TEX as described in section 453 on page 157 of the August 1981 revision. This error affected the use of *leqno*. The correction involves replacing the word *linq* with *link* and adding a missing line of *shift* := 0.0; as shown below:

```
453. ⟨Attach equation number 453⟩ =
  begin q := getnode(gluenodesize);
  typ(q) := gluenode; gluelink(q) := fillglue;
  if leqno then
    begin link(q) := b; link(eqnobox) := q;
    b := hpack(eqnobox, dw − shift, false);
        { eqno will be left-justified }
    shift := 0.0;
    end
  else begin link(q) := eqnobox; link(b) := q;
    b := hpack(b, dw − shift, false)
        { eqno will be right-justified }
    end
  end
This code is used in section 444.
```

* * * * * * * * * * *

THE FORMAT OF PXL FILES

David Fuchs

A PXL file is a raster description of a single font at a particular resolution. These files are used by driver programs for dot matrix devices; TEX itself knows nothing about PXL files. Let's say a user creates a file called FOO.MF, which is the **METAFONT** language description of a new font, called FOO. In order for everyone to be able to run TEX jobs that use this font and get their output on our 200-dot-per-inch proof device, we must first run the **METAFONT** program on FOO.MF, and ask it to make both a TFM file and a PXL file for it. These files (called FOO.TFM and FOO.PXL) are then put in a public directory so that anyone using TEX may access them. Now, whenever a TEX job is run that refers to FOO (`\font A=FOO`), the TEX program reads in FOO.TFM to get all the width, height, depth, kerning, ligature, and other information it needs about any font. To get output on the proof device, the DVI file produced by TEX must now be processed by a device-driver program. This program reads the postamble of the DVI file to find out the names of all the fonts referred to in the job, and for each font, it opens the corresponding PXL file. In our example, the driver would find the file FOO.PXL, which it would then use along with the main body of the DVI file to produce the actual output. The DVI file tells where to put all the characters on each page, while the PXL file(s) tell which pixels to turn 'on' in order to make each character.

In fact, there is a little lie in the preceding paragraph. The actual name of the PXL file would be something like FOO8.1000PXL. This means that the PXL file represents the font FOO in an 8-point face for a 200-dot-per-inch device with a magnification of 1. (If you don't fully understand the term 'magnification' as it is used in the TEX world, the rest of this paragraph might not make a lot of sense. The end of this document contains more information on magnified fonts.) If we also had a 100-dot-per-inch device, we would also want to have the file FOO8.0500PXL (which we can get by asking **METAFONT** nicely). This PXL file could also be used by the higher resolution device's driver for any TEX job that asked for `\font B=FOO8 at 4pt`; or one that used `\font C=FOO8`, but then got TEXed with magnification 500; or one that used `\font C=FOO8`, but then got spooled with magnification 500. Note that we are assuming that the font FOO is like the CM family in that it does not scale proportionately in different point sizes—we are only talking about the 8-point face. If it turns out that 8-point FOO magnified by 1.5 is exactly the same as 12-point FOO, then we can also use FOO12.1000PXL in place of FOO8.1500PXL, and so forth. For fonts that scale proportionately like this, a point-size should not be included as part of the font name, and FOO.1000PXL is by convention the 10-point size of FOO for a 200-dot-per-inch machine.

Now for an explanation of where the bits go. A PXL file is considered to be a series of 32-bit words (on 36-bit machines, the four low-order bits of each word are always zero). In the discussion below, "left half word" means the highest-order 16 bits in a word, and "right half word" means the 16 next-highest-order bits in the word (which are exactly the lowest-order 16 bits on 32-bit machines).

Both the first and last word of a PXL file contain the PXL ID, which is currently equal to 1001 (decimal). The second-to-last word is a pointer to the first word of the Font Directory. (All pointers are relative to the first word in the file, which is word zero.)

