XEROX
PALO ALTO RESEARCH CENTER
Computer Sciences Laboratory
March 5, 1977

For Xerox Internal Use Only

To: File

From: Bob Sproull

Subject: Font Representations and Formats

Filed on: <GR-DOCS>FontFormats.Press

This report presents the various standard and device-dependent font formats in use at PARC.

1. Introduction

A font is a collection of character descriptions, indexed by a character code. These descriptions represent, in one fashion or another, the appearance of the character. The ultimate purpose of maintaining a font is for use when generating a raster-scanned image of a document. This image may be created on a display and used for interactive purposes, or it may be generated by a printing service as part of a "hard copy" function. In both cases, for purposes of space and device independence, the document itself does not normally contain the character representations, but only codes used to identify the characters that comprise the document.

It is important to distinguish *font representations* from *font formats.*

We use two generically different *representations* for character shapes. The first, loosely termed "splines" or "spline fonts," represents the outline of the each character shape with a series of parametric cubic spline curves (see Figure 1). This representation is handy because it is independent of the particular output device and its resolution: the outlines describe the desired appearance of the character. The second representation we use is a *raster* (sometimes loosely termed a "bit map"), as shown in Figure 2. This representation records, in some way, a two-dimensional (binary) occupancy map: it tells where the character lies on a two-dimensional grid. This representation is handy for actually building raster images of documents: the occupancy map is combined with color information, often at very high speed, to generate a larger raster image of the document. The raster character description is in effect *merged* into the page raster at the proper position.

When characters are recorded in *font files*, we choose a particular *format* for the file; quite a number of different formats have emerged. This is because there are many ways to encode digitally the information in either an outline or raster representation of a character. The details of the encoding are often of vital concern when making a particular piece of hardware or software generate page rasters rapidly.

Fortunately, we can write conversion programs that are able to generate the various specialized formats from *standard formats.* When an artist (or a needy user) devotes a large amount of effort to designing and debugging a font, it should be recorded and disseminated in one of the standard formats. Clients can then easily convert to one of the subsidiary formats, or to their own private format.

Widths

An important adjunct to the font descriptions themselves is the "widths' file," which summarizes the dimensions of all characters in the font data base. This summary must be available to a text editor when it formats a document for hard copy: the widths are used to determine how many characters will fit on a line and to perform justification calculations. Because the information in this file can be independent of any particular output device, the hard-copy file produced by the editor can be printed on any of a number of printing devices.

The widths summary is, in effect, extracted from information recorded in the standard formats of the relevant fonts.

Software

The PARC font descriptions are supported by a reasonably full set of software:

> FRED: Interactive program for building outline font representations. Documentation is on <AltoDocs>Fred.Press. The program is on <ALTO>Fred.Dm.
>
> PREPRESS: Interactive program for building standard raster font representations. The program also contains numerous options for converting from standard to subsidiary formats. Documentation is on <AltoDocs>PrePress.Press. The program is on <ALTO>PrePress.Run.
>
> COMPRESS: A program that converts .CU format to EARS (.EP and .EL) formats. The program is on <EARS>Compress.Run.

The reader is invited to consult PrePress documentation for miscellaneous lore relating to fonts and for "standard operating procedures" for maintaining font files.

People

This document is simply a convenient summary of formats and techniques developed by a large number of individuals. The people behind the formats include Patrick Baudelaire, Peter Deutsch, Diana Merry, Ron Rider, Bob Sproull, Larry Tesler, and Chuck Thacker.

2. Terminology

The terminology that has developed around fonts is hopelessly inconsistent. This section is intended to serve as a glossary for the descriptions in the remainder of this document. Be forewarned that terminology used elsewhere may not match.

2.1 Characters

Family is the term given to a particular design of characters. Examples of families are "Times Roman," or "Helvetica."

Point size of a character refers to size measurements used in the printing industry. If text is *n* points high, this means that closely-spaced lines of text will fall *n*/72 inches apart on the page. Note that the point size does not relate in any consistent way to the geometry of characters, e.g., to the height of an upper case A.

Face denotes a number of attributes of a particular font: *italic, bold, light, condensed, expanded* are all attributes of the font. Sometimes this is called a "style." Sometimes the face is defined with a three-letter code: the first letter is L for light, M for medium, or B for bold; the second is R for regular or I for italic; the third is C for condensed, R for regular or E for expanded.

Rotation refers to the orientation of the character. If a string of characters is intended to be horizontal, it has rotation zero; if a string runs vertically upward, it has a rotation of 90 degrees.

Font, as we use the term, refers to a collection of characters of the same family, the same size, the same rotation, and the same face attributes.

Character code refers to a number (usually only 8 bits) that identifies a character. All our fonts use standard ASCII conventions, when the conventions are meaningful. For special-character fonts (e.g., mathematics, logic design), another mapping must generally be devised.

Origin of a character (sometimes called "the (0,0) point") is conceptually a reference mark that is used to describe a character's location on a page or display. Thus a directive to "display an A at x=103, y=204" is interpreted to mean "place an instance of the symbol A on the display so that the character origin coincides with the coordinate x=103, y=204." Figures 1 and 2 show the origin of a sample character.

Width of a character is a *two-dimensional vector* that represents the incremental translation that should take place to determine the placement of the origin of the next character to be displayed in a (conventionally aligned) string of characters. In the example of Figure 3, if we assume the *x* direction points to the right and the *y* direction up, we see that the width vector has a zero *y* component.

