Timing Techniques

1 Introduction

Anyone can time the seconds that elapse between requesting the production of a PostScript™ language document from within an application and receiving the completed document from a given printer. However, to determine what that time indicates about the parts of the system producing the document requires some knowledge concerning the basic components of the document production system, and how each part’s performance affects the final result.

This knowledge is helpful when deciding which part of a particular system would be most usefully altered to reduce the total printing time. It is also a necessary foundation for application developers’ efforts to make their applications produce good looking finished documents more rapidly than their competitors.

Developers not only need to know about the systems of which their software is a part, they also need tools and strategies that allow them to more closely determine where and why time is being spent and to explore ways through which that time can be reduced. While that is too large a topic to cover completely, in the following sections we supply both some system information and simple support for performing useful timing studies.

2 System Components

It is easy to see the physical components of a document production system based on one or more PostScript language devices. There is the computer where the PostScript language program is created, the connection over which it is delivered, and a device that executes the PostScript language programs it receives and produces the documents described by those programs.

This is too simplified a view, however, to be very helpful for conducting timing studies and analyzing the results. Instead, if the steps that occur between deciding to print and getting the final document are examined, a picture emerges that is not much more complex, but does represent enough of the factors that affect timing to be useful. See Figure 1.
The chief areas of concern when performing timing studies on PostScript language devices include

- The host platform on which an application runs
- The application itself
- Software that produces and delivers PostScript language statements (which may or may not be included in the application), referred to as the PostScript printer driver
- The communications channel
- The PostScript interpreter
- The printer controller board on which the interpreter executes
- The print engine that produces the physical document

Considering this entire sequence as a PostScript language pipeline, where one slow area can make other areas appear slow, makes it easier to determine where time is being spent, and to deduce and eliminate bottlenecks that degrade the performance of a system as a whole. Each of these different areas and its possible effect on the apparent speed of the others is discussed in the following sections.

2.1 Host Platform

The host platform (or system) refers to the computer and operating system that support the application that ultimately produces a PostScript language program. The speed of the computer’s hardware, and how well the operating system takes advantage of that hardware, affect how quickly the application can execute its tasks. Both an application’s ability to produce a PostScript language program rapidly and its ability to rapidly deliver that program to the printer are potentially limited by the platform on which it is running.
As desktop machines offer more features, they become increasingly configurable and correspondingly require more care when timing. The memory available to an application affects its speed, as does the form of the memory: the amount of buffer space allocated for performing I/O functions, the kind of caching being done, and how the memory is managed by the system. The amount of hard disk workspace available to the application and whether that available space is contiguous or fragmented makes a difference. The application that produces PostScript language output might not be the only task running on its host; sharing the system can significantly affect the apparent performance of an application.

2.2 Application Software

Application software typically performs many tasks. Regardless of what those tasks are, an application must convert its internal data representation into some printable description when it is time to print. The application itself might perform a direct conversion to PostScript language statements, relying chiefly on its own internal resources to produce its PostScript language program output.

Applications that use an external piece of software to create their PostScript language output must translate their internal representations into an intermediate format. Some applications producing PostScript language programs might also use an intermediate description to interface their internal representation with their own internal module producing the PostScript language output. Depending in part on the speed of the host platform, applications that use an intermediate internal format might take some time to perform that translation, in addition to the time required by a final translation to PostScript language statements.

Software whose task is the translation of input into a corresponding PostScript language program and the delivery of that program to its destination is referred to as a PostScript printer driver, whether internal or external to the application software under discussion. PostScript printer drivers are described in section 2.3.

2.3 PostScript Printer Driver

The functions of a PostScript printer driver were mentioned previously: to produce a PostScript language program corresponding to an input description and to deliver that program to some specified destination. A PostScript printer driver can also be referred to as a PostScript language driver. The driver might be internal or external to an application.
Familiar examples of external PostScript printer drivers include ones that produce PostScript language statements from GDI calls for Microsoft® Windows™ applications and ones that translate QuickDraw™ commands into PostScript language statements for Apple® Macintosh® applications.

There are several areas where the question of speed arises when examining the production of a PostScript language program. First is the speed with which the PostScript language statements are produced by the driver. On most platforms, this time should be small compared to other speed concerns (such as transmission and interpretation times) found farther along the pipeline between application and printed paper.

The remaining concerns have to do with the quality of the PostScript language program produced: the size of the code (affecting the time it takes to get the complete document description to the printer), and the length of time it takes to execute (the time the program takes once it gets there). For more detail, there are classes available on this topic, as well as developer support offered by the Adobe Systems Developers Association.

