Inter-Office Memorandum

To	Whom it May Concern	Date	April 17, 1980
From	Lyle Ramshaw	Location	Palo Alto
Subject	Kerned Strike Fonts (Revised version)	Organization	CSL

XEROX

Filed on: [Maxc1]<Fonts>KenrnedStrikes.bravo
[Maxc1]<Fonts>KenrnedStrikes.press

It's time for a new font format! The Strike format for font files was devised to permit the graceful use of BITBLT for writing characters onto the Alto screen; the authoritative description of this original Strike format can be found in the memo *Font Representations and Formats* by Bob Sproull, filed on
[Maxc1]<PrintingDocs>FontFormats.Press.

The original Strike format, however, has a serious difficulty: it does not distinguish between the spacing width of a character, and the width of that character's bounding box. Let us say that a character *kerns to the left* if its raster includes bits in columns to the left of the character's origin. A character is said to *kern to the right* if its raster includes bits in columns to the right of the end of its width vector (the origin of the next character). The original Strike format is incapable of handling characters that kern in either direction. The newer PARC fonts contain lots of zero-width accent characters that are intended to overprint the following character. These accent characters overhang the ends of their width vectors by quite a bit, that is, they kern heavily to the right; hence, conventional Strike format can't handle them.

There are basically two ways to adjust Strike format so that it can handle characters that kern. Consider left kerning, for example. One possibility might be called the *padding approach*. The idea here is to pad every character raster in the strike body with enough blank columns so that all of the overhanging bits fit into those padding columns.

A padding approach to the left kerning problem would work like this. We would look through the entire font, and determine the size of the largest left kern of any character; say that it is four columns. Thus, at least one character's bounding box includes four columns to the left of the origin, while no character overhangs to the left of the origin by more than four columns. When putting character rasters into the Strike array, then, we include four columns for every character to the left of that character's origin. We also find a field somewhere early in the file to put the number 4'. A user of the resulting file would paint a character on the display by BitBlt'ing the entire character raster, padding columns and all. The first column of the destination of the BitBlt would be given by the formula (<desired origin> 4).

A symmetric padding strategy could be used to handle right kerning. But any padding strategy has problems. The padding columns appear in every character's raster; they take up space in the file, and in memory, and they slow down the process of painting the character, since they are included in the BitBlt's.

The other approach to the kerning problem might be called the *auxiliary table approach*. In this scheme, we store the character rasters into the strike body by their bounding box dimensions, and then we use an auxiliary table of small integers to tell the location of the origin relative to the bounding box, and the length of the width vector. The auxiliary table approach seems to be a better choice than a padding approach. The purpose of this memo is propose a new Strike file format that uses the auxiliary table approach to handle both left and right kerning. If you object to the following plan, or have suggestions to improve it, please get in touch with me as soon as possible.

History:

The definition of Strike file format in Sproull's memo *Font Representations and Formats* suggests the use of a padding scheme to handle left kerning. As I read it, this memo states that a negative integer in the xoffset word implies that every character raster in the strike is padded by (xoffset) columns on the left. To the best of my knowledge, no one ever wrote any programs that would correctly produce or consume such a left-padded strike font. In addition, left kerning doesn't happen to be nearly as common in PARC fonts as right kerning anyway. We could, of course, legislate against left kerning completely, and start devoting the xoffset word to a right kerning scheme. In fact, this is what the current version of PrePress does when the Kerned flag is set in MakeStrike.

But thinking things over, it seems to be a better plan to take advantage of the fact that no padding scheme has been widely used, and drop all such schemes in favor of a format based on the auxiliary table approach. From now on, if you want to handle kerning, you should use the new KernedStrike format, and its auxiliary tables.

