PostScript Speedup Secrets

Don Lancaster

Those long PostScript page makeready times are simply not needed. Our resident PostScript guru reveals how to optimize your way around them.

The great PostScript general purpose computer language from *Adobe Systems* is often wrongly accused of being slow. In reality, the latest versions of PostScript can be made blindingly fast. With some care, all of those long page makeup times can be outright eliminated.

They are flat out not needed.

The problem lies not in today's fast PostScript language, but in klutzy applications code, the abysmally bad drivers, glacial comm, and the poor end user understanding of the "not me!" con that's coming down.

Ferinstance, when I'm reprinting a self-collating duplex page in one of my new Book-on-demand published volumes, I'll routinely have a page makeup time of *zero* to four seconds. Many pages typically print at the full mechanical speed of the printer. Page after new page after new page. That is for a 6000 character multi-column page with an ultra fancy justification, headers, footers, and one or even two technical figures having a medium complexity. All done from a typical page file that is between 8K and 12K in length. I also consider the "speed" measurements you will find in most PostScript printer reviews and ads to be laughingly absurd.

Let us look at some of the secret insider tricks to blinding PostScript speed. Some of these will apply to anyone anytime. The others are best applied as speed optimization hacks specifically for any copy that is to be reprinted a number of times in the future. We might as well...

Start With The Obvious

By far your quickest and cheapest PostScript speedup trick is to *leave the printer on between jobs!* Whenever any font character first gets used, it gets built up from an outline description. A bitmap copy of that character then gets saved to a *font cache* for potential later reuse on current or future jobs. Reuse of a cached character can be as much as one thousand times faster, which translates to an overall 2:1 to 3:1 speedup. On power down, your old font cache vanishes.

A second obvious ploy is to *make sure the printer resets after each job.* To a solid green LED display. Otherwise several minutes will pass by before a timeout and allowable reuse. Most drivers do take care of this resetting automatically. If not, you can inquire your printer status by a control-T, and reset it with a control-C followed by a control-T one second later. Be certain to get the *status:idle* message back onto your host screen.

One rather popular speedup stunt is to disable your test page. But you should not be turning your machine off and on that often in the first place. And losing your first job because the toner cartridge needed shaken or the density dial needed adjusted negates any time saved. Finally, a test page that starts taking excessively long to output could warn you of impending hard disk disaster and the need for an immediate rebuild.

Most systems do offer a software utility or front panel option to select a test page. Ultimately, the following PostScript code is generated...

```
serverdict begin 0 exitserver
statusdict begin
false setdostartpage end quit
```

Your choice of a PostScript printer and firmware can make an incredible speed difference. We have now gone through five or more generations of firmware, each of which gave you a thirty percent speedup from earlier versions. Except for that quantum leap up to PostScript level II, which more than doubled throughput.

Thus, for maximum speed, you'll want to be certain to use a genuine Adobe PostScript level II and a RISC, "turbo", "co-processed" or otherwise

enhanced printer controller board.

One fine source of retrofit printer upgrades and speedups is *Thompson and Thompson* at (714) 855-3838.

Many mid- to high-range printers include a local SCSI hard disk option. If a hard disk is available, do be sure to use it. First, because you now have a "permanent" and a much expanded font cache. Second, because you can now immediately access hundreds of fonts in a fraction of a second.

Third, because you can put files, utilities, and downloads onto disk. Eliminating the need for any comm hassles. And fourth, your hard disk lets you do many longer production tasks largely unattended by either a human operator or a computer.

Two tips: The Apple LaserWriter printers do not necessarily require Apple SCSI hard drives. But they do *demand* a SCSI drive that can return its *size* back to the printer. Be sure to get a written compatibility guarantee before buying any third party drive. Most other PostScript printers also demand SCSI drives that can return their size to the requestor.

Also, some hard disks can *not* be initially formatted with a PostScript printer. Instead, you *may* have to do your very first track-creating SCSI formatting using some other host machine. And then *reformat* with PostScript. Watch this detail.

The Click-to-clunk Curve

Your central key to all PostScript print time speedups will lie in first understanding and then optimizing the top secret *Click-to-clunk* curve.