> In all our font representations, we associate the width vector with each character code. If this width vector is used for character positioning, the spacing between the origin of a A (say) and the origin of the next character is independent of that next character. This is not always desirable: because of the different shapes of characters, spacing between differing pairs may want to be adjusted slightly to make the text line appear more pleasing.

Bounding box is the term for a rectangle that just barely surrounds the character (see Figure 3). It is characterized by its *width* and *height*, and by a two-dimensional vector that specifies where the lower-left corner of the bounding box is with respect to the origin of the character inside. These four numbers are named (in this document) BBdx, BBdy, BBox, and BBoy.

The *font bounding box* is a bounding box that applies to *all* characters in the font. That is, if all the characters in the font were placed with their origins coincident, the smallest rectangle that encloses every part is the font bounding box. The four parameters of the font bounding box are named (in this document) FBBdx, FBBdy, FBBox, and FBBoy

The *coordinate system* assumed for this document is that *x* points to the right on a (portrait-oriented) page, and *y* points up. A *mica* is a unit of measure, equal to 10 microns or 1/2540 inch. Both of these conventions are identical to those used by Press.

Scanning mode refers to the way a *raster* is laid upon a character description. This in effect defines a coordinate system in which one direction is measured in *scan-lines* and the other direction is measured in *bits* (along a scan line). To describe the modes, we use a single number that relates the scanning regime to the conventional (x,y) coordinate system: the mode is bit-direction-description*4 + scan-line-direction-description, where a direction-description is:

> 0 if the coordinate increases as *x* increases

1 if the coordinate decreases as *x* increases
2 if the coordinate increases as *y* increases
3 if the coordinate decreases as *y* increases

This convention is identical to the one used by Press and AIS. We use it in this document to characterize character encodings: if a raster is encoded in mode 8, then the first bit of the bit stream defining the character will be at the lower left-hand corner of the character; the next bit will be just above the first, and so on *up* the page (because the bit-direction-description is 2); then the next scan-line to the right will be given (because the scan-line-direction-description is 0).

Note that there is a relation between *rotation* and *scanning mode*. For example, a character encoded with rotation=0, scanning mode=3 is identical to one recorded with rotation=90 degrees, scanning mode=8.

2.2 File terminology

A file is a homogeneous sequence of data bits. (We at PARC do not have any file systems that have the concept of "record" as implemented in XDS and IBM operating systems. We view a file as an unbroken sequence of data.)

A *word* is 16 bits, a *byte* is 8 bits. If these are to interpreted as *signed* integers, the representation is two's complement.

Several files use the concept of *self-relative pointers*. The idea is that the pointer specifies a file position *relative* to the file position of the pointer itself. The following example may help clarify the notion of self-relative pointers. Suppose that the character encoding for character 101b starts at word 1650b of the file, and that a self-relative pointer to that encoding is at word 105b of the file. Then word 105b of the file will contain 1543b=1650b-105b.

2.3 Numbers

Numbers in this document are decimal unless followed by a "b," in which case they are octal. 12b=10.

A FloatingPoint number is a two-word structure that contains a sign, an 8-bit exponent and a 23-bit mantissa. This representation is identical to the 32 most significant bits of the representation used by the PDP-10 and MAXC. The Alto BCPL subroutine package FLOAT manipulates these numbers as well. (Further information about the actual encoding of numbers can be found in PDP-10 documentation or in FLOAT documentation.)

3. File Naming Conventions

A standard naming convention is used for font files. In some cases, programs depend on adherence to the convention (e.g., extracting width information from EARS fonts). The convention permits programs to "parse" the font name to discover various parameters. The convention is:

{family-name-in-full}{point-size}{[B|L]}{[I]}{[C|E]}.{extension}

The optional B stands for "bold;" L for "light," I for "italic," C for "condensed," and E for "expanded." If a font file applies to all sizes of character (e.g., a spline file), the {point-size} is omitted. Examples:

Helvetica12.Ep	12-point Helvetica font for EARS
Helvetica12b.Ep	12-point bold Helvetica font for EARS

The {extension}s are chosen to identify the *format* of the file. Standard extensions are given below, together with the MAXC directory (inside brackets < >) where such files are traditionally found.

Standard formats:

.XX-SF	Spline representations edited with FRED. <PRESSFONTS>
.AC	Raster representations, edited or created with PrePress. <PRESSFONTS> (Usually, though not always, these are Alto fonts.)

Subsidiary formats:

Fonts.Widths	Summary of widths. <FONTS>
.SD	Compact spline representations (SDtemp format). <PRESSFONTS>
.CU	"Carnegie-Mellon University" format.

Subsidiary formats (device-dependent):

.AL	Alto-format (CONVERT) font. <ALTOFONTS>
.STRIKE	Alto-format font (BITBLT). <ALTOFONTS>
.EP	EARS-format portrait font. <FONTS>
.EL	EARS-format landscape font. <FONTS>
.XH	XGP-format font, for XPRINT. Archived from <FONTS>
.VT	VTS-format font. Archived from <FONTS>

4. PrePress File Format

Several of the file formats are variants of a generic file created and modified by PrePress. The format was designed to be easily extendable to include new sorts of information and to permit many different fonts to be included in one file. PrePress documentation refers to this files with names like SD, SDtemp, CD, CDtemp, WD, WDtemp. An index at the head of the file describes each font *segment* that is contained within the file. The intention is that a reader will scan the index to find a pointer to the font he desires. Thus a file is:

```
structure PrePressFile:
    [
    index word howeverMany
    @IX                                 //Index entry with type=0 (end of index)
    stuff word howeverManyAgain
    ]
```

Each index entry begins with a common form of header:

```
structure IX:
    [
    type bit 4                          //Various type codes are assigned
    length bit 12                       //Length of entry in words, counting this one
    ]
```

A particular kind of index entry establishes a correspondence between a *code* and a string:

```
structure IXN:
    [
    @IX                              //Header with type =1
    code word                        //The numeric code
    nameLength byte                  //The number of characters in the name
    characters ↑1,19 byte            //Room for the name
    ]
```

Note that a name entry has a fixed length, although the name itself can be of any length up to 19. The final 20 bytes in the IXN structure are in the same format as a BCPL string. By convention, an IXN entry must establish a correspondence between a name and a code before any index entries that use the code appear.

