Inter-Office Memorandum

To	Cyprians	Date	January 29, 1976
From	Tesler, Merry, Sproull, Lampson, Ingalls	Location	Palo Alto
Subject	Font Format	Organization	PARC

XEROX

This memo explains how we arrived at a font format that is twice as compact as .AL format and is easy to use with BitBlt.

Background

Everybody who ever tried to make the Alto do graphics wondered why there was no BitBlt, so now there is one, and it is wonderful for moving rectangles around the screen, making gray boxes, and painting. But some of us have been less than delighted with its use for "scan conversion" of characters. This has been a surprise, because we always thought of BitBlt as a way of making character display easier.

The Tableau Scheme

Our first idea long ago was to have something like the 8 character wide by 16 character high Tableau of the Smalltalk font editor, a bitmap tiled with the characters of a font. With ascii 0 placed in the upper left corner, one can easily find the upper-left-origin source rectangle of any character by:

```
source x = fontmaxwidth * (ascii & 7)               <==
source y = fontheight * (ascii rshift 3)            <==
source bca (base core address) = tableau
source bmw (bit map width) = fontmaxwidth lshift 3
```

The destination rectangle to display the character at (x,baseline) is:

```
dest width = widthtable ! ascii                     <==
dest height = fontheight
dest x = left                                       <==
dest y = baseline + fontascent
dest bca = displayBitMapAddress
dest bmw = displayBitMapWidthWords
```

Only the four values marked by <== need to be computed for every character in a run of the same font and baseline. Note that it is unnecessary to pre-clear the space that the characters will occupy, as long as spaces and leading are explicitly cleared.

When it came around to designing the font format, we noted some disadvantages of this simple scheme.

The tableau has wasted white space above and below short characters and next to narrow characters.

Unnecessary time is taken to "Blt" the white space (unless it is used as a feature to pre-clear).

Kerned characters like *f* require special handling.

The Offset/Align Scheme

To remedy these difficulties, we adopted the following cousin of AL format. Each character has its own bit map in which white space on all sides is suppressed, and three extra words for each character help to determine the display parameters as follows:

```
source x = 0
source y = 0
source bca (base core address) = glyphtable+glyphtable!ascii      <==
source bmw (bit map width) = bca>>bwcount                         <==

dest width = widthtable ! ascii                                   <==
dest height = bca>>scount                                         <==
dest x = left      + bca>>balign + bca>>fontboffset               <==
dest y = (baseline-fontsoffset) - bca>>salign - height            <==
dest bca = displayBitMapAddress
dest bmw = displayBitMapWidthWords
```

Six values instead of four are computed for every character, and they are a little harder to compute, but it seems a small price to pay for a more compact font, faster scan conversion after pre-clearing, and uniform specification of kerned and unkerned characters.

Disappointment

As it has turned out, the code to implement this scheme is messy and time-consuming. A couple of us started getting that depressed feeling last acquired after disillusionment with the "two-array" tree representation last month. The way out last time was to do some calculations of total space requirements, which exploded the myth fostered by a worst case analysis done last Spring, and to think of some ways around the algorithmic shortcomings. Let's do that again.

If the maximum and average widths and heights in a 96-character font are Wmax, Wavg, Hmax, and Havg, and wds(x)=(x+15)/16, then the space in words taken by each format is theoretically:

AL:	96*Havg*wds(Wavg) + 448
Offset/align:	96*Havg*wds(Wavg) + 288 + widthtable
Tableau:	6*Hmax*Wmax + widthtable

For Helvetica-10, Wavg=Havg=7, Wmax=Hmax=12, wds(Wavg)=1, the width table can be packed into 24 words, and the theoretical space requirements are:

AL: 96*7*1+448	=	1120 (actual figure is 1147)
Offset/align: 96*7*1+288+24	=	984 (actual figure is 1029)
Tableau: 6*12*12+24	=	8 888

This represents a saving of 13%. It wins because the glyph/align format wastes more than half of every 16-bit word for the average 7-wide character. For fonts closer to 16 dots wide, the saving disappears; thus, Helvetica-14 with Wavg=Havg=9, Wmax=Hmax=16, would require:

AL: 96*9*1+448	=	1312
Offset/align: 96*9*1+288+48	=	1200
Tableau: 6*16*16 + 48	=	1584

which is 32% worse. This kind of reasoning originally led us away from the tableau scheme. Furthermore, trouble could arise in the tableau scheme with a font of special symbols of radically differing sizes, where the tile size would be determined by the largest character.