The *Click-to-clunk* parameter is the ultimate end user measurement of printer throughput. It is a measure of the time which will elapse between your keystroke or the mouse "click" requesting a page and the "clunk" of that page dropping onto the output tray. Factors which enter into your Click-to-clunk curve do include your host processing speeds, your driver klutziness, network behavior, the de-facto effective baud rates, your comm supervisory overhead, and the actual printer mechanical speed.

Oh yes, the speed of your current PostScript interpreter sometimes might also have a slight effect here. But often a *totally negligible* one.

And, as we'll shortly see, *faster* PostScript code might even end up *slowing you down!*

A typical Click-to-clunk curve is shown in figure one. It turns out that *for maximum possible speed on any page, there is one clearly optimum source file length associated with any PostScript printing job.* This optimum dip very much shifts with your selection of printer, host, comm, hard disk usage, and programming style. But a good "in the dip" solution for what you have now often will form a superb starting point for future speedups.

Why the unusual curve shape? The answer to this lies in how PostScript works. Some characters first have to be sent to the Postscript interpreter. Sooner or later, enough characters are received that the interpreter can start doing something useful.

Then a race begins.

Characters go in at a comm speed I like to define as the *effective baud rate*, set regardless of which selected comm method is in use. The effective baud rate does include all of your supervisory handshaking, host file operations, network collisions, and any similar delays.

Anyhow, characters get consumed at a rate that is set by the PostScript interpreter's needs. Eventually all characters are received and finally processed. Sometimes the characters pile up in an input buffer; other times the interpreter has to sit around and wait for new input. Your optimum speed obviously happens whenever the characters arrive at precisely the speed they can be processed, with no pileups and no waiting.

The longer file lengths are often associated with simpler PostScript processing tasks, while the short file lengths often may require extensive PostScript computation. Hence the sharp dip in the Click-to-clunk curve at a clearly defined optimum.

There are three distinct areas to the Click-to-clunk curve. Each area has profoundly different implications for your speedup optimizations...

Area "III" – Baud Rate Limited

In area III, you will be *baud rate limited.* The interpreter is starved for characters and has to wait around twiddling its bit buckets until more data arrives. *Most casual PostScript users most of the time usually end up ridiculously baud rate limited.*

Some typical effective baud rates are shown to you in figure two. In general, and regardless of the comm

method you use, it will be far slower than you suspect. It is thus super important to both measure and know all your effective comm rates. Until quite recently, AppleTalk PostScript comm was actually *slower* than plain old serial comm. Which led to the hilarious ridiculosity of that Apple IIe game paddle port ending up four times faster than the best and fastest Macs whenever it came to sending PostScript code. But don't smirk just yet. The game paddle port at a totally honest 59600 baud was also twice as fast as IBM parallel ports and eight to ten times faster than most IBM serial communications.

Note that many of the brand new *Ethernet* comm schemes still have to overlay *AppleTalk*, and thus will only end up modestly faster.

By far your optimum PostScript comm is to talk directly to your SCSI hard disk or to create some sort of *Shared SCSI Comm* that lets both your host and your printer interact with the same hard disk or CD drives. This gives you effective baud rates in the one million range. Sadly, the major players are all less than enthusiastic over properly documenting and then providing for shared SCSI comm. So, you have to set this up by yourself on a custom basis.

Although *adjudication* is the main criticism against a multi-host shared SCSI comm, it can become an easily avoided non-issue in the PostScript environment. More info on shared SCSI comm appears on *GEnie* PSRT.

If you are baud rate limited, the speed of your PostScript processing does not matter in the least. It is thus totally pointless to try and do any PostScript area III code speedups.

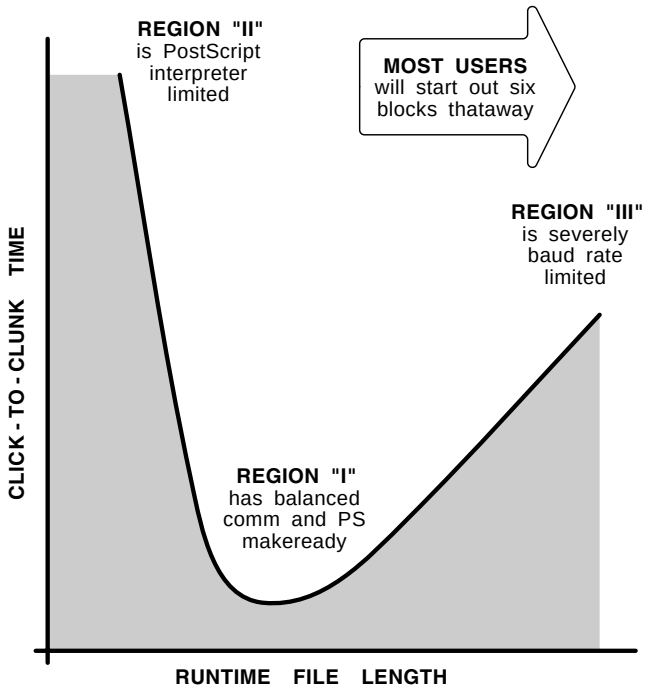
In fact, any PostScript speedups done while in area III may even *slow you down*, if your text files happen to get *longer* in the process.