Each segment of the file will have an index entry that points to it (SplineSegment, CharacterSegment, or WidthSegment). They all have roughly the same form:

```
structure STDIX:
    [
    @IX                              //Header with various types
    family byte                      //Family name, using a name code
    face byte                        //Encoding of the face properties
    bc byte                          //Code for the "beginning character"
    ec byte                          //Code for the "ending character"
    size word                        //Size of the font segment
    rotation word                    //Rotation of the font segment
    segmentSA word 2                 //Starting address in file of the font segment
    segmentLength word 2             //Length of the segment
    ]
```

The family name is identified by referring to a name-code correspondence established with an IXN entry. The face is encoded as:

```
        (if bold then 2 elseif light then 4 else 0)+
        (if italic then 1 else 0)+
        (if condensed then 6 elseif expanded then 12 else 0)
```

The two entries bc and ec give the character codes for the first and last characters represented in the segment. This allows partial fonts to occupy less space. Size gives the size of the font description in micas. Rotation gives the rotation, in minutes of arc. segmentSA and segmentLength specify the location of the segment in the file (both entries are double-word integers, in units of file words): these are included to permit random access to a large number of segments in one file.

A common special case of a PrePress file is a font file that contains only one segment, and consequently a very brief index (a name entry, and entry pointing to the segment, and an End entry). The AC and SD files are examples.

5. Standard Formats

5.1. Outline representation -- SF format.

The standard format for outline representations is a specially-organized text file. The file is normally read and written by FRED, the interactive editor for outlines, and by PrePress, the program for converting the outline representations to other formats. We designed the SF format to be based on a text file, and further to be readable by the INTERLISP programming system, in anticipation of the need to make transformations on outlines once they were defined (the transformations could be made by hand with a text editor, or by writing a suitable LISP program). This approach has several times saved us from some very messy effort to repair a damaged binary file--the text file has been a good idea.

The definition of the file follows normal INTERLISP conventions for atoms, numbers, strings, and lists. (A number is either an integer of the form 123 or an octal number followed by Q, i.e., 12Q=10, or a floating-point number with an exponent heralded by E, e.g., 1.23E-4.) In the description below, vertical bar (|) is used to separate alternatives, and

<...>	is a list,
{...}	is a string,
[...]	is a number.

A single SF file may contain definitions for several characters, although the definitions are independent. The file is a sequence of <character description>s, terminated by the atom STOP:

<character description> ... <character description> STOP

Normally, a full font will consist of about 7 SF files. These are conventionally given names like:

family.LC1-SF	Lower case, first file
family.LC2-SF	Lower case, continuation file
family.UC1-SF	Upper case, first file
family.UC2-SF	Lower case, continuation file
family.NUM-SF	Numerals
family.S1-SF	Special characters, first file
family.S2-SF	Special characters, continuation file

A <character description> is:

```
((FAMILY   {family name})
 (CHARACTER   [code])
 (FACE    { B | M | R }    { R | I }    { C | R | E })
 (WIDTH   [width in x]   [width in y])
 (FIDUCIAL   [dimension in x]   [dimension in y])
 (VERSION   [number]   {date})
 (MADE-FROM   {file name}
      [x character origin]   [y character origin]
      [x fiducial origin]    [y fiducial origin])
 (SPLINES   <closed curve> ... <closed curve>))
```

Alternatively, a <character description> may specify that some other character is to be copied into this one (not universally implemented):

```
((FAMILY   {family name})
 (CHARACTER   [code])
 (USE {family name} [code]   { B | M | R }    { R | I }    { C | R | E }))
```

Within the top-level list for <character description>, a construct of the form (COMMENT {any string}) may be inserted at will.

The FACE characters stand for:

BOLD | MEDIUM | LIGHT
REGULAR | ITALIC
CONDENSED | REGULAR | EXPANDED

It is important to understand the normal use of coordinates in a SF file. The coordinates of knots, for the width, origins in the MADE-FROM description, and in the FIDUCIAL annotation, are all Alto screen units: these are recorded directly by FRED. However, these coordinates must ultimately be related to a more standard system common to all characters in the world. The FIDUCIAL serves this purpose: it gives *the distances, in x and y, that correspond to the point size of the character.* For example, if we use FRED to design a (nominal) 12-point character, we set the fiducials to the dimension (in Alto screen units) that should be mapped into 12/72 inch on the final page image.