The general layout of a PXL file looks like this:

PXL ID	[1 word long – First word of the PXL file]
RASTER INFO	[many words long – begins at second word]
FONT DIRECTORY	[512 words – 517th-to-last through 6th-to-last word]
CHECKSUM	[1 word – fifth-to-last word]
MAGNIFICATION	[1 word – fourth-to-last word]
DESIGNSIZE	[1 word – third-to-last word]
DIRECTORY POINTER	[1 word – second-to-last word]
PXL ID	[1 word – Last word of the PXL file]

The Font Directory is 512 words long, and contains the Directory Information for all the 128 possible characters. The first four words of the Font Directory have Directory Information about the character with ascii value of zero, the next four words are for the character with ascii value of one, etc. Any character not present in the font will have all four of its Directory Information words set to zero, so the Directory Information for the character with ascii value X will always be in words $4 * X$ through $4 * X + 3$ of the Font Directory. For example, if the second-to-last word in a PXL file contained the value 12000, then words 12324 through 12327 of the PXL file contain information about the ascii character "Q", since ascii "Q" = '121 octal = 81 decimal, and 4*81 = 324. The meanings of a character's four Directory words are described below.

The first word of a character's Directory Information has the character's Pixel Width in the left half-word, and its Pixel Height in the right half-word. These numbers have no connection with the 'height' and 'width' that TEX thinks the character has (from the TFM file); rather, they are the size of the smallest bounding-box that fits around the black pixels that form the character's raster representation, i.e. the number of pixels wide and high that the character is. For example, here is a letter "Q" from some PXL file:

```
00 ....*******....
01 ...*********...
02 ..****...****..
03 .***.......***.
04 ****.......****
05 ***.........***
06 ***.........***
07 ***:.*****..***
08 **********.****
09 .*****..******.
10 ..****...****..
11 ...*********...
12 ..X.*******..**
13 ........***.***
14 .........******
15 ..........*****
   0000000000111111
   0123456789012345
```

(The rows and columns are numbered, and the reference point of the character is marked with an 'X', but only the stars and dots are actually part of the character—stars represent black pixels, and dots represent white pixels.)

Note that the Pixel Width is just large enough to encompass the leftmost and rightmost black pixels in the character. Likewise, the Pixel Height is just large enough to encompass the topmost and bott most black pixels. So, this 'Q's Pixel Width i and its Pixel Height is 16, so word 12324 in the ample PXL file contains $15 * 2^{16} + 16$.

The second word of a character's Direc Information contains the offset of the character's erence point from its upper-left-hand corner of bounding box; the X-Offset in the left half-word Offset in the right half-word. These numbers be negative, and two's complement representatio used. Remember that the positive x direction me 'rightward' and positive y is 'downward' on the p The offsets are in units of pixels. In our 'Q' exam the X-Offset is 2 and the Y-Offset is 12, so w 12325 of the example PXL file contains $2 * 2^{16} +$

The third word of a character's Direct Information contains the number of the word this PXL file where the Raster Description for character begins. This number is relative to beginning of the PXL file, the first word of whic numbered zero. The layout of Raster Descript is explained below. The Raster Description consecutive characters need not be in order wi a PXL file—for instance the Raster Descrip for character 'Q' might be followed by the Ra Description of the character 'A'. Of course, a si character's Raster Description is always contai in consecutive words.

If a character is 'totally white' then the t word contains a zero. (The Pixel Width, P Height, X-Offset and Y-Offset of a 'totally wh character must be zero too, although the T Width may be non-zero. A non-zero TFM Wi would mean that the character is a fixed wi space. TEX's standard CM fonts do not contain such characters.) For the "Q" example, word 12 might contain any number from 0 thru 12000 – (since Q's Raster Description is 16 words long), say it is 600.

The fourth word contains the TFM Width of character. That is the width that TEX thinks character is (exactly as in the TFM file). The wi is expressed in FIXes, which are $1/(2^{20})$th of design size. The TFM Width does not take into count the magnification at which the PXL file prepared. Thus, if "Q" had a width of 7 points 12-point font, word 327 in the PXL file would tain $trunc((7/12) * 2^{20})$. See the TFM docume tion for more information on FIXes.

