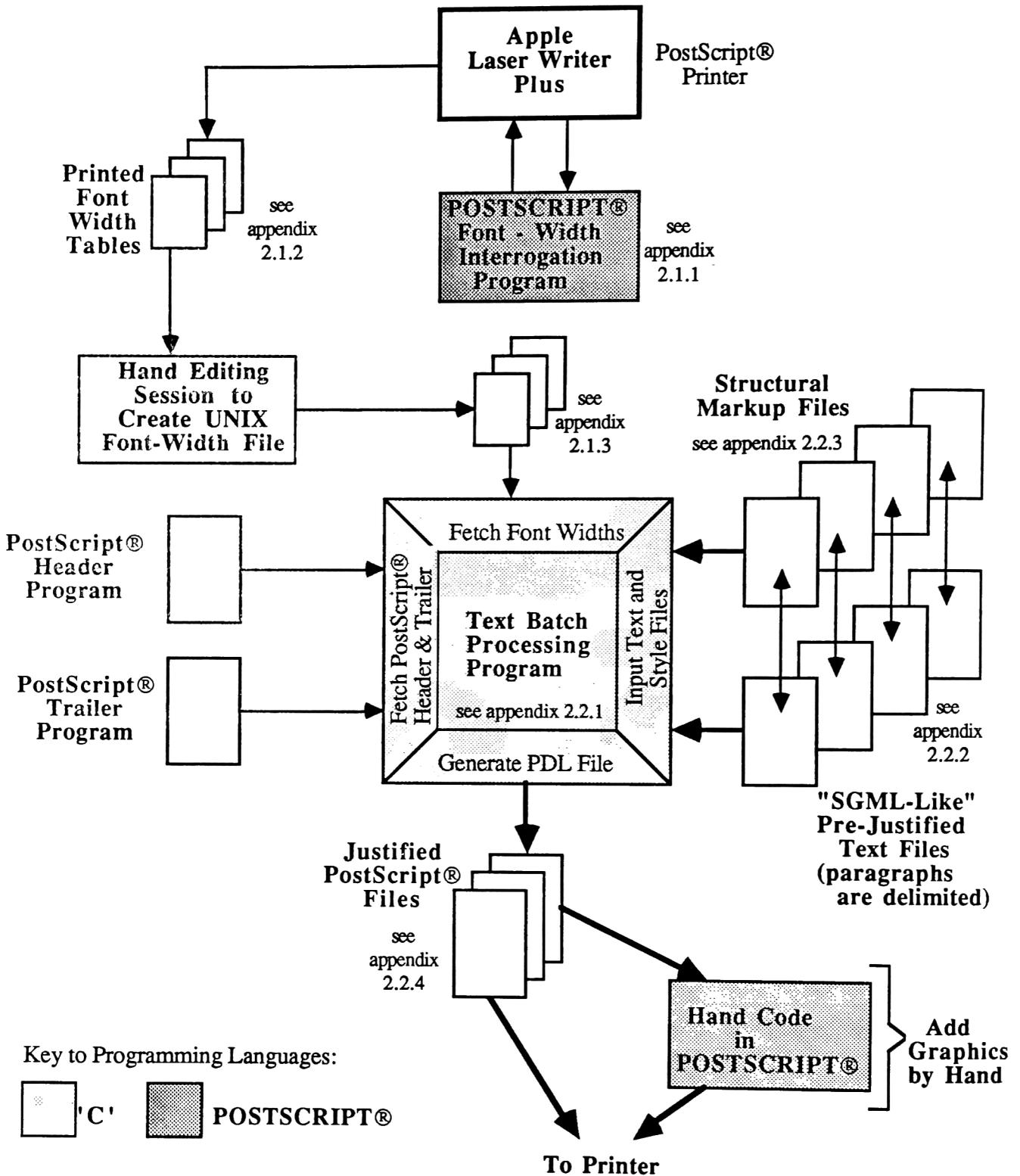
A set of POSTSCRIPT® programs are generated to test the performance of one implementation of POSTSCRIPT®, namely that of the Apple Laser Writer Plus, while printing text in a variety of ways. Several support programs and packages that aid in the generation of these test pages are described below:

### **2.3.1 Extraction of Font Widths from Apple Laser Writer Plus**

The POSTSCRIPT® program shown in appendix 2.1.1 interrogates the Apple Laser Writer printer to extract three sets of font widths at a font size of 10 points. The numbers extracted are real numbers, in points, representing the font width of each printable character specified in strings "st1" and "st2". These widths are needed by the Batch Composition Program, described in section 2.3.2 below, which implements a simple line break algorithm.

Figure 2.3.1 shows how the POSTSCRIPT® Font-Width Interrogation Program fits into the task of automatically generating POSTSCRIPT® text pages. The program shown in appendix 2.1.1 requests the Times Roman™, Times Italic, and Times Bold fonts. Similar programs were written to get the other four fonts required of the Batch Composition Program.

Appendix 2.1.2 shows the page of font widths printed on the Apple Laser Writer Plus when sent the program shown in appendix 2.1.1. In order to put the font width information into a file that the Batch Composition Program could use, a Unix file was hand created using the "vi" editor on a Sun workstation. The format of this file, consisting of (character,width) pairs, is shown in appendix 2.1.3.



Logical Flow to Generate Benchmark PostScript® Pages

Figure 2.3.1

### 2.3.2 Batch Composition Program

A batch composition program, written in C, generates a POSTSCRIPT® file using two SGML-like files for input. Appendix 2.2.2 shows one type of an input file that contains the printable text with beginning and end of paragraph markers, <p> and </p> respectively. The second type of an input file, shown in appendix 2.2.3, contains "structure" markup information. Each line of the "structure" file of appendix 2.2.3 corresponds to one paragraph of appendix 2.2.2. The parameters that can be set are:

- Font typeface (number corresponding to a name) and font size in points;
- Leading multiplier (1.0 indicates that the vertical spacing is equal to the font size);
- Width of the printed justified line, in inches;
- Justification method (one of 6 possible types, corresponding to section 3.3);
- (x,y) Cursor position, in points, at which the first character is printed.

In the example shown in appendix 2.2.3, structure commands for nine paragraphs are specified. There are nine lines of commands contained within the < and > symbols, corresponding to exactly nine paragraphs in appendix 2.2.2, which are delimited by the pair <p> and </p>. The structure commands of the first paragraph are:

- F 1 9.0** Font number 1 (Times Roman); 9 point
- L 1.2** Leading Multiplier of 1.2 meaning the vertical spacing from line to line is 1.2 times the size of the font (9 point). For the first paragraph the distance from the baseline of one line of text to the baseline of the line of text directly below it is 10.8 points (1.2 \* 9).
- W 6.5** Width of the lines of text are limited to 6.5 inches. In other words, the batch composition program inserts a line break after 6.5 inches of text read in.
- J 1** Justification method number 1: simple ragged right utilizing the simple POSTSCRIPT® show command. See section 5.1.3 for the mapping of the other methods that are experimented with.
- C 72.0 720.0** Starting (cursor) position at (x=72 points, y=720 points)  
Since 72 points equals one inch, the point at which the printing starts is one inch from the left edge of the page and 10 inches from the bottom edge of the page (or one inch from the top).

For paragraphs 2 through 9 only font commands are issued. All of the other (non-font related) commands shown above are still in effect until changed. Note that when the font size changes, the vertical line spacing changes along with the size of the font even though the leading multiplier **L** stays constant at 1.2. The command for:

paragraph 2 is **F 1 11.0** == Font 1 (Times Roman); 11 point

paragraph 3 is **F 1 13.0** == Font 1 (Times Roman); 13 point

paragraph 4 is **F 2 9.0** == Font 2 (Times Italic); 9 point

paragraph 5 is **F 2 11.0** == Font 2 (Times Italic); 11 point

paragraph 6 is **F 2 13.0** == Font 2 (Times Italic); 13 point

paragraph 7 is **F 3 9.0** == Font 3 (Times Bold); 9 point

paragraph 8 is **F 3 9.0** == Font 3 (Times Bold); 11 point

paragraph 9 is **F 3 9.0** == Font 3 (Times Bold); 13 point

Appendix 2.2.1 is the *pseudocode* for the C program that generates the POSTSCRIPT® file from these two SGML-like input files. This Batch Composition program calculates how many words, along with their associated interword spaces, fit within the line width specified in the "Markup File". In POSTSCRIPT® fonts and their widths are linearly scaled. So the Batch Composition program uses the font widths shown in Appendix 2.1.3 and scales these widths appropriately (i.e. linearly). The POSTSCRIPT® file generated from the input files of appendices 2.2.2 and 2.2.3 is shown in Appendix 2.2.4, with the actual Apple Laser Writer Plus printer output shown in Appendix 2.2.5.

### 2.3.3 Downloadable POSTSCRIPT® Bitmap Font Program

The program shown in figure 2.3.3.1 is a part of one that is listed in the POSTSCRIPT® Language Tutorial and Cookbook (*Reference 17*). Two small changes were made to make this example and the experiment of section 3.1.1 simpler. First, the number of characters defined was limited to ten: only ten of the twenty-five characters (plus a special "not defined" blank character) were specified in the encoding vector, and similarly bitmaps for the same ten characters were defined when building the **CharData** dictionary. Second, the string to be printed consisted of only one occurrence of each of the ten characters instead of five lines (one sentence) of text. In other words, all ten characters were printed exactly once.

A user-defined font must contain the following entries:

**FontMatrix** transforms the character coordinate system into the user coordinate system. In the program shown in figure 2.3.3.1 it is the unity matrix.

**FontType** must be the integer "3" for user defined fonts.

**FontBBox** specifies the (x,y) coordinates for the lower left and upper right corners of an imaginary box that represents the smallest rectangle in which all the characters of the font being defined would simultaneously fit, assuming all of their origins were coincident. In the program shown in figure 2.3.3.1 the lower left corner has the coordinate (-0.16, -0.24), and the upper right corner has the coordinate (1.28, 1.2). Note that later in the program the *imagemaskmatrix*, an input to the *imagemask* operator, is set up to map a 25 pixel character (baseline to baseline) to the character coordinate system which is one unit wide and high. Taking this into account the coordinates shown above map into the following pixel coordinates: lower left corner is (-4, -6); upper right corner is (32, 30). Figure 2.3.3.2 shows this mapping graphically for the character "g" of the user defined font of figure 2.3.3.1.

**Encoding** is a 256 element array which maps the array indices to character names. In the case of printer resident fonts these names, in turn, serve as keys in the **CharStrings** dictionary to reference the (proprietary) executable character drawing programs. In the case of a user defined font, a procedure provided by the user, which must be named **BuildChar**, uses the **Encoding** vector to find the character name which serves as a key into the **CharData** dictionary. Information in this dictionary is used by the **BuildChar** procedure to render the character. More information on the **BuildChar** procedure is given on the following page.

In the program shown in figure 2.3.3.1 the **Encoding** vector maps ASCII character codes to the letters, "a", "b", "c", "d", "e", "f", "g", "h", "i", and "m". All other character codes have been set up to reference the ".notdef" character which does nothing.

**BuildChar** is a procedure provided by the user to render the desired character. The specific **BuildChar** routine shown in figure 2.3.3.1 is described below in detail.

The current font dictionary, whose name is *Bitfont* , and the character code is passed to **BuildChar**.

```

/BuildChar
{ 0 begin                                current dict. is temporarily stored on stack

/char exch def                            character code and
/fontdict exch def                        font dictionary are stored locally

/charname
  fontdict /Encoding get                put the Encoding vector on top of the stack
  char get                               access the character name
  def                                    and store it in "charname"

/charinfo
  fontdict /CharData get                put the CharData dict. on top of the stack
  charname get                           access info associated with "charname"
  def                                    and store it in "charinfo"

/wx charinfo 0 get def                    1st element of "charinfo" array is char width

/charbbox charinfo 1 4
  getinterval def                        next four elements are lower left (x,y) and
                                          upper right (x,y) coordinates of the
                                          individual character bounding box

wx 0 charbbox
  aload pop                              need to call setcachedevice with x and y
  setcachedevice                          cursor advancement values, and the
                                          character bounding box llx, lly, urx, and ury

charinfo 5 get                            width and height of bitmap image for the
charinfo 6 get                            imagemask command

true                                       invert the image (imagemask input)

fontdict /imagemaskmatrix get
  dup 4                                   get x translate from the "charinfo" array and
  charinfo 7 get put                       add it to the imagemaskmatrix
  dup 5                                   get y translate from the "charinfo" array and
  charinfo 8 get put                       add it to the imagemaskmatrix
                                          matrix is used as imagemask input

charinfo 9 1 getinterval                 get the character bitmap
  cvx                                     make it executable and use as the "proc"
imagemask                               for the imagemask command

end                                       make previous dictionary the current one

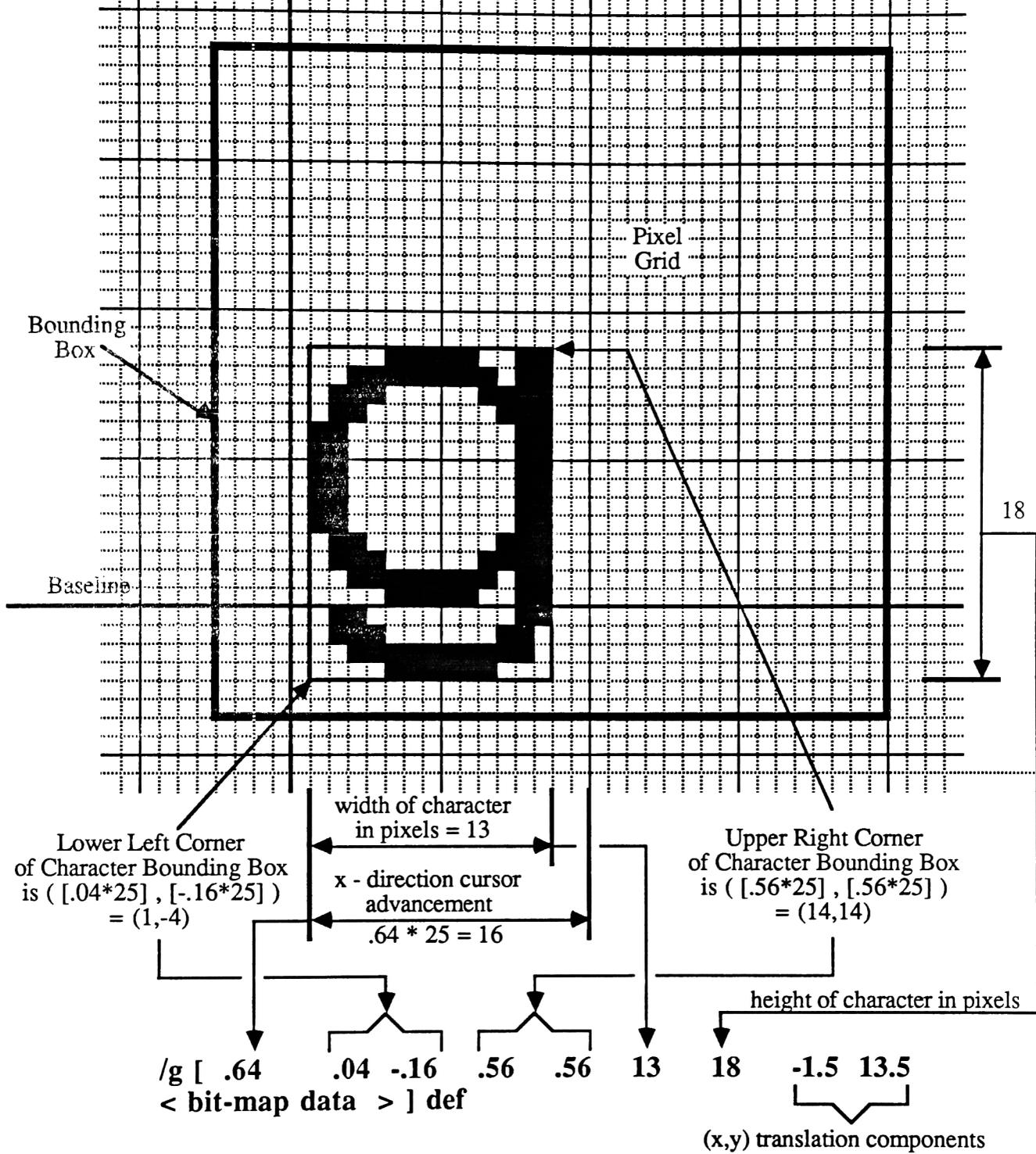
} def                                     end of BuildChar procedure

```



Figure 2.3.3.2 shows one line of the **CharData** dictionary which was put into the **charinfo** variable in the **BuildChar** procedure. The **charinfo** data for "g" is as follows:

.64 (16 pixels)	x direction cursor advancement
(.04, -.16) (1, -4) pixels	lower left corner coordinate of the character bounding box
(.56, .56) (14, 14) pixels	upper right corner coordinate of the character bounding box
13	width of character in pixels
18	height of character in pixels
(-1.5, 13.5)	x and y translation components in pixels



## Character Definition of a PostScript® Bitmap Font

Figure 2.3.3.2

### 2.3.4 Installing Downloadable POSTSCRIPT® Outline Fonts

Provided with the Adobe Stone™ Serif font disk is a program, called *Font Downloader*, that establishes a connection with the Apple LaserWriter Plus via the Appletalk network and downloads the Stone™ Serif outline fonts. This type of *manual* downloading causes the downloaded font to remain in the printer until it is turned off or restarted. Once the font has been downloaded it can be referenced by its assigned name, *StoneSerif* for the Stone Serif font with the following POSTSCRIPT® instructions:

```
/StoneSerif findfont 12 scalefont setfont
```

In addition to the manual method of downloading fonts there is another one, which is called *automatic*. To use the automatic method, first the screen font must be installed with the *Font/DA Mover* program so that the application program can access it for display on the screen. Font metric information is also included in the screen font file and is needed to calculate line and character positioning. Second, the downloadable printer fonts must be placed in the *System* directory. When *Print* is activated through the application program, the printer driver immediately interrogates the printer to get a list of printer resident fonts. If the font selected with the application program is not resident inside the printer, then the printer driver looks for the downloadable font in the *System* directory. If it is found, the font is automatically downloaded. If not, a scaled version of the corresponding screen bitmap font is downloaded instead. This causes a very significant degradation of font image quality. After the POSTSCRIPT® file has finished printing, the downloaded font is deleted from the printer. For this reason the font is said to be *temporary*.

Only the manual download method was used for this paper. The results are reported in sections 3.2.2 and 3.2.3.

## 2.4 Assumed RIP Architectural Model

The hardware and software models of the Apple Laser Writer Plus is presented below. The information was extracted from the references specified (see section 6). Full bibliography entries can be found in section 7.

### RIP Hardware

- MC68000 microprocessor
  - + 7.455 MHz (*Reference 28*)
  
- 1.0 MByte of ROM (Read Only Memory) (*Reference 17*)
  - + printer control routines
  - + Adobe Systems POSTSCRIPT® page description language interpreter (version 42.2)
  - + printer diagnostic routines
  - + printer emulator routines
  - + Appletalk routines
  - + error-reporting routines
  - + outline fonts (35 fonts)
  - + bit-map fonts ( see Pre-cached Bit-maps reference below)
  
- 2.0 MBytes of RAM (Random Access Memory) (*Reference 28*)
  - + Appletalk data buffer (*Ref. 28*)
  - + frame buffer for constructing the bitmapped page image (*Ref. 28*)
  - + font cache buffer for caching character bitmaps (*Ref. 28*)
  - + display list buffer for storing compiled POSTSCRIPT® (*Ref. 28*)
  - + virtual memory - 175KBytes (*Ref. 23*)
  - + downloadable fonts (*Ref. 28*)

### Fonts (*Reference 18*)

All fonts are printed in 14 point. Resident fonts on the Apple LaserWriter and LaserWriter® Plus include:

#### Apple Laser Writer (original 13 faces)

Times ®

Roman

**Bold**

*Italic*

***Bold Italic***

Helvetica ®

Regular

**Bold**

*Oblique*

***Bold Oblique***

Courier

Regular

**Bold**

*Oblique*

***Bold Oblique***

Symbol

â|çð'f©'∧Δ°

New Century Schoolbook

Roman

**Bold**

*Italic*

***Bold Italic***

ITC Bookman ®

Light

*Light Italic*

**Demi**

***Demi Italic***

ITC Avant Garde ®

Book

*Book Oblique*

**Demi**

***Demi Oblique***

Helvetica Narrow

Regular

**Bold**

*Oblique*

***Bold Oblique***

Palatino ®

Roman

**Bold**

*Italic*

***Bold Italic***

*ITC Zapf Chancery* ®

*Medium Italic*

ITC Zapf Dingbats ®

\* \* \* ◆ ● ⚙ □

### Internal resident outline fonts (Reference 19)

The Apple LaserWriter Plus stores fonts in an outline format, and rasterizes individual characters at the desired font, style, size, and orientation when it is specified in the POSTSCRIPT® program

### Font bit-map cache (References 20, 21)

In a POSTSCRIPT® printer, the first time a character appears in a particular font, style, point size and rotation angle, its dot image is stored in a "font cache" (a reserved portion of the printer's memory), so that the next time it appears it doesn't need to be re-imaged. 160 KBytes of the total 1.5 MBytes of RAM is used for font caching.

### Full page bit-map memory (References 22, 23)

The Apple Laser Writer Plus bit-map memory is normally limited to an area of 8 inches by 10.75 inches. This uses up 967,500 bytes, which is 61.5 % of the available 1.5 MBytes of RAM:

**RAM required to image an 8 inch by 10.75 inch area**  
(8 inches \* 300 dots/inch) \* (10.75 \* 300 dots/inch)  
= 7,740,000 bits = **967,500 bytes**

**Percent of the total RAM available**  
967,500 / 1,572,864 = **61.5 %**

It is possible to image a full letter-size page on the LaserWriter Plus, but the memory must be "borrowed" from other functions, such as space for downloadable fonts. To image the entire 8.5 inch by 11 inch area, 1,051,875 bytes of RAM is required, which is 66.9 % of the total 1.5 MBytes. This is an additional 84,375 bytes, or an additional 5.4 % of the total RAM in the system, over the smaller, and generally acceptable, 8 inch by 10.75 inch area.