A <closed-curve> is:

(<spline> ... <spline>)

A <spline> is:

([n] <knot list> <weight list> <derivative list> {solution method})

where [n] is the number of knots, <knot list> is:
(([X_1] [Y_1]) ([X_2] [Y_2]) ... ([X_n] [Y_n]))
<weight list> is either NIL, in which case all knots are weighted equally, or:
([W_1] [W_2] ... [W_n])
and <derivative list> is:
(([X_1'] [Y_1'] [X_1'']. [Y_1''] [X_1'''] [Y_1''']) ...
... ([X_{n-1}'] [Y_{n-1}'] [X_{n-1}''] [Y_{n-1}''] [X_{n-1}'''] [Y_{n-1}''']))
and {solution method} is:
{ NATURAL | CYCLIC | PSEUDOCYCLIC }

The numbers in this description are handled slightly differently: derivatives and weights are floating point numbers, character code is octal (e.g. 101Q) or decimal, all other numbers (in particular knot coordinates) are integers.

5.2 Raster representations -- AC format.

The standard format for raster representations is the AC file, usually edited with the PrePress font editor. This format is used because it contains more information about characters than any other font format we have. Consequently, one can always convert to formats that demand less information. *By convention, AC files assume a scanning mode of 8.* (Note: PrePress "character" files such as CD and CDtemp are in this format.)

The file is a *segment* of a "PrePress font file" (see section 4 for a general discussion of PrePress files). The font file contains some identification information, and a directory that points to a *character segment*, which itself contains the information about the font. An index entry that points to a character segment is:

```
structure CharacterIndexEntry:
    [
    @STDIX                      //Standard header with type=3.
    resolutionS word            //Resolution in scan-lines/inch * 10
    resolutionB word            //Resolution in bits/inch * 10
    ]
```

This index entry points to a CharacterSegment:

```
structure CharacterSegment:
    [
    charData ↑bc,ec @CharacterData          //Useful data about each character
    directory ↑bc,ec @relFilePos            //Relative file positions of rasters
    rasters ↑bc,ec @rasterDefn              //The actual raster encodings
    ]
```

```
structure CharacterData:
    [
    XWidth @Fraction                        //X Width (scan-lines)
    YWidth @Fraction                        //Y Width (bits)
    BBox word                               //Bounding box offsets
    BBoy word
    BBdx word                               //Width of bounding box (scan-lines)
    BBdy word                               //Height of bounding box (bits)
    ]
```

The first two entries are signed fractions (a fraction is two words: the first is the integer part, the second the fractional part) that give the width vector (with reference to the origin of the character). The four parameters of the bounding box follow. However, BBdy=-1 is reserved to indicate that a character of this code does not really exist in the font (such a code is necessary because CharacterData structures are recorded for all character codes in the range bc through ec).

The *directory* portion is a table that points to the raster definitions of each character in the range bc through ec. Each pointer is 32 bits long (a double-word integer) that gives the position in the file in words, relative to the beginning of the *directory* table, of the rasterDefn for the appropriate character. If a character of the given code is not in the font, both words of the relFilePos are -1.

A rasterDefn is:

```
structure rasterDefn:
    [
    BBdyW bit 6                             //Height of raster (in words)
    BBdx bit 10                             //Same as BBdx in CharacterData
    raster word BBdyW*SW                    //The actual raster bits!
    ]
```

The value of BBdyW is simply K(BBdy+15)/16l, the number of words required to specify one scan-line. Each scan-line in the raster encoding begins on a word boundary.

6. Subsidiary Formats

6.1 Fonts.Widths format.

The file Fonts.Widths is used to disseminate width information to all formatting and editing programs. Its basic format is that of a PrePress font file, with index entries that point to *WidthSegments.* An index entry is of the form:

```
structure WidthIndexEntry:
    [
    @STDIX                                  //Standard header, type=4
    ]
```

The interpretation of the *size* entry in this index is somewhat subtle. If it is non-zero, then it is the size of the font, measured in micas. Thus, a 12-point font would have *size*=453. In this case, the width information is said to be *absolute*. On the other hand, if *size* is zero, then the width information will be usable for fonts of any size (i.e., we shall scale it by the actual font size), and the information is said to be *fractional*. If the data are *absolute*, then all dimensions are measured in micas. If they are relative, dimensions cited in the WidthSegment must be scaled by 2540P/72000, where P is the point size of the desired font, in order to convert the numbers to micas (You will note that this simply means that entries are measured in thousandths of the point size). The widths file may contain entries for both *absolute* and *fractional* information for the same font; in this case the *absolute* information takes precedence.

The index entry points to a *WidthSegment*, which has the following format:

```
structure WidthSegment:
    [
    FBBox word                          //X offset for font bounding box
    FBBoy word                          //Y offset for font bounding box
    FBBdx word                          //X width for font bounding box
    FBBdy word                          //Y height for font bounding box
    YWidthFixed bit                     //=1 if all Y widths equal
    XWidthFixed bit                     //=1 if all X widths equal
    spare bit 14
    widthData word howEverMany
    ]
```

The first four numbers are the dimensions of the *font bounding box*. At the end of the entry comes (widthData) the width information for individual characters. First comes the X width information. If the XWidthFixed flag is set, there is only one number given, which applies to all characters in the font. If the XWidthFixed flag is zero, then there are ec-bc+1 words that give the X widths of the characters with codes from bc to ec inclusive. Then follows the Y width information, correspondingly encoded. In order to identify "non-existent" characters in the range bc to ec, a width (either absolute or fractional) of 100000b (the most negative number) signals a non-existent character.

Note: The widths file should really be able to deal with device-dependent widths as well: this is a tremendous help with photocomposers, etc. Consequently, a WidthIndexEntry should really include a *deviceCode*, which identifies (by correspondence with some string in a IXN entry) the relevant device. If the device is PRESS, then the font would be assumed to be standard across a variety of devices; a width entry with an exact match of device name would take precedence over standard (PRESS) widths.