After the 512 words of Directory Informa come 3 words of font information:

First, the checksum, which should match checksum in any DVI file that refers to this (otherwise TEX prepared the DVI file under

wrong assumptions—it got the checksum from a TFM file that doesn't match this PXL file). However, if this word is zero, no validity check will be made. In general, this number will appear to contain 32 bits of nonsense.

Next is an integer representing 1000 times the magnification factor at which this font was produced. If the magnification factor is XXXX, the extension to the name of this PXL file should be XXXXPXL.

Next comes the design size (just as in the TFM file), in units of FIXes (2^{-20} unmagnified points; remember that there are 72.27 points in an inch). The design size should also be indicated by the last characters of the PXL's file name. The design size is not affected by the magnification. For instance, if the example font is CMR5 at 1.5 times regular size, then the PXL file would be called CMR5.1500PXL, word 12513 would contain 1500, and word 12514 would contain $5 * 2^{20}$.

The word after the design size should be the pointer to the Directory Information, and the word after that should be the final PXL ID word. Thus, if the number of words in the PXL file is p (i.e. word numbers zero through $p-1$) then the Directory Information Pointer should equal $p - 512 - 5$.

All of the PXL file from word 1 up to the Font Directory contains Raster-Descriptions. The Raster Description of a character is contained in consecutive words. Bits containing a '1' correspond to 'black' pixels. The leftmost pixel of the top row of a character's pixel representation corresponds to the most significant bit in the first word of its Raster Description. The next most significant bit in the first word corresponds to the next-to-leftmost pixel in its top row, and so on. If the character's Pixel Width is greater than 32, the 33rd bit in the top row corresponds to the most significant bit in the second word of its Raster Description. Each new raster row begins in a new word, so the final word for each row probably will not be "full" (unless the Pixel Width of the character is evenly divisible by 32). The most significant bits are the ones that are valid, and the unused low order bits will be zero. From this information, it can be seen that a character with Pixel Width W and Pixel Height H requires exactly $(ceiling(W/32)) * H$ words for its Raster Description.

In our "Q" example, words 600 through 615 would have the binary values shown here (high order bit on the left):

```
600 00001111111000000000000000000000
601 00011111111100000000000000000000
602 00111100011110000000000000000000
603 01110000000111000000000000000000
604 11110000000111100000000000000000
605 11100000000011100000000000000000
606 11100000000011100000000000000000
607 11100111110011100000000000000000
608 11111111110111100000000000000000
609 01111100111111000000000000000000
610 00111100011110000000000000000000
611 00011111111100000000000000000000
612 00001111111001110000000000000000
613 00000000111011100000000000000000
614 00000000011111100000000000000000
615 00000000001111100000000000000000
```

As an example of the case where the Pixel Width is greater than 32, consider a character with Pixel Width = 40 and Pixel Height = 30. The first word of its Raster Description contains the the leftmost 32 pixels of the top row in the character. The next word of the Raster Description contains the remaining 8 pixels of the first row of the character in its most significant 8 bits, with all remaining bits zero. The third word contains the left 32 pixels of the second row of the character, etc. So, each row takes 2 words, and there are 30 rows, so the Raster Description of this character requires 60 words.

Finally, some implementation notes and advice for DVI-to-device program writers: First, please note that PXL files supersede our older raster description (VNT) files. One notable difference is that VNT files claimed to have two representations for each character, one being the 90 degree rotation of the other. While a rotated copy of every character is useful in many circumstances, there is no reason that installations using only one character orientation should be burdened with so much wasted space. **METAFONT** outputs PXL files as described above, and there is a separate utility that can read a PXL file and write a rotated version into a new PXL file. Naming conventions to keep various rotations of the same font straight are currently under consideration.