2.4 Communications Channel

Once produced, how quickly the PostScript language statements can be sent to their destination depends on the channel over which they must be sent: what the hardware is and how well it is supported. Although this determines how quickly it is possible to send data, the actual speed of delivery also depends on how well the driver software uses its opportunities. For instance, a driver using smaller packets than necessary to send a file over a network might take longer to deliver that file than a driver using larger packets.

The speed of the communications channel over which a PostScript language program is delivered to the printer is a well-known determining factor in how rapidly a PostScript language document appears to print. Some frequently used channels, in order of increasing speed, include serial, AppleTalk®, parallel, Ethernet, and SCSI connections. There can be a considerable difference in the length of time it takes to print the same document to the same printer over a serial versus a parallel port. Although each of these channels has some maximum rated speed, the actual speed with which data can be sent depends on the hardware and software at either end and in some cases on the nature of the data being transmitted.

An additional level of complexity is added when networked printers are considered. Files might have to traverse print managers, print spoolers, file and print servers, and various combinations of connections to get to a printer. Even for a relatively simple network, it can be difficult to determine how much the network itself is affecting printing time. For instance, the presence of other equipment on an AppleTalk network affects performance results. Even an unshared network connection might give variable results for large
files, depending on how often it tends to lose packets. Unless the timing target is the network connection, more consistent results can usually be obtained over one of the unshared types of connection.

2.5 PostScript Interpreter

In the type of system under discussion, the PostScript interpreter resides on the printer, and executes PostScript language programs to create rasterized printable images for the printer. The interpreter is a piece of software, just as a BASIC interpreter or an application program is a piece of software, and its performance can be affected by the same factors that affect other software.

These factors include the current configuration of the interpreter itself (for instance, the current font caching policy for the job being timed), the amount of RAM memory available on the printer, how quickly it receives PostScript language statements to execute, and so on. Like application software, its apparent speed depends in part on its host: the printer controller board.

2.6 Printer Controller

A PostScript printer is in many respects exactly like any computer system set up to read and execute program statements directly. It is very much like a computer running a BASIC interpreter that processes unnumbered BASIC language program statements entered from its keyboard, and even more like one that executes BASIC statements received over a communications port.

Like the BASIC interpreter, the PostScript interpreter runs on a host platform, in this case the printer controller board. Like the BASIC interpreter on the computer, perceived performance of the PostScript interpreter is affected by processor speed, hardware support, and system software on the printer controller.

Communications channels fall into one of several major categories when considering their speed. Similarly, printer controller boards can be categorized by the type and speed of the processor on which they are based.

Low end printers usually have a controller board based on the Motorola 68000, which is considered slow relative to more recently developed microprocessor technology. There are now printer controller boards based on the faster Motorola 68020 and 68030 chips, as well as RISC based controllers built on a variety of chip sets.

For more discussion regarding other aspects of a host platform that affect software performance, see section 2.1, “Host Platform.”
2.7 Print Engine

The speed of the print engine, like that of the communications channel, is an important limiting factor affecting document printing speed. A printer with a print engine rated at eight pages per minute can print at most eight pages in a minute, no matter how fast any of the other parts of the system are.

It is possible to have two PostScript language documents that print at the same speed on an eight page per minute (ppm) printer, but print at two different speeds on a seventeen page per minute printer, where fast execution time is less limited by the print engine speed.

Jobs that print as rapidly as a particular print engine allows are referred to as “printing at engine speed” on that printer. Jobs that run at engine speed on a particular printer can be difficult to compare further on that printer, but may be easier to compare on a faster print engine. There are PostScript printers that print at one hundred thirty-five pages per minute, but fifteen to forty ppm printers are more common.

3 Click-to-Clunk Timing

Click-to-clunk timing refers to a method of timing in which a person uses a stopwatch and times the creation of a PostScript language document from the moment a key is pressed in an application to request printing (the “click”) to the moment a printed piece of paper falls into the printer delivery tray (the “clunk”).

This type of timing can be done with a variety of goals. It is frequently done in comparative studies in which different versions of the same system component are being evaluated. It is also a useful reality check for applications under development, where typical and less typical print jobs are tested both before and after revisions are made. The true test of an optimization is the effect it has on click-to-clunk timing results. Regardless of the goal, it is important to hold constant every factor except the one under examination to obtain meaningful results.