Your big two area III optimization strategies are obvious: *Measure and raise your effective baud rate. Then you shorten and rearrange your files.*

The simplest way to measure your effective baud rate is to make up a text file of 10,000 or 100,000 comment characters, ending with a *showpage* command. Then, you use a plain Jane stopwatch to measure your de-facto Click-to-clunk time. For simplicity, assume *ten* bits per character.

Be *absolutely certain* to select a PostScript comm, print buffering, and networking method which give you a full *two way* data transfer, that include the on-screen error messages and your ability to fully record host returned data. More details on this appeared in my *PostScript Startup Secrets* in the Dec/Jan 1992 issue of *PC Techniques.*

▲ FIGURE 1 – The secret PostScript Click-to-clunk curve.



Note that a sending echo on your host screen can dramatically reduce your PostScript comm speeds. Watch your characters going out *only* when debugging.

To reduce the characters in your files, first print-to-disk. And then inspect the runtime files with a word processor or editor. A first and very obvious step is *send nothing that is not needed.* Comments can be reduced or eliminated. Downloaded preambles, dictionaries, or fonts can get done once persistently rather than on a job-by-job basis. Or read from hard disk. If several figures need the same dictionary, consider sending just *one* dictionary with common access.

Characters can get eliminated by throwing away the spaces adjacent to self-delimiting symbols. Changing numeric formats to avoid leading zeros and excessive accuracy can also dump unneeded characters. There's not normally any logical reason to request any page location to better than 0.2 pixel accuracy.

Shorter proc names can make a surprising difference in file lengths. Use *mt* or just *m* instead of *moveto*, and so on. But be careful with *lineto*, because *lt* and *ln* are both reserved commands. *li* works fine here.

Another character reducing ploy is to eliminate those oft retransmitted characters such as the "0", "0" and "32" in most *awidthshow* commands.

Consider doing more with your PostScript computer and less with your host. By giving PostScript more to do, you can usually shorten your "do this" files and thus move away from a severe baud rate limit.

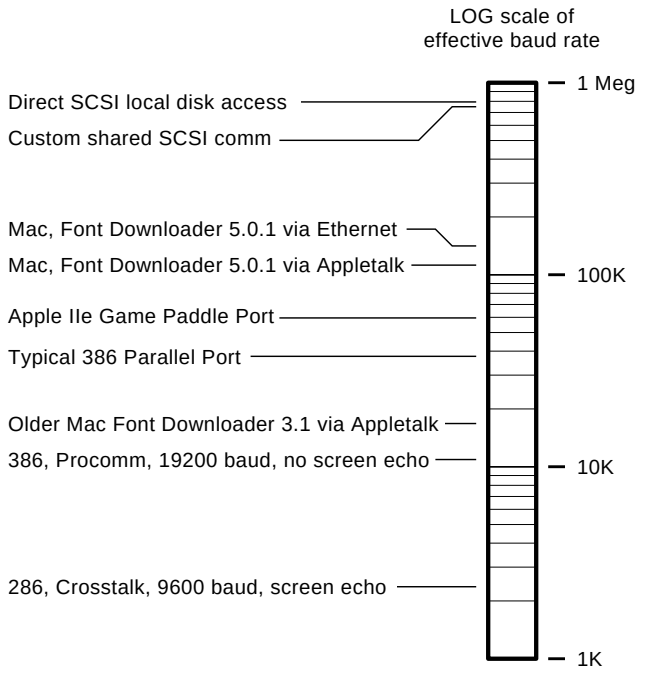
One blatant example is to justify entire text lines using *awidthshow*, rather than individual words using *ashow*. This can sometimes shorten your files by a factor of three.