Some terminology:

The following section is a substitute for section 7.2 of the *Font Representations and Formats* memo. Before we start on that, though, let me review a little terminology from that memo, in case the reader is rusty. The origin of a charcter is a reference point located at the corner where four pixels touch. The width vector of a character is a two-dimensional vector that specifies the desired displacement from the origin of the current character to the origin of the next character in a string. The components of the width vector are written Wx and Wy. Wx is always nonnegative; all of the font formats intended for the Alto screen assume in addition that Wx is an integral number of pixels, and that all characters have a Wy of zero. The bounding box of the black pixels in the character raster is a rectangle with width BBdx and height BBdy. These numbers are always nonnegative. They are both zero if the character has no black bits in its raster (such characters are called *empty characers*), and are both positive in all other cases. If we use the origin to define a Cartesian coordinate system on the plane, the lower left corner of the bounding box is located at the point <BBox, BBoy>. Thus, the left edge of the bounding box is located BBox units to the right of the origin (left if BBox is negative), while the bottom of the bounding box is located BBoy units above the origin (below if BBoy is negative). Empty charcters have BBox and BBoy equal to zero, by convention. Finally, if all of the characters in the font are superimposed with their origins coincident, the dimensions of the resulting bounding box are called FBBdx, FBBdy, FBBox, and FBBoy respectively.

7.2 (New version) STRIKE format:

There are four kinds of files in the Strike class: a PlainStrike file (conventional extension .Strike), a KernedStrike file (conventional extension .KS), a PlainStrikeIndex file (conventional extension .StrikeX), and a KernedStrikeIndex file (conventional extension .KSX). In a PlainStrike file, the individual rasters of the characters are assembled in ascending order of character code into one large raster, called the *strike*. The baselines of the characters are aligned, and the origin of each character is made coincident with the end of the width vector of the preceding character. The PlainStrike file also contains a table indexed by character code that points to the leftmost column of the raster for each character in the strike. Warning: since the rasters in a PlainStrike file are positioned by their origins and width vectors, it must be the case that all of the black bits of the character lie between these two bounds. No character may include bits to the left of its origin (left-kerning) or the right of the end of its width vector (right-kerning).

A KernedStrike file handles kerned characters, and does so in the following way: the individual rasters are put into the strike by their bounding box widths. Just think about taking the bounding boxes of all of the characters, lining up their baselines, and packing them tightly into one long raster array; in this format, there are no blank columns between characters in the strike. A KernedStrike file has three additional tables, indexed by character code. One gives the position in the strike of the first column of the character's raster, which is also the leftmost column of its bounding box. The other two tables consist of small integers that specify the left-to-right location of the origin with respect to the bounding box, and the length of the width vector.

A StrikeIndex is essentially a table that maps character codes into <strike, code> pairs, together with the associated strikes. An index can be used to achieve sharing if several character codes map to the same <strike, code> pair, and hence refer to the same raster. Or it can help to save space, by grouping the rasters into several strikes to save top and bottom scanlines. [By the way, to the best of my knowledge, no one has ever used StrikeIndex format.]

PlainStrike and KernedStrike files have the following format:

```
structure PlainStrike:
  [
  @StrikeHeader             // header common to all Strike files
  @StrikeBody               // the actual strike
  ]

structure KernedStrike:
  [
  @StrikeHeader             // header common to all Strike files
  @BoundingBoxBlock         // dimensions of the font bounding box
  @StrikeBody               // the actual strike
  @WidthBody                // table of width data
  ]

structure StrikeHeader:
  [
  format word =
              [
              oneBit bit    // always =1, meaning   new style''
              index bit     // =1 means StrikeIndex,  =0 otherwise
              fixed bit     // =1 if all characters have same value of Wx, else =0
              kerned bit    // =1 if KernedStrike, =0 if PlainStrike
              blank bit 12
              ]
  min word                  // mimimum character code
  max word                  // maximum character code
  maxwidth word             // maximum spacing width of any character = max{Wx}
  ]

structure BoundingBoxBlock:
  [
  FBBox                     //  as defined above
  FBBoy                     //  as defined above
  FBBdx                    //  as defined above
  FBBdy                    //  as defined above
  ]

structure StrikeBody:
  [
  length word                         // total number of words in the StrikeBody
  ascent word                         // number of scan-lines above the baseline, which is
                                      //   normally max{BBdy+BBoy} over the chars in this strike
  descent word                        // number of scan-lines below the baseline, which is
                                      //   normally max{( BBoy)} over the chars in this strike
  xoffset word                        // always =0  [used to be used for padding schemes]
  raster word                         // number of words per scan-line in the strike
  bitmap word raster*height           // the bit map, where height=ascent+descent=FBBdy
  xinsegment ^ min, max+2 word        // pointers into the strike, indexed by code
  ]

structure WidthBody:
  [
  widthtable ^ min, max+1 @WidthEntry              // spacing information, indexed by code
  ]

structure WidthEntry:
  [
  spacing word =           // the entire spacing word will be =( 1)  (both bytes =377b)
                           // to flag a non-existent character, else the bytes are:
             [
             offset byte              // =BBox FBBox
             width byte               // =Wx
             ]
  ]
```