Note When timing with a stopwatch, be aware of what constitutes a meaningful difference. Given human reaction time and variation in the point where the person perceives that a piece of paper is fully delivered (some people prefer to time only until they hear the engine start—an easier point to determine consistently), it would be unreasonable to distinguish between reported times that vary by tenths, let alone hundredths of seconds. Also, only jobs with typical times on the order of at least several seconds should be timed in this way to allow differences in test timings to outweigh possible inaccuracies in measurement. Only differences that clearly do not fall within the margin of error should be regarded as significant.
It is important to realize that when click-to-clunk timing is performed, all contributing aspects of the delivery and execution of a PostScript language file are being timed. The different system components described previously are playing a part in the time obtained. When comparing performance across different printers, it is important not to invalidate timing results by inadvertently timing a difference that was introduced by another component. For instance, a person timing job execution on several printers should not use a serial connection for one printer and a parallel for another, and then compare those two times, because two different communications overheads affect the results.

Another kind of interaction not documented previously is that between PostScript language jobs. One job can alter the printer environment in such a way as to affect the time it takes the next job to run. Some applications download their prologs only with the first file they send. This might make the first job seem much slower than the second job. Another example has to do with the printer font cache, which caches characters across jobs. Not only will a given job print more quickly once the characters it uses have been executed and cached, but the same job sent again will run faster the second time, because it will find the previously used characters already in the cache.

We recommend that you re-boot the printer between test jobs when sending test files directly, and that the same job be sent a second time before re-booting. For text-based files, the second time might more accurately reflect what an end user will see. An allied strategy for sending files from within applications is to send a test page containing no text (perhaps a rectangle) to get the application prolog to download and then to send the same test job twice before re-booting the printer.

Click-to-clunk timing has its limits in software development, where performance improvements made in one area might mask performance degradation in other areas. Developers need a way to get inside the PostScript language program files their applications produce to more accurately understand the files’ performance. Fortunately, there are PostScript language operators and methods useful for evaluating PostScript language program performance. These operators, strategies for their use, and some examples of timing tools are discussed in section 4.
4 PostScript Language Timing Methods

All timing methods that enable a developer to evaluate a PostScript language program file and determine where there might be opportunities to make it produce a printed document more rapidly depend on the support of PostScript language operators. It is possible on systems that provide two-way communication with the printer, to do stopwatch timing of intervals inside programs if those intervals are not too small. To do so, the developer flags the interval with print instructions.

```plaintext
... PostScript language statements ...
% about to begin timed section
(Begin Timing) print flush

... code section being timed ...

(End Timing) print flush
... possibly some more statements ...
```

The accuracy of the above method may be less than satisfactory, and also will not be very helpful to developers lacking two-way communications with their test printers.

Two PostScript language operators, usertime and realtime, save developers from using a stopwatch. All timing tools and strategies not based on stopwatch timing rely on these operators, which provide the same function in two slightly different ways.

4.1 usertime and realtime

Both usertime and realtime are operators documented in the PostScript Language Reference Manual, Second Edition. The usertime operator was originally the only timing operator available.

These time-related operators support valuable timing tools for making timed interval comparisons on the same printer. Each returns the value of a clock in millisecond units. The actual size of the “milliseconds” reported and the consistency of the elapsed time measurement can vary from printer to printer. The intervals at which the realtime clock updates its value can vary; for instance, a particular realtime clock might increment only in 16 millisecond steps. The time reported by each operator may also include communications overhead.

The realtime operator is usually preferred over usertime for interval timing.
4.2 Timing an Interval

Click-to-clunk timing only allows timing of a PostScript language file as a single lump. Interval timing allows a developer to bracket sections within the file in order to decide where time is being spent, and also allows timing of those sections of PostScript language code for which potential performance enhancements are being explored.

For instance, a developer who needs to draw a lot of rectangles on a PostScript device might want to explore alternate ways of drawing those rectangles, and to time only that section of code within the PostScript language program file that does the actual work of drawing the rectangles. An example of using interval timing to determine optimal ways of performing this task is found in Technical Note #5126, “PostScript Language Code Optimization: Rectangles.”

The following procedures time the PostScript language code placed between them:

```
/systemdict /realtime known
{/timerop (using realtime: ) def
 /gettime /realtime load def }
{/timerop (using usertime: ) def
 /gettime /usertime load def }
ifelse

/beginTiming { % - beginTiming - begin interval timing
 /tmptime gettime def
} bind def

/endTiming { % - endTiming - end interval timing and save result
 gettime tmptime sub /elapsedtime exch def
} bind def

beginTiming
... PostScript language code being timed ...
endTiming

(Time obtained ) print timerop print elapsedtime == ( ms) print
```

These procedures present a problem for developers who have not established two-way communication with the printer they are using. Alternate methods for obtaining the elapsed time are shown in the following sections.