Yet another area III ploy is to *give PostScript something intense to do early in your file*, so that characters can pile up at the same time that PostScript is busy on something exotic. While a great paper idea, this may be hard to implement. Especially since many files start with long preambles.

Area "II" – Processor Limited

Much more rarely, you might end up in area II, where you really will be PostScript *processor limited*. In this case, enough characters are always available and the PostScript speed is in fact your bottleneck.

▲ FIGURE 2 – Some popular PostScript effective baud rates.



For instance, a full page fractal fern can be requested with only a few hundred bytes of code. The math-intensive results will take far longer to generate than it does to send the code, even at a glacial comm rate.

Your specific solution to the fern problem is to do the fern *once* and capture it back to host as a reusable *image* file. Your general solution to area II speedups is to *give PostScript fewer and simpler tasks to do, even if these new tasks may require somewhat longer file lengths.*

Timing utilities could get used to measure which tasks take how long. This might get tricky, though. Some operations do get deferred till paper feeding time. Note that baud rate limited comm times could easily be separated from execution times by defining and deferring a proc, then executing it later inside a timing loop. Note also that short times can be accurately measured by repeating the process a hundred or a thousand times inside of a *repeat* loop.

Some obvious speedup tricks here include avoiding irregular clipping intervals, pixel line remapping, large oddball screens, trig, and fractals.

Excessive detail can chew up time. On typical engineering curves, there is usually no point in plotting things beyond sub-pixel accuracies. Use the minimum number of points you can to still give you just barely acceptable results at your target resolution.

Inelegantly selecting your fonts is a great way to waste both proc time and virtual memory. Do observe that *findfont*, *scalefont*, and *makefont* are all slow and gobble up VM. But *setfont* is fast and needs zero VM. Thus, you should initially predefine all of your fonts as *prescaled font dictionaries*. Then call them later as needed with *setfont*. Ferinstance, at the start of your file, do a…

```
/font1 /Helvetica findfont
10 scalefont def
```

Note the absence of {..} deferred action. Which *immediately* predefines *font1* as an ultra powerful, but little used and less understood *prescaled font dictionary*.

Later on, as needed, do a…

```
font1 setfont
```

My favorite PostScript speedup tool is *table lookup*. Which, besides walking the dog and dispensing soft ice cream, can leap tall buildings in a single bound. *Never calculate (or even test for) something that you can look up instead!*

While PostScript seemingly lacks the *case* command of other high level languages, this elegant construct quickly handles option picking and loads more…

```
[{proc0} {proc1} ... {procn}]
exch get exec
```

Any zero on stack quickly executes *proc0*. A one does *proc1*, and so on. Test and logic free. Note that table lookup is also handy for redefining characters or actions taken on any given character.

PostScript level II offers a slew of new features that could lead to very dramatic speedup possibilities. My favorites here now include *cached user defined procs, forms capabilities, newly open font path grabbing, binary sequences* and file conversion *filters*.

Your most powerful and the most general PostScript speedup tools do involve *compiling* and *distilling*. We will look at these shortly. Because of the extra front end time and operator effort, these tools are best used on files that are to get reprinted at least once in the future.

Area "I" – Balanced Execution

The dip in area I is the best you can do with what you have so far. Your optimization strategy to go beyond area I is to *pick a new Click-to-clunk curve*. You might do this with faster hardware, less-glacial comm, newer firmware, rethunk apps, improved algorithms, better hard disk usage, a more elegant programming style, and a fresh look at what really needs done and what does not.

Once again, your Click-to-clunk curve very much depends upon you, your hardware, your applications, the comm, and your programming style. But any present "in the dip" solution is usually a good starting point for future speedups.

Using Images

The PostScript *image* operator can be a very powerful tool – whenever it is not being severely abused. While great for high quality photographic halftones, it is totally inexcusable to use *image* as a sloppy crutch to avoid "real" PostScript translations from other graphics standards. Or to use *image* for a plain old logo where a few dozen bytes of hand-crafted or application-traced code can do a far cleaner job much faster.

Typically, a PostScript 300 DPI image needs 22K bytes per square inch per bit of gray level per color. It is thus rather easy to get up into megabyte file sizes. And, of course,

severely baud rate limit yourself.