The Strike Scheme

A slight variation of the tableau scheme has recently arisen which remedies some of these problems. The tableau is made 96 characters wide by 1 character high, i.e., a line of text containing the whole font (some call it the "strike"). Now the space taken by each character can be proportional to its width. The space requirements are:

Strike: 6*Hmax*Wavg words + (96 for x table)

where the x table doubles as a way to locate the source rectangle and to calculate its width (the difference of the following x and the current x). Now Helvetica-10 takes:

AL: 96*7*1+448	=	1120
Offset/align: 96*7*1+288+24	=	984
Tableau: 6*12*12+24	=	888
Strike: 6*12*7+96	=	600 (actual figure is 622)

and Helvetica-14 requires:

AL: 96*9*1+448	=	1312
Offset/align: 96*9*1+288+48	=	1200
Tableau: 6*16*16+48	=	1584
Strike: 6*16*9+96	=	960

This is obviously advantageous for typical fonts, but there are some disdvantages for odd fonts. In a mixed-size symbol font a lot of space is wasted, and very large fonts are hard to window. To handle kerned fonts there must be as much white space to the left of every character as is required for the worst character, and a font-wide parameter (le 0) tells how much that is (this also works for the tableau). In its favor, the algorithms are utterly simple:

```
source x = xtable!ascii                              <==
source y = 0
source bca (base core address) = strike
source bmw (bit map width) = strikewidth

dest width = xtable!(ascii+1)-xtable!ascii           <==
dest height = fontheight
dest x = left        + fontkern                      <==
dest y = baseline + fontascent
dest bca = displayBitMapAddress
dest bmw = displayBitMapWidthWords
```

with only three things to compute for each character. With a planned change to BitBlt that speeds up the or'ing of zero, and the microcoding of the BitBlt setup routine, this scheme can be made nearly as fast as the standard Alto Convert.

Variations

Refinements to this scheme are necessary to window large fonts and to compress mixed-size symbol fonts. Unfortunately, they make the algorithms more complicated.

Vertical Strike Scheme

To make the font windowable, a vertical strike can be used. However, a width table and a displacement is then necessary in addition to a ytable, and the dest height and y must be set for every character.

Segmented Strike Schemes

The font can be segmented. Each segment has its own strike, so the font can be windowed on a segment basis. One method of identifying the segment is by high order bits in the xtable entry. This necessitates extraction of the segment number from the xtable entry, but only adds a minor adjustment to the width computation, and the bmw and bca must be computed for each character.

Pinching Schemes

To handle mized size symbol fonts, each segment can have its own top and bottom "pinch", which allows suppression of white space above and below all the characters in a segment. This helps the A-Z in standard fonts, and helps carefully organized symbol fonts. The space requirements for Helvetica don't improve much, but strange symbol fonts could: a font with 96 characters of sizes 1x96, 2x95, ..., 96x1 in 32 segments of 3 characters each would require only a few thousand words as opposed to almost 28,000 words without pinching. Fewer bits need to be Blt'ed. However, it is now necessary to recompute dest y and height for every character.

It is tempting to pinch out the vertical white stripe that is normally on the left or right of each character. However, this saves only a few per cent space and adds even more computation.

Permuting Schemes

The next complication is to permute the font so that characters with similar pinches are in the same segment. This requires separating the xtable into a table to find x and one to find width, although the latter usually can be packed into 24 words. Now even Helvetica gains, because the space requirement is:

Permuted pinch: 6*Havg*Wavg + 96 + (24, 48, or 96 width table)

which comes out for Helvetica-10:

AL: 96*7*1+448	=	1120
Offset/align: 96*7*1+288+24	=	984
Tableau: 6*12*12+24	=	888
Strike: 6*12*7+96	=	600
Permuted pinch: 6*7*7+96+24	=	414

and for Helvetica-14 with an average height of 9:

AL: 96*9*1+448	=	1312
Offset/align: 96*9*1+288+48	=	1200
Tableau: 6*16*16 + 48	=	1584
Strike: 6*16*9+96	=	960
Permuted pinch: 6*9*9+96+24	=	606

The width calculation is not very much harder than in the unpinched strike method, but the dest y and height must be adjusted frequently. This is what you must do for the gain of 35% in space:

```
let seg = (xtable!ascii<<seg)

source x = (xtable!ascii<<x) + segx!seg               <==
source y = 0
source bca (base core address) = segbca!seg           <=
source bmw (bit map width) = segbmw!seg               <=
dest width = widthtable!ascii [unpacked]              <==
dest height = segheight!seg                           <=
dest x = left       + fontkern                        <==
dest y = (baseline+fontascent) - topPinch!seg         <=
dest bca = displayBitMapAddress
dest bmw = displayBitMapWidthWords
```

Those things marked <= need be recomputed only when the segment changes, but if extensive packing has been done, this happens on almost every character, so it may as well be done every time. If FillBits is called directly, multiplication can be avoided by computing a short table of (topPinch!seg * dest bmw) when the font is read. Then the setup takes only 15 or 20 emulated instructions more than in the simplest strike scheme. The segmented but unpinched scheme is somewhere in between.

Variable Tableau

One could make segments be equal in total width, yielding a tableau with a varying number of characters per line. The only advantage is that the bmw stays the same for every character; bca could stay the same but then y would have to vary.

File Format

PREPRESS has been modified to generate single-segment unpermuted strikes. The format chosen allows for future addition of segmenting, permuting, and pinching. At this time, Cypress and Smalltalk implementors expect to use the simple scheme for all fonts except very large ones (>30 points) and very odd ones. Cypress will use a separate algorithm much slower than the standard to display characters in the harder formats.

```
structure MMGLYPHS:
          [
          format      word =                 // all zeroes for simple strike
                      [
                      strike bit             //distinguish from .al format
                      pinched bit            // some segments are pinched
                      permuted bit           // the width table is explicit
                      fixedpitch bit         // all characters are maxwidth
                      blank bit 12
                      ]
          maxwidth word                      // width of widest character
          ascent      word                   // in bits
          descent     word                   // in bits
          xoffset     word                   // in bits (negative for kerned font, 0 normally)
          max         word
          nsegs       word                   // =1 if unsegmented
          @SEGMENTPOINTER ↑ 0,nsegs-1
          if permuted & not fixedpitch then @CHARWIDTH ↑ min,max+1
          @CHARPOINTER ↑ min,max+2           // index ↑ ascii into the strike
                                             // dummy at max+1
                                             // max+2 is for unpermuted width calculation
          @SEGMENT ↑ 0,nsegs-1               // the strike, possibly segmented
          ]

structure SEGMENTPOINTER:
          [
          segwidth    word                   // bits (total of all chars; last word padded)
                                             //(segwidth+15)/16 = raster for bitblt
          pinchTop    word                   // bits down from ascent top
                                             //0 for the simple case
          pinchBottom word                   // bits up from descent bottom
                                             //0 for the simple case
          ]

structure CHARWIDTH:
          [
          width       bit(maxwidth ls 16? 4, maxwidth ls 256? 8, 16)
          ]

segheight = (ascent + descent) - (pinchTop + pinchBottom)

structure CHARPOINTER:
          [
          segment bit 5                      // numbered from zero, always 0 if unsegmented
          xinsegment bit 11                  // source X for bitblt
                                             //all 16 bits if unsegmented font (nsegs = 1)
          ]

structure SEGMENT:
          [
        bits word ((segwidth+15)/16) * segheight
        ]

Rules:  SegmentX(segment) = Sum(s=0 to segment-1)(segwidth(s))
        VirtualX(char) = xInSegment(char)+SegmentX(Segment(char))
        If unpermuted then Width(char) = VirtualX(char+1)-VirtualX(char)
        DestX = DesiredDestx + xoffset

Note:   if nsegs eq 1, then both pinches will be zero
        Iff Width(char) eq 0, char is illegal, try max+1 where
                a blob ought to be stashed
```