**RAM required to image an 8.5 inch by 11 inch area**  
(8.5 inches \* 300 dots/inch) \* (11 \* 300 dots/inch)  
= 8,415,000 bits = **1,051,875 bytes**

**Percent of the total RAM available**  
1,051,875 / 1,572,864 = **66.9 %**

**Difference of RAM used for the two imaging areas**  
1,051,875 - 967,500 = 84,375 bytes = **5.4%**

### Pre-cached Bit-maps for selected fonts and sizes (Reference 24)

Times-Roman-12 (*point*), Helvetica-12 (*point*), and Courier-10 (*point*) are the most frequently used fonts in business communications. By having these fonts already bit-mapped in ROM, the page can be assembled much faster than if the fonts had to be reconstructed for each printing job.

### Print Engine Specifications

Canon LBP-CX Laser Beam Printer

- laser-scanning
- xerographic (electrophotographic) printer
- 8 page per minute maximum page rate
- 300 dots per inch resolution



## 3. Analysis of Experimental Results

### 3.1 Font Formats: Outline vs. Bit-Maps

#### 3.1.1 Real Time Outline Font Rasterization and Caching

##### 3.1.1.A. Experimental Results

Two experiments were run according to the procedure described in section 2.2. For each of the experiments two runs of nine pages were printed: the first run immediately followed the printer power-up sequence; the second run followed the first run. The programmed wait times in between pages ranged from 0 seconds (no delay) to 16 seconds. Only the times of the first page of each run is reported in figure 3.1.1.A.3. The program which measures the time to generate the first page is described in section 2.2 and shown in figure 2.2.2.1

Two test programs were used which resulted in the printed pages shown in figures 3.1.1.A.1 and 3.1.1.A.2. The first program prints 96 characters of the 11 point Times Roman® font in two lines. The page this first program prints is shown in figure 3.1.1.A.1. The *Body* of code contains the following sequence of POSTSCRIPT® instructions:

```
Times-Roman findfont 11 scalefont setfont
72 720 moveto
(!"$%&' ... MNOP) show
72 720 moveto
(QRST ... yz{|}~) show
```

The second program, which prints the page shown in figure 3.1.1.A.2, prints the same two lines as above, but in addition prints an additional 41 lines that contain a total of an additional 2,243 characters.

The timing measurements are shown in figure 3.1.1.A.3 in a **bold** typeface. The first column of figure 3.1.1.A.3, with the heading of *96 char.*, is split with two different shades of gray. The entry in the first row, labeled *1st page after power-up*, appearing in the light gray region is the measured time to print the page shown in figure 3.1.1.A.1 the first time after a power-down / power-up sequence. This measured time of **34.860** seconds includes the time to scan convert all of the 96 character outlines, since powering down and powering up the printer clears the bitmap font cache. The entry in the second row, labeled *1st page pre-cached*, appearing in the dark gray region is the measured time to print this same page for the first time of the second run. During the first run all of the characters were scan converted and placed into the font cache, so this second measured time of **5.154** seconds includes only the time to move (BitBLT) all of the 96 character bitmaps from the bitmap font cache to the page buffer.

The second column of figure 3.1.1.A.3, with the heading *2,339 (96 + 2,243) characters*, is split into a white and a gray region. The entry in the first row, labeled *1st page after power-up*, in the white region is the measured time to print the page shown in figure 3.1.1.A.2 after a power-down / power up sequence. This measured time of **35.746** seconds includes the time to scan convert the initial 96 characters, printed in the first two lines of the page, as well as the time to move (BitBLT) the remaining 2,243 characters from font cache containing the newly entered character bitmaps. The entry in the second row, labeled *1st page pre-cached*, appearing in the gray region is the measured time to print this same page for the first time of the second run. This second measured time of **5.946** seconds includes only the time to move all 2,339 characters from the bitmap font cache to the page buffer.

Note that it is impossible to completely isolate the scan conversion time or the BitBLTing time from the overall time of printing a POSTSCRIPT® page. In an attempt to extract a major portion of the printing overhead a very simple page was printed and timed. The job consists of a single character being positioned and printed on the page. The character being printed was pre-cached by running one of the programs discussed earlier. A time of **5.124** seconds was measured to print this almost blank page. The remaining values shown in figure 3.1.1.A.3 are calculated in the analysis section which follows.

### 3.1.1.B. Analysis

Four methods were used to calculate the speed, in characters per second, that the Apple Laser Writer Plus™ can print from the font cache:

1. Methods one and two are similar. Focus on the first row of figure 3.1.1.A.3 entitled *1st page after power-up*. The entry in the first column, **34.860** seconds, is the total time taken, including overhead, to scan convert and print 96 characters. The entry just to the right of this, in the second column, **35.746** seconds, is the total time taken to do exactly the same task as before, plus move an additional 2,243 characters from font cache to the page buffer. So, the time that can be attributed specifically to moving, or Bit BLTing, the additional 2,243 characters can be calculated by taking the difference of these two values. This yields the value **0.886** seconds and can be found in the third column labeled *BitBLT 2,243 char. from cache - calculated*. Dividing the number of characters that are moved from cache (2,243 characters) by the time needed to move them (0.886 seconds) yields the rate at which the printer can render characters from font cache (2,532 characters per second).
2. Now focus on the second row of figure 3.1.1.A.3 entitled *1st page pre-cached*. The entry in the first column, **5.154** seconds, is the total time taken, including overhead, to BitBLT the 96 printable ASCII characters from font cache. Similar to the case described in (1) above, the entry in the second column, **5.946** seconds, is the time to do the same task plus BitBLT an additional 2,243 characters. Once again, the difference of these numbers representing the time to BitBLT 2,243 characters is calculated and placed in column three. The time to BitBLT the 2,243 characters is **0.792** seconds. Dividing 2,243 by 0.792 yields a BitBLT character rate of 2,832 characters per second.

3. Methods three and four are similar. Focus on the dark gray "L" shaped band in that highlights the *96 char.* column and the *BitBLT 95 char from cache* column. The time to print 95 pre-cached characters is **5.154** seconds. The overhead to print one pre-cached character is **5.124** seconds. Subtracting out this overhead yields **0.030** seconds to BitBLT 95 characters (i.e.  $96 - 1$ ). Dividing these numbers produce a character BitBLT rate of **2,843 characters per second.**
4. Now focus on the light gray "L" shaped band in that highlights the *2,339 (96 + 2,243) characters* column and the *BitBLT 2,338 char from cache* column. The time to print 2,339 pre-cached characters is **5.946** seconds. Subtracting out this overhead yields **0.822** seconds to BitBLT 2,338 characters (i.e.  $2,339 - 1$ ). Dividing these numbers produce a character BitBLT rate of **3,167 characters per second.**

The calculated character BitBLT rates range from 2,532 to 3,167 characters per second. The accuracy of method four is suspect due to the small time value to *BitBLT 95 characters from cache*, 0.030 seconds. In section 2.2 it was shown that the methods used for the experiments throughout this paper can routinely be off, from run to run, by 0.010 second or more (see figure 2.2.2.3). With such a combined change in either the 96-character page print time or the 1-character page print time, the character per second rate would change from 2,375 (+ 0.010 change) to 4,750 (- 0.01. change). Therefore the results produced with method four will not be used. Methods one, two and three produce results within 11% of each other. Since methods two and three produce very similar results (less than 0.4%) and since method two produced a result between the results from methods one and three, the value produced by method two (2,832) will be used for further evaluation.

Now focus on the outermost "L" shaped gray band that highlights the *96 char.* column and the *Scan Convert 96 characters* row. It takes 34.860 seconds to print a 96-character page the first time after a power-down / power-up sequence. This time includes scan converting the 96 characters. When the overhead associated with printing an almost blank page, 5.124 seconds, is subtracted from 34.860 seconds, the difference generated, 29.736 can be specifically attributed to the scan conversion of the 96 character outlines to bitmaps. Dividing 96 characters by the time to scan convert 96 characters yields the character per second **scan conversion rate of 3,228 characters per second.**

In this case, printing from the font cache was 877 times faster than scan converting the original outline. The POSTSCRIPT® Language Reference Manual states that "printing a character that is already in the font cache is typically a thousand times faster than scan converting it from the character description in the font." (*reference 12*) 877 is rather close to the expected value of 1,000.

### 3.1.1.C. Proposed Improvements

Add hardware speedup for the scan conversion.

!"#\$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ[\]^\_`{|}~

Note : The printed areas on this page excluding the region  
bounded by this rectangle is the printed test page output.

96 characters of 11 point Times-Roman®

!"#\$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ[^\\_`{|~abcdefghijklmnopqrstuvwxyz{|}~

**A. IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

**B. PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

**C. EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

**D. AUXILIARY ERASE.** Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

**E. DEVELOPER STATION ASSEMBLY.** The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

**F. SCAVENGER ROLLER.** Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

**G. POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Page of 11 point Times-Roman text; all 96 printable ASCII characters are printed at least once; 43 lines of text; 2,339 non-space characters; 427 inter-word spaces.

	Times-Roman		BitBLT 2,243 char. from cache calculated	char / sec calculated
	96 char.	2,339 (96 + 2,243) characters		
1st page after power up	<b>34.860</b>	<b>35.746</b>	0.886	2,532
1st page pre- cached	<b>5.154</b>	<b>5.946</b>	0.792	2,832
simple 1 char pre-cached	← 5.124 →			
BitBLT 2,338 char. from cache		0.822		2,843
BitBLT 95 char. from cache	0.030			3,167
Scan Convert 96 characters	29.736			3.228

All times are given in seconds.

Measured times are printed in **BOLD**.

Calculated times are printed in normal text.

RS-232 communications was used.

Printing from cache memory is  
877 times faster than scan conversion:

( 2,832 / 3.228 )

### Times-Roman® Printing Timing

#### Figure 3.1.1.A.3

## 3.1.2 Pre-Cached Bit-Maps

### 3.1.2.A. Experimental Results

For each run of this experiment a single page, along with its associated timing page, was printed. The POSTSCRIPT® file specifying these two pages were downloaded to the printer after a printer power-up sequence. Figure 3.1.2.A.1 shows the results of the three runs, each of which prints a full page of a single font. For the first run an 11 point Helvetica® font is used; for the second run a 12 point Helvetica® font is used; and for the third run a 13 point Helvetica® font is used. All three pages were "set" at 12 point. That is the leading, or vertical spacing between lines were all the same. This insured that the three POSTSCRIPT® files were identical except for the variable preceding the *scalefont* operator.

All three runs were printed twice: once immediately following a printer power-down / power-up sequence, and a second time immediately following the first, keeping the printer powered up. The page used for this experiment is identical to the text page shown in appendix 3.2.5 except that one (font & size & orientation) is used instead of the nine used in the appendix.

<u>Run</u>	<u>Right After Power-Up</u>	<u>Second Time</u>
1 (11 point)	20.858 seconds	6.082 seconds
2 (12 point)	6.194 seconds	6.078 seconds
3 (13 point)	21.330 seconds	6.084 seconds

**Figure 3.1.2.A.1**

### 3.1.2.B. Analysis

The 12 point Helvetica® font is one of the designated fonts that is pre-scan converted and permanently resident in the RIP in bit-map form (*Ref. 13*). The experimental results show this to be true. The page of 12 point Helvetica® text printed in 6.194 seconds immediately following a power-up sequence, only 0.110 seconds slower than that recorded for a second run. By comparison, the 11 point and 13 point Helvetica® text pages took much longer to print the first page after power up vs. the second time (about 15 seconds). Clearly this is due to the additional time needed to scan convert these fonts during the POSTSCRIPT® page itself.

### 3.1.2.C. Proposed Improvements

Expand the size of the bit-map font storage ROM;

Strongly advise the user to keep the printer powered up;

Expand the non-volatile memory - possibly disk.

**A. IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

**B. PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

**C. EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

**D. AUXILIARY ERASE.** Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

**E. DEVELOPER STATION ASSEMBLY.** The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

**F. SCAVENGER ROLLER.** Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

**G. POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

**H. REGISTRATION.** While the developed electrostatic image moves around the CORE, a sheet of copy paper is advanced to the REGISTRATION ASSEMBLY (not shown in Figure 1). At precisely the right time, the copy paper is directed into contact with the IMAGE LOOP and its developed image. This aligns the copy paper and the image on the IMAGE LOOP.

**I. TRANSFER CHARGER.** The IMAGE LOOP and copy paper now pass under the TRANSFER CHARGER, which produces a negative charge on the paper surface to attract the positive charged developer toner. This effectively transfers the copy image to the paper.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Full page of 11 point Helvetica®.

## 3.1.3 Background Outline Rasterization

### 3.1.3.A. Experimental Results

For each run of this experiment a single page, along with its associated timing page, was printed. The POSTSCRIPT® file specifying these two pages were downloaded to the printer after a printer power-up sequence. Figure 3.1.3.A shows the results for the case when a full page of exclusively 10 point Helvetica® was printed. The shaded columns show the time waited from the completion of the power up sequence (ie. the header sheet was fully ejected) until the the POSTSCRIPT® file started to download. The white columns show the measured printing time of the first page using the method described in section 2.2.2. The page used for this experiment is identical to the text page shown in figure 3.1.2.A.2. In addition to the 11 point case, 9 and 10 point Helvetica® were also used. The data shown in figure 3.1.3.A refers to the 10 point Helvetica® case.

#### 10 Point Helvetica Case

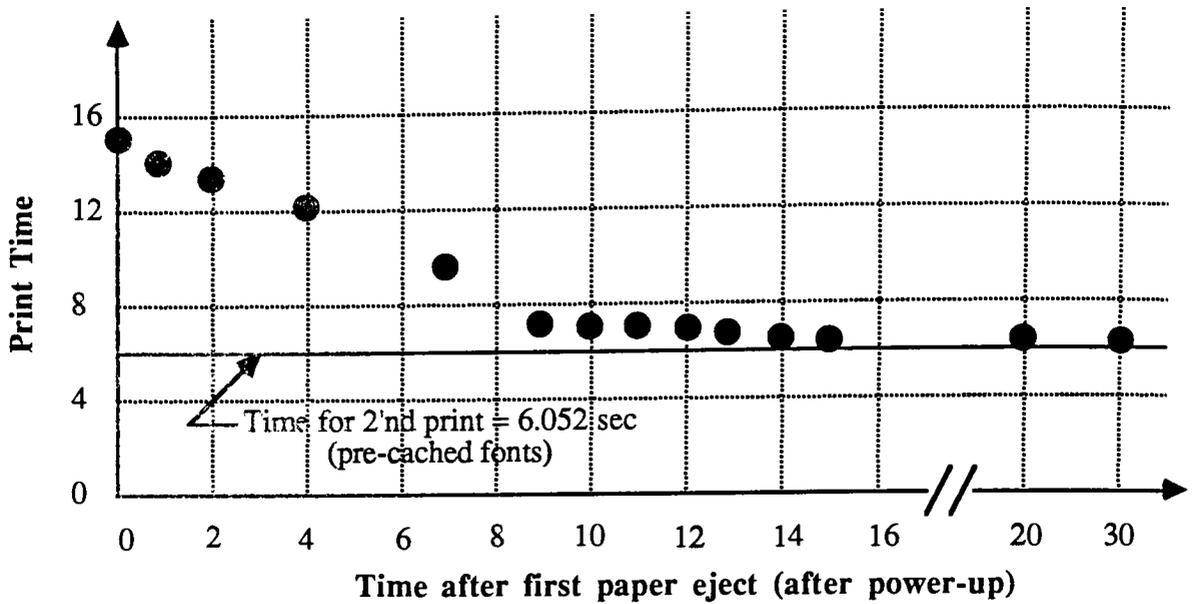
As shown in figure 3.2.3.A the page that was downloaded immediately following the header page ejection (Time After 1'st paper eject = 0 seconds) took the longest to print, 15.888 seconds. As more time was allowed to expire after the power-up sequence was complete the printing times decrease until, at 14 seconds, the printing time appears to reach the lower limit at about 6.65 seconds. The printing time stays at this limit for all of the recorded wait times above 14 seconds up to 30 seconds. Another page was printed without powering down the printer. The second print time, for which all characters have already been cached, took 6.052 seconds, about one half second faster than the limit that was reached.

Two additional small experiments were run. The font specified in the POSTSCRIPT® file was changed from 10 point Helvetica® to 9 point Helvetica® for the first experiment and 11 point Helvetica® for the second experiment. As in 3.2.1 both pages were "set" at the same point size, 10 point in this case, to insure that all the three POSTSCRIPT® files were identical except for the variable preceding the *scalefont* operator. The non-cached printing times were as follows:

9 point printing time	20.110 seconds.
11 point printing time	20.842 seconds.

#### 14 Point Times-Roman Case

The experimental results for the 14 point Times-Roman did not vary as the time delay after the first paper eject varied. Times delays ranged from 0 seconds to over one hour with exactly the same results: a print time of 26.374 seconds. As in the previous case, a second page was also printed without powering down the printer. The second print time took 6.062 seconds.



Time After 1'st paper eject	Print Time (Seconds)	Time After 1'st paper eject	Print Time (Seconds)
0	15.888	11	7.516
1	14.554	12	7.526
2	13.886	13	7.220
4	12.144	14	6.646
7	9.762	15	6.660
9	7.634	20	6.658
10	7.516	30	6.658

## Results of Background Outline Rasterization Experiment

Full Page of 10 Point Helvetica® Text

Figure 3.1.3.A

### 3.1.3.B. Analysis

#### 10 Point Helvetica Case

The 10 point Helvetica® font is one of the designated fonts that is scan converted during idle time (*Reference 13*). The experimental results show the idle time span from 0 to 14 seconds to be the time during which all (or most) of the 10 point Helvetica® outline fonts are scan converted. The scan conversion was most likely already in process by the time the first page was downloaded to the printer (ie. the Time = 0 sec. case). It is assumed that, to scan convert a 10 point Helvetica® page of text, it takes somewhere between the times to scan convert a 9 point page of Helvetica® text and an 11 point page of Helvetica® text. A simple approximation is to average the two, which yields 20.476 seconds. This implies that the scan conversion process began approximately 4.6 seconds before the header sheet was completely ejected:

Ave of 9 & 11 point scan conversion times	20.476 seconds
- 10 point scan conversion from T=0	<u>15.888 seconds</u>
scan conversion time before T=0	4.588 seconds

This is still a rough estimate for the following reasons:

- times to print the 9 and 11 point font pages did require the scan conversion of all of the alphanumeric and common punctuation;
- also, these times include the POSTSCRIPT® interpretation and rendering tasks.

The measured times decrease as the wait time increases. This is exactly the expected results.

Note that between 10 and 12 seconds there was virtually no change in printing time. A possible reason for this is that the remaining characters that still needed scan conversion were scan converted after the 12 second mark. In between 10 and 12 seconds the idle time scan converter worked on characters not needed by the current job.

#### 14 Point Times-Roman® Case

The experimental results indicate that no idle time scan conversion of the 14 point Times-Roman® font is done. It is possible that the information given in the "Red Book" once applied for the Apple Laser Writer™ but not for the Apple Laser Writer Plus™.

### 3.1.3.C. Proposed Improvements

Currently a font is scan converted during the RIP's idle time if:

1. it is one of the default fonts, specified by the printer manufacturer (Apple in this case) or
2. it is explicitly specified via the **setidlefonts** operator by the user.

One possible addition to this scheme would be to automatically scan convert the remainder of the character set of a font previously selected by the POSTSCRIPT® **findfont**, **scalefont**, and **setfont** (and possibly a few other) commands. If some significant number of characters have already been used, then the remaining few characters of the same font could be idle time scan converted. This assumes that more characters of the same font will *probably* be used in subsequent printing jobs. If, however, only one or a few characters were used, this mechanism should not be activated.

## 3.1.4 Effect of Scaling and Rotation

### 3.1.4.A. Experimental Results

The 96 printable ASCII characters were all printed exactly one time using the following POSTSCRIPT® code (or variations of it):

```
/Times-Roman findfont 11 scalefont setfont
```

```
350 300 translate
```

```
70 rotate
```

```
0 75 moveto
```

```
(!"#$%&'()*+,-./0123456789:;) show
```

```
0 50 moveto
```

```
(<=>?@ABCDEFGHIJKLMNOP) show
```

```
0 25 moveto
```

```
(QRSTUVWXYZ[\]^_`|~abcdefg) show
```

```
0 0 moveto
```

```
(hijklmnopqrstuvwxyz{|}~) show
```

Two main variables are modified:

1. The number specifying the font point size is set to either:  
11 (point), as shown above;  
or 22 (point);
2. The number specifying the angle of rotation is set to either  
0 (degrees), for which case the line of code is completely removed;  
70 (degrees) as shown above;  
or 90 (degrees).

The x and y coordinate variables that are input to the **translate** command also need to be modified as the angle changes to assure that all of the text maps inside the page boundaries (8.5 inches by 11 inches). These variables are shown above as **350** and **300** (points).

Six different POSTSCRIPT® programs (two point sizes at three orientations) were run two times (with and without pre-cached fonts). The choices are:

- font is Times-Roman®;
- two font sizes are 11 and 22 point;
- three orientations are 0°, 70°, and 90°.

!"#\$%&'()\*+,-./0123456789:;  
=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ[\]^\_`!~abcde  
fghijklmnopqrstuvwxyz{|}~

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

96 characters of 11 point Times-Roman® rotated 70 degrees

Two tables containing six entries each are shown in figures 3.1.4.A.2 and 3.1.4.A.3. Both tables show the measured time to print the first page of the standard nine page sequence. Note that the page 2 through 9 times were not relevant for this experiment since these pages automatically use pre-cached fonts and also do not have the same startup overhead associated with them. Figure 3.1.4.A.2 gives the measured times immediately after a power-down/power-up sequence. This insures that none of the fonts being printed were in the font cache and therefore all six of these times include the font outline scan conversion times (i.e outline font rasterization). Figure 3.1.4.A.3 gives the measured times for a second run, immediately following the previously described run. This insures that all of the fonts being printed were already resident in the bit-map font cache.

Character Size	← Angle of Rotation →		
	0 Degrees	70 Degrees	90 Degrees
11 point	35.350 seconds	31.356 seconds	36.264 seconds
22 point	40.730 seconds	37.098 seconds	41.442 seconds

First Time After Power-Up (ie. Without Pre-caching)

Figure 3.1.4.A.2

Character Size	← Angle of Rotation →		
	0 Degrees	70 Degrees	90 Degrees
11 point	5.162 seconds	5.172 seconds	5.178 seconds
22 point	5.160 seconds	5.178 seconds	5.180 seconds

First Page Time After Previously Printed Job (ie. With Pre-caching)

Figure 3.1.4.A.3

A blank page was printed and timed. The measured time to print a single blank page was 5.110 seconds. This blank page time is treated as an overhead time that is inherently imbedded in the times of figures 3.1.4.A.2 and 3.1.4.A.3.