6.2. Compact outline representations -- SD format.

Because the SF files that describe outline representations are somewhat bulky and tiresome to interpret, there is an alternative format: SD. This format is created from the SF files by the PrePress READSF command (i.e., SD files are in the same format as is SDtemp). The file is in the general " PrePress font file" format, with an index entry:

```
structure SplineIndexEntry:
    [
    @STDIX                              //Standard header, type=2
    ]
```

The *size* entry in the index must be zero. This index entry points to a SplineSegment:

```
structure SplineSegment:
    [
    splineData ↑bc,ec @SplineData          //Useful information about each character
    directory ↑bc,ec @relFilePos           //Directory pointing to spline encodings
    splines ↑bc,ec @splineCodes            //The encodings of each character
    ]
```

The information about each character is:

```
structure SplineData:
    [
    XWidth @FloatingPoint                  //Width in X direction
    YWidth @FloatingPoint                  //Width in Y direction
    BBox @FloatingPoint                    //Left edge of bounding box
    BBoy @FloatingPoint                    //Bottom edge of bounding box
    RightX @FloatingPoint                  //Right edge of bounding box (=BBox+BBdx)
    TopY @FloatingPoint                    //Top edge of bounding box   (=BBoy+BBdy)
    ]
```

All of these coordinates are relative to the origin of the character, and use the convention that 1.0 is equal to the point size of the final character. Consequently, most are usually fractional. A special (illegal) value of XWidth is used to flag SplineData structures that correspond to non-existent characters in the font (this problem arises because there are SplineData structures for all characters bc through ec, even though they may not all exist). The special value is 0 in the first word, and -1 in the second word.

The interpretation of the *directory* is precisely the same as for AC files.

The encoding of each character (splineCodes) is essentially a list of commands to a scan-conversion algorithm, such as the one used in PICO. Five different kinds of entries may appear:

```
structure SMoveTo:
    [
    codeMoveTo word                        //Command code =1
    X  @FloatingPoint
    Y  @FloatingPoint
    ]

structure SDrawTo:
    [
    codeDrawTo wor d                       //Command code =2
    X  @FloatingPoint
    Y  @FloatingPoint
    ]

structure SDrawCurve:
    [
    codeDrawCurve word                     //Command code =3
    X'  @FloatingPoint
    Y'  @FloatingPoint
    X"/2  @FloatingPoint
    Y"/2  @FloatingPoint
    X"'/6  @FloatingPoint
    Y"'/6  @FloatingPoint
    ]
```

```
structure SNewObject:
    [
    codeNewObject word                    //Command code =-1
    ]

structure SEndDefinition:
    [
    codeEndDefinition word                //Command code =-2
    ]
```

Each closed curve is specified with a sequence that begins with SMoveTo, and uses subsequent SDrawTo and SDrawCurve entries to trace the outline. An entirely new object is initiated with SNewObject (this is presently unnecesssary, and unimplemented). The entire character is terminated with SEndDefinition.

The SDrawCurve entry gives the parameters for a parametric cubic spline:

$$x = X_0 + X' t + (X''/2) t^2 + (X'''/6) t^3$$
$$y = Y_0 + Y' t + (Y''/2) t^2 + (Y'''/6) t^3$$

where t ranges from 0 to 1, and (X_0, Y_0) is the starting point of the curve.

The SD files created by PrePress from SF files have an additional property: each SDrawCurve entry defines a curve segment that is monotonic in *both* x and y directions. This simplifies scan-conversion for both portrait and landscape printing devices, provided the font characters are rotated a multiple of 90 degrees (or 0 degress, of course).

6.3 CU format.

The CU format was once our standard format for raster representations; some vestigial software in fact still uses this format. It has the great virtue of simplicity, but is rather bulky and lacks some crucial information.

The file has the structure:

```
structure CU:
    [
    H word                                //Height of font (number of scan lines)
    WW word                               //"Word width" of font
    character ↑1,howeverMany              //Character codings
    ]
```

Each character is a separate encoding with a character code, a width (in bits) for the character, and a raster. Every character in the file is placed within a raster of the same size: this raster size is thus analogous to the *font bounding box*, but is actually somewhat larger because the width of the box is a multiple of 16.

```
structure CUChar:
    [
    ASCIICode word                        //ASCII character code
    Width word                            //Width of character in bits
    raster word H*WW                      //The actual encoding of the raster
    ]
```

The raster is a sequence of scan-lines, each encoded in WW words. The first scan-line is at the "top" of the character. Within a scan-line, bits are given from left to right (more significant bits to less significant bits). Characters are, in general, "at the left" in the font bounding box; white space is provided on the right.

This font format omits some useful information: the location of the origin within the bounding box. There is a convention used to remedy this lack: the lower leftmost 1 bit in the encoding of upper case A (ASCII code 101b=65 decimal) is at the origin.

7. Subsidiary formats -- device dependent

7.1 AL format.

The AL format is designed to simplify the use of the Alto CONVERT instruction for creating displays (see the Alto Hardware Manual for a description of CONVERT).

```
structure AL:
    [
    Height word                          //Height of font (scan-lines)
    proportional bit                     //True if proportionally spaced font
    baseline bit 7                       //(see below)
    maxWidth bit 8                       //Width of widest character
    pointers ↑0,nCharsX                  //Self-relative pointers to XW entries
    charData word howEverMany
    ]
```

The Height entry must be > FBBdy. The baseline entry equals the height of the font bounding box above the origin (=FBBoy+FBBdy). If the AL font dates from a somewhat earlier vintage, the baseline may be recorded as 0.