The bitmap’’ entry is one large bit map; there are height=ascent+descent scanlines in the bitmap, each of which is raster words long. Unless something funny is going on, ascent will be simply FBBdy+FBBoy, while descent will be simply (FBBoy).

The font includes characters for some of the ASCII codes from min through max inclusive. The bitmap includes a dummy character associated with the charcter code (max+1), which can be displayed for any non-existent character.

A PlainStrike works as follows: Given a character code c, in the range [min, max], we first compute:

```
            xLeft _ xinsegment ^ c;
            xRight _ xinsegment ^ (c+1);.
```

If xLeft=xRight, then c is a non-existent character in the current font, and should be replaced by the raster with code (max+1). Otherwise, the columns of the bitmap from xLeft through (xRight 1) inclusive contain the raster for character c, and the width of charcter c is Wx=(xRight xLeft).

A KernedStrike works a little differently. We first compute xLeft and xRight as above, and also compute

```
            Spacing _ WidthTable ^ c;.
```

If Spacing=(1), then c is a non-existent character in the current font, and should be replaced by the dummy character at (max+1). Otherwise, the columns of the bitmap from xLeft through (xRight 1) constitute the bounding box of the raster of character c. In this case, we decompose the Spacing value into its two bytes:

```
                    Offset _ Spacing<<WidthEntry.offset;
                    Width _ Spacing<<WidthEntry.width;.
```

Now assume that we want to paint the character c starting at destination column xDest. The source of the BitBlt is columns xLeft through (xRight 1) of the bitmap inclusive. We can compute the proper destination from xDest, Offset (which =BBox FBBox), and the FBBox word of the BoundingBoxBlock: the first column of the destination is (xDest+Offset+FBBox)=(xDest+BBox). Note: the offset portion of the WidthEntry was chosen to be (BBox FBBox) rather than BBox itself, since the former quantity is always nonnegative, while the latter quantity can have either sign; and signed 8-bit numbers are a pain in the ass. Finally, we replace xDest by (xDest+Width) to prepare for the painting of the following character.

Two fine points concerning KernedStrikes: A non-existent character is flagged by a (1) value in the WidthTable. Since a non-existent character doesn’t have a bounding box, the xLeft and xRight entries for such a character will be equal. But there can also be perfectly legal characters for which xLeft=xRight; in particular, all of the empty characters will have this property: figure space, em quad, word space, etc. If you are painting an empty character, there is no need to actually perform the BitBlt, since the rectangle being Blt’ed would have zero width. All that must be done is to replace xDest by (xDest+Width) to make the space happen. Secondly, some extra efficiency can be gained when using a KernedStrike font by keeping track of the quantity (xDest+FBBox) in the character-painting loop, instead of xDest itself. This moves one addition out of the inner loop.