Since the measurements for the 11 point - 0 degree case were substantially different than the measurements from a very similar page printed in section 3.1.1 an additional experiment was run. Both the translate and rotate commands were removed. Four lines were still printed. The results are as follows:

First time after power-up: 34.878 seconds

Pre-cached time: 5.160 seconds

### 3.1.4.B. Analysis

The purpose of this section is to compare the outline font rasterization printing performance (i.e. without pre-cached fonts) to the bit-map font printing performance (i.e. with pre-cached fonts) as the character size and orientation changes. This expands on the experiments performed in section 3.1.1 in that two additional variables are introduced into the experiment, namely font size and orientation.

The "Blank Page" time, of 5.110 seconds, is considered to be baseline overhead that any POSTSCRIPT® job has in it. This is close to the "almost blank" page time of section 3.1.1 (5.124 seconds) which printed one pre-cached character. The overhead "Blank Page" time of 5.110 seconds is subtracted from all of the measured times in figures 3.1.4.A.2 and 3, and these results (differences) are recorded in figures 3.1.4.A.5 and 6, respectively. What is left in figures 3.1.4.A.4 and 5 are the times that are more attributable to the scan-conversion and accessing from font cache tasks. This is not entirely true, since the POSTSCRIPT® interpreting also occurs, but this is the lowest level that can be investigated from the outside of the Apple Laser Writer Plus® printer.

Character Size	← Angle of Rotation →		
	0 Degrees	70 Degrees	90 Degrees
11 point	30.240 seconds	26.246 seconds	31.154 seconds
22 point	35.62 seconds	31.988 seconds	36.332 seconds

First Time After Power-Up (ie. Without Pre-caching)

Less Blank Page Overhead

Figure 3.1.4.A.4

Character Size	← Angle of Rotation →		
	0 Degrees	70 Degrees	90 Degrees
11 point	0.052 seconds	0.062 seconds	0.068 seconds
22 point	0.050 seconds	0.068 seconds	0.070 seconds

First Page Time After Previously Printed Job (ie. With Pre-caching)

Less Blank Page Overhead

Figure 3.1.4.A.5

### Scaling Outline Fonts

Figure 3.1.4.A.4 shows that it takes approximately 20 % longer to scan convert the 22 point font when compared to the 11 point font:

- at 0°, 5.380 seconds more or 17.7 %
- at 70°, 5.742 seconds more or 21.9 %
- at 90°, 5.178 seconds more or 16.6 %

It also appears that the performance of scaling a rotated font not on a quadrant boundary is slightly more affected by size variation than for one that is on a quadrant boundary (i.e. about 4% for the 70° case over the other two cases). This difference may be due to the font "intelligence" that is used for the 0° and 90° cases (see Rotating Outline Fonts below). The "intelligence" algorithm may be less size dependent and therefore did not slow down as much as the 70° case, for which font intelligence is not used.

It was expected to take 1 to 4 times longer to scan convert the 22 point font vs. the 11 point font. The experimental data shows it taking 1.2 times longer. This implies that the task of applying the transform to the outline is dominant over the rendering task.

Once the font is in the bit-map cache it seems that the time to pull the bit-map characters out of cache and onto the page is not dependent on font size. The numbers may be too small to see a definitive trend. See the following page for a discussion on the accuracy of the numbers in figure 3.1.4.A.6

### Font Outline Scan Conversion vs. Pre-Cached Bit-Map Fonts

Figures 3.1.4.A.4 and 5 contain the measured times to scan convert and print from cache, respectively, less the overhead of printing a blank page. Figure 3.1.4.A.6 shows the ratios of the entries in figure 3.1.4.A.4 as compared to figure 3.1.4.A.5. That is, figure 3.1.4.A.6 shows how much faster printing from font cache is vs. printing from outlines that have yet to be scan converted.

Character Size	← Angle of Rotation →		
	0 Degrees	70 Degrees	90 Degrees
11 point	582	423	458
22 point	712	470	519

Ratio of Scan Converted Font Timing to Bit-mapped Font Timing

Figure 3.1.4.A.6

The ratios range from 423 to 712, with the average being 527. This is about one half of the expected 1000 and substantially different than the results of section 3.1.1, which yielded a ratio of 877. The accuracy of these numbers should not be taken as absolutes, but merely as a general approximation for the following reasons:

1. The times recorded in figure 3.1.4.A.5 are differences, and therefore are subject to the potential cumulative error of two measured values, that of the printing text from outline or cache, and that of printing the blank page.
2. The numbers shown in figure 3.1.4.A.5 are very small. Since these numbers are so small, small fluctuations (errors) in these values can cause large deviations in the ratios found in figure 3.1.4.A.6. For example, for the (22 point, 0° angle) case a 10 millisecond difference in the 50 millisecond recorded value will make the 712 value range from 594 to 891.
3. As previously mentioned, these numbers are not purely attributable to the scan conversion or printing from font cache processes. The POSTSCRIPT® interpretation was included in printing both types of jobs. The larger this number is, the smaller the ratios in figure 3.1.4.A.6 tend to be.

Comparison to Results in Section 3.1.1

The experiment that was run in section 3.1.1 is very similar to the 11 point, 0 degree case of this section. Both print 96 characters of an 11 point Times-Roman® font in the portrait orientation (i.e. 0 degrees). There are two small differences:

1. the **translate** command was used in this section but not in section 3.1.1;
2. four lines were printed in this section instead of two lines in section 3.3.1 (i.e. two additional **moveto** commands).

In an attempt to isolate the variable that caused the difference of BitBLT to scan conversion ratios (877 vs. 582) the POSTSCRIPT® **translate** command was taken out of the program. The four discrete **moveto** coordinate pairs were changed to position the text in the middle of the page. Also, there was no **rotate** command. Figure 3.1.4.A.7, below shows the results.

	First page time	Pre-cached fonts
with "translate" command	35.350 seconds	5.162 seconds
w/o "translate" command	34.878 seconds	5.160 seconds
section 3.1.1 results (w/o "translate")	34.860 seconds	5.154 seconds

Page Printing Times with and without POSTSCRIPT® "translate" command for the 11 point, 0 degree case

Figure 3.1.4.A.7

It seems that only the first page time is effected by the presence of the **translate** command. When the **translate** command is eliminated from the experiment run in this section the results are very close to the results of section 3.1.1: 34.878 seconds vs. 34.860 seconds, respectively. The additional 0.018 second difference may be due to the additional two **moveto** commands.

One possible explanation for the larger 0.490 second discrepancy (i.e. 35.250 seconds vs. 34.860 seconds) is that the scan conversion software may be sensitive to any modification of the current transformation matrix (CTM). The CTM is changed by the **scale**, **rotate**, and **translate** commands. Once the font is in bitmap form in the font cache, however, the effect seems to disappear.

An even more significant factor that influenced the 877 vs. 582 discrepancy in BitBLT vs. scan conversion time ratios is the times that were used for the "base" page. There was a 0.014 second difference in these numbers which becomes a 28% difference to the 0.050 second value. These values should only be used as general guidelines for RIP performance. Any small change in measured values reflect large ratio differences.

#### Rotating Outline Fonts

The 70° rotated case yielded the best results:

- 13.2 % faster than 11 point at 0° orientation;
- 15.7 % faster than 11 point at 90° orientation;
- 10.2 % faster than 22 point at 0° orientation;
- 12.0 % faster than 22 point at 90° orientation.

This indicates that the font "intelligence" overhead is more dominant than the overhead incurred when rotating a font outline. The 0 and 90 degree cases are both slower due to the font intelligence algorithm used. As expected, the 0° case is slightly faster than the 90° case, although by only about 3%. The font outline scan conversion algorithm evidently takes advantage of knowledge of the 90° rotation for the performance to take such a small hit.

#### 3.1.4.C. Proposed Improvements

A faster processor and/or math accelerator hardware could help in generating the font bitmaps from the master font outlines. Once the font is in bitmap form in the font cache the time to move these bitmaps onto the page is limited by memory speeds. Using faster memory could help. Also, for larger fonts data compression / decompression could help the potential memory speed bottleneck.

### 3.1.5 Variations of Font Style

#### 3.1.5.A. Experimental Results

The experiment of 3.1.1 used the Times-Roman® font. The same experiment was run in this section using two other fonts: Helvetica® and Zapf Chancery ® Medium Italic. The Times-Roman printed pages are shown in figures 3.1.1.A.1 and 3.1.1.A.2, along with the detailed measured and calculated times in figure 3.1.1.A.3. One of the two pages of ZapfChancery® text is shown in figure 3.1.5.A.1. The Helvetica® pages are not shown.

All three fonts are resident in the Apple Laser Writer Plus®. Scan convert times and BitBLT times were measured and calculated in the same manner described in section 3.1.1. The results are shown below:

	Characters per second		Ratio BitBlt / Scan
	Scan Convert	BitBLT from cache	
Helvetica®	3.976	2,797	703
Times-Roman ®	3.228	2,832	877
<i>Zapf Chancery® Medium Italic</i>	2.875	2,868	998

#### 3.1.5.B. Analysis

"Complex fonts ... will take longer to print because of the intricate curves and shapes". (*reference 25*) It can be seen from the data shown above that as the font gets "more complex" the character per second scan conversion rate decreases. The simple Helvetica® style is 38% faster to scan convert than the more complex Zapf Chancery® font. However, once the font has been cached there is only a minimal effect on speed of moving the cached bitmaps to page memory for printing. Consequently, the significant effect that font complexity has on the cache vs. scan converting the font outlines.

Once the font has been placed in the bitmap font cache all three fonts are yielded performance that were vey close to each other. There was a slight difference in speed that seems to corrolate with the apparent size of the fonts:

- the font that appears to be the smallest (i.e. Zapf Chancery®) was the fastest, printing 2,868 characters per second;
- the font that appears to be the second smallest (i.e. Times Roman®) was "in between", printing 2,832 characters per second;
- the font that appears to be the largest (i.e. Helvetica®) was the slowest, printing 2,797 characters per second;

!'#\$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN(OP  
QRSTU VWXYZ[\]^\_`'-abcdefghijklmnopqrstuvwxyz{|}~

*A. IMAGE LOOP.* The *KODAK EKTA*PRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

*B. PRIMARY CHARGER.* The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

*C. EXPOSURE.* The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

*D. AUXILIARY ERASE.* Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

*E. DEVELOPER STATION ASSEMBLY.* The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged *KODAK EKTA*PRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

*F. SCAVENGER ROLLER.* Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

*G. POST-DEVELOPMENT ERASE LAMP.* To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Page of 11 point Zapf Chancery Medium Italic™ text; all 96 printable ASCII characters are printed at least once; 43 lines of text; 2,339 non-space characters; 427 inter-word spaces.

**3.1.5.C. Proposed Improvements**

Same as section 3.1.1.C.

## 3.2 Resident vs. Downloaded Fonts

In section 2.2 the measurement techniques are explained using several experiments to demonstrate the validity of the techniques used. All experiments that were run in section 2.2, as well as in almost all other sections in chapter 3, use the RS-232 communications interface (see section 2.2.1 for more details). Sections 3.2.1 through 3.2.3 are the exception to this rule. In these next three sections Appletalk is used exclusively, instead. The reason Appletalk is used instead of RS-232 is because two (of the three) downloadable fonts were purchased and come in a form that is set up for use with Appletalk. The "Font Downloader" software, provided by Adobe, uses Appletalk. Once these fonts have been downloaded into the Apple Laser Writer Plus printer via Appletalk it is impossible to change the communications mode of the printer without powering it down. Since the downloaded fonts are stored in volatile memory (RAM), all downloaded fonts are effectively erased when the printer is powered down. Therefore all experiments that use a font that has been downloaded with Appletalk must also be run with Appletalk.

In addition to downloading fonts the "Font Downloader" tool also allows a POSTSCRIPT® file to be downloaded via Appletalk. All of the techniques described in section 2.2 are still valid when using Appletalk, including the measurement of download times of POSTSCRIPT® fonts and programs. The one drawback to using Appletalk is that experiments involving measurement of download times cannot easily be translated back to the equivalent times with RS-232. This is because of the uncertainty of the Appletalk protocol overhead and the "shared" nature of the Appletalk network. To address the latter point to some extent the printer is connected point-to-point with the Mac-II providing the POSTSCRIPT® file with no other computers or printers physically connected.

For more information on the "Font Downloader" tool and how it is used to download POSTSCRIPT® fonts as well as POSTSCRIPT® programs refer to section 2.3.4.

### 3.2.1 Downloaded Bitmaps vs Printer Resident (Permanently-Cached) Bitmaps

#### 3.2.1.A. Experimental Procedure and Results

A very simple page of text is printed and timed. The page consists of a single string of ten characters each of which were printed exactly one time. The string is positioned once. The POSTSCRIPT® commands that were used are:

```
200 400 moveto  
(abcdefghim) show
```

Two bitmap fonts are used:

- A. A hand-coded downloadable POSTSCRIPT® bit-map font. The program shown in section 2.3.3 is a downloadable POSTSCRIPT® font that was largely taken from a section in the POSTSCRIPT® Language Tutorial and Cookbook (*see reference 16*). Ten characters of a simple six point font are defined in that program. Unfortunately no six point font was available in permanent bit-map form on the Apple Laser Writer Plus® for this comparison, but a 12 point font was available - a factor of two larger than the six point downloadable bit-map font. For this experiment the bit-map definitions of the ten characters were doubled in size in both the x and the y directions. This was done by hand editing the file. Simple pixel replication was used.

In addition to modifying the bit-maps four other entries needed to be modified:

1. the `imagemaskmatrix` was changed from  
[25 0 0 -25 0 0]  
to  
[50 0 0 -50 0 0] ;
2. the width of each character, in pixels, was doubled;
3. the height of each character, in pixels, was doubled;
4. the y translation component was doubled (+0.5) .

Once the new 12 point font, named *Bitfont* is defined, it is callable by the command:

**`/Bitfont findfont 12 scalefont setfont`**

- B. The printer resident Helvetica® 12 point font. The 12 point Helvetica® font is pre-scanned and permanently resident in ROM, in bitmap form (*reference 13*). Section 3.1.2 of this thesis also discusses the topic of printer resident bit-map fonts. To select the 12 point bit-map font the following simple sequence of POSTSCRIPT® code is used:

**`/Helvetica findfont 12 scalefont setfont`**

The bit-map version of this font is automatically referenced. This command precedes the pair of commands shown earlier which positions and specifies the string to be printed (i.e. the `moveto` and `show` commands).

The size of the bitmap font is:

2,712 bytes: character bitmaps and metric information  
959 bytes: additional overhead  
3,671 bytes total

The measured time to download the bitmap font was 1.044 seconds.

Nine pages were output with varied delays between page printing. The following times were measured:

	<u>Bitfont downloaded</u>	<u>Helvetica resident</u>
1st page	5.580 seconds	5.110 seconds
Min. Time	3.498 seconds	3.510 seconds

abcdefghim

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

10 characters of the 12 point downloaded bitmap font

Figure 3.2.1.A.1

page 83

abcdefghim

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

10 characters of the 12 point printer resident bitmap font (Helvetica®)

### 3.2.1.B. Analysis

The actual size of the bitmap font was slightly smaller than anticipated: 3,671 bytes actual vs. 3,940 bytes expected. The combination of the character bitmaps and the character metrics information was larger 2,712 bytes actual vs. 2,940 bytes expected (2,540 bytes of bitmaps + 400 bytes of metrics). Given the difficulty of predicting exact sizes of ten specific characters, these results look very good.

The measured download time of 1.044 seconds, on Appletalk, is about four times faster than the 4.10 second time predicted for slower RS-232 communications interface. This indicates that, for this particular case (file size, Appletalk loading, etc.) the Appletalk network provided a throughput of approximately 40 KBits / sec. This is about 20 % of its maximum speed.

Two times are recorded above:

**First Time:** The time taken to print the first page. It took 0.470 seconds more to print the first page using the downloaded bitmap font vs. printing the printer resident Helvetica® bitmap font. This does not include font download time, but instead is a measure of the time to move the bitmap font (i.e. a POSTSCRIPT® program defining a font) to the font cache.

**Minimum Time:** Of the nine pages that were printed, this time shows how fast the page can be printed under as close to ideal conditions as possible. All other pages are effected by external influences (see sections 2.1.6 and 3.6). The measured time to print the page with the ten bitmap characters, 3.498 seconds, was only 0.34 % less than the measured time to print the page with the ten printer-resident bitmap characters, 3.510 seconds. This is exactly what was expected. Once a font is in cache memory, whether it originated in the printer or was downloaded, the time to access this font is the same.

### 3.2.1.C. Proposed Improvements

Use a faster communication interface;

Use a communications interface that handles binary information;

Use data compression.

### 3.2.2.A. Experimental Results

The two POSTSCRIPT® test pages that were printed in section 3.1.1 are identical to one set of test pages printed in this section. This set consists of two programs. The first program prints 96 characters of the 11 point Times Roman® font in two lines as described in section 3.1.1. The page this first program prints is shown in figure 3.1.1.A.1. The second program, which prints the page shown in figure 3.1.1.A.2, prints the same two lines as above, but also prints an additional 41 lines that contain a total of 2,243 characters. The only difference in running the experiment in section 3.1.1 and this section is the communications interface used: in section 3.1.1, the serial RS-232 interface is used; in this section, the Appletalk local area network is used.

In addition to the pages described above that use the Times-Roman® font, two more pages were printed for comparison that were exactly the same except for the font used. The two new pages use the Adobe downloaded Stone Serif font. They are shown on the following two pages in figures 3.2.2.A.1 and 3.2.2.A.2.

The data in figure 3.2.2.A.3 shows the experimental results when the four pages were printed. Following the convention set forth in section 3.1.1 all of the measured times are printed in a bold face and all of the the calculated values are printed in a normal weight face. Furthermore, the values that correspond to the pages that use the Stone Serif font are printed on top in the Roman style (Roman and Roman Bold); the values that correspond to the pages that use the Times-Roman® font are printed on the bottom in the Italic style (*Italic* and *Italic Bold* ).

Since Appletalk and the font / program downloader programs were provided, very little control was available to accurately time the font download time. A stop watch was used to measure the time from the program selection (i.e. begin execution) to the return from the program.

It took 4 seconds for the Font Downloader program, provided with the Adobe Stone Serif font, to establish a connection with the Apple LaserWriter Plus with an additional 26 seconds needed to download the font. The size of the outline font was measured to be 34,260 bytes with the *Get Info* utility program on the Macintosh *Finder*. The space this font occupies in the Apple Laser Writer Plus is 28 KBytes, as measured by the *Printer Font Directory* function of the *Font Downloader* program.

### 3.2.2.B. Analysis

The same analysis procedure that was used in section 3.1.1 is used here. For a detailed explanation of this procedure, refer back to section 3.1.1. It will not be repeated here.

As expected the printing performance of the resident and the downloaded fonts were very close:

- scan conversion rate
  - 3.23 characters per second for Stone Serif
  - 3.19 characters per second for Times-Roman®

printing bitmaps from cache

2,776 (2,375 to 2,776) characters per second for Stone Serif

2,825 (2,371 to 2,969) characters per second for Times-Roman®

The download time of 26 seconds includes all of the Appletalk overhead combined with the Font Downloader application program overhead. Assuming the entire 34,260 byte Stone Serif font is actually downloaded in this time, the overall throughput is only 10.5 KBits/sec. As predicted, font download time is quite significant. The time to download the font via this means is almost as long as the time taken to scan convert a full 96 character set of the font itself.

### 3.2.2.C. Proposed Improvements

Optimize the program path that downloads the font. Appletalk has a burst rate of 230.4 KBits/sec. The effective data rate of only 10.5 KBits/sec can probably be improved by optimization.

Use a faster communications interface.

Spool downloadable fonts on a disc to reduce future needs for download.

!"#\$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ[\]^\_`~-abcdefghijklmnopqrstuvwxy  
z{|}~

Note : The printed areas on this page excluding the region  
bounded by this rectangle is the printed test page output.

96 characters of 11 point Stone Serif

!"#\$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN OP  
QRSTUVWXYZ[\]^\_`!~abcdefghijklmnopqrstuvwxyz{|}~

A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

B. PRIMARY CHARGER. The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

C. EXPOSURE. The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

D. AUXILIARY ERASE. Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

E. DEVELOPER STATION ASSEMBLY. The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

F. SCAVENGER ROLLER. Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

G. POST-DEVELOPMENT ERASE LAMP. To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Page of 11 point Stone-Serif text; all 96 printable ASCII characters are printed at least once; 43 lines of text; 2,339 non-space characters; 427 inter-word spaces.

	96 char.	2,339 (96 + 2,243) characters	BitBLT 2,243 char. from cache calculated	char / sec calculated
1st page	<b>36.330</b>	<b>37.188</b>	0.858	2,614
after power up	<i>35.260</i>	<i>36.206</i>	<i>0.946</i>	2,371
1st page	<b>5.158</b>	<b>5.966</b>	0.808	2,776
pre- cached	<i>5.162</i>	<i>5.956</i>	<i>0.794</i>	2,825
simple 1 char pre-cached				
BitBLT 2,338 char. from cache		0.848 0.826		2,756 2,831
BitBLT 95 char. from cache	0.040 <i>0.032</i>			2,375 2,969
Scan Convert 96 characters	31.212 <i>30.130</i>			3.23 3.19

Stone Serif times are printed in:

Roman print for calculated values;

**Bold print for measured values.**

Times-Roman® times are printed in:

*Italic print for calculated values;*

***Bold Italic for measured values.***

All times are given in seconds.

Appletalk was used.

Adobe Stone Serif (Downloaded Outlines)

vs.

Adobe Times-Roman® (Printer Resident)

Figure 3.2.2.A.3

### 3.2.3 Adobe Downloaded Outlines vs. 3rd party

#### 3.2.3.A. Experimental Results

As in sections 3.1.1 and 3.2.2 two sets of two pages each were printed. Two of the four POSTSCRIPT® test pages that were printed in section 3.2.2 are used in this section with no changes. They are the two pages that use the Adobe downloaded Stone Serif font. The first program prints 96 characters of the 11 point Stone Serif font in two lines as described in section 3.1.1 (see figure 3.1.1.A.1). The second program prints the same two lines as above, but also prints an additional 41 lines that contain a total of 2,243 characters. The two pages that correspond to the programs described above are shown in figures 3.2.2.A.1 and 3.2.2.A.2.

In addition to the pages described above that use the Stone Serif font, two more pages were printed for comparison that were exactly the same except for the font used. The two new pages use the CasadyWare downloaded Galileo Roman™ font. They are shown on the following two pages in figures 3.2.3.A.1 and 3.2.3.A.2.