The pointers table contains self-relative pointers to character encodings. Each character encoding in the charData region can describe at most 16 (horizontal) bits of character data; if the character requires more data bits, an "extension character" is used to contain the rest of the data. Characters may have as many extensions as necessary.

By convention, the first 377b entries in the pointers table are assumed to be self-relative pointers for the corresponding ASCII characters codes. Following these entries are entries for any necessary extension characters.

The data for a character encoding is represented as:

```
structure XHdata:
    [
    bitData word XH                      //Top scan-line first
    XW word                              //(see below)
    HD byte                              //(see below)
    XH byte                              //Number of scan-lines of bit data
    ]
```

In order to conserve space, the bit data omits all-zero words at the top and bottom of the character. The HD entry records the number of scan-lines at the top of the character (relative to the font bounding box) that are omitted. (Technically, HD = FBBdy+FBBoy-(BBdy+BBoy).)

The XW word is interpreted in one of two ways. If the width of the character is 16 or fewer bits, then XW is (2*width)+1. Otherwise, the character must require an extension character, and XW contains 2*xCode, where xCode is the character code of the extension character. The final extension character will have an XW that contains (2* width of final extension)+1. rather than the total width. The self-relative pointers in the pointers table point to the XW word.

By convention, the first character encoding in the charData region is a "dummy" to which all non-existent character codes point. This dummy has XH=0, HD=0, and XW=1.

7.2. STRIKE format.

The STRIKE format was devised to permit graceful use of BITBLT for writing characters onto the Alto screen.

There are two kinds of font files: a STRIKE (conventional extension .Strike) and a STRIKEINDEX (conventional extension .StrikeX). A Strike is very simply a bunch of characters that are grouped together in a bit map and given a table of x-positions of each character. A StrikeIndex is essentially a map that maps character codes into (strike, code) pairs, together with the associated strikes. An index can thus be used to achieve sharing because several character codes may map to the same (strike, code) pair. Or it can help save space, because the characters of a font may be grouped into several strikes in order to save top and bottom scan-lines; the index can then establish the proper correspondence.

First, the simple "Strike" format that most people would use:

```
structure STRIKE:
    [
    @StrikeHeader                       // header of standard format
    @StrikeBody                         // the actual strike
    ]

structure StrikeHeader:
    [
    format word = [
      oneBit bit                        //=1 to mean new style fonts
      index bit                         //=0 for simple strike, 1 for index
      fixed bit                         //=1 if all chars have width maxwidth
        blank bit 13
    ]
    min word                            // minimum ASCII code
    max word                            // maximum ASCII code
    maxwidth word                       // width of widest character
    ]

structure StrikeBody:
    [
    length word                         // number of words in this StrikeBody
    ascent word                         // in scan-lines (=FBBdy+FBBoy)
    descent word                        // in scan-lines (=-FBBoy)
    xoffset word                        // in bits (<0 for kerning, else 0 ) (=FBBox)
    raster word                         // # words per scan-line in bit map
    bitmap word raster*height           // the bit map
                                        // define: height = (ascent + descent) = FBBdy
    xinsegment ↑ min,max+2 word         // index ↑ ASCII into the strike
                                        // dummy at max+1
    ]
```

(Note: When a StrikeBody is in memory, separate pointers to xinsegment and bitmap would probably be kept. The reason for keeping the xinsegment entry last is that in the StrikeIndex form, below, we don't know the value of max and hence could not compute the address of the bit map.)

The "bitmap" entry is a bit map -- there are height=ascent+descent scan-lines, each of which is raster words long. The font defines pictures for characters corresponding to ASCII codes min through max. Given a code c, with min < _ c ≤ max, we get:

```
    xLeft ← xinsegment ↑ c
    xWidth ← (xinsegment ↑ (c+1))- xLeft
```

If xWidth is 0, there is no picture for the character (i.e., the character does not exist in this font). Otherwise, the portion of the bit map corresponding to xLeft < _ x ≤ xLeft+xWidth-1 contains the picture for the character.

Each font is blessed with a dummy picture that can be displayed for all illegal characters -- it is given code = max+1.

The ascent is a measure of the number of scan-lines in the bit map that are above the putative baseline of the font; descent is the number of scan-lines below the baseline. The xoffset is an offset to be applied to the "destination x" of all characters: if a character is to be "placed" at (x,y), the destination x given to BITBLT should be x+xoffset; this feature is for kerned characters.

A StrikeIndex is simply an index at the front of some StrikeBodies:

```
structure STRIKEINDEX:
    [
    @StrikeHeader                       // common header
    maxascent word                      // max ascent of all strikes
    maxdescent word                     // max descent of all strikes
    nStrikeBodies word                  // number of strike bodies
    map ↑min,max+1 @mapEntry            // map to (strike, code)
                                        // dummy at max+1
    bodies ↑1,nStrikeBodies @StrikeBody // the strike bodies.
    ]

structure mapEntry:
    [
    strike byte                          // which strike
    code byte                            // which code
    ]
```

In this case, the StrikeBodies all have implicit values min=0; 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 max+1 in the index maps to a dummy picture. Undefined characters in the range min to max are identified by mapping to a strike number >127, i.e. by the sign bit of the map entry being 1.

7.3 EL and EP format for EARS fonts

Font formats for the EARS system are compressed (all other raster representation formats mentioned in this document use no compression). The extension .EP is used, by convention, to denote "portrait" fonts (font strings will run horizontally on the page if it is oriented as a portrait). The EL extension is used for "landscape" fonts.