So, step uno on image speedup is to *never use that image operator when much shorter lower-level PostScript code can reasonably be substituted.* One trick that dramatically simplifies logos is to capture the actual full sized paths of those few special font characters used. This eliminates either the need for bulky image code or the need to tow along an entire special font.

Second, if possible, do keep any essential images on a local SCSI hard disk or CD ROM to avoid baud rate comm limits. Third, always use the minimum image size, resolution, and number of gray levels.

Fourth, watch your *image* string coding. The *image* operator expects a string of data values. On your typical non-transparent serial data channels, this used to need a hex-ASCII coding of *two* characters per byte.

PostScript level II allows you to substitute an *ASCII85* coding which recodes four bytes as five printable characters. You also have the option of using a transparent data channel or *binary encoding* formats. Any of these tricks can cut your image file sizes nearly in half.

Finally, PostScript Level II offers lots of brand new image compaction schemes. These include run length encoding, fax, DCT, LZW, and TIFF. See the Red Book II for details.

A tradeoff is involved between processing times and compressed file sizes. One that is highly application dependent. Thus, the extreme image compression algorithms may not help as much as you expect.

Compiled PostScript?

Pretty near any general purpose computer language will give you an option of *interpreting* or *compiling* your code. Compiling usually gives you much faster executing code that needs far less in the way of machine resources. Compiling also often can divorce your final output from the applications generating code, leading to license-free and restriction free run-time executions. Disadvantages of compiling include the extra front end time and effort required, and that compiled code is often very hard to read or edit.

Compiled code often gets longer, and sometimes appallingly so.

A true compiling is tricky to do in PostScript, so we will use a looser *pseudocompiling* definition of "Do stuff once now to speed up future print times". Compiling is best for Book-on-demand publishing where you already know you will do lots of reprinting in the future. On very long or complex files, compiling could be used for one-use phototypesetting. Especially if a few minutes head end work can save you hours of big buck typesetting charges.

Another really big advantage of PostScript compiling is that you can now create compiled EPS files that can be mailed, emailed, or modemed anywhere. Device independently.

This ends up insanely great for such things as giving first generation digitally mastered printed circuit layouts to readers of an electronic magazine. Or dialplates, meter faces, layout templates, any quilt patterns, weaving drafts, or craft decals. As plain old textfiles that will quickly run on *any* computer. With no trace (and thus no royalties) of the fancy CAD-CAM layout or whatever code that created the original art.

PostScript has a *bind* operator that substitutes underlying procedures for name lookups when it gets used. This gives you a "free" ten to fifteen percent speedup. On the other hand, binding takes time. So it should not be used if a proc is only rarely called. Binding also locks you into all your definitions as they exist at the time *bind* is executed. So you cannot later change the meaning or purpose of bound code.

PostScript Level II also offers *user paths*. This works the same as the font cache. The first time your path gets used, it gets executed in the usual way *and* saved as a bitmap to your cache. Reuse of your path can be hundreds or even thousands of times faster. On any business card where most of the time is spent on the logo, user paths could give you nearly a twelve times speedup, since the final eleven logos come out of the cache.

Tricks with forms also give you somewhat of a compiling action. On older PostScript, you would put a background down, add a first name and address, and then do a *copypage*.

Then you erase *only the name and address*, put down the next address, and use *copypage* again. There's no need to reprint the fancy background each time. Level II PostScript makes background sharing even faster by way of a cached *forms* capability.