The data in figure 3.2.3.A.3 shows the experimental results when the four pages described above were printed. Following the convention set forth in section 3.1.1 all of the measured times are printed in a bold face and all of the the calculated values are printed in a normal weight face. Furthermore, the values that correspond to the pages that use the Stone Serif font are printed on top in the Roman style (Roman and Roman Bold); the values that correspond to the pages that use the Galileo Roman™ font are printed on the bottom in the Italic style (*Italic* and *Italic Bold* ).

Since Appletalk and the font / program downloader programs were provided, very little control was available to accurately time the font download time. A stop watch was used to measure the time from the program selection (i.e. begin execution) to the return from the program.

It took 4 seconds for the Font Downloader program, provided with the Adobe Stone Serif font, to establish a connection with the Apple LaserWriter Plus® with an additional 26 seconds needed to download the font. The size of the Stone Serif outline font was measured to be 34,260 bytes with the *Get Info* utility program on the Macintosh *Finder*. The space this font occupies in the Apple Laser Writer Plus is 28 KBytes, as measured by the *Printer Font Directory* function of the *Font Downloader* program.

The Adobe Font Downloader program was not able to download the CasadyWare Galileo Roman™ font. The reverse procedure also did not work. That is, the Altsys *Laser Writer Downloader* program (version 1.3), provided with the CasadyWare Galileo Roman™ font, was not able to download the Adobe Stone Serif font. When using the Altsys program to download the Galileo Roman™ font it took a total of about 10 seconds to connect with the printer and download the font. This is three times faster than downloading the Stone Serif font with the Adobe *Font Downloader* program, above. The size of the Galileo Roman™ outline font was measured to be 44,341 bytes with the *Get Info* utility program on the Macintosh *Finder*. The space this font occupies in the Apple Laser Writer Plus is 28 KBytes, as measured by the *Printer Font Directory* function of the Adobe *Font Downloader* program.

!"#\$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ[\]^\_`-abcdefghijklmnopqrstuvwxyz{|}~

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

96 characters of 11 point Galileo Roman™

!"#\$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ[\]^\_`-abcdefghijklmnopqrstuvwxyz{|}~

**A. IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

**B. PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

**C. EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

**D. AUXILIARY ERASE.** Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

**E. DEVELOPER STATION ASSEMBLY.** The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

**F. SCAVENGER ROLLER.** Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

**G. POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increase its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Page of 11 point Galileo Roman™ text; all 96 printable ASCII characters are printed at least once; 43 lines of text; 2,339 non-space characters; 427 inter-word spaces.

	96 char.	2,339 (96 + 2,243) characters	BitBLT 2,243 char. from cache calculated	char / sec calculated
1st page after power up	<b>36.330</b>	<b>37.188</b>	0.858	2,614
	<i>40.528</i>	<i>41.346</i>	<i>0.818</i>	<i>2,742</i>
1st page pre- cached	<b>5.158</b>	<b>5.966</b>	0.808	2,776
	<i>5.168</i>	<i>5.972</i>	<i>0.804</i>	<i>2,790</i>
simple 1 char pre-cached	← <b>5.118</b> →			
		<b>5.130</b>		
BitBLT 2,338 char. from cache		0.848		2,756
		0.842		2,776
BitBLT 95 char. from cache	<b>0.040</b>			<b>2,375</b>
	<i>0.038</i>			<i>2,500</i>
Scan Convert 96 characters	31.212			3.23
	<b>35.398</b>			<b>2.71</b>

Stone Serif times are printed in:

Roman print for calculated values;

**Bold print for measured values.**

All times are given in seconds.

Appletalk was used.

Galileo Roman™ times are printed in:

*Italic print for calculated values;*

***Bold Italic for measured values.***

Adobe Stone Serif (Adobe Downloaded Outlines)

vs.

CasadyWare Galileo Roman (3rd Party Downloaded Outlines)

Figure 3.2.3.A.3

### 3.2.3.B. Analysis

The same analysis procedure that was used in section 3.1.1 is used here. For a detailed explanation of this procedure, refer back to section 3.1.1. It will not be repeated here.

It was expected that scan converting the Adobe Stone Serif font would be slower than scan converting the CasadyWare Galileo Roman™ font because the Adobe algorithm incorporates "intelligence" whereas the CasadyWare algorithm does not. The results were exactly the opposite. Scan conversion times are:

- 3.23 characters per second for Stone Serif;
- 2.71 characters per second for Galileo Roman™.

The two possible reasons why this discrepancy exists are font style complexity and font vendor encoding differences. Although every attempt was made to select a font with a similar complexity to the Stone Serif font, this judgement is very subjective. Some the performance difference may be due to this. Secondly, the font vendor encoding difference, which is the largest unknown in this case, is probably the biggest cause of the performance difference. Whether Adobe downloadable fonts will consistently outperform third party fonts cannot be predicted from this single experiment.

Once the fonts have been downloaded and scan converted (i.e. put into the font cache) the printing bitmaps from cache performance of the two are very close, as expected:

- 2,776 (2,375 to 2,776) characters per second for Stone Serif;
- 2,790 (2,500 to 2,790) characters per second for Galileo Roman™.

The file size of the two downloadable fonts were expected to be close. They were not:

- 34,260 bytes for the Stone Serif font;
- 44,341 bytes for the Galileo Roman™ font.

*(This may indicate higher complexity which supports the earlier "possible reason" for the scan conversion performance discrepancy.)*

Yet the space these fonts occupy in the Apple Laser Writer font memory is the same: 28 KBytes. Compaction may occur at the printer. Or it is possible that not all of the Galileo Roman™ printer font is downloaded.

It was expected that the download times of the two downloadable fonts would be very close to the same. This was the biggest surprise. Even though the Adobe font was smaller by more than 10,000 bytes (34,260 vs. 44,341 bytes) it took three times as long to connect with the printer and download the font (30 vs. 10 seconds). Assuming the entire files were sent to the printer, the overall system throughput was approximately 9,000 bits/sec to download the Stone Serif font with the Adobe *Font Downloader* program vs. approximately 35,000 bits/sec to download the Galileo Roman™ font with the Altsys *Laser Writer Downloader* program. From this comparison it seems that the Altsys font download program utilizes the Appletalk network much more efficiently than Adobe's.

In either case, the font download time is quite significant.

### 3.2.3.C. Proposed Improvements

It seems that using the Altsys program to download the POSTSCRIPT® font is a substantial improvement over using Adobe's download program. Still to be resolved is why it does not work with Adobe fonts.

Use a faster communications interface.

Spool downloadable fonts on a disc to reduce future needs for download.

## 3.2.4 Appletalk vs. RS-232 Comparison

### 3.2.4.A. Experimental Results

The two POSTSCRIPT® test pages that were printed in section 3.1.1 are identical to one set of test pages printed in section 3.2.2. The first program prints 96 characters of the 11 point Times Roman® font in two lines as described in section 3.1.1. The page this first program prints is shown in figure 3.1.1.A.1. The second program, which prints the page shown in figure 3.1.1.A.2, prints the same two lines as above, but also prints an additional 41 lines that contain a total of 2,243 characters. The only difference in running the experiment in section 3.1.1 and 3.2.2 is the communications interface used: in section 3.1.1, the serial RS-232 interface running at 9600 baud is used; in section 3.2.2, the Appletalk local area network running at a burst rate of 230.4 Kbits/second is used. The data in figure 3.2.4.A.1, on the following page, is a reiteration of the experimental results reported in figures 3.1.1.A.3 (for the RS-232 data) and 3.2.2.A.3 (for the Appletalk data). The printer resident Times Roman® font is used for both cases. Following the convention set forth in section 3.1.1 all of the measured times are printed in a bold face and all of the the calculated values are printed in a normal weight face. Furthermore, the values that correspond to the RS-232 case are printed on top in the Roman style (Roman and Roman Bold); the values that correspond to the Appletalk case are printed on the bottom in the Italic style (*Italic* and *Italic Bold*).

Figure 3.2.4.A.2 shows the size of the "bodies" of the two downloaded pages, along with the measured download times of each for both RS-232 and Appletalk communication interfaces. Again, the measured values are reported in bold with the calculated values in normal weight.

The methods used to download these jobs are described below:

RS-232 On the Sun workstation shown in Appendix 1 the UNIX command

```
cat file_to_be_sent > /dev/tty07
```

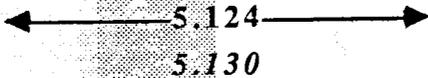
was typed which sends the file *file\_to\_be\_sent* to the printer via port 7 of the 16-port multiple terminal interface.

Appletalk On the Macintosh II personal computer shown in Appendix 1 the program FontDownloader was selected (executed). The file to be downloaded to the printer was then selected and downloaded. This procedure is described in more detail in section 2.3.4.

### 3.2.4.B. Analysis

The measured times to print the first page after a power-down / power-up sequence is the only noticeable difference in the printing time measurements shown in figure 3.2.4.A.1. It takes 0.400 seconds more in the 96 character case and 0.460 seconds more in the full page case for the page to print using Appletalk than RS-232. It appears that the Apple Laser Writer Plus™ processor is using some of its time performing Appletalk functions during the execution time of the POSTSCRIPT® page even though, the point at which these measurements were made, the POSTSCRIPT® page had already been fully downloaded.

The calculated character per second printing rates are very close for both the printing from cache case and the scan conversion case.

	Times-Roman		BitBLT 2,243 char. from cache calculated	char / sec calculated
	96 char.	2,339 (96 + 2,243) characters		
1st page after power up	<b>34.860</b>	<b>35.746</b>	0.886	2,532
	<i>35.260</i>	<i>36.206</i>	<i>0.946</i>	<i>2,371</i>
1st page pre- cached	<b>5.154</b>	<b>5.946</b>	0.792	2,832
	<i>5.162</i>	<i>5.956</i>	<i>0.794</i>	<i>2,825</i>
simple 1 char pre-cached				
BitBLT 2,338 char. from cache		0.822		2,843
		<i>0.826</i>		<i>2,831</i>
BitBLT 95 char. from cache	<b>0.030</b>			3,167
	<i>0.032</i>			<i>2,969</i>
Scan Convert 96 characters	29.736			3.23
	<i>30.130</i>			<i>3.19</i>

RS-232 times are printed in:

Roman print for calculated values;

**Bold print for measured values.**

All times are given in seconds.

Appletalk times are printed in:

*Italic print for calculated values;*

***Bold Italic for measured values.***

Appletalk vs. RS-232 Comparison

Figure 3.2.4.A.1

As shown in figure 3.2.4.A.2 the maximum raw data rate of Appletalk, at 230.4 KBits / second, far exceeds that of RS-232, at 9600 Bits / second. Using these base numbers Appletalk has a raw communications speed that is 24 times faster than the raw data rate of the RS-232 communications interface used. However, the point to point nature of the RS-232 interface and the low protocol overhead (i.e. 2 bits per 8 bit character), make this interface rather efficient (80%). On the other hand, Appletalk is set up to be a local area network and has with it several inherent overheads:

1. several layers of protocol overhead including:
  - the "to" address
  - the "from" address;
  - the type of frame (i.e. information, acknowledgement, etc.);
  - flow control information;
  - error detection via a CRC check.
2. retransmission if an error occurred;
3. contention with other computers and printers sharing the Appletalk network;
4. the requirement to break up large files into packets no larger than 512 bytes.

Due to these factors Appletalk tends to be less efficient. The calculated efficiency values, based on the measured times to download two files of different size, are 20% and 30% for the larger and smaller files respectively. Instead of being 24 times faster than RS-232, Appletalk is only 9 times faster for the 96 character case and 6 times faster for the larger full page of text case. The reason Appletalk gets less efficient as the file size increases is listed as factor #4 above. The 4,322 byte file needs to be "chopped up" into 9 smaller sub-file and sent to the printer through 9 separate packets. The smaller 363 byte file can easily fit in one packet and is subsequently faster. It is not known how badly the throughput of Appletalk degrades as files sizes get much larger.

Size of the downloaded file is a very important factor especially when considering downloadable fonts. Font sizes of the downloaded fonts used in this section 3.2 are as follows:

Stone Serif	34,260 bytes
Galileo Roman™	44,341 bytes

Also note that for this experiment no other computers were active on the Appletalk network. So the factor listed in #3 above did not even come into play.

Still it should be noted that, although Appletalk is "less efficient" than RS-232, it still is shown delivering between 6 and 9 times the throughput of the RS-232 interface for the experiments run in this section.

	96 characters		2,339 (96 + 2,243) characters	
	RS-232	Appletalk	RS-232	Appletalk
Download Page Size	← [ 363 Characters 2,904 Bits ] →		← [ 4,322 Characters 34,567 Bits ] →	
Measured Download Time	<b>378 msec</b>	<b>42 msec.</b>	<b>4.502 sec.</b>	<b>765 msec.</b>
Maximum Bit Rate (Bits/Sec)	9600 Bits / Sec.	230,400 Bits / Sec	9600 Bits / Sec	230,400 Bits / Sec
Calculated Bit Rate (Bits/Sec)	7,683 Bits / Sec	69,143 Bits / Sec	7,678 Bits / Sec	45,186 Bits / Sec
% Comm Protocol Efficiency	80 % Efficiency	30 % Efficiency	80 % Efficiency	20 % Efficiency
% Comm. Protocol Overhead	20 % Overhead	70 % Overhead	20 % Overhead	80 % Overhead

Measured values are printed in Bold print.

Calculated values are printed in Roman print.

### Appletalk vs. RS-232 Download Times

Figure 3.2.4.A.2

### 3.2.4.C. Proposed Improvements

Faster local area networks, like Ethernet at 10 MBits/seconds could improve the communications bottleneck present with large files (fonts, images, etc.). The cost of Ethernet may still be a drawback to this improvement.

Faster dedicated point-to-point communications interfaces, like a faster RS-232 running at 19.2 K or higher, or a Centronics parallel interface running at more than 100 KBits / second.

An architecture that supports spooling input files as fast as the communications interface allows with faster rates available from the internal disc (i.e. SCSI interface).

### 3.3 Methods of Printing Strings

#### 3.3.A. Experimental Results

Strings of characters can be placed on a given page in a number of ways. Each method outlined in this section will be used to print the pages shown in figures 3.3.A.1 through 3.3.A.3.

For the following examples please note that words in **bold** are POSTSCRIPT® tokens; all other variables, numbers, and words are parameters that influence the effect of the command. Note that all pages use a the Times-Roman font set at 10 point. All times given assume the font has already been scan converted

##### A. Simple unjustified text (i.e. "ragged right")

```
72.00 710.00 moveto
(A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP is a ....)show
72.00 700.00 moveto
(electrically charged, and is sensitive to direct light. The IMAGE LOOP...)show
:
:
72.00 280.00 moveto
(effectively transfers the copy image to the paper. )show
```

*Note: The printed page produced with this method is shown in figure 3.3.A.1.*

##### B. Justified text with the front end application providing the extra incremental spacing to be applied between adjacent characters.

```
72.00 710.00 moveto
0.054 0(A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP...) ashow
```

for the above line, the characters are spaced proportionally with 0.054 points added to relative x increment to every character; note that "0" is the y increment.

```
72.00 700.00 moveto
0.044 0(electrically charged, and is sensitive to direct light. The IMAGE...)ashow
:
:
72.00 290.00 moveto
0.356 0(which produces a negative charge on the paper surface to attract...)ashow
72.00 280.00 moveto
(effectively transfers the copy image to the paper. )show
```

*Note: The printed page produced with this method is shown in figure 3.3.A.2.*

**A. IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

**B. PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

**C. EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

**D. AUXILIARY ERASE.** Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

**E. DEVELOPER STATION ASSEMBLY.** The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

**F. SCAVENGER ROLLER.** Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

**G. POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

**H. REGISTRATION.** While the developed electrostatic image moves around the CORE, a sheet of copy paper is advanced to the REGISTRATION ASSEMBLY (not shown in Figure 1). At precisely the right time, the copy paper is directed into contact with the IMAGE LOOP and its developed image. This aligns the copy paper and the image on the IMAGE LOOP.

**I. TRANSFER CHARGER.** The IMAGE LOOP and copy paper now pass under the TRANSFER CHARGER, which produces a negative charge on the paper surface to attract the positive charged developer toner. This effectively transfers the copy image to the paper.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Simple Unjustified Text (i.e. "ragged right")

**A. IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

**B. PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

**C. EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

**D. AUXILIARY ERASE.** Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

**E. DEVELOPER STATION ASSEMBLY.** The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

**F. SCAVENGER ROLLER.** Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

**G. POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increase its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

**H. REGISTRATION.** While the developed electrostatic image moves around the CORE, a sheet of copy paper is advanced to the REGISTRATION ASSEMBLY (not shown in Figure 1). At precisely the right time, the copy paper is directed into contact with the IMAGE LOOP and its developed image. This aligns the copy paper and the image on the IMAGE LOOP.

**I. TRANSFER CHARGER.** The IMAGE LOOP and copy paper now pass under the TRANSFER CHARGER, which produces a negative charge on the paper surface to attract the positive charged developer toner. This effectively transfers the copy image to the paper.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Justified text with the front end application providing the extra incremental spacing to be applied between adjacent characters.

C. Justified text with the front end application providing the extra incremental spacing to be applied between adjacent words.

72.00 710.00 moveto

0.293 0 32 (A. IMAGE LOOP. The KODAK EKTAPRINT IMA...)widthshow

for the line above, the characters are spaced proportionally with 0.293 points added to relative x increment to every character with an ASCII value of "32" (i.e. a space); note that "0" is the y increment.

72.00 700.00 moveto

0.263 0 32 (electrically charged, and is sensitive to direct light. Th...)widthshow

⋮

72.00 290.00 moveto

2.242 0 32 (which produces a negative charge on the paper surface...)widthshow

72.00 280.00 moveto

(effectively transfers the copy image to the paper. )show

*Note: The printed page produced with this method is shown in figure 3.3.A.3.*

A. **IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

B. **PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

C. **EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

D. **AUXILIARY ERASE.** Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

E. **DEVELOPER STATION ASSEMBLY.** The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

F. **SCAVENGER ROLLER.** Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

G. **POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

H. **REGISTRATION.** While the developed electrostatic image moves around the CORE, a sheet of copy paper is advanced to the REGISTRATION ASSEMBLY (not shown in Figure 1). At precisely the right time, the copy paper is directed into contact with the IMAGE LOOP and its developed image. This aligns the copy paper and the image on the IMAGE LOOP.

I. **TRANSFER CHARGER.** The IMAGE LOOP and copy paper now pass under the TRANSFER CHARGER, which produces a negative charge on the paper surface to attract the positive charged developer toner. This effectively transfers the copy image to the paper.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Justified text with the front end application providing the extra incremental spacing to be applied between adjacent words.

D. Justified text with the downloaded POSTSCRIPT® program calculating excess space and distributing it between all adjacent characters. The number of characters in each line is passed to the "js" routine.

<b>/xlen 468 def</b>	width of line set at 468 points or 6.5 inches
<b>/js {</b>	"js" - justify string procedure name: assumes a string to be printed is on top of the stack
<b>/sl exch def</b>	"sl" is a variable that is assigned the length of the string on the stack in an integer number of characters, including spaces
<b>dup stringwidth pop /sw exch def</b>	"sw" is a variable that is assigned the width of the string on the stack in current units; if no <b>scale</b> command was issued, then the current unit is the "point"
<b>/s exch def</b>	"s" is the name of the string on top of the stack
<b>xlen sw sub</b>	the excess space is calculated
<b>sl 1 sub div</b>	and distributed between adjacent characters
<b>0 s ashow</b>	"0" indicates that the incremental character positioning is only in the x direction; the string "s" is printed
<b>} def</b>	

72.00 710.00 **moveto**  
(A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP is a contin...) 99 js  
characters are automatically spaced  
proportionally with the extra space  
evenly distributed between adjacent  
characters

72.00 700.00 **moveto**  
(electrically charged, and is sensitive to direct light. The IMAGE...)108 js  
⋮  
72.00 290.00 **moveto**  
(which produces a negative charge on the paper surface to attract...)108 js  
72.00 280.00 **moveto**  
(effectively transfers the copy image to the paper. )show

*Note: The page produced is the same as the one shown in figure 3.3.A.2.*

- E. Justified text with the downloaded POSTSCRIPT® program calculating the excess space and the number of characters in the string, and distributing the space between all adjacent characters. The number of characters in each line is not passed to the "js" routine, as in (D), but instead is calculated within "js" itself.

<code>/xlen 468 def</code>	width of line set at 468 points or 6.5 inches
<code>/js {</code>	<code>"js"</code> - justify string procedure name: assumes a string to be printed is on top of the stack
<code>dup stringwidth pop /sw exch def</code>	<code>"sw"</code> is a variable that is assigned the width of the string on the stack in current units; if no <code>scale</code> command was issued, then the current unit is the <code>"point"</code>
<code>dup length /sl exch def</code>	<code>"sl"</code> is a variable that is assigned the length of the string on the stack in an integer number of characters, including spaces
<code>/s exch def</code>	<code>"s"</code> is the name of the string on top of the stack
<code>xlen sw sub</code>	the excess space is calculated
<code>sl 1 sub div</code>	and distributed between adjacent characters
<code>0 s ashow</code>	<code>"0"</code> indicates that the incremental character positioning is only in the x direction; the string <code>"s"</code> is printed
<code>} def</code>	

72.00 710.00 moveto

(A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP is a continuo.....)js

72.00 700.00 moveto

(electrically charged, and is sensitive to direct light. The IMAGE .....)js

72.00 290.00 moveto

(which produces a negative charge on the paper surface to attract.....)js

72.00 280.00 moveto

(effectively transfers the copy image to the paper. )show

*Note: The page produced is the same as the one shown in figure 3.3.A.2.*

- F. In addition to the functions performed by the "js" routine in (E), the "js" routine for this section also handles the carriage return - line feed functions.

```

/x 72 def
/y 730 def                                starting (x,y) coordinates

/xlen 468 def                              width of line = 6.5 inches

/D1 {x y yneg sub dup /y exch def moveto} def    single line feed
/D2 {x y yneg 2 mul sub dup /y exch def moveto} def  double " "
                                                    carriage return - line feed

/js{dup stringwidth pop /sw exch def dup length /sl exch def
  /s exch def xlen sw sub sl 1 sub div 0 s ashow D1}def
                                                    same as "js routine in (E) plus
                                                    line feed - carriage return

/yneg 10 def                                leading

```

D1

(A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP is a continu....)js  
 (electrically charged, and is sensitive to direct light. The IMAGE LOOP....)js

(which produces a negative charge on the paper surface to attract the....)js  
 (effectively transfers the copy image to the paper. )show

The fastest measured print time (of the nine pages per run) is used for comparison:

<u>Case</u>	<u>Print Time</u>
A	4.460 seconds
B	4.514 seconds
C	4.538 seconds
D	4.872 seconds
E	4.870 seconds
F	4.926 seconds

### 3.3.B. Analysis

As the experiments progress from case A through case F the POSTSCRIPT® RIP is progressively handling more of the job:

- A. All positioning information is supplied by the POSTSCRIPT® program; characters are positioned according to their normal proportional spacing rules; no adjustment is made for justification. The show command is used.