Another item still under consideration is alternate packing schemes. You may have noticed that the current way that rasters are stored is fairly wasteful of space: Why should a new raster row begin in the next word rather than the next byte of the raster description? The answer is that this is for the sake of the poor people who are doing page-painting for their Versatec/Varian on their 32-bit mainframe computer. All the extra zeros help them write a faster paint program. An alternate, as yet unimplemented, PXL format would pack the rasters

tighter for the sake of those who have a minicomputer dedicated to doing the painting process. Such byte-packed PXL files will be identified by a PXL ID of 1002. A straightforward utility program can convert between word-packed and byte-packed PXL files.

For those of you still in a fog about character widths, here's more prose: The intent is that a DVI-to-device program should look like DVITYP, always keeping track of the current-position-on-the-page in RSU-coordinates. DVITYP looks at TFM files in order to get the character width info necessary to interpret a DVI file in this way. The DVI-to-device program shouldn't have to open a lot of TFM files in addition to the PXL files it will be needing, so the system has redundant width information for each character—a PXL file has all of the character widths exactly as they appear in the TFM file (in units of FIXes). Thus, the DVI-to-device program can completely interpret a DVI file by getting character widths from PXL files rather than TFM files. The purpose of the CHECKSUM is to ensure that the TFM files used by TEX when writing the DVI file are compatible with the PXL files the DVI-to-device program sees (so if someone changes a font and makes new TFMs but not new PXL files, you'll have some way of knowing other than seeing ragged right margins).

In the places where DVITYP would print a message indicating that "Character C in Font F should be placed at location $\langle H, V\rangle$ on the page" (where H and V are in RSUs), the DVI-to-device program should cause character C to be put on the paper at the point closest to $\langle H, V\rangle$ that the resolution of the device allows. The important point is that the DVI-to-device program should *not* attempt to keep track of the current-position-on-the-page in device-units, since this will lead to big roundoff problems that will show up as ragged right margins.

Note again that the character widths are different things than pixel-widths: The width of the character "A" in CMR10 is (say) 6.7123 points, independent of the representation of that character on the page. For a 100-dot-per-inch device, the raster representation of "A" might be 18 pixels wide, while for a 200-dot-per-inch device, the best representation might be 33 pixels wide.

Here is some more information to help clear up misunderstandings concerning magnification. (Much of this is taken from TEX's errata list, and is destined to be included in the next TEX manual.) One point to keep in mind is that the character widths in a PXL file are exactly as in the TFM file. Since the magnification factor is not taken into account, a DVI-to-device program should multip
widths by the product

(font design size) × (overall job magnificatio
× (font magnification) × (254000 RSU/i
× ($1/2^{20}$ point/FIX) × (1/72.27 inch/poi

to get the effective character width in RSUs course, this multiplication should only be done per character per job!)

It is sometimes valuable to be able to c the magnification factor at which document printed. For example, when preparing docu masters that will be scaled down by some fac a later step in the printing process, it is help be able to specify that they be printed blown the reciprocal factor. There are several new fea in TEX to allow for greater ease in the producti such magnified intermediate output.

TEX should be thought of as producing as put a "design document": a specification of the final result of the printing process should like. In the best of worlds, this "design docur would be constructed as a print file in a genera device-independent format. Printing a mag copy of this document for later reduction shou viewed as the task of the printer and its contr software, and not something that TEX should about. But real world constraints may force deviate from this model somewhat.

First, consider the plight of a TEX user who to print a document magnified by a factor of t a printer that only handles 8.5″ by 11″ paper. der to determine an appropriate \hsize and \v this user will have to divide the paper dimensio the planned magnification factor. Since comp are so good at dividing, TEX offers this use option of setting the "magnification" paramet 2000, warning TEX of the anticipated factor blow up, and then specifying \hsize and \v in units of "truein" instead of "in". When i ting a "true" distance, TEX divides by the scal tor that "magnification" implies, so as to canc effect of the anticipated scaling. Normal units to distances in the "design document", while "t units refer to distances in the magnified printe put.