4.3 Reporting the Time

As shown in section 4.2, it is quite easy to define a procedure that reports back to the host the elapsed time saved by timing procedures, which allows for easy capture into spreadsheets, and so on. The following code reports the elapsed time back to the host.
Such a procedure can be inserted into the file being downloaded at whatever point reporting is desired. It could also be included as part of the endTiming procedure. However, this type of reporting will not be useful on all systems.

It is also possible to print results directly on the page of the job being run. The following code prints the results of the job being run.

```
/reportTime { % - reportTime - report value of elapsed time on page
gsave
  0 0 moveto 0 72 rlineto 288 0 rlineto 0 -72 rlineto closepath
  1 setgray fill % make sure report will be visible
  /Times-Roman findfont 8 scalefont setfont 0 setgray
  40 56 moveto currentpoint % initial line position
  statusdict begin
    /product where { pop product show ( ) show } if
    /printername where
    { pop { known as } show 64 string printername show } if
  end moveto 8 -10 rmoveto % next line
  (elapsed time measured ) show timerop show
  elapsedtime 10 string cvs show ( milliseconds) show
  grestore
} bind def
```

Examples of downloadable PostScript language program files that define similar timing and reporting procedures are found in Appendix A.

### 4.4 Factoring Out Communication Time

An interval timed using either `usertime` or `realtime` often includes overhead that occurs during that interval due to communication time. This overhead can represent a very large proportion of the time reported for the interval. It is important to know the time that includes communications overhead. However, in the situation where two strategies give the same time result, it is useful to know whether the two strategies also have a similar execution time, excluding that overhead.

A strategy with a faster execution time might perform better than the other on printers with a faster communications channel than the printer currently being tested; also, it might give better performance over the same channel
combined with a faster printer. The ability to factor out communications time permits evaluation of performance in terms of both data size reduction and execution speed.

A developer who tailors files to test specifically the aspect of PostScript language programming practice under evaluation can get an idea of total time versus execution time by performing the operations under test in a loop. If the original test file has a timed interval of one hundred circular object procedures, the file for determining execution time might call the circle procedure in a repeat loop:

```
100 { circleproc } repeat
```

as opposed to the original file’s more usual inline invocation of the procedure:

```
circleproc
circleproc
circleproc
circleproc
circleproc
... and so on ...
```

The data transmission time is then trivial compared to the execution time.

Another method for factoring out communication time, and thus greatly reducing data transmission time, involves running the original test file from an SCSI device (usually the printer hard drive). This makes it unnecessary to write loop versions of test code sufficiently similar to the original code to give useful results. The following PostScript language code, inserted at the beginning of a test file, downloads that test file to disk using the file name specified:

```
/buff 128 string def
/fd { ... filename ... } (w) file def
{ %loop
  currentfile buff readstring { %ifelse
    fd exch writestring
  }{ %else
    dup length 0 gt { %ifelse
      fd exch writestring
    }{ %else
      pop
    } ifelse
    fd closefile
    exit
  } ifelse
} bind
% file should follow the "loop" token....
loop
```
The code above appears in PostScript Language Program Design, section 13.8, listing 13-2. The file can be executed subsequently from disk for timing. This is done by downloading a file containing the statement

\[( \ldots \text{filename} \ldots ) \text{ run} \]

### 4.5 Factoring Out Page Printing Time

Factoring out the print engine part of the pipeline may not be realistic. It is possible to obtain the execution speed of some code distinct from the page-per-minute speed of a particular printer, however, if the interval being timed includes `showpage` or its equivalent, as in job-level interval timing.

There are two main approaches to factoring out the print engine. One approach includes page printing, but subtracts its estimated duration, as in `timepages.ps`—a PostScript language program timing utility available from the Adobe Developers Association.

Another approach disables the print mechanism. On PostScript Level 1 devices, it is possible to disable paper motion and therefore avoid including that time in the measure of elapsed time by defining `#copies` to 0.

**Note** The `#copies` variable controls the behavior of `showpage`; it is documented in the “PostScript Language Reference Manual, Second Edition.”

On PostScript Level 2 devices, it is preferable to use the operator `setpagedevice` to disable paper motion. The following code allows all operations, except producing the printed page, to occur as usual.

```
<</OutputPage false>> setpagedevice
```


Also note that while page printing is disabled, a report procedure cannot print results on a page of the job being run.
Appendix A: PostScript Language Interval Timing Utilities

1 Adding Begin/End Timing Procedures to userdict

Download the following file to make beginTiming and endTiming available to subsequently downloaded files. Note that in their current form, beginTiming and endTiming require a separate call to reportTime for a report of the elapsed time. These procedures and the following report procedures are the basic building blocks from which more complex and powerful tools may be built.