- B. Positioning information is still supplied by the POSTSCRIPT® program, as in (A), but now the position of every character of each printed string is adjusted by an amount that is also specified by the POSTSCRIPT® program. This is to achieve full justification of the text. The **ashow** command is used.
- C. Once again, the positioning information is supplied by the POSTSCRIPT® program, as in both (A) and (B). Now, instead of adjusting the position of every character, as in (B), only the position of characters that have the ASCII code represented by the decimal number "32" (i.e. the "space" character) will be adjusted. The amount of adjustment is provided by the POSTSCRIPT® program, same as in (B). This case is more difficult than (B) in that the RIP needs to check each character and only adjust the positioning of one particular character, namely the space character in this case.
- D. The positioning information is still supplied by the POSTSCRIPT® program. A separate routine, called **js**, calculates the length of the string passed to it, subtracts the calculated width from the preassigned width of the line held in **xlen**, divides this number by the number of characters in the string (less one) which is also supplied to the **js** program, and uses this result as a parameter for the **ashow** command. Many more calculations are executed here than in cases (A), (B), or (C).
- E. This case is similar to (D) except the number of characters in the string is also calculated in the **js** routine using the **length** operator, instead of this number being provided to **js**, as in (D).

*Note the time to print cases (D) and (E) are nearly the same. This indicates that the overhead to get the length of the string to be printed using **length** is similar to the overhead incurred by passing the length of the string itself.*

- F. This case is similar to (E) except the "carriage return / line feed" function is automated with programs **D1** and **D2**.

It can be seen that as the complexity of the program to print text increases, the time to print them also increases. The only apparent contradiction to this rule is explained in the italicized print for case (E).

### 3.3.C. Proposed Improvements

Encourage the printer driver developers to write simple POSTSCRIPT® code.

## 3.4 Resolution "Targeting" vs. Total Resolution Independence

### 3.4.A. Experimental Results

Strings of characters can be placed on a given page in a number of ways. Each method outlined in this section will be used to print the pages shown in figures 3.3.A.1 through 3.3.A.3.

For the following examples please note that words in **bold** are POSTSCRIPT® tokens; all other variables, numbers, and words are parameters that influence the effect of the command. Note that all pages use a the Times-Roman font set at 10 point. All times given assume the font has already been scan converted

Simple unjustified text (i.e. "ragged right") with the CTM set to device resolution, 300 units to the inch.

```
72 300 div dup scale
/Times-Roman findfont 3000 72 div scalefont setfont
300 3000 moveto
(A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP is a cont....)show
300 2958 moveto
(electrically charged, and is sensitive to direct light. The IMAGE LOOP...)show

300 1250 moveto
(which produces a negative charge on the paper surface to attract the....)show
300 1208 moveto
(effectively transfers the copy image to the paper. )show
```

*Note: The printed page produced with this method is shown in figure 3.4.A.1.*

The fastest measured print time (of the nine pages per run) is used for comparison:

<u>Case</u>	<u>Print Time</u>	
A	4.460 seconds	the point is used as the positioning unit
"target"	4.458 seconds	the 300 dpi pixel is used as the positioning unit

### 3.4.B. Analysis

Setting up the CTM to directly map user coordinates to print engine pixels did not seem to have an effect on speed nor appearance (compare figure 3.3.A.1 to 3.4.A.1).

### 3.4.C. Proposed Improvements

None.

**A. IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

**B. PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

**C. EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

**D. AUXILIARY ERASE.** Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

**E. DEVELOPER STATION ASSEMBLY.** The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

**F. SCAVENGER ROLLER.** Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

**G. POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

**H. REGISTRATION.** While the developed electrostatic image moves around the CORE, a sheet of copy paper is advanced to the REGISTRATION ASSEMBLY (not shown in Figure 1). At precisely the right time, the copy paper is directed into contact with the IMAGE LOOP and its developed image. This aligns the copy paper and the image on the IMAGE LOOP.

**I. TRANSFER CHARGER.** The IMAGE LOOP and copy paper now pass under the TRANSFER CHARGER, which produces a negative charge on the paper surface to attract the positive charged developer toner. This effectively transfers the copy image to the paper.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.  
Simple Unjustified Text (i.e. "ragged right") with all positioning coordinates given in device units (300 dpi).

## 3.5 Effect of Varied Page Complexity on POSTSCRIPT® Processing Time

### 3.5.1 Effect on the Inter-Page Time Delay Program sec

#### 3.5.1.A. Experimental Results

The sec program, shown below, is measured and analyzed in this section.

```
/sec                                % one second delay under no load
{ usertime /T exch def              % keep track of time actually taken
  {
    1228 {
      373.737 737.373 mul pop
    }repeat
  } repeat
usertime T sub def
} def
```

In each of the five cases the sec program is called seven times. Each time the parameter selecting the intended time delay, in seconds, changes until all seven are completed in the sequence 1, 2, 3, 5, 8, 12, and 16. These are the target delay times shown in the figure below. As the cases progress from one through five, the amount of work that the RIP does outside the sec delay program increases. In case one, nothing is done other than calling the sec program and reporting the measured times in the printed trailer page. In case two, a trivial page routine which prints a blank page is called in between calls to the sec routine. In cases three and four, the page routine includes the printing of a few lines of text at the top and the bottom of the page, respectively. And in case five a full page of text is printed. Detailed descriptions of the five cases follow.

#### Case 1 Experiment:

The sec program is called seven separate times. Each time it is called one of the seven target delay times, shown above, is passed to it. The page procedure is not part of this particular POSTSCRIPT® job and is therefore never called. Only the trailer page with the seven measured delay times is printed. No other showpage instruction is executed. Note that the loop count of 1228 was heuristically generated to produce results as close as possible to the target delay times under this "no load" condition.

```
/sec
{
  sec delay program is shown above
} def

/W1 1 sec
/W2 2 sec
⋮
/W16 16 sec

print trailer page
```

### Case 2 Experiment:

This case is similar to case one with one exception: the `showpage` operator is executed inside the `page` procedure and timed between each pair of sequential calls to the `sec` delay program. This causes blank pages to be printed followed by the trailer page containing the seven measured delay times.

```
/page { showpage } def

/sec
{
  sec delay program is shown on previous page
} def

/Pr
{
  calls and times the sec procedure;
  shown in figure 2.2.2.2 and
  page 3 of appendix 2.2.4
} def

/T2 Pr

/W1 1 sec
/T3 Pr

/W2 2 sec
/T4 Pr
:
/W16 16 sec
/T9 Pr

  print trailer page
```

### Case 3 Experiment:

This case is similar to case two except for the operations executed in the `page` procedure. Thirteen lines of left justified text plus 2 additional line spaces (to separate 3 paragraphs) are "printed" in `page` using the `POSTSCRIPT® show` operator. The text is positioned at the top of the page occupying almost two inches of the page (in the y direction). It starts at 10 inches from the bottom of the page (or 1 inch from the top) and progressing downward to 8.1 inches from the bottom of the page.

```
/page
{
  72 700 moveto ( string 1 ) show
  72 691 moveto ( string 2 ) show
  :
  72 574 moveto ( string 13 ) show
  showpage
} def
```

*The sec and Pr procedures, the calling sequence to these procedures, and the printing of the trailer page are the same as shown in case two above.*

#### Case 4 Experiment:

This case is identical to case three with the exception of the position of the thirteen lines of left justified text printed in page. Here the text is positioned at the bottom of the page instead of the top of the page as in case three. It starts at 2.8 inches from the bottom of the page and progressing downward to 0.9 inches from the bottom of the page.

```
/page
{
    72 190 moveto ( string 1 ) show
    72 181 moveto ( string 2 ) show
        :
    72 64 moveto ( string 13 ) show
    showpage
} def
```

*The sec and Pr procedures, the calling sequence to these procedures, and the printing of the trailer page are the same as shown in case two.*

#### Case 5 Experiment:

This case is identical to case three with the exception of the number of lines printed: 29 (with 8 additional line spaces) instead of 13 (with 2 additional line spaces). It starts at 10 inches from the bottom of the page and progressing downward to 5.2 inches from the bottom of the page.

```
/page
{
    72 700 moveto ( string 1 ) show
    72 691 moveto ( string 2 ) show
        :
    72 376 moveto ( string 29 ) show
    showpage
} def
```

*The sec and Pr procedures, the calling sequence to these procedures, and the printing of the trailer page are the same as shown in case two.*

<u>Target Delay</u> <u>Time</u>	<u>Case1</u>	<u>Case 2</u>	<u>Case 3</u>	<u>Case 4</u>	<u>Case 5</u>
1	1.074	1.006	1.686	1.716	1.716
2	2.020	2.006	2.686	3.540	3.446
3	3.000	3.008	3.684	5.362	4.446
5	4.998	5.004	5.684	7.440	6.444
8	7.998	8.004	8.684	10.438	9.442
12	11.994	12.002	12.682	14.436	13.440
16	15.992	15.998	16.678	18.434	17.438

*all times given in seconds*

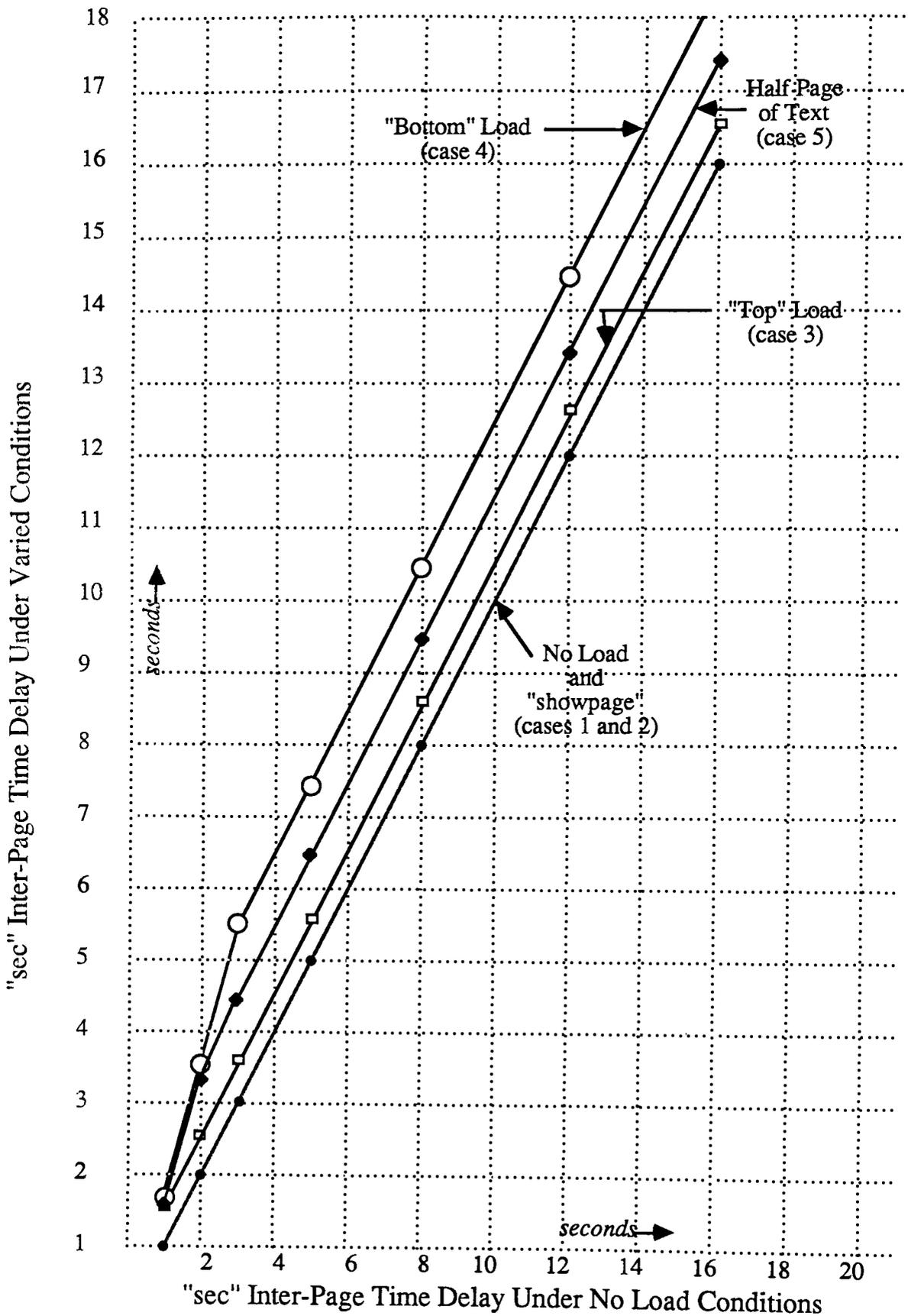


Figure 3.5.1.A.1

### 3.5.1.B. Analysis

Note the correlation between distance from bottom of page that characters are placed and delay time discrepancy.

Top (8.1 inches) <i>case 3</i>	~ .7 sec	for all target delays of 1 sec and up
Middle (5.2 inches) <i>case 5</i>	~ 1.4 sec	for all target delays of 2 sec and up
Bottom (.9 inch) <i>case 4</i>	~ 2.4 sec	for all target delays of 3 sec and up

The time taken to execute sec seems only to be influenced by the position of the mark on the page that is closest to the bottom of the page. Furthermore, this relationship is linear:

Assume  $x$  is the distance of the bottom-most mark on the page and  $y$  is the time discrepancy in seconds.

Using the top and bottom points of (8.1,0.7) and (0.9,2.4) the slope ( $m$ ) and  $y$  intercept ( $b$ ) can be calculated to be:

$$m = -0.236$$

$$b = 2.613$$

Calculating the third point,  $y$ , with the  $x$  value equal to 5.2 yields:

$$y = mx + b = (-0.236) 5.2 + 2.613 = 1.385 \approx 1.4$$

Page complexity does not seem to have any effect: case 5 prints the more than twice as much text as either cases 3 or 4, yet the sec delay program time discrepancy lies in between these two cases.

One possible explanation for this strange behavior is that immediately following the `showpage` command, the POSTSCRIPT® interpreter needs to clear the page buffer memory before the next page is allowed to use the page. The algorithm seems to start from the top of the page and work its way down until it is determined that no more page memory was used. In case 3 this occurs close to the top of the page so this overhead is minimal, but in case 4 this occurs almost at the very bottom of the page, so the overhead is great.

### 3.5.1.C. Proposed Improvements

If a dual page buffer were used with additional hardware to do the necessary memory housecleaning this phenomena should disappear.

## 3.5.2 Effect on the Compilation and Rasterization of Text Pages

### 3.5.2.A. Experimental Results

Experimental cases 3 and 4, described in the previous section, are used here. Sample output for case 3 is shown in figure 3.5.2.A.1, along with its corresponding timing page in figure 3.5.2.A.2. Two observations will be made:

1. The time to execute (i.e. print) the text page from case 3 will be examined as the induced delay between subsequent pages changes.
2. How the data from (1) above changes as the position of the text changes will also be examined. This is done by comparing the measured times of case 4, which prints several lines of text close to the bottom of the page, to the measured times of case 3, which prints the same text close to the top of the page (see figure 3.5.2.A.3).

### 3.5.2.B. Analysis

As shown in figure 3.5.2.A.2 the time to print the very simple page of text (with pre-cached fonts) is 5.542 seconds for the first page. With no induced delay the second page prints in 7.322 seconds. As the induced delay times increase in between pages the time needed to print the next three test pages steadily decreases. With a 3.734 second delay (target delay of 3 seconds) the measured print time is 3.898 seconds. As the delay time increases to 5.736 seconds the measured print time stays constant. This is the point at which the printer motor turns off (as observed by hearing). As the delay times increase from 8 to 16 seconds the print time is again constant at a much higher print time, ~5.45 seconds.

Possible explanations for this behavior is broken into three sections:

1. Print time decreases as interpage delay time increases. During this region printing the previous page "gets in the way" of printing the current page. As more time is allowed between pages, the previous page gets in the way less and therefore the current page waits less time to access the page buffer and therefore takes less time to render the page.
2. Constant minimum time; increasing delay time does not effect print time. Once the point has been reached where enough time between pages is waited the previous page no longer interferes with the printing of the current page, the page prints at its maximum possible speed. As long as nothing else changes, (i.e. status of print engine, etc.) changing the interpage delay times has no effect on the printing times.
3. Second constant time region (after motor shuts off): no effect of increasing delay time. Once the print engine motor shuts off, additional time is required to turn it back on.

A. **IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

B. **PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

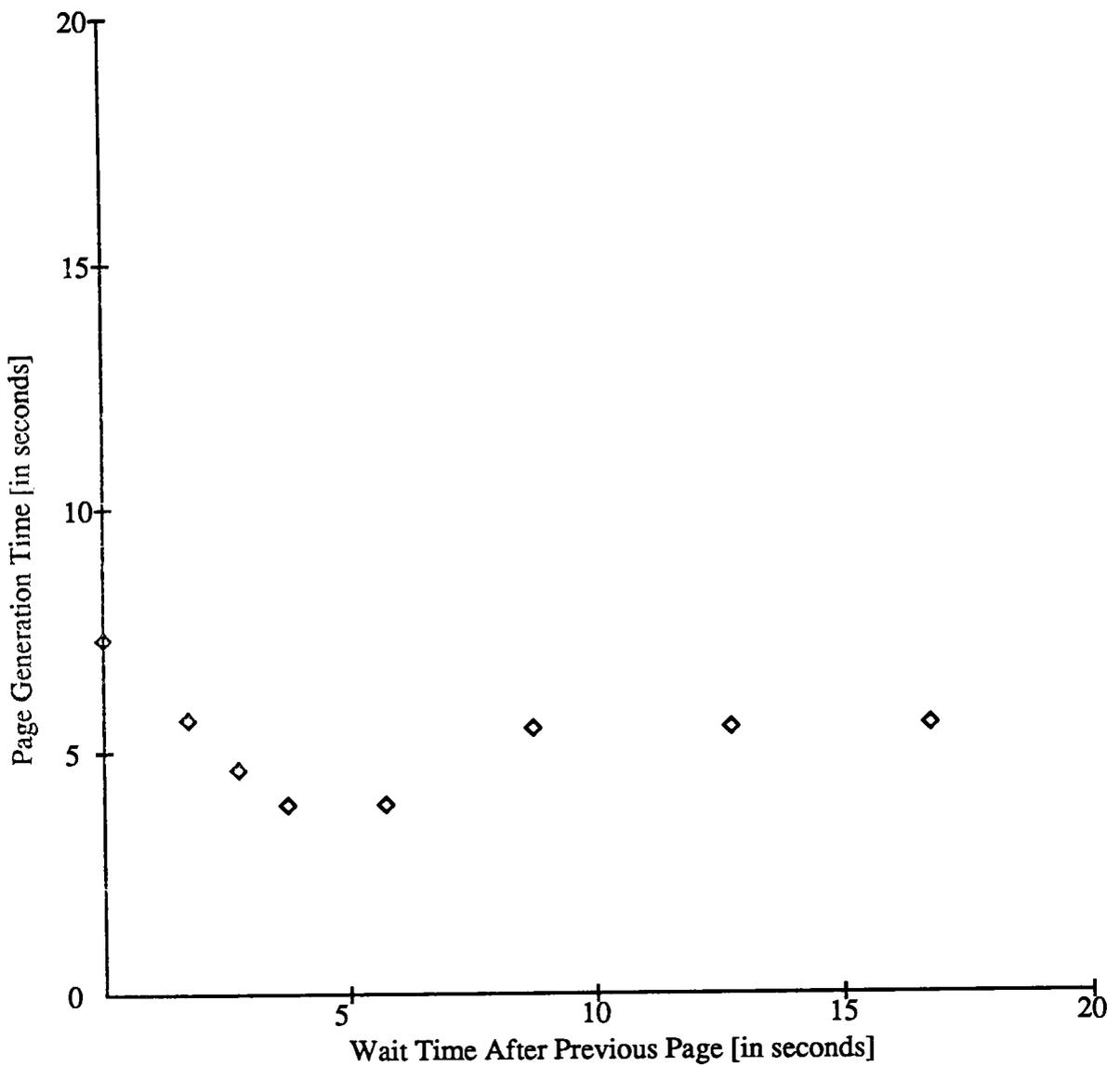
C. **EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

Note : The printed areas on this page excluding the region bounded by this rectangle is the printed test page output.