Both sorts of font have the same format (remember that EARS scans in mode 8):

```
structure ELEP:
    [
    @Record0                            //General information
    @Record1                            //Character information
    Record2 word howEverMany            //Actual character encodings
    @Record3                            //Font specification table
    ]

structure Record0:
    [
    MRLILength word                     //Length of Record2 (in words)
    maxWidth word                       //Maximum character width (scan-lines)
                                        // max (over all Record1Entry's) of Width
    maxHeight word                      //Maximum character height (bits) FBBdy
    TTYTab word                         //How many bits or scan-lines for a tab
    defaultFSN word                     //Default font set number (PSPOOL)
    reserved word 3                     //Used by PSPOOL
    blank word 56
    ]

structure Record1:
    [
    characterData ↑0,127 @Record1Entry  //Descriptions of each character
    ]

structure Record1Entry:
    [
    FontAddress word                    //Address (in words) into Record2 of encoding
                                        // (relative to beginning of Record2)
    FontLength word                     //Number of words of encoding in Record2
    Width word                          //"Width" of character (amount to "space" over)
    W word                              //Width of bounding box BBdx
    H word                              //Height of bounding box BBdy
    baseline word                       // BBoy (portrait) or BBox (landscape)
    codingType word                     //(see below)
    alignment word                      // FBBdx+FBBox-BBox (landscape only)
    ]
```

The codingType is 0 if the character does not really exist in the font. It is <0 if the encoding within Record2 is RLI (run length increments). It is >0 if the encoding is a matrix (in this case, the value of codingType is the height of the matrix in bytes).

Record2 contains the encodings of the rasters for the individual characters (as pointed to by Record1 and Record3 entries). If the encoding is a matrix, the entry in Record2 in an uncompressed raster for the character (scanning mode=8), with (1) the height rounded up to the next multiple of 8 bits, and (2) a possible 1-byte padding at the end of the matrix encoding to make the entry an integral number of 16-bit words long. For example, the K of Figure 4 would have a matrix encoding of:

```
    100004b                             (first scan-line, rounded up to 16 bits high)
    177777b                             (second scan-line, ...)
```

```
177777b
103004b
001400b
003600b
006300b
014140b
130064b
160034b
140014b
100004b                                 (last scan-line)
```

Most characters will be encoded in Record2 with a more economical scheme: RLI. This is a compression scheme that reduces font storage for high-resolution characters (compression of 3.5:1 is typical for a 12-point font at 500 bits/inch). We shall describe RLI by referring to Figure 4. Each scan-line could be coded as a series of number pairs, where the first number of each pair represents a number of "white" bits to be followed by the number of "black" bits specified by the second number of the pair. With this scheme, the first scan-line of the K would be represented by the two pairs (0,1) and (12,1). We can omit the parentheses and write simply 0,1,12,1. The entire K is encoded into runs as follows:

Scan-line	Runs		RLI
0	0,1,12,1	(R)	0,1,12,1
1	0,14	(R)	0,14
2	0,14	(I)	0,0
3	0,1,4,2,6,1	(R)	0,1,4,2,6,1
4	6,2	(R)	6,2
5	5,4	(I)	-1,2
6	4,2,2,2	(R)	4,2,2,2
7	3,2,4,2	(I)	-1,0,2,0
8	0,1,1,2,6,2,1,1	(R)	0,1,1,2,6,2,1,1
9	0,3,8,3	(R)	0,3,8,3
10	0,2,10,2	(I)	0,-1,2,-1
11	0,1,12,1	(I)	0,-1,2,-1

The second column gives simply the runs. The third column gives the run-length-*increment* format: a given scan-line is represented as increments on the runs for the previous scan-line, provided there are the same number of runs as in the previous scan-line. Thus scan-line 10 is represented by the increments 0,-1,2,-1, which are added to the runs for scan-line 9 (0,3,8,3) to yield runs 0,2,10,2 for scan-line 10. For high resolution characters (our example is not high resolution), the incremental mode (I) dominates.

The RLI information is encoded as follows. The character encoding starts in Record2 at the location specified by Record1 and Record3 entries; RLI information is recorded for each scan-line (starting with the left-most scan-line, scan-line 0 in our example). Runs appear in 8-bit bytes, where the first bit of a byte is a flag which is set to mark the last run for a scan-line. Thus, scan-line 9 is represented by the 4 8-bit bytes 0, 3, 10b and 203b; these are packed into words as 3b and 4203b. Because of this encoding, runs are limited to the range 0-127; if a longer run is needed, two runs may be spliced with a zero-length connector (e.g., 100,0,100,10 is equivalent to 200,10). A limit of 8 runs is imposed for each scan-line (characters requiring more than 8 runs can be represented in matrix format).

The increments for RLI are specified in 4-bit groups in which the first bit is used as a flag and the remaining 3 bits are 2's complement increments (range -4 to 3). As with runs, the flag bit for the last increment of the scan-line is set. In addition, the flag bit of the first increment on the scan-line is set (this allows runs to be differentiated from increments, because there are always at least 2 runs per scan-line). For example, the increments to scan-line 10 are encoded as the 4-bit quantities 10b, 7b, 2b, 17b; these are packed into 8-bit bytes as 207b, 57b; or into a 16-bit word as 103457b. Note that if increments do not fall in the range -4 to 3, you can always use a *run* representation rather than an *increment* representation.