Finally, it is time to say a few words about StrikeIndex format: a StrikeIndex is simply an index at the front of some StrikeBodies.

```
structure PlainStrikeIndex:
  [
  @StrikeHeader                         // common header
  maxascent word                        // maximum ascent of all the strikes
                                                    //  [probably =FBBdy+FBBoy]
  maxdescent word                       // maximum descent of all the strikes
                                                    //  [probably =( FBBoy)]
  nStrikeBodies word                    // the number of strike bodies
  map ^ min,max+1 @mapEntry             // table of <strike, code> pairs, dummy at max+1
  bodies ^ 1, nStrikeBodies @StrikeBody // the strike bodies themselves
  ]

structure KernedStrikeIndex:
  [
  @StrikeHeader                         // common header
  @BoundingBoxBlock                     // bounding box data for the entire font
  maxascent word                        // maximum ascent of all the strikes
                                                    //  [probably =FBBdy+FBBoy]
  maxdescent word                       // maximum descent of all the strikes
                                                    //  [probably =( FBBoy)]
  nStrikeBodies word                    // the number of strike bodies
  map ^ min,max+1 @mapEntry             // table of <strike, code> pairs, dummy at max+1
  bodies ^ 1, nStrikeBodies @StrikeBody // the strike bodies themselves
  @WidthBody                            // table of width data
  ]

structure mapEntry:
```

```
[
missing bit 1               //  =1 if character is non-existent, else =0
strike bit 7                //which strike  in range [0:127]
code byte                   // which code
]
```

In a StrikeIndex font, all of the StrikeBodies have implicit min values of zero; the max value is unimportant, as the map will never generate a reference outside the range. The individual StrikeBodies do not have separate pictures for illegal characters; instead, the (max+1) entry in the global map defines a single dummy picture. Non-existent characters in the range [min, max] are indicated in the global map by a mapEntry that specifies a strike number larger than 127=177b, that is, by the sign bit of the map entry being 1. In KernedStrikeIndex fonts, non-existent characters will also be indicated by having a WidthEntry of (1).

In StrikeIndex fonts, the ascent and descent words in each StrikeBody give the dimensions of that particular StrikeBody; thus, they probably are the y dimensions of the bounding box of those characters that are included in that StrikeBody, rather than of the entire font.

BitBlt modes:

There are evidently lots of programs in the world that paint charcters on the screen by calling BitBlt in Replace mode, in which the new bits simply smash whatever used to be at the destination. If you want to handle characters that kern, you simply can’t do this! The bounding boxes of successive characers may actually overlap, and hence a Replace Blt might overwrite valuable bits. If you want kerning specified in a KernedStrike font to work, you must use one of the other BitBlt modes: Paint, Erase, or Invert.

Conversion issues:

I propose the following plan. In the near future, some combination of Doug Wyatt and myself will modify PrePress 1.12 to produce a PrePress 1.13. This new PrePress will include a MakeKS command, that converts a font from .AC format to .KS format, and a ReadKS command, that converts in the opposite direction. The MakeStrike command will remain, to convert from .AC format to the original Strike format. However, the Kerned flag will be replaced by a new flag called Clipped’. And the code that implements the MakeStrike command will be altered to treat overhanging characters as follows. If any character of the .AC font kerns to the left, the MakeStrike command will fail, as it does currently. Characters will be allowed to kern to the right. If the Clipped flag is true, the portion of the bounding box that overhangs the end of the width vector will be ignored. If the Clipped flag is false, the width of any right-kerned characters will be artificially increased just enough to eliminate the overhang. Thus, calling MakeStrike on a font with right kerning will have one of two effects: if Clipped is true, the resulting Strike font will space correctly but the overhanging portions of characters won’t print. If Clipped is false, the overhanging portions will print, but the characters will be spaced too far apart. Neither of these two possibilities is very pleasant, of course; but if you want kerned characters to work, you should use .KS format.

Note that the xoffset word of all StrikeBodies produced under the new rules will always be zero, no matter what file format is involved. Non-zero values of xoffset were originally reserved for handling kerning by padding schemes; since padding is now handled by the auxiliary table approach, the xoffset word is not needed.

PrePress 1.13 will also include new code to handle the exotic face byte values that are being introduced as part of the upcoming font cataclysm.

Once PrePress 1.13 is up and running, we will take all of the .AL fonts on [Ivy]<AltoFonts> and produce a .KS version for each of them, sticking all of those fonts on [Ivy]<AltoFonts> as well. That should give us plenty of KernedStrike fonts to play with, as people start to produce software that uses the new format.