Case 3 Output: Printing on Top of Page

Figure 3.5.2.A.1

page 119



Download Time = 2350

Wait Time 0 sec = 0  
 Wait Time 1 sec = 1716  
 Wait Time 2 sec = 2738  
 Wait Time 3 sec = 3736  
 Wait Time 5 sec = 5736  
 Wait Time 8 sec = 8734  
 Wait Time 12 sec = 12732  
 Wait Time 16 sec = 16728

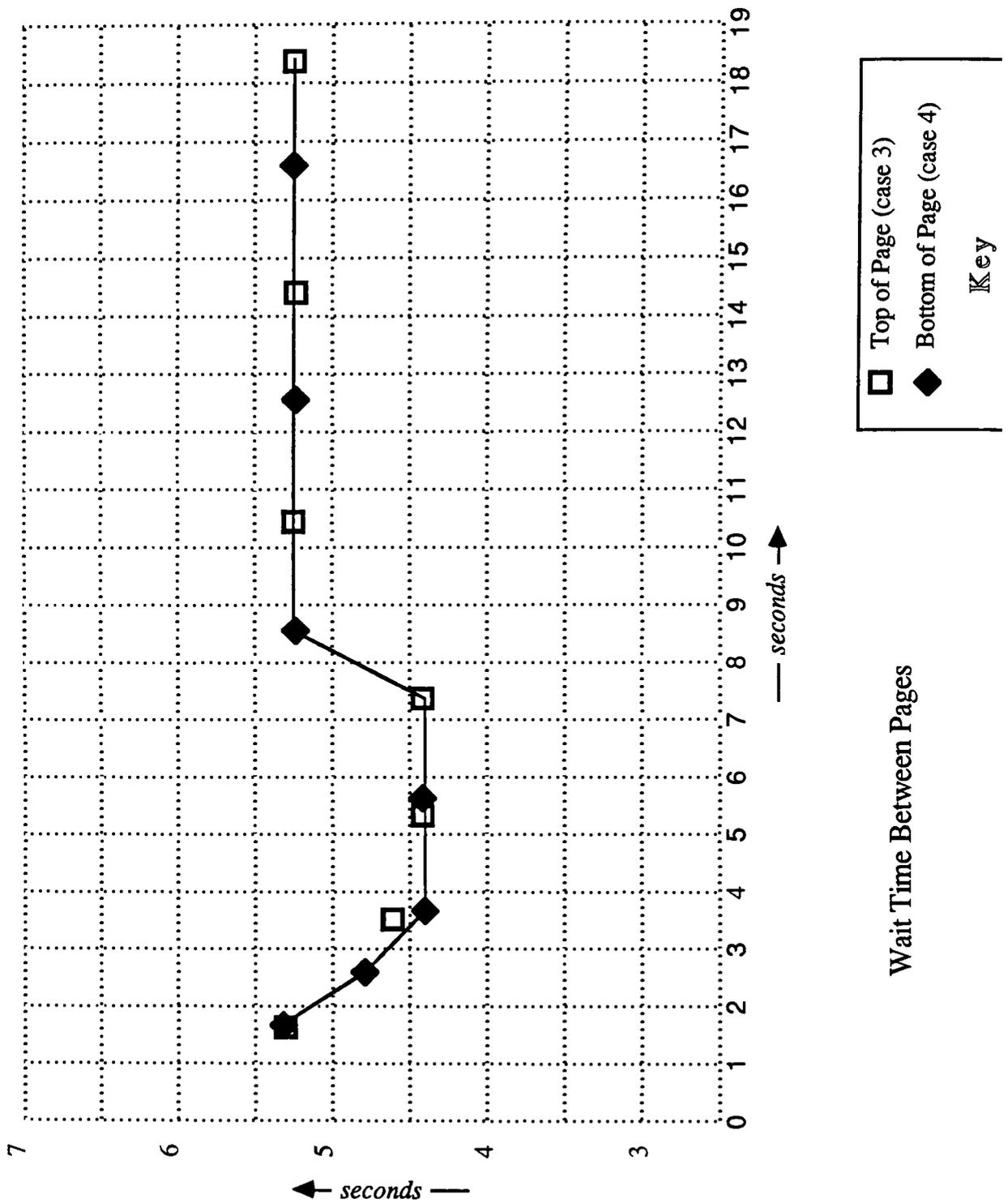
Page Time 1 (A) = 434  
 Page Time 1 (B) = 5532  
 Page Time 2 = 7320  
 Page Time 3 = 5644  
 Page Time 4 = 4626  
 Page Time 5 = 3894  
 Page Time 6 = 3894  
 Page Time 7 = 5430  
 Page Time 8 = 5440  
 Page Time 9 = 5484  
 Ave Pg Time 2-9 = 5216.5

*Note: All times are specified in milliseconds.*

The second part of this analysis deals with comparing printing times when the positions of the text that is placed on the page changes. Data that has been extracted from figure 3.5.2.A.2 for case 3 and a similar timing page for case 4 (not shown) was compiled and put in graph form. The data, shown in figure 3.5.2.A.3, indicates that only variations in the delay time effect the printing times. Region 1 ranges from 1.7 seconds to about 3.7 seconds; region 2 from 3.7 seconds to 7.4 seconds, and region 3 above 7.4 seconds

### 3.5.2.C. Proposed Improvements

Once again, a dual page buffer system with hardware assist would help eliminate the extra time in region 1. Secondly, entrance into region 3 could be delayed or eliminated if the printer were left in a "more ready" state (i.e. without stopping the motor) when the RIP is still processing a print job. On the negative side, more power would be taken up when the print engine itself would be in the idle state.



Page Rasterization Performance Under Varied Wait Times

Figure 3.5.2.A.3

## 4. Conclusions

### 4.1 Scan Converting Outline Fonts vs. Caching Bitmap Fonts

Printing bitmap fonts that are in the bitmap font cache are about 500 times faster than scan converting the corresponding outline font (see section 3.1.4). The character scan conversion or rasterization rate is approximately 3 characters per second (see sections 3.1.1 and 3.1.5). Note that this time includes the POSTSCRIPT® interpretation overhead. Because this is such a dominant factor, several methods are used to increase the probability of a font being in bitmap form when it is specified in a POSTSCRIPT® program:

1. a small set of fonts are stored in ROM in bitmap form (see section 3.1.3). These fonts are specified by the manufacturer of the LaserWriter® Plus Printer, Apple® Computer Inc.
2. an area of RAM memory in the RIP is reserved to serve as a bitmap font cache (see section 2.4). When a character of an outline font is specified in a POSTSCRIPT® program, the time consuming scan conversion program is executed only one time (see section 3.1.1). The rendered bitmap font is stored in bitmap font cache. When the same character is subsequently specified, it is accessed from this font cache.
3. a small set of fonts, which are stored in outline form, are scan converted during the printer's idle time (see section 3.1.3). The newly generated font bitmaps are stored in the bitmap font cache. This process generates certain default fonts at pre-selected sizes and orientations that are specified, once again, by the manufacturer of the printer which, in this case is Apple® Computer Inc. Since much of the time of the printer is idle, this seems to be a very good way to build up a library of bit-map fonts in cache.
4. a process similar to that specified in #3 above allows the user to select which outline fonts are to be scan converted during the printer's idle time (see section 3.1.3).
5. a bitmap font can be downloaded to the printer and stored in the bitmap font cache, eliminating the scan conversion overhead. Unfortunately, the communications overhead can similiary slow performance (see section 3.2.1).

There are several potential ways to increase the performance of the POSTSCRIPT® printer:

1. "encourage" the customer (e.g. person running the application program) to make extensive use of the pre-cached bitmap fonts described in #1 above or the idle time scan converted fonts described in #3 above.
2. "encourage" the customer to use a small number of fonts so that once they are scan converted by one of the means specified above they will be kept in the bitmap font cache for future use at the higher speed.
3. "encourage" the printer driver developers to make extensive use of the user-selectable idle time scan conversion function described in #4 above.

*Note that #1 through #3 are ways to increase performance that are accessible externally to the printer.*

4. use a faster processor and/or math accelerator hardware. In the event that the selected font has not been scan converted, the added hardware should scan convert faster than the 3 character per second rate that was observed.
5. increase the ROM size to store more bitmap fonts. This only helps if the printer manufacturer correctly "guesses" which fonts the customer will use.

6. increase the amount of font cache RAM. This will delay the time at which previously placed bitmap fonts must be cleared out of the bitmap font cache.
7. extend the bitmap font cache with disk memory. Accessing a bitmap font from disk is still faster than scan converting an outline font.
8. make use of data compression for larger fonts (see section 1.4).
9. use a faster and/or more efficient communications interface to download the bitmap (or outline) fonts (see sections 2.2.1 and 3.2.4).

*Note that #4 through #9 are ways to increase performance that require internal changes to the RIP that drives the print engine.*

#### **4.2. Scan Converting Font Outlines with "Quality Hints" Applied**

At a printer resolution of 300 dots per inch, "quality hints" are applied to the outline fonts during the scan conversion process (see sections 2.1.1.4 and 3.1.4). At typical sizes ranging from 6 to 14 point, the scan conversion algorithm uses these hints to insure that a uniform look of the fonts are provided. For example, stems and curves of a given character are forced to be the same dot patterns (e.g. "m"). Also similar characters are forced to have similar dot patterns (e.g. "c" and "o", or "m" and "n").

The experiments reported in section 3.1.4 reveal that these intelligent hints:

1. are only applied if the orientation of the font lies on a quadrant angle (i.e. 0°, 90°, 180°, or 360°). Only the 0° and 90° cases were tested. A font that was rotated 70° did not have the "hints" applied.
2. take additional time to apply. Figure 3.1.A.4 shows both the 0° and 90° cases taking 12 % to 16 % longer to scan convert as compared to the 70° case. This indicates that the overhead of applying the hints was greater than the overhead of rotating the font set to a non-quadrant angle.

As the industry progresses to higher resolutions in the future, the issue of needing these font quality hints will probably fade away.

#### **4.3 Font Style Complexity**

The performance of scan converting a font outline varies with the complexity of the font (see section 3.1.5). The simple Helvetica® font style was measured to scan convert 38 % faster than the more complex Zapf Chancery® font style. Performance of scan converting the Times Roman® font was in between the simple Helvetica® and the complex Zapf Chancery® font styles, as could be expected.

**Fastest** Helvetica® scan converted at a rate of 3.976 characters per second.

**Medium** Times Roman® scan converted at a rate of 3.228 characters per second.

**Slowest** Zapf Chancery® scan converted at a rate of 2.875 characters per second.

#### 4.4 Downloading Fonts

The key parameter that influences the performance of downloaded fonts is the download process itself. The time to download a font is influenced by:

1. the size of the font;
2. the speed of the communications interface used;
3. the efficiency of the communications interface used;
4. the efficiency of the application program and communications interface driver software.

One surprising result that was found in section 3.2.3 shows point #4 above to be a very significant factor. When the Adobe Stone Serif font was downloaded using Adobe's *Font Downloader* program, it took about 30 seconds to download the 34 KByte font. When the CasadyWare™ Galileo Roman font was downloaded using the Altsys *Laser Writer Downloader* program, it took about 10 seconds to download the 44 KByte font. This is three times faster to download about 30% more data. Both used the Appletalk communications interface. It was disappointing to find that the Adobe font would not download using the Altsys program, nor would the CasadyWare™ font download using the Adobe program. Without any more data available, it seems that point #4 above, the efficiency of the Altsys download program vs the Adobe download program, was the key difference.

Once the font has been downloaded, the performance of downloaded fonts were similar to the performance of internal fonts. The performance of two different downloaded fonts were also rather close. When the scan conversion time of the Adobe internal Times Roman® font was compared to the Adobe downloaded Stone Serif font the times were very close: 3.19 vs. 3.23 characters per second respectively. The third party downloaded CasadyWare Galileo Roman font scan converted at a slightly slower speed, 2.71 characters per second. This speed difference could be attributable to many different factors, like less efficient coding of the fonts themselves or the effect of the font style complexity variation.

#### 4.5 Methods of Printing Strings

Since POSTSCRIPT® is a full functionality programming language it is possible to print justified and unjustified strings of text in a variety of different ways. Section 3.3 shows six of them. One trend was shown. As more POSTSCRIPT® instructions had to execute per "showing" each line of text, the time to execute these instructions increased causing the page performance to decrease.

The way to achieve the best performance is by calculating the start positions of each line of text in the front end composition computer and send this information down for each line within the POSTSCRIPT® page. For justification the *ashow* or *widthshow* operators can be used, depending on the effect that is desired (i.e. adjusting the intercharacter or interword spacing). In the experiments, the page using *ashow* yielded slightly better performance than the page using *widthshow*.

#### **4.6 Resolution Targeting**

On page 76 of the POSTSCRIPT® Language Reference Manual it states that if:

1. a sampled image is a binary image that uses one bit per sample, and
2. the combination of the image matrix and the current transformation matrix is such that one unit in image space corresponds to one unit in device space,

*then*

The produced results are precisely predicatable down to the pixel level and executes a great deal faster than general imaging.

The above statement applies to scanned bitmap images, not to fonts. Section 3.4 tested to see if a similar speedup could be observed if the starting positions of each line were specified in print engine coordinates. No effect could be seen. This speedup seems to be only active for a special case of bitmap images.

#### **4.7 Printer Induced Delays**

Execution of POSTSCRIPT® programs on the Apple LaserWriter® Plus are influenced by several factors:

1. the location of the lowest (smallest "y" value) mark on the page. On the Apple LaserWriter® Plus, the page writes from top to bottom. All lines below the line which prints the lowest mark on the page is all white. It seems that the Apple LaserWriter® Plus RIP keeps track of this information (sometimes called a "highwater mark"). When this point is reached the RIP can start to use the page buffer to generate the next page, while at the same time providing a "white" signal to the print engine to finish printing the current page. This phenomena is shown in section 3.5.1.
2. similar to #1 above, the printing of the current page holds up the generation of the next page, since a page buffer is required to operate in. A dual page "ping-pong" buffer can alleviate this problem at an added cost.
3. print engine control and communications interface control. Since the Apple LaserWriter® Plus printer RIP has a single processor generating the page it must be shared to perform non-POSTSCRIPT® functions, thereby slowing down the page generation task. Dedicated microcontrollers could be used to handle these ancillary tasks and help improve the overall performance. See section 3.5.2 for more on this topic.

## 5. Glossary

**batch composition** - A program that executes on a front end system that accepts, as its input, a text file with markup commands (line width, type of justification, etc.), and outputs a page description file ready to be output on a printer.

**cache memory** (also bitmap font cache) - memory (RAM) on the Apple LaserWriter® Plus RIP that temporarily stores bitmap fonts. Outline fonts are scan converted and placed in cache memory. When a reference is made to print a character of a certain font that has previously been placed in the font cache the bitmap is quickly BitBLT'ed from the font cache to the page buffer. This process is much faster than scan converting the outline character.

**composition** - The process of integrating text, vector graphics, and bit-map images on pages in a form that enhances the information being conveyed by the written prose. Typical features of a composed page are justified text using multiple fonts, charts that graphically show a trend, and screened photographs of items being described in the written portion of the page.

**CTM (Current Transformation Matrix)** - A 3x3 matrix that describes how user coordinate space maps into device space. Changing variables in the CTM changes rotation, scaling, and translation. The matrix is shown below:

$$\begin{bmatrix} S_x \cos \theta & S_x \sin \theta & 0 \\ -\sin \theta & S_y & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

$\theta$  is angle of rotation in a positive direction (counterclockwise)

$S_x$  is the scale factor in the x direction.

$S_y$  is the scale factor in the y direction.

$T_x$  is the translate number in the x direction.

$T_y$  is the translate number in the y direction

**device coordinate system** - The cartesian coordinate system that describes the printable dots of a particular print engine (i.e. "device"). The units of the device coordinate system have a one to one correlation to the dots of the printer. For the Apple® LaserWriter Plus® printer the device resolution is 300 dots per inch. The Current Transformation Matrix, or CTM, maps the user space into device space.

**dot** - see *printable dot*

**EPRM (Electrically Programmable Read Only Memory)** - A type of ROM that is programmable using a computer controlled tool that provides sequenced electrical pulses to the EPRM. EPRMs are typically erasable by irradiating the EPRM chip with ultraviolet light for several minutes. Because they are erasable and (re)programmable, EPRMs are commonly reused during product development and for product code updates.

**font** - A complete assortment of a given size of type, including capitals, small capitals, and lowercase, together with figures, punctuation marks, ligatures, and the commonly used signs and accents. The *italic* of a given face is considered a part of the equipment of a font of type but is spoken of as a separate font.

**font family** - A group including all the styles and sizes of the characters in that font. For example Times® is the font family that contains all sizes of all characters in the Times Roman, Times Italic, Times Bold, and Times Bold Italic fonts.

**full justification** - modification of the intercharacter and/or interword spacing within a line to position both the left and right ends of the line at certain points. This is done to fill in a complete rectangular space with solid text.

**front end** - A workstation with interfaces to other computers typically via a network interface, a user interface to allow text, graphical, and scanned image input, and an interface to printers.

**H & J (Hyphenation and Justification)** - The process of breaking up strings of text into substrings that fit well within a specified line width using a specified font. A complex set of rules are used to determine whether a line break should occur between adjacent words or the line break should occur between words. If a word is broken up into two sub-words, then the two sub-words are placed on consecutive lines with a hyphen placed after the first sub-word.

**interactive composition** - A program that executes on a front end system that accepts, as its input, keystrokes and possibly mouse clicks defining the text and style that the user wants the page to look like, and displays a softcopy image on the computer CRT screen and ultimately outputs a page description file ready to be output on a printer.

**Interpress** - A family of page description languages, invented by Xerox Corporation.

Interpress has three language sets:

1. **Commercial Set** includes the specification of basic text files, similar in functionality to many line printers.
2. **Publication Set** specifies multiple bit-map font capability and graphics, similar in functionality to *imPress*.
3. **Professional Graphics Set** is the high functionality PDL allowing arbitrarily rotated and scaled font, among other high end features. It is similar in functionality to *POSTSCRIPT®*.

**left justification** - Placement of consecutive lines so that the left edge of each line starts at the same x coordinate. This type of justification is also called "ragged right" since the right edges do not line up, but appear to be "ragged".

**justification** - The process of positioning strings of text to get a more legible effect. See *left justification* and *full justification* for more information.

**kerning** - The process of altering the spacing between two adjacent characters such that one character engulfs (ie. gets closer to) the other. Kerning is an advanced method of generating very high quality text. Common kerning pairs are ""VA" and "Ti".

**landscape** - A page orientation where the horizontal dimension is greater than the vertical direction. On a typical 8.5" x 11" page the horizontal dimension is 11 inches with the vertical dimension being 8.5 inches. See "portrait" for comparison.

**ligature** - A character combination that is made up of two or three characters that "fit" well with each other. These characters are graphically combined and are typographically considered a single character. Common ligatures are the "fi" and the "fl" character combinations.

**outline font rasterization** - see *scan conversion*

**PDL (Page Description Language)** - A file format which specifies how the RIP is to place marks on a page. These marks can be characters in a variety of fonts, vector graphics, and bit-map images.

**pixel** - see *printable dot*

**point** - A dimension that is approximately 1/72 of an inch. The point is the default unit of POSTSCRIPT®.

**portrait** - A page orientation where the horizontal dimension is less than the vertical direction. On a typical 8.5" x 11" page the horizontal dimension is 8.5 inches with the vertical dimension being 11 inches. Most letters are typed on a portrait page. See "landscape" for comparison.

**POSTSCRIPT®** - A high functionality page description language, defined by Adobe Systems, Inc. that boasts full functionality. The language itself is stack oriented and "forth"-like, and has the ability to define callable procedures and looping constructs. It is used in the Apple Laser Writer Plus printer, the printer used throughout this thesis. It is similar in functionality to the Interpress Professional Graphics Set.

**print engine** - A device that put black or color marks onto paper at a typical resolution of 300 to 400 dots per inch. Note that at 300 dots per inch, over 1 MByte of data is needed to "paint" a full 8.5 x 11 inch page at 1 bit/dot.

**printable dot** - The smallest mark on a page that a print engine can render. Size of the printable dots, or pixels ("picture elements") are the unit measures of the "device coordinate system".

**printer** - The combination of a RIP and a print engine.

**process (electrophotographic)** - The process used in most copiers and many page printers in which the following steps take place to produce copied or printed pages:

1. an area of electrophotographic material is charged with a uniform static electrical charge;
2. light, provided by a flashed image, laser or LED, exposes the page area on this charged material;
3. charge dissipates in the areas that were exposed, creating the latent image;
4. toner particles are charged such that either they will be attracted to the area that was exposed or not exposed, depending on the polarity of the process. A mirrored image on toner is layed onto the page area.
5. paper is charged such that the toner image is attracted to it;
6. the toner transfers to the paper;
- 7 the toner is fused to the paper using heat and/or pressure.

**process color** - The ability to produce a large number of different colors (thousands to millions) on a printed page to closely resemble a continuous tone color picture. See *spot color* for comparison.

**screen** - A method, called "halftone screening", is used to render an intermediate shade of gray, using a grouping of dots that can, individually, be either white or black. The parameters that define a screen in POSTSCRIPT® are:

1. frequency, which specifies the number of halftone cells per inch in device space;
2. angle, which specifies the number of degrees that the halftone screen is to be rotated with respect to the device coordinate system;
3. a procedure defining the spot function, which determines the order in which pixels within a halftone cell are whitened to produce the desired shade of gray.

**spot color** - The ability to produce several (typically 2 to 4) different color to accentuate key words or graphics for effect. See *process color* for comparison.

**rasterization** (alternatively, "rasterizing" or "scan converting" a font) - see *scan conversion*

**RIP (Raster Image Processor)** - A computer that inputs a page description language file and generates a bit-map image to transfer to the print engine, usually in a raster form.

**ROM (Read Only Memory)** - Computer memory that is written into only once, at the factory, and read many times. The data that is stored in ROM is permanent and non-volatile. It is typically used to store program code and other static data structures (like resident fonts).

**scan conversion** - The process of converting a geometric description of a shape and generating the bitmap that corresponds to that shape. Outline fonts and graphical objects are scan converted in RIPs that accept POSTSCRIPT®.

**user coordinate system** - A cartesian coordinate system, specifiable through POSTSCRIPT®, defined by the units (per inch), the origin, and x and y incrementing directions. The default user coordinate system of POSTSCRIPT® defines the units to be 1/72 of an inch (close to a "point"), the x incrementing direction to be "right", the y incrementing direction to be "up", and the origin to be the lower left hand corner of the page.

**device coordinate system** - The units of the device coordinate system have a one to one correlation to the dots of the printer. For the Apple® LaserWriter Plus® printer the device resolution is 300 dots per inch. The Current Transformation Matrix, or CTM, maps the user space into device space.

**WYSIWYG (What-You-See-Is-What-You-Get)** - The concept, in interactive composition systems, where the image that is viewed on the CRT screen is the same image that will be printed out on the printer.