This encoding will produce an integral number of 8-bit bytes for each character. Consequently, a character may be followed by a 1-byte padding in order to stard the subsequent character at a word (16-bit) boundary.

Record3 is a very compact description of each character, and is actually examined by the RCG hardware:

```
structure Record3:
    [
    fontSpecTable ↑0,127 @CharSummary
    ]

structure CharSummary:
    [
    baseline bit 13                //Two's complement baseline (0 for landscape)
    matrix bit                     //True if encoding is a matrix (not RLI)
    endOfPage bit
    notEndOfLine bit
    Width word                     //Amount to space over to next character
    W bit 10                       //Bounding box width -1

    Hb bit 7                       //⌈(Height+7)/8⌉ -1
    fontAddress bit 15             //Relative address in Record 2 of encoding
    ]
```

7.4 XH format XGP fonts for XPRINT

The XH format was devised to simplify the inner loop of XPRINT, a program for printing text on the XGP. The XGP scans in mode 3. The file has the format:

```
structure XH:
    [
    nChars word                    //The number of characters in the font
    nData word                     //Number of words of font data
    H word                         //Height of the font (in scan-lines)
    W word                         //Maximum width (in words) of any character
    pointers ↑0,nChars-1  word     //Self-relative pointers to  charData (see below)
    widths ↑0,nChars-1 word        //Width to space to next character
    data word nData                //Character encodings (see below)
    ]
```

nChars is usually 128 or 256. The height H must be > FBBdy. A width of zero identifies a non-existent character; any width up to 12 W is legal.

The character encodings are represented as follows:

```
structure charData:
    [
    ↑1,K(width+11)/12l @block          //Each block defines up to 12 bits
    ]

structure block:
    [
    ↑1,H [ bitData bit 12              //Up to 12 bits of character data
          validBits bit 4 ]            //Number of bits in bitData that are valid
    ]
```

Thus a character is defined by successive blocks of H words; each block defines up to 12 horizontal bit positions of the character. The first word in the block defines the top scan-line, the next word the next scan-line, etc. Words of the block define up to 12 bits of character data: the validBits field contains the number of valid bits in the word (1 is minimum; 12 is maximum). All blocks except the last have validBits=12.

ASCII Character Codes

underline	30b	A	101b	a	141b
space	40b	B	101b	b	142b
!	41 b	C	103b	c	143b
"	42b	D	104b	d	144b
#	43b	E	105b	e	145b
$	44b	F	106b	f	146b
%	45b	G	107b	g	147b
&	46b	H	110b	h	150b
’	47b	I	111b	i	151b
(	50b	J	112b	j	152b
)	51b	K	113b	k	153b
*	52b	L	114b	l	154b
+	53b	M	115b	m	155b
,	54b	N	116b	n	156b
-	55b	O	117b	o	157b
.	56b	P	120b	p	160b
/	57b	Q	121b	q	161b
0	60b	R	122b	r	162b
1	61b	S	123b	s	163b
2	62b	T	124b	t	164b
3	63b	U	125b	u	165b
4	64b	V	126b	v	166b
5	65b	W	127b	w	167b
6	66b	X	130b	x	170b
7	67b	Y	131b	y	171b
8	70b	Z	132b	z	172b
9	71b	[	133b	{	173b
:	72b	\	134b	|	174b
;	73b	]	135b	}	175b
<	74b	↑	136b	~	176b
=	75b	←	137b		
>	76b	’ (left quote)	140b		
?	77b				
@	100b				

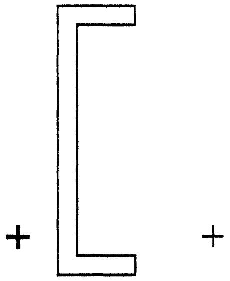

Figure 1. Outline representation

+ Origin

+ Origin of next character

Width = + - +

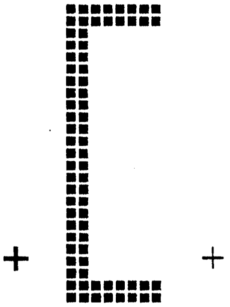

Figure 2. Raster representation

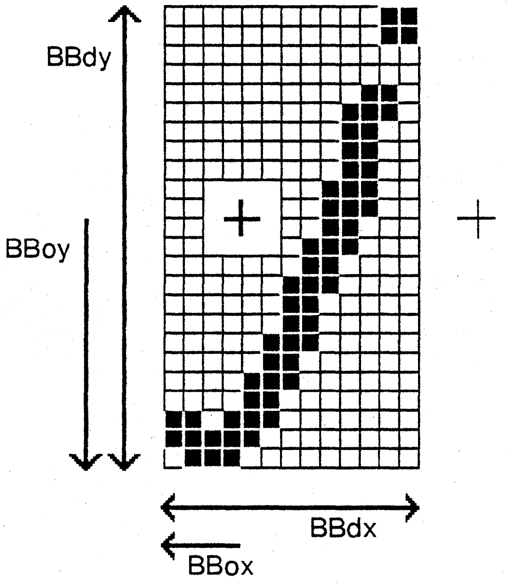

Figure 3. Bounding box conventions. In the example, BBdx = 13, BBdy = 24, BBox = -4, and BBoy = -13

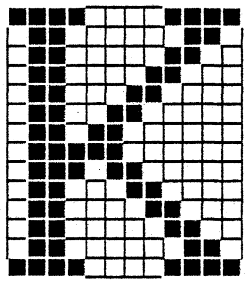

Figure 4. RLI coding example.