Secondly, some existing print file format printer combinations have no current provisio magnified printing. This is not generally the for DVI files, but a Press file, for example, absolute distances internally in all positioning mands, and Press printers treat these distanc concrete instructions without any provision for ing. There is a program that takes a Pres and a scale factor as input and produces as ou

a new Press file in which all distances have been appropriately scaled. But it is inconvenient to be forced to use this scaling program on a regular basis. Instead, the Press output module of TEX chooses to scale up all distances by the "magnification" factor when writing the output Press file. Thus, the Press files that TEX writes are not representations of TEX's abstract "design document", but rather representations of the result of magnifying it by the factor (\parval12)/1000. On the other hand, the DVI files written by other versions of TEX contain normal units of distances, and the software that translates DVI files to instructions that drive various output devices will do the magnification by themselves, perhaps even using a magnification that was not specified in the TEX source program; if the user has not specified "true" dimensions, his or her DVI output file will represent the design document regardless of magnification.

Caveat: Due to the manner in which the current implementation of TEX writes Press files, it is not permissible to change the value of parameter 12 in the middle of a TEX run. If you want to produce magnified output, you should reset parameter 12 once very early in your document by using the \chpar12 control sequence, and from then on leave it alone. Another caveat below discusses the situation in more detail.

The magnification mechanism has been extended to include font specifications as well: in order to print a document that is photographically magnified, it is essential to use magnified fonts. A font is specified by the "\font" control sequence, which now has the syntax

\font ⟨fontcode⟩=⟨filename⟩ at ⟨dimen⟩.

The "at" clause is optional. If present, the dimension specified is taken as the desired size of the font, with the assumption that the font should be photographically expanded or shrunk as necessary to scale it to that size times the magnification factor specified by parameter 12. For example, the two fonts requested by the control sequences

```
\font a=CMR10 at 5pt
```

and

```
\font b=CMR5 at 5pt
```

will look somewhat different. Font a will be CMR10 photographically reduced by a factor of two, while font b will be CMR5 at its normal size (so it should be easier to read, assuming that it has been designed well).

The dimension in a font specification can use any units, either standard or "true". The interpretation of "true" here is identical to its interpretation in the specification of any other distance: asking for a font "at 5pt" requests that the font be 5 points in size in TEX's "design document", while asking for a font "at 5truept" requests that the font be 5 points in size after the scaling implied by the "magnification" factor.

If the "at ⟨dimen⟩" clause is omitted, TEX defaults the requested size to the design size of the font, interpreted as a design (non-"true") distance. Thus, the control sequence "\font a=CMR10" is equivalent to the sequence "\font a=CMR10 at 10pt", assuming that the designer of CMR10 has indeed told TEX that CMR10 is a 10-point font.

Caveat: This extension allows the TEX user to request any magnification of any font. In general, only certain standard magnifications of fonts will be available at most raster printers, while most high-resolution devices have scalable fonts. The user of TEX at any particular site must be careful to request only those fonts that the printer can handle.

Caveat: As mentioned above, you shouldn't change the value of parameter 12 in the middle of a run. TEX uses the value of parameter 12 in the following three ways:

(i) Whenever the scanner sees a "true" distance, it divides by the current magnification.

(ii) At the end of every page, TEX's output module may scale all distances by the current magnification while converting this page to format for an output device (this doesn't happen with DVI output).

(iii) At the very end of the TEX run, the output module uses the current magnification to scale the requested sizes of all fonts. Given this state of affairs, it is best not to change parameter 12 once any "true" distance has been scanned and once any page has been output.

Some device-drivers give the user the option of overriding the magnification at which the TEX job was run. Note that running a given TEX job with \magnify{2000} is not the same as running it with \magnify{1000} and then asking the driver to override the magnification to 2000. The difference will be in the dimensions of the pages; in the first case, the output will be, say, 8.5″ by 11″ pages filled with double size fonts, while in the second case the output will be 17″ by 22″ pages with double size fonts.

This is a new document, so it is bound to contain outright errors along with the portions that are merely misleading. I would certainly be glad to hear of any errors, but I am also interested in which parts of the explanation need clearing up, either by adding to or changing the text.