**6. References (see section 7 for full bibliography entries)**

1. A Comparison of Interpress™ and POSTSCRIPT®
2. page 109 Apple LaserWriter® Reference for LaserWriter, LaserWriter Plus, LaserWriter II NT, and LaserWriter II NTX
3. page 221 POSTSCRIPT® Language Reference Manual
4. page 222 POSTSCRIPT® Language Reference Manual
5. pp. 117, 118 POSTSCRIPT® Language Reference Manual
6. page 140 POSTSCRIPT® Language Reference Manual
7. page 229 POSTSCRIPT® Language Reference Manual
8. page 156 POSTSCRIPT® Language Reference Manual
9. page 150 POSTSCRIPT® Language Reference Manual
10. page 216 POSTSCRIPT® Language Reference Manual
11. page 128 POSTSCRIPT® Language Reference Manual
12. page 97 POSTSCRIPT® Language Reference Manual
13. page 287 POSTSCRIPT® Language Reference Manual
14. page 288 POSTSCRIPT® Language Reference Manual
15. pp. 85 - 102 POSTSCRIPT® Language Reference Manual
16. pp. 222-225 POSTSCRIPT® Language Tutorial and Cookbook
17. page 41 Printers Buyers Guide and Handbook Test Reports
18. page 6 Apple LaserWriter® Reference for LaserWriter, LaserWriter Plus, LaserWriter II NT, and LaserWriter II NTX
19. pp. 3,17 Apple LaserWriter® Reference for LaserWriter, LaserWriter Plus, LaserWriter II NT, and LaserWriter II NTX
20. page 17 Apple LaserWriter® Reference for LaserWriter, LaserWriter Plus, LaserWriter II NT, and LaserWriter II NTX
21. page 90 Laser Wars
22. page 18 Apple LaserWriter® Reference for LaserWriter, LaserWriter Plus, LaserWriter II NT, and LaserWriter II NTX
23. pp. 87, 92 Laser Wars
24. page 76 The Laser's Edge: Anatomy of a Printing Job
25. page 20 Fluent Laser Fonts™ User's Guide

26. pp 287, 301      POSTSCRIPT® Language Tutorial and Cookbook
27. page 7-5        MC68000 - 16-/32-Bit Microprocessor
28. page 122        Apple LaserWriter® Reference for LaserWriter, LaserWriter Plus,  
                         LaserWriter II NT, and LaserWriter II NTX
29. pp. I 230 - 235    Inside Macintosh™ Volumes I, II, and III
30. pp. IV 33 - 48    Inside Macintosh™ Volume IV

## 7. Bibliography

Adobe Type Library User's Manual for the Macintosh  
Version 2.0 - October, 1987  
Copyright 1987  
Adobe Systems Incorporated

Apple LaserWriter Reference  
Apple Programmer's and Developer's Association (APDA)  
APDA #: KNBLRM  
290 SW 43rd Street  
Renton, WA. 98055  
Tel. No. (206) 251-6548

Apple LaserWriter® Reference  
for LaserWriter, LaserWriter Plus, LaserWriter II NT, and LaserWriter II NTX  
Apple Computer, Inc.  
Addison-Wesley Publishing Company, Inc.  
Copyright© 1988 by Apple Computer, Inc.

The Chicago Manual of Style  
Thirteenth Edition, Revised and Expanded  
The University of Chicago Press  
Copyright 1982

A Comparison of Interpress™ and POSTSCRIPT®  
Jerry Mendelson  
April, 1985  
Xerox Corporation

Font and Function - The Adobe Type Catalog Spring / 1988  
Adobe Systems Incorporated  
1988

Fluent Laser Fonts™ User's Guide  
Robin Casady and Richard Ware  
CasadyWare Inc.  
Copyright 1987

Information Processing - Text and Office Systems  
Standard Generalized Markup Language  
ISO / DIS8879

Inside Macintosh™ Volumes I, II, and III  
Addison-Wesley Publishing Company, Inc.  
Copyright© 1985 by Apple Computer, Inc.  
*Sixth Printing, September 1987*

Inside Macintosh™ Volume IV  
Addison-Wesley Publishing Company, Inc.  
Copyright© 1986 by Apple Computer, Inc.  
*Third Printing, April 1987*

The Laser's Edge: Anatomy of a Printing Job  
Danny Goodman  
MACWORLD The Macintosh™ Magazine  
February, 1985  
Article: The Laser's Edge - pp. 70 - 79  
Anatomy of a Printing Job - pp.76

Laser Wars  
Henry Bortman  
MacUser - The Macintosh™ Resource (magazine)  
October, 1987  
Article: Laser Wars - pp. 84 - 94

MC68000 - 16-/32-Bit Microprocessor  
Advance Information  
March, 1985  
Motorola Publication Number ADI814R5

Page Description Languages  
Robert A. Morris  
Interleaf, Inc. and the University of Massachusetts at Boston  
Copyright 1985, Boole Press, Dublin  
Proceedings of the ProText II Conference, Boole Press, 1985

Pocket Pal - a graphics arts production handbook  
International Paper Company  
Thirteenth Edition  
Copyright May 1983

POSTSCRIPT® Language Program Design (aka - the Green Book)  
Adobe Systems Incorporated  
Addison-Wesley Publishing Company, Inc.  
Glenn C. Reid  
Copyright© 1988

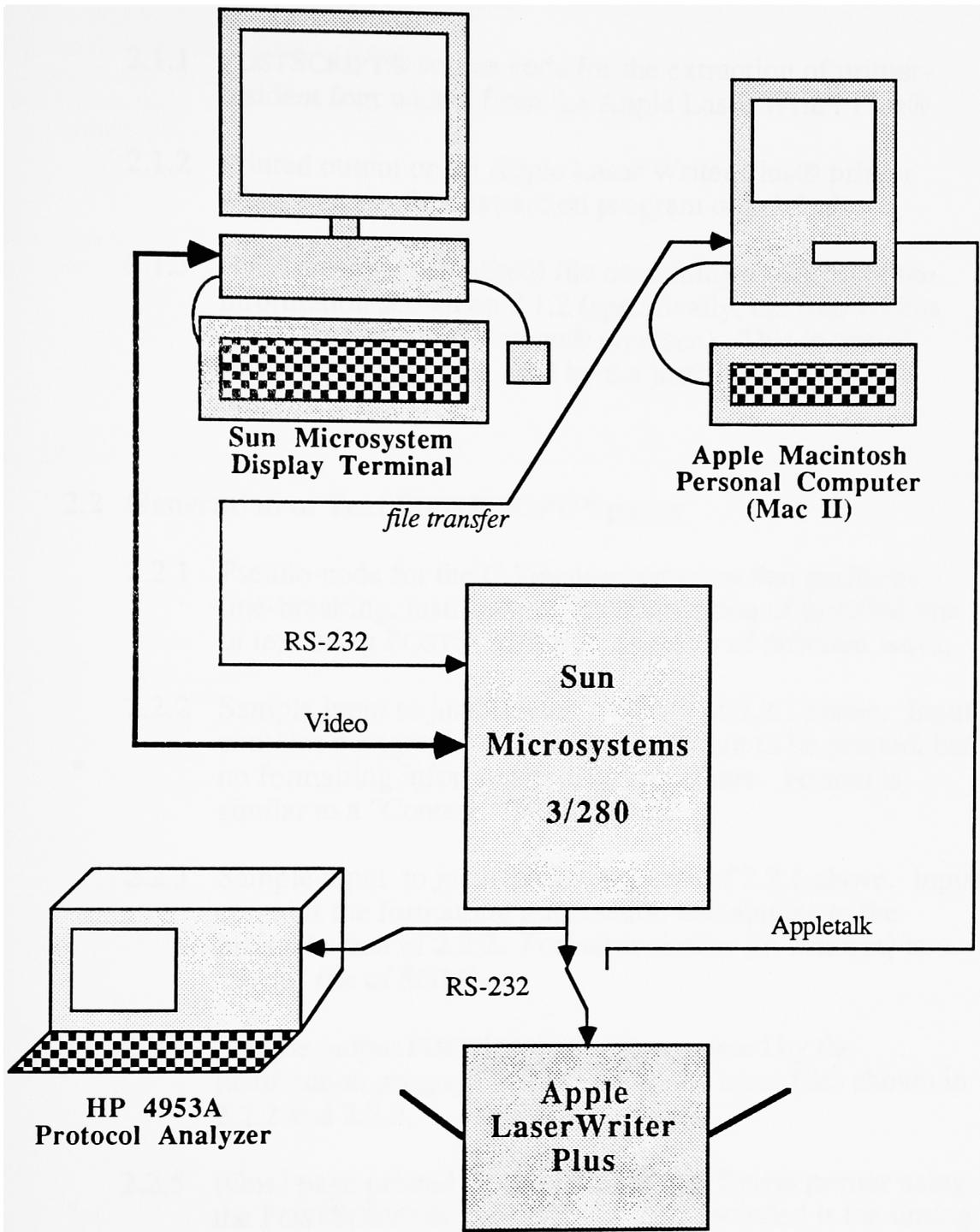
POSTSCRIPT® Language Tutorial and Cookbook (aka - the Blue Book)  
Adobe Systems Incorporated  
Addison-Wesley Publishing Company, Inc.  
Copyright© 1985

POSTSCRIPT® Language Reference Manual (aka - the Red Book)  
Adobe Systems Incorporated  
Addison-Wesley Publishing Company, Inc.  
Copyright© 1985

Postscript®: Master of the Raster  
Ted Nace  
PC World  
August, 1985  
pages 256-262

Printers Buyers Guide and Handbook Test Reports  
Printers Buyers Guide and Handbook  
Guide #7  
Copyright© 1987  
Article: Apple Laser-Writer Plus - page 41

Supporting Downloadable POSTSCRIPT® Fonts  
Glenn Reid  
Adobe Systems Incorporated  
August 14, 1987



## PostScript® Evaluation Equipment Configuration

### Appendix 1



## **2.1 POSTSCRIPT™ Font Widths**

- 2.1.1 POSTSCRIPT® source code for the extraction of printer-resident font widths from the Apple Laser Writer Plus®
- 2.1.2 Printed output on an Apple Laser Writer Plus® printer when sent the font extraction program of 2.1.1 above.
- 2.1.3 Hand created (and edited) file containing a portion of the information shown on 2.1.2 (specifically, the font widths for a 10 point Time-Roman® typeface). This is one of seven font width files used by the justification program described in 2.2.1.

## **2.2 Generation of Text POSTSCRIPT™ pages**

- 2.2.1 Pseudo-code for the C-language program that performs line-breaking, justification, and generation of justified line of text using POSTSCRIPT® in a variety of different ways.
- 2.2.2 Sample input to justification program of 2.2.1 above. Input contains paragraphs of information meant to be printed, but no formatting information is included here. Format is similar to a "Content" file of SGML.
- 2.2.3 Sample input to justification program of 2.2.1 above. Input contains the formatting information that applies to the printable data of 2.2.2. Format is similar (in concept) to a "Style" file of SGML.
- 2.2.4 Sample output POSTSCRIPT® file generated by the justification program of 2.2.1 using the input files shown in 2.2.2 and 2.2.3.
- 2.2.5 (One) page printed on the Laser Writer Plus® printer using the POSTSCRIPT® file of 2.2.4. Also, included is the timing page that is ejected after a series of pages is printed.

*Note: The pages that are shown in 2.2.2 through 2.2.5 are used to generate timing data recorded in section 3.3 using justification method # 1.*

## **Appendix 2: Support Programs**

```

% Written for the Masters Thesis at RIT; Thomas L. Kowalczyk
% Performance Analysis of Text Oriented Printing Using PostScript
%
% This procedure interrogates the Apple Laser Writer Plus for the font widths
% of three of its printer resident fonts.  The font widths of the normal ASCII
% printable characters are extracted, and font width tables for three font
% faces are printed.  A 10 point font was assumed for all three fonts.

% Default units of POSTSCRIPT is the point, which is 1/72 of an inch.  The
% "inch" procedure allows the program to specify coordinates in inches instead.
%
/inch {72 mul} def

% The top of the font width columns is this distance from the bottom of the
% page.  Note that in POSTSCRIPT the origin is at the lower left corner of the
% page with x increasing in value to the right and y increasing in value up,
% like a normal Cartesian coordinate system.
%
/ytop 9.25 inch def

% Set up two strings of characters corresponding to the two columns of font
% width tables.
%
/st1 ( !"#$$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNPO ) def
/st2 (QRSTUVWXYZ[\]^_`|~abcdefghijklmnopqrstuvwxy{|}~ ) def

% Set up variables and position cursor before each column is printed
%
/reset_to_top
{
  /n 0 def % character index into string
  /y ytop def % set y cursor value to top of column position
  x y moveto % position POSTSCRIPT cursor on page
} def

% Put title on the top of the page
%
/page_header
{
  /Times-Roman findfont 24 scalefont setfont
  2.5 inch ytop .5 inch add moveto
  (Font Extraction Program) show
} def

```

## Appendix 2.1.1 Font Extraction Program

```

% Selecting each individual character from the string passed to this procedure,
% request its character width. Then, starting at the existing (x,y) cursor
% position, place each (character,width) pair at locations successively below
% one another (i.e. print a column).
%

```

```

/print_col

```

```

{
  /st exch def          % pass string of characters to procedure
  1 1 49                % loop from the first number to the last
                        % number in increments of the second number

  {
    st n 1 getinterval % get the n'th character of the string
    dup /s exch def    % save it in the variable "s"
    show               % print it at the current (x,y) position
    x .2 inch add y    % position cursor to print width to the right
    moveto             % of the character
    s stringwidth      % get the width of the individual character
    pop                % x width and y height are returned, so get
                        % rid of the height, since it is of no interest
    100 mul round      % round the font width to the nearest 1/100 of
    cvr 100 div        % a point; note that the "round" operator yields
                        % an integer and "cvr" converts it back to real
    (          ) cvs   % convert the number which represent the
                        % character width to a printable string of
                        % characters
    show               % and print it.
    /y y 12 sub def   % set up y to go down to the next line
    x y moveto        % position the cursor for next pair
    /n n 1 add def    % increment index to reference next character
  } for
} def

```

```

% Print two columns of font width tables.
%

```

```

%

```

```

/print_2_col

```

```

{
  reset_to_top          % set up variables and position cursor
  st1 print_col         % generate and print first column of
                        % (char,width) pairs
  /x x .8 inch add def % set up second column to be to the right of
                        % the first one just printed
  reset_to_top          % set up variables and position cursor
  st2 print_col         % generate and print second column of
                        % (char,width) pairs
} def

```

```

% Code starts to execute here. All of the above are procedures are referenced
% by the calls below.
%

% Print title on the page.
page_header

% Select fonts, position the x cursor position, print a column title, and print
% the (char,width) pairs for three font faces.
%
/Times-Roman findfont 10 scalefont setfont
                                % select the font
/x 1 inch def                    % set up x position for first set of columns
x ytop 15 add moveto             % position
(10 point Times Roman) show     % and print title
print_2_col                      % extract and print table

/Times-Italic findfont 10 scalefont setfont
                                % select the font
/x 3.25 inch def                % set up x position for second set of columns
x ytop 15 add moveto            % position
(10 point Times Italic) show    % and print title
print_2_col                     % extract and print table

/Times-Bold findfont 10 scalefont setfont
                                % select the font
/x 5.5 inch def                 % set up x position for last set of columns
x ytop 15 add moveto           % position
(10 point Times Bold) show      % and print title
print_2_col                    % extract and print table

showpage                        % print and eject the page

```

## Appendix 2.1.1 Font Extraction Program

Following Page  
is the  
Actual Output  
from the  
Apple Laser Writer Plus®  
Printer

**Appendix 2.1.2 Printed Font Width Tables from Laser Writer Plus®**  
single printed page to follow

# Font Extraction Program

## 10 point Times Roman

2.5	Q	7.22
! 3.33	R	6.67
" 4.08	S	5.56
# 5.0	T	6.11
\$ 5.0	U	7.22
% 8.33	V	7.22
& 7.78	W	9.44
' 3.33	X	7.22
( 3.33	Y	7.22
) 3.33	Z	6.11
* 5.0	[	3.33
+ 5.64	\	2.78
, 2.5	]	3.33
- 3.33	^	4.69
. 2.5	_	5.0
/ 2.78	`	3.33
0 5.0		2.0
1 5.0	~	5.41
2 5.0	a	4.44
3 5.0	b	5.0
4 5.0	c	4.44
5 5.0	d	5.0
6 5.0	e	4.44
7 5.0	f	3.33
8 5.0	g	5.0
9 5.0	h	5.0
: 2.78	i	2.78
; 2.78	j	2.78
< 5.64	k	5.0
= 5.64	l	2.78
> 5.64	m	7.78
? 4.44	n	5.0
@ 9.21	o	5.0
A 7.22	p	5.0
B 6.67	q	5.0
C 6.67	r	3.33
D 7.22	s	3.89
E 6.11	t	2.78
F 5.56	u	5.0
G 7.22	v	5.0
H 7.22	w	7.22
I 3.33	x	5.0
J 3.89	y	5.0
K 7.22	z	4.44
L 6.11	{	4.8
M 8.89		2.0
N 7.22	}	4.8
O 7.22	~	5.41
P 5.56	2.5	

## 10 point Times Italic

2.5	Q	7.22
! 3.33	R	6.11
" 4.2	S	5.0
# 5.0	T	5.56
\$ 5.0	U	7.22
% 8.33	V	6.11
& 7.78	W	8.33
' 3.33	X	6.11
( 3.33	Y	5.56
) 3.33	Z	5.56
* 5.0	[	3.89
+ 6.75	\	2.78
, 2.5	]	3.89
- 3.33	^	4.22
. 2.5	_	5.0
/ 2.78	`	3.33
0 5.0		2.75
1 5.0	~	5.41
2 5.0	a	5.0
3 5.0	b	5.0
4 5.0	c	4.44
5 5.0	d	5.0
6 5.0	e	4.44
7 5.0	f	2.78
8 5.0	g	5.0
9 5.0	h	5.0
: 3.33	i	2.78
; 3.33	j	2.78
< 6.75	k	4.44
= 6.75	l	2.78
> 6.75	m	7.22
? 5.0	n	5.0
@ 9.2	o	5.0
A 6.11	p	5.0
B 6.11	q	5.0
C 6.67	r	3.89
D 7.22	s	3.89
E 6.11	t	2.78
F 6.11	u	5.0
G 7.22	v	4.44
H 7.22	w	6.67
I 3.33	x	4.44
J 4.44	y	4.44
K 6.67	z	3.89
L 5.56	{	4.0
M 8.33		2.75
N 6.67	}	4.0
O 7.22	~	5.41
P 6.11	2.5	

## 10 point Times Bold

2.5	Q	7.78
! 3.33	R	7.22
" 5.55	S	5.56
# 5.0	T	6.67
\$ 5.0	U	7.22
% 10.0	V	7.22
& 8.33	W	10.0
' 3.33	X	7.22
( 3.33	Y	7.22
) 3.33	Z	6.67
* 5.0	[	3.33
+ 5.7	\	2.78
, 2.5	]	3.33
- 3.33	^	5.81
. 2.5	_	5.0
/ 2.78	`	3.33
0 5.0		2.2
1 5.0	~	5.2
2 5.0	a	5.0
3 5.0	b	5.56
4 5.0	c	4.44
5 5.0	d	5.56
6 5.0	e	4.44
7 5.0	f	3.33
8 5.0	g	5.0
9 5.0	h	5.56
: 3.33	i	2.78
; 3.33	j	3.33
< 5.7	k	5.56
= 5.7	l	2.78
> 5.7	m	8.33
? 5.0	n	5.56
@ 9.3	o	5.0
A 7.22	p	5.56
B 6.67	q	5.56
C 7.22	r	4.44
D 7.22	s	3.89
E 6.67	t	3.33
F 6.11	u	5.56
G 7.78	v	5.0
H 7.78	w	7.22
I 3.89	x	5.0
J 5.0	y	5.0
K 7.78	z	4.44
L 6.67	{	3.94
M 9.44		2.2
N 7.22	}	3.94
O 7.78	~	5.2
P 6.11	2.5	

8 2.5  
! 3.33  
" 4.08  
# 5.0  
\$ 5.0  
% 8.33  
& 7.78  
' 3.33  
( 3.33  
) 3.33  
\* 5.0  
+ 5.64  
, 2.5  
- 3.33  
. 2.5  
/ 2.78  
0 5.0  
1 5.0  
2 5.0  
3 5.0  
4 5.0  
5 5.0  
6 5.0  
7 5.0  
8 5.0  
9 5.0  
: 2.78  
; 2.78  
< 5.64  
= 5.64  
> 5.64  
? 4.44  
@ 9.21  
A 7.22  
B 6.67  
C 6.67  
D 7.22  
E 6.11  
F 5.56  
G 7.22  
H 7.22  
I 3.33  
J 3.89  
K 7.22  
L 6.11  
M 8.89  
N 7.22  
O 7.22  
P 5.56  
Q 7.22  
R 6.67  
S 5.56

## Appendix 2.1.3 Times-Roman® Font Width Table

T 6.11  
U 7.22  
V 7.22  
W 9.44  
X 7.22  
Y 7.22  
Z 6.11  
[ 3.33  
\ 2.78  
] 3.33  
^ 4.69  
\_ 5.0  
` 3.33  
a 4.44  
b 5.0  
c 4.44  
d 5.0  
e 4.44  
f 3.33  
g 5.0  
h 5.0  
i 2.78  
j 2.78  
k 5.0  
l 2.78  
m 7.78  
n 5.0  
o 5.0  
p 5.0  
q 5.0  
r 3.33  
s 3.89  
t 2.78  
u 5.0  
v 5.0  
w 7.22  
x 5.0  
y 5.0  
z 4.44  
{ 4.8  
| 2.0  
} 4.8  
~ 5.41

### Appendix 2.1.3 Times-Roman® Font Width Table

# Batch\_composition

Leading = 1        factor times font size for vertical spacing  
Static\_line\_width = 7 Inches  
Justification\_method = ragged right  
Font = Times Roman  
Font\_size = 10 point  
X\_cursor = 1 Inch  
Y\_cursor = 10 Inches

```
{  
Input filename from command line  
  
filename.postscript << postscript header file
```

```
for fonts 1 through 7  
  load font widths from font width files
```

```
Word_char_cnt = 0  
Line_char_cnt = 0
```

```
Old_line_width = 0  
New_line_width = 0
```

```
Space_cnt = 0
```

```
Inside_word_flag = FALSE
```

```
  Loop to get next character until done
```

```
  {  
  Get next character
```

```
  If Character is "normal"
```

```
  then
```

```
    {  
    inside_word_flag = TRUE  
    Current_word = Current_word + Character  
    }
```

```
  else
```

```
    if inside_word_flag = FALSE  
    then  
      {  
      if Character is a space or carriage return  
      then  
        Increment Space_cnt  
      else  
        Call Get_command  
      }  
    }
```

main program

Initialize default global static composition state variables; these variables are also set as specified in the *filename.structure* file

Initialize dynamic composition state variables

current # of characters in word  
current # of characters in line (including spaces)

before current word width is added  
after space width and current word width is added