%!PS-Adobe-3.0 ExitServer
%%Title: Download Interval Timing Tool Procedures
%%Creator: RH
%%CreationDate: July 19, 1991
%%For:
%
%%EndComments
%%BeginExitServer: 0
serverdict begin 0 exitserver

/TIMETOOL (incomplete) def
/elapsedtime 0 def
/timeToolDict 20 dict def

systemdict /realtime known
{ /timerop (using realtime: ) def /gettime /realtime load def }
{ /timerop (using usertime: ) def /gettime /usertime load def }
ifelse

/beginTiming { % - beginTiming - begin interval timing
    /tmptime gettime def
} bind def

/endTiming { % - endTiming - end interval timing
    gettime tmptime sub 1000 div /elapsedtime exch def
} bind def

/TIMETOOL (interval) def
2 Installing a Report Procedure in userdict

To see the result of timing an interval, it is necessary to somehow report that time. Many people prefer to have the time reported directly to the host, while others prefer to have the time printed on paper. Depending on the preference, use the code in section 2.1 or 2.2 that seems most suitable. Switching between the two reporting methods is as simple as downloading the desired file.

2.1 Reporting to the Host

Download the following file if times can and should be reported on screen.

```
%!PS-Adobe-3.0 ExitServer
%%Title: Download Report To Screen Timing Tool Procedure
%%Creator: RH
%%CreationDate: July 19, 1991
%%For:
%
%%EndComments
%%BeginExitServer: 0
serverdict begin 0 exitserver

/reportTime { % - reportTime - report value of elapsedtime to host
    (Time obtained ) print timerop print elapsedtime == ( seconds)
    print
} bind def
```
2.2 Reporting on the Printed Page

Download the following to cause reportTime to report results on the printed page.

%!PS-Adobe-3.0 ExitServer
%%Title: Download Report On Page Timing Tool Procedure
%%Creator: RH
%%CreationDate: July 19, 1991
%%For:
%
%%EndComments
%%BeginExitServer: 0
serverdict begin 0 exitserver

/reportTime { % - reportTime - report value of elapsed time on page
  save
  0 0 moveto 0 72 rlineto 288 0 rlineto -72 rlineto closepath
  1 setgray fill % make sure report will be visible
  /Times-Roman findfont 8 scalefont setfont 0 setgray
  40 56 moveto currentpoint % initial line position
  statusdict begin
    /product where { pop product show ( ) show } if
    /printername where
      { pop ( known as ) show 64 string printername show } if
    end
  moveto 8 -10 rmoveto % next line
  (elapsed time measured ) show timerop show elapsedtime 10 string cvs show ( seconds) show
  restore
} bind def
3 Using the Timing Procedures

Using the timing procedures is quite straightforward: Use beginTiming where timing should begin, and endTiming where timing should end. If using the report-on-the-page version of reportTime, it might be wise to use that procedure just before a call to showpage to avoid writing over the results. The report-on-the-page version of reportTime in section 2 does not ensure that the default coordinate system is in effect before placing its report on the page.

An example of the use of the timing procedures:

```
.001 setlinewidth

beginTiming % start timing
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
200 600 moveto 200 700 lineto 400 700 lineto 400 600 lineto 200 600
lineto stroke
endTiming % stop timing and save result

reportTime % show result

showpage
```
Appendix B: Changes Since Earlier Versions

Changes since August 13, 1991 version

• Document was reformatted in the new document layout and minor editorial changes were made.
Index

A
application software  7

B
beginTiming  17, 20

C
communication time
   factoring out  14–16
communications channel  8
#copies  16

device driver
   speed  8

D
device driver
   speed  8

e
endTiming  17, 20

F
files
   timepages.ps  16

H
host platform  6

I
interval timing  13

O
on screen reporting  18

P
PostScript interpreter  9
print engine  10
   factoring out  16
printed page reporting  19
printer controller board  9

R
realtime  12
reportTime  17, 19, 20

S
setpagedevice  16
system components  5–10
   document production system  6

timepages.ps  16
timing  5–16
   click-to-clunk  10–11
      between PostScript language
      jobs  11
      methods  12–16
      reporting elapsed time  13
      studies, main areas  6
      timing procedures
         using  20

U
userdict
   adding beginTiming and
      endTiming  17
   installing a report procedure  18
usertime  12