▲ FIGURE 3 – Typical Adobe Distillery "compiled" PostScript code.

```
/bdef { bind def } bind def /ldef { load def } bdef
/W { moveto widthshow } bdef /AW { moveto awidthshow } bdef
/MF { exch findfont exch makefont setfont } bdef
/DF { selectfont currentfont def } bdef /F /setfont ldef

/Palatino-Bold [10 0 0 10.1 0 -8 ] MF
-0.3 0 32 0.1 0 (Start  With  The  Obvious) 236.339 325.1 AW
/F1 /Palatino-Roman 9.5 DF
0.292065 0 32 0.198678 0 (By far your quickest and cheapest ) 225.0 303.1 AW
-0.292759 0 32 0.101207 0 (PostScript speedup trick is to ) 215 292.1 AW
/F2 /Palatino-Italic 9.5 DF
-0.292759 0 32 0.101207 0 (leave the) 341.399 292.1 AW
F2 F -0.144955 0 32 0.125841 0 (printer on between jobs!) 215 281.1 AW
F1 F -0.144955 0 32 0.125841 0 ( Whenever any) 310.039 281.1 AW
F1 F 0.428717 0 32 0.221453 0 (font character first gets used, it gets ) 215 270.1 AW
F1 F 0.2575 0 32 0.192917 0 (built up from an outline description. ) 215 259.1 AW
F1 F 0.0425076 0 32 0.157085 0 (A bitmap copy of that character then ) 215 248.1 AW
F1 F -0.171851 0 32 0.121358 0 (gets saved to a ) 215 237.1 AW
F2 F -0.171851 0 32 0.121358 0 (font cache) 280.246 237.1 AW
F1 F -0.171851 0 32 0.121358 0 ( for potential) 319.364 237.1 AW
F1 F 0.380548 0 32 0.213425 0 (later reuse on current or future jobs. ) 215 226.1 AW
F1 F -0.0919617 0 32 0.134673 0 (Reuse of a cached character can be as ) 215 215.1 AW
F1 F 0.799381 0 32 0.28323 0 (much as one thousand times faster, ) 215 204.1 AW
F1 F 0.548458 0 32 0.24141 0 (which translates to an overall ) 215 193.1 AW
/F3 /Palatino-Roman 9 DF
0.548458 0 32 0.24141 0 (2:1) 351.11 193.1 AW
F1 F 0.548458 0 32 0.24141 0 ( to ) 363.085 193.1 AW
F3 F 0.667083 0 32 0.26118 0 (3:1) 215 182.1 AW
F1 F 0.667083 0 32 0.26118 0 ( speedup. On power down, your ) 227.034 182.1 AW
F1 F 0.06 0 32 0.16 0 (old font cache vanishes.) 215 171.1 AW
F1 F 0.647822 0 32 0.25797 0 (A second obvious ploy is to ) 225.0 160.1 AW
F2 F 0.647822 0 32 0.25797 0 (make ) 354.604 160.1 AW
F2 F 0.144363 0 32 0.17406 0 (sure the printer resets after each job) 215 149.1 AW
F1 F 0.144363 0 32 0.17406 0 (. To ) 358.473 149.1 AW
F1 F -0.119264 0 32 0.130123 0 (a solid green ) 215 138.1 AW
F3 F -0.119264 0 32 0.130123 0 (LED) 272.552 138.1 AW
F1 F -0.119264 0 32 0.130123 0 ( display. Otherwise) 290.906 138.1 AW
F1 F 0.68911 0 32 0.264852 0 (several minutes will pass by before  ) 215 127.1 AW
F1 F 0.214405 0 32 0.185734 0 (a timeout and allowable reuse. ) 215 116.1 AW
showpage

% Total non-header length:        2088 bytes
% Baud rate limited run time:     1.09 seconds at an honest 19200 baud
% Raw PostScript run time:        0.31 seconds on LaserWriter G
```

▲ FIGURE 4 – Typical Guru Double Distilled PostScript code.

```
/b {bind def} bind def
/D {exch findfont exch scalefont setfont} b
/M {exch findfont exch makefont setfont} b
/a {moveto 0 32 4 2 roll 0 exch awidthshow} b

/Palatino-Bold [10 0 0 10.1 0 -8 ] M
-.3 .1(Start With The Obvious)236 325 a
/Palatino-Roman 9.5 D
.29 .2(By far your quickest and cheapest)225 303 a
-.29 .10(PostScript speedup trick is to)215 292 a
-.14 .13( Whenever any)310 281 a
.43 .22(font character first gets used, it gets)215 270 a
.26 .19(built up from an outline description.)215 259 a
.04 .16(A bitmap copy of that character then)215 248 a
-.17 .12(gets saved to a)215 237 a
-.17 .12( for potential)319.3 237 a
.38 .21(later reuse on current or future jobs.)215 226 a
-.09 .13(Reuse of a cached character can be as)215 215 a
.80 .28(much as one thousand times faster,)215 204 a
.55 .241(which translates to an overall)215 193 a
.55 .24( to)363 193 a
.67 .26( speedup. On power down, your)227 182 a
.06 .16(old font cache vanishes.)215 171 a
.65 .26(A second obvious ploy is to)225 160 a
.14 .17(. To)358.4 149 a
-.12 .13(a solid green)215 138 a
-.12 .13( display. Otherwise)290.9 138 a
.69 .26(several minutes will pass by before)215 127 a
.21 .19(a timeout and allowable reuse.)215 116 a
/Palatino-Italic 9.5 D
-.29 .10(leave the)341.3 292 a
-.14 .13(printer on between jobs!)215 281 a
-.17 .12(font cache)280.2 237 a
.65 .26(make)354.6 160 a
.14 .17(sure the printer resets after each job)215 149 a
/Palatino-Roman 9 D
.67 .261(3:1)215 182 a
.55 .24(2:1)351 193 a
-.12 .13(LED)272.5 138 a
showpage

% Total non-header length:      1375 bytes
% Baud rate limited run time:   0.716 seconds at an honest 19200 baud
% Raw PostScript run time:      0.22 seconds on LaserWriter G
```