# spaces before current word

i.e. not a "<", space, or carriage return

set flag for next time through loop

character is not "normal"  
a delimiter has been detected

more than one space or carriage return

it's a "<"  
interpret command; look at structure file if necessary

## Appendix 2.2.1 Batch Composition *Pseudo-code*

else	was inside a word
{	first delimiter after a word
Inside_word_flag = FALSE	
New_line_width = Old_line_width +	
Old_space_cnt * width of space +	
width of Current_word	Does the Line + Word fit on the line?
	it fits, so add word to line
If New_line_width < Static_line_width	
then	
{	
Line = Line + Spaces + Current_word	
Line_char_cnt = Line_char_cnt + Space_cnt + Word_char_cnt	
Total_space_cnt = Total_space_cnt + Space_cnt	
Word_char_cnt = 0	get ready for next word
Old_line_width = New_line_width	
Space_cnt = 1	
If Character = "<"	end of paragraph marker
then	
{	
Print (Line)	Print unjustified line
Reset all dynamic comp variables	
Call Get_command	get rid of "/p>", and "beginning of paragraph" marker
}	if it exists along with "structure" info. for new par.
else	
Space_cnt = 1	space and CR both count as a space
}	end of "it fits"
else	
{	
Justified_print (Line)	it doesn't fit,
if Character = "<"	so print justified line
then	end of paragraph marker
{	
Print (Word)	Print unjustified line consisting of one word
Reset all dynamic comp variables	
Call Get_Command	get rid of "/p>", and "beginning of paragraph" marker
}	if it exists along with "structure" info. for new par.
else	
{	
Line = Word	Start building new line with the word that didn't fit
Space_cnt = 0	
Reset all dynamic comp variables	
}	
}	end of "it doesn't fit"
}	end of "first delimiter after line"
}	end of loop "get next character"
Print_trailer	"cat" the trailer postscript file that sets up timing
}	tests to the filename.postscript file

## Appendix 2.2.1 Batch Composition *Pseudo-code*

## Get\_command

{

Get next character from *filename.data*

If Character = '/'  
then

read next three characters from *filename.data*

procedure to detect beginning and end of paragraphs  
and setting of the global static composition  
variables from the "structure" file

get rid of the "p", ">", and carriage return of the  
end-of-paragraph command to get the input file  
to the position for reading more data

else

If Character = 'p'

{  
read next two characters from *filename.data*

get rid of the ">", and carriage return of the  
beginning-of-paragraph command to get the input file  
to the position for reading more data

Get next character from *filename.structure*

Loop until Character = '>'

reset the global static composition parameters when  
a beginning-of-paragraph marker is detected

{  
Get command Character from *filename.structure*

all structure cmnds are read from the "structure" file

If Character = 'J'

Justification\_method = Integer from *filename.structure*

If 'L'

Leading = real number from *filename.structure*

If 'F'

{  
Font = Integer from 1 to 7 from *filename.structure*  
Font\_size = real number from *filename.structure*  
*filename.postscript* << postscript font setup instructions  
}

If 'C'

{  
X\_cursor = real number (In points) from *filename.structure*  
Y\_cursor = real number (In points) from *filename.structure*  
*filename.postscript* << postscript cursor positioning instructions  
}

If 'W'

Static\_line\_width = real number (In Inches) from *filename.structure*

}

end of "loop until done"

}

end of "character is a 'p' "

}

end of Get\_command routine

## Appendix 2.2.1 Batch Composition *Pseudo-code*

<p>

A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

</p>

<p>

B. PRIMARY CHARGER. The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

</p>

<p>

C. EXPOSURE. The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

</p>

<p>

D. AUXILIARY ERASE. Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

</p>

<p>

E. DEVELOPER STATION ASSEMBLY. The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

</p>

<p>

F. SCAVENGER ROLLER. Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

</p>

## Appendix 2.2.2 Input Data "Content" File (All paragraphs)

<p>

G. POST-DEVELOPMENT ERASE LAMP. To reduce the electrostatic stress on the IMAGE LOOP and thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

</p>

<p>

H. REGISTRATION. While the developed electrostatic image moves around the CORE, a sheet of copy paper is advanced to the REGISTRATION ASSEMBLY (not shown in Figure 1). At precisely the right time, the copy paper is directed into contact with the IMAGE LOOP and its developed image. This aligns the copy paper and the image on the IMAGE LOOP.

</p>

<p>

I. TRANSFER CHARGER. The IMAGE LOOP and copy paper now pass under the TRANSFER CHARGER, which produces a negative charge on the paper surface to attract the positive charged developer toner. This effectively transfers the copy image to the paper.

</p>

```
<F 1 9.0 L 1.2 W 6.5 J 1 C 72.0 720.0>  
<F 1 11.0>  
<F 1 13.0>  
<F 2 9.0>  
<F 2 11.0>  
<F 2 13.0>  
<F 3 9.0>  
<F 3 11.0>  
<F 3 13.0>
```

## Appendix 2.2.3 Input Markup "Structure" File

```

/Tsd userstime def
/Page {
/Times-Roman findfont 9.0 scalefont setfont
72.00 709.20 moveto
(A. IMAGE LOOP. The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically)show
72.00 698.40 moveto
(charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion)show
72.00 687.60 moveto
(for as long as copy exposures are being made \see Figure 1\). )show
/Times-Roman findfont 11.0 scalefont setfont
72.00 663.60 moveto
(B. PRIMARY CHARGER. The function of the PRIMARY CHARGER is to place a negative charge on)show
72.00 650.40 moveto
(the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE)show
72.00 637.20 moveto
(LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns)show
72.00 624.00 moveto
(on the PRIMARY CHARGER. )show
/Times-Roman findfont 13.0 scalefont setfont
72.00 595.20 moveto
(C. EXPOSURE. The charged IMAGE LOOP continues around the CORE to the)show
72.00 579.60 moveto
(EXPOSURE area, where it is exposed to a reflected light copy image that is focused on)show
72.00 564.00 moveto
(the IMAGE LOOP at precisely the right time, as determined by LOGIC AND)show
72.00 548.40 moveto
(CONTROL. The original document is illuminated by high intensity flash lamps for a)show
72.00 532.80 moveto
(short duration, which prevents blurring of the image as it is exposed on the moving)show
72.00 517.20 moveto
(IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are)show
72.00 501.60 moveto
(exposed to light. The charge remains in the areas that are not exposed. The exposure is)show
72.00 486.00 moveto
(said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused)show
72.00 470.40 moveto
(copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as)show
72.00 454.80 moveto
(an electrostatic image. )show
/Times-Italic findfont 9.0 scalefont setfont
72.00 428.40 moveto
(D. AUXILIARY ERASE. Just before each first, and just after each last, exposure area is an improperly charged segment.
These)show
72.00 417.60 moveto
(segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned
off)show
72.00 406.80 moveto
(during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the
moving)show
72.00 396.00 moveto
(IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development. )show
/Times-Italic findfont 11.0 scalefont setfont
72.00 372.00 moveto
(E. DEVELOPER STATION ASSEMBLY. The properly charged and exposed IMAGE LOOP area now)show
72.00 358.80 moveto
(enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner)show
72.00 345.60 moveto
(particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner)show
72.00 332.40 moveto

```

## Appendix 2.2.4 Text POSTSCRIPT® File

```

(particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the)show
72.00 319.20 moveto
(IMAGE LOOP surface for later transfer to a copy paper. )show
/Times-Italic findfont 13.0 scalefont setfont
72.00 290.40 moveto
(F. SCAVENGER ROLLER. Any developer carrier granules \(\iron\) left on the IMAGE)show
72.00 274.80 moveto
(LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the)
show
72.00 259.20 moveto
(DEVELOPER STATION ASSEMBLY. )show
/Times-Bold findfont 9.0 scalefont setfont
72.00 232.80 moveto
(G. POST-DEVELOPMENT ERASE LAMP. To reduce the electrostatic stress on the IMAGE LOOP and)show
72.00 222.00 moveto
(thereby increases its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge)show
72.00 211.20 moveto
(that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to)show
72.00 200.40 moveto
(prevent residual image retention. )show
/Times-Bold findfont 11.0 scalefont setfont
72.00 176.40 moveto
(H. REGISTRATION. While the developed electrostatic image moves around the CORE, a)show
72.00 163.20 moveto
(sheet of copy paper is advanced to the REGISTRATION ASSEMBLY \(\not shown in)show
72.00 150.00 moveto
(Figure 1\). At precisely the right time, the copy paper is directed into contact with the)show
72.00 136.80 moveto
(IMAGE LOOP and its developed image. This aligns the copy paper and the image on)show
72.00 123.60 moveto
(the IMAGE LOOP. )show
/Times-Bold findfont 13.0 scalefont setfont
72.00 94.80 moveto
(I. TRANSFER CHARGER. The IMAGE LOOP and copy paper now pass)show
72.00 79.20 moveto
(under the TRANSFER CHARGER, which produces a negative charge on)show
72.00 63.60 moveto
(the paper surface to attract the positive charged developer toner. This)show
72.00 48.00 moveto
(effectively transfers the copy image to the paper. )show

```

```

usertime /T0 exch def          % record time before showpage
showpage                      % operator is executed
} def

```

```

usertime /Ted exch def        % record time at end of file download
Ted Tsd sub /Tdl exch def     % calculate download time

```

```

%-----*
%  Procedures to time the printing of pages and generate and time the delay between pages.
%-----*

```

```

/sec                          % one second delay under no load
{                              % keep track of time actually taken
  usertime /T exch def {
    1228 {373.737 737.373 mul pop } repeat
  } repeat
  usertime T sub def} def

```

## Appendix 2.2.4 Text POSTSCRIPT® File

```

...
% Print one page and keep time
{
usertime /T exch def
Page
usertime T sub def
} def

%-----*
%   Calling sequences for the print (and time) pages
%   and the delay (again with timing)
%-----*

/Print_Pages
{
/T1 usertime def
Page
usertime T1 sub /T1B exch def           % calculate time that the first page
                                         % took to execute

T0 T1 sub /T1A exch def                 % calculate time that the first page
                                         % took to execute, less the "showpage"

/T2 Pr                                  % T2 gets time to print second page

/W1 1 sec                               % W1 gets actual delay time (1 second under no load)

/T3 Pr                                  % T3 gets time to print third page

/W2 2 sec                               % W2 gets actual delay time (2 seconds under no load)

/T4 Pr                                  % T4 gets time to print fourth page

/W3 3 sec                               % W3 gets actual delay time (3 seconds under no load)

/T5 Pr                                  % T5 gets time to print fifth page

/W5 5 sec                               % W5 gets actual delay time (5 seconds under no load)

/T6 Pr                                  % T6 gets time to print sixth page

/W8 8 sec                               % W8 gets actual delay time (8 seconds under no load)

/T7 Pr                                  % T7 gets time to print seventh page

/W12 12 sec                             % W12 gets actual delay time (12 seconds under no load)

/T8 Pr                                  % T8 gets time to print eighth page

/W16 16 sec                             % W16 gets actual delay time (16 seconds under no load)

/T9 Pr                                  % T9 gets time to print ninth page

} def

```

## Appendix 2.2.4 Text POSTSCRIPT® File

```

%-----*
% Routines to draw a simple graph relating delay time between pages
% on the x axis) to the time to print the page (on the y axis)
%-----*
/Draw_Graph
{
/Times-Roman findfont 12 scalefont setfont
/x 160 def /y 280 def

/GraphXInit 125 def
/GraphYInit 350 def
/GraphUnit 0.02 def % 20 points per second

GraphXInit GraphYInit 400 add moveto % draw the graph axes
0 400 neg rlineto
400 0 rlineto stroke

/GX GraphXInit 16 sub def
/GY GraphYInit 4 sub def

GX GY moveto (0) show % put numbers in
GX GY 100 add moveto (5) show % on y axis
-20 0 moveto
gsave 90 rotate
(Page Generation Time [in seconds]) show grestore
GX GY 200 add moveto (10) show
GX GY 300 add moveto (15) show
GX GY 400 add moveto (20) show

/GX GraphXInit 7 sub def
/GY GraphYInit 12 sub def

GX GY 10 sub moveto
GX 100 add GY moveto (5) show % on x axis
0 -15 rmoveto (Wait Time After Previous Page [in seconds]) show
GX 200 add GY moveto (10) show
GX 300 add GY moveto (15) show
GX 400 add GY moveto (20) show

GraphXInit GraphYInit moveto % put tick marks in
100 -3.5 rmoveto
0 7 rlineto % x axis, 5
100 0 rmoveto
0 -7 rlineto % x axis, 10
100 0 rmoveto
0 7 rlineto % x axis, 15
100 0 rmoveto
0 -7 rlineto stroke % x axis, 20

GraphXInit GraphYInit moveto % put tick marks in
-3.5 100 rmoveto
7 0 rlineto % y axis, 5
0 100 rmoveto
-7 0 rlineto % y axis, 10
0 100 rmoveto
7 0 rlineto % y axis, 15
0 100 rmoveto
-7 0 rlineto stroke % y axis, 20
} def

```

## Appendix 2.2.4 Text POSTSCRIPT® File

```

{
/yy exch def
3 add yy moveto
-3 3 rlineto
-3 -3 rlineto
3 -3 rlineto
3 3 rlineto
stroke
} def

/ShowPageTime                                % procedure to display time to generate page
{
x 200 add y moveto
(Page Time ) show show
x 290 add y moveto
( = ) show
dup /Ty exch def
(      ) cvs show
} def

/ShowWaitTime                                % procedure to display actual delay time
{
d
(Wait Time ) show show ( sec ) show
x 90 add y moveto
( = ) show
dup /Wx exch def
(      ) cvs show
} def

/GraphIt                                     % procedure to graph a (Delay,Page) time pt.
{
Wx GraphUnit mul GraphXInit add
Ty GraphUnit mul GraphYInit add
Diamond
} def

```

## Appendix 2.2.4 Text POSTSCRIPT® File

```

%-----*
% Calls to the above routines to print the timing data.
%-----*
/Report_Times
{
  Draw_Graph
  x y moveto
  (Download Time) show
  x 100 add y moveto
  (= ) show
  Td! ( ) cvs show

  /d {x y 15 sub dup /y exch def moveto} def

  d T1A (1 \{A\}) ShowPageTime
  d T1B (1 \{B\}) ShowPageTime

  0 (0) ShowWaitTime T2 (2) ShowPageTime GraphIt
  W1 (1) ShowWaitTime T3 (3) ShowPageTime GraphIt
  W2 (2) ShowWaitTime T4 (4) ShowPageTime GraphIt
  W3 (3) ShowWaitTime T5 (5) ShowPageTime GraphIt
  W5 (5) ShowWaitTime T6 (6) ShowPageTime GraphIt
  W8 (8) ShowWaitTime T7 (7) ShowPageTime GraphIt
  W12 (12) ShowWaitTime T8 (8) ShowPageTime GraphIt
  W16 (16) ShowWaitTime T9 (9) ShowPageTime GraphIt

  d x 200 add y moveto
  (Ave Pg Time 2-9) show
  x 290 add y moveto
  (= ) show
  T2 T3 add T4 add T5 add T6 add T7 add T8 add T9 add 8 div
  ( ) cvs show

  /Times-Italic findfont 12 scalefont setfont
  d d (Note: All times are specified in milliseconds.) show

} def

Print_Pages

Report_Times

showpage

```

Following Pages  
are the  
Actual Output  
from the  
Apple Laser Writer Plus  
Printer

**Appendix 2.2.5: Printed Text & Timing Pages from Laser Writer Plus®**

two printed pages to follow

**A. IMAGE LOOP.** The KODAK EKTAPRINT IMAGE LOOP is a continuous loop of film that is capable of being electrically charged, and is sensitive to direct light. The IMAGE LOOP is driven around the IMAGE LOOP CORE in a continuous motion for as long as copy exposures are being made (see Figure 1).

**B. PRIMARY CHARGER.** The function of the PRIMARY CHARGER is to place a negative charge on the IMAGE LOOP. This prepares the IMAGE LOOP for exposure and development. The IMAGE LOOP starts moving on command from LOGIC AND CONTROL. LOGIC AND CONTROL then turns on the PRIMARY CHARGER.

**C. EXPOSURE.** The charged IMAGE LOOP continues around the CORE to the EXPOSURE area, where it is exposed to a reflected light copy image that is focused on the IMAGE LOOP at precisely the right time, as determined by LOGIC AND CONTROL. The original document is illuminated by high intensity flash lamps for a short duration, which prevents blurring of the image as it is exposed on the moving IMAGE LOOP. The charge on the IMAGE LOOP is removed from the areas that are exposed to light. The charge remains in the areas that are not exposed. The exposure is said to discretely alter the charge characteristics of the IMAGE LOOP so that the focused copy image is recorded on the IMAGE LOOP. This IMAGE LOOP image is known as an electrostatic image.

*D. AUXILIARY ERASE.* Just before each first, and just after each last, exposure area is an improperly charged segment. These segments are produced when the PRIMARY CHARGER is turned on at the time of initial IMAGE LOOP movement and turned off during final IMAGE LOOP movement. As the unwanted areas pass under the AUXILIARY ERASE LAMP, it floods the moving IMAGE LOOP base with light that desensitizes the IMAGE LOOP to prevent unwanted development.

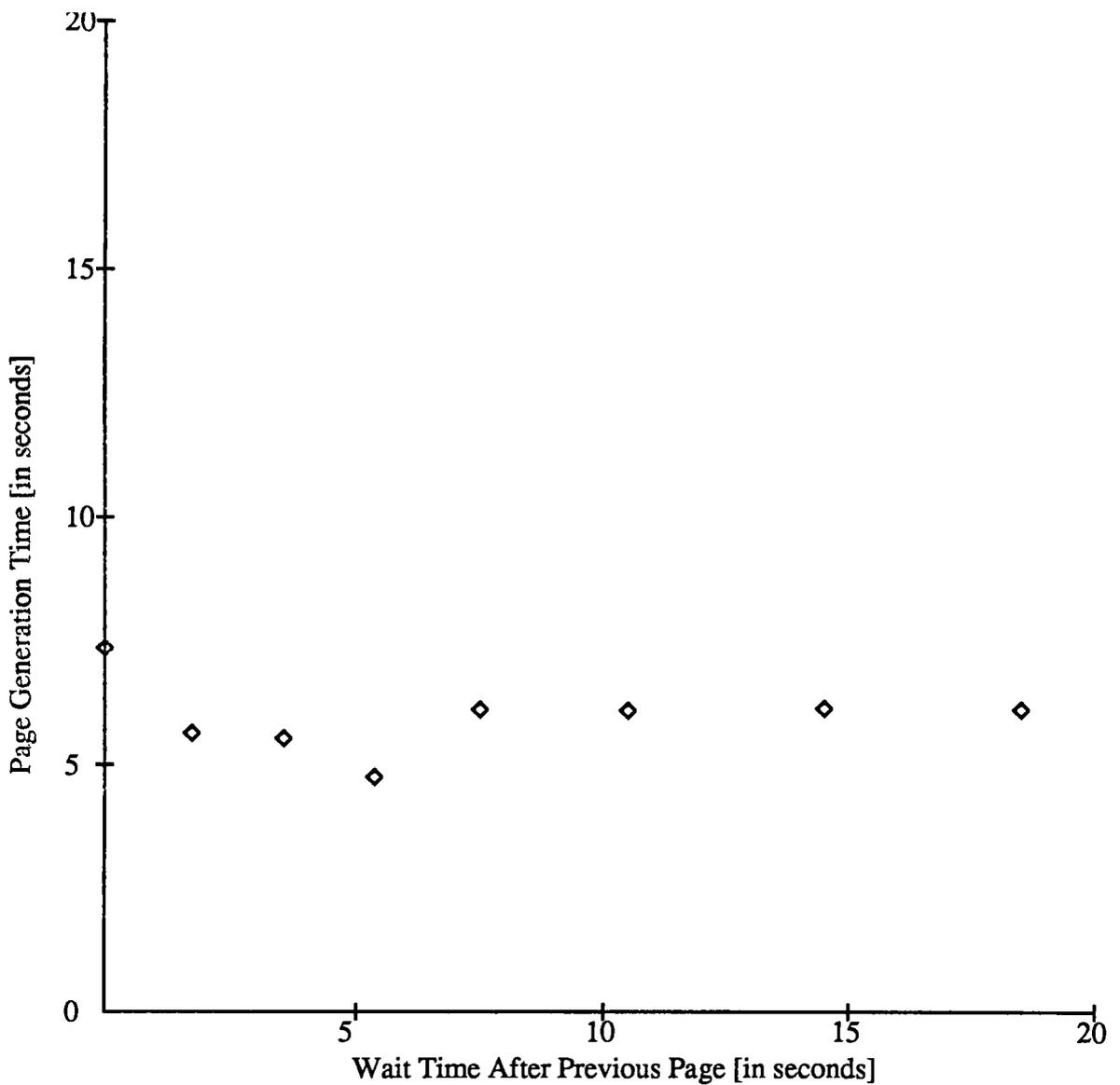
*E. DEVELOPER STATION ASSEMBLY.* The properly charged and exposed IMAGE LOOP area now enters the DEVELOPER STATION ASSEMBLY where positively charged KODAK EKTAPRINT K Toner particles are attracted to the IMAGE LOOP. Development occurs as the result of attraction of the toner particles to the electrostatic image on the IMAGE LOOP. The toner particles are carried away on the IMAGE LOOP surface for later transfer to a copy paper.

*F. SCAVENGER ROLLER.* Any developer carrier granules (iron) left on the IMAGE LOOP are salvaged at this point by the SCAVENGER ROLLER and returned to the DEVELOPER STATION ASSEMBLY.

**G. POST-DEVELOPMENT ERASE LAMP.** To reduce the electrostatic stress on the IMAGE LOOP and thereby increase its life, the POST DEVELOPMENT ERASE LAMP is used to lower the high level charge that was required for proper image development. This POST-DEVELOPMENT ERASE process also helps to prevent residual image retention.

**H. REGISTRATION.** While the developed electrostatic image moves around the CORE, a sheet of copy paper is advanced to the REGISTRATION ASSEMBLY (not shown in Figure 1). At precisely the right time, the copy paper is directed into contact with the IMAGE LOOP and its developed image. This aligns the copy paper and the image on the IMAGE LOOP.

**I. TRANSFER CHARGER.** The IMAGE LOOP and copy paper now pass under the TRANSFER CHARGER, which produces a negative charge on the paper surface to attract the positive charged developer toner. This effectively transfers the copy image to the paper.



Download Time = 5396

Wait Time 0 sec = 0  
 Wait Time 1 sec = 1714  
 Wait Time 2 sec = 3538  
 Wait Time 3 sec = 5360  
 Wait Time 5 sec = 7500  
 Wait Time 8 sec = 10500  
 Wait Time 12 sec = 14498  
 Wait Time 16 sec = 18496

Page Time 1 (A) = 127394  
 Page Time 1 (B) = 132492  
 Page Time 2 = 7364  
 Page Time 3 = 5648  
 Page Time 4 = 5552  
 Page Time 5 = 4762  
 Page Time 6 = 6158  
 Page Time 7 = 6160  
 Page Time 8 = 6218  
 Page Time 9 = 6212  
 Ave Pg Time 2-9 = 6009.25

*Note: All times are specified in milliseconds.*