One extremely versatile PostScript compiling process involves a popular shareware program called…

The Adobe Distillery

To use the Adobe Distillery, your PostScript code has to be in the *exact* final form you want it, and you have to want to reprint your code at least one time in the future. You also must definitely *not* be baud rate limited, since the somewhat longer distilled files will profoundly speed you up in area II but significantly *slow you down* in area III. Thus, distilled code is best used from a local SCSI hard disk.

At any rate, the distillery simply asks "What is the absolute minimum information needed to make full size marks on this page?" That essential information is returned to your host for recording. Thus, instead of doing any nonlinear perspective graphics transformation, *only the fixed position results* of that transformation will get saved. Instead of making complex fill justification calculations, *only those specific results* get saved.

Figure three shows an example of distilled code. Properly applied, the Distillery can dramatically speed up your files. It can sometimes shorten them if any long preambles can be eliminated.

There are several Distillery bugs, such as ignoring superscripted and subscripted fonts unless their height and width differ. The clipping is not done properly. The Distillery doesn't know how to reuse an existing font path for opaque icons and such.

Because it trys to process anything from anybody, the distillery is also very much program style dependent. So you may want to work with it for a while before you can get in sync with it and best tap its abilities.

Certain program constructs could cause the Distillery to generate lots of unbelievably long code. So you just might want to manually intervene. Use Distilled results for the majority of stuff that Distillery does well. And revert to bits of the origional code or substitute a hand crafted something whenever you can clearly gain any speed or size advantage.

To go beyond the Distillery, I use a sneaky trick I call…

Double Distilling

Properly used, Distillery files are fast and compact. But I found I could make the code even faster and more compact by playing around with each individual Distillery line. One example appears in figure four. I often gain 33 percent length and 25 percent speed.

The modified lines eliminate any characters that are not really needed, reformat all numbers to drop leading zeros, and reduce the final position accuracy to one tenth of a point. I also rearrange the line positions in the file so that each font only will get chosen *once* per page.

Double distilling is very much a custom process. I've uploaded a MAUDEDOC.PS tool to *GEnie* PSRT that could get you started. This is a very general tool. It lets you scan any document line by line and modify whatever lines you wish in any way you care to.

You could further triple distill by going to binary format, but you may lose human readability (and what limited editability now remains) in the process. Besides, most of my files already do have page makeup times much faster than mechanical feed times, leading to a virtually "zero" page makeready time.

But the whole game starts all over again when you switch to a 17 PPM high end printer.

Sigh.

For More Info

More info on PostScript in general appears in that *blue* book (Adobe's *PostScript Cookbook*), in the *red* book (Adobe's *PostScript Reference Manual II*) and also in my *LaserWriter Secrets* book / disk combo. I try to stock these and several other major PostScript resources here at *Synergetics*.

I've also posted lots of PostScript optimization utilities plus detailed distilling examples to *GEnie* PSRT. You can call (800) 638-9636 for voice connect info.❖

Microcomputer pioneer and guru Don Lancaster is the author of 27 books and countless tech articles. Don runs a no-charge technical helpline found at (602) 428-4073, besides offering all his own books, reprints, and his technical services. He also has a free brochure of his insider desktop publishing secrets waiting for you. Your best calling times are often 8-5 on weekdays, Mountain Standard Time. Or you can reach Don by way of Synergetics, *at Box 809, Thatcher, AZ 85552.*