Extending Typeman Talk

**Control Program**

The following two “settings” declarations have been added:

ClearTypeCtrl = <n>

LinearAdvanceWidths = <n>

ClearTypeCtrl is a “macro” that uses INSTCTRL[] to set the appropriate flags indicating a font that is CT aware (and hence the rasterizer shouldn’t e.g. bypass delta instructions). Acceptable values for n are 0 and 1 (for off and on, respectively).

LinearAdvanceWidths permits to override the default usage of USE\_INTEGER\_SCALING flag in the ‘head’ table. Acceptable values for n are 0 (default) and 1 (don’t use integer scaling). Non-integer scaling yields advance widths that are closer to the designed advance widths because it does not e.g. take 11 pt at 96 dpi and round the resulting 14 2/3 ppem up to 15 ppem. Internal to the rasterizer e.g. MPPEM will still return an integer ppem size, but instructions keying off of this ppem size may no longer produce the same results. For instance, at 11 pt, 96 dpi (MPPEM rounds to 15 ppem from 14 2/3), the rasterization may get one pixel pattern, while at 15 pt, 72 dpi (or true 15 ppem), the rasterization may get a different pixel pattern, but since both ppem sizes are reported as the same number, there is no easy way to distinguish. It would be better if this flag was not in the head table, but could be passed into the rasterizer, e.g. through its client interface, or by extending the INSTCTRL, such that a client could choose integer ppem size along with bi-level rendering, while choosing linear advance widths with CT, particularly fractional pixel positioned CT. Notice that this flag *does make a difference* even in fractional pixel positioning. To see this simply look at the waterfall view at various display resolutions (especially 96 dpi and 120 dpi) and compare line lengths with the flag on or off.

The following TT functions are modified or new (generated by the auto-hinter):

* CALL[], <rel cvt>, <ppem size>, <abs cvt>, 85

(modified) equalizes double specified heights below a given ppem size

* CALL[], <cvt>, 89

(new) rounds <cvt> based on rendering environment and current projection vector

* CALL[], <p>, <ch0>, <fraction>, <ch1>, <ppem>, 90

(new) below ppem threshold make both children the same as the averaged parent cvt, at and above calculate child cvt 0 as fraction of child cvt 1. This function is used to control lc/UC main vertical stem contrast.

* CALL[], <ch>, <fraction>, <p>, <ppem>, 91

(new) below ppem threshold make child cvt the same as parent cvt, at and above calculate child cvt as fraction of parent cvt. This function is used to control y/x contrast (main thin and main thick stroke), separately for lc and UC characters.

* CALL[], <ch>, <blend>, <p0>, <p1>, 92

(new) calculate a child cvt as a blend of two parent cvts (alpha blending). This function is used to force inheritance clusters between the extremes defined by the main thin and main thick stroke. Note that this function may be used for implementing optical scaling by using the cvts of the optical extremes.

Some of the above functions use a ppem threshold. These thresholds currently apply only in the case of 1 sample/pixel (if a child cvt is allowed to split off of a parent cvt at any ppem size, it may happen that the weight of the child cvt decreases as the ppem size increases, which looked odd to me. Hence I change the split ppem size to a size at which the (larger) parent cvt size increases by a pixel, at which point I simply keep the child cvt size the same) but not to cases of multiple samples/pixel (where I allow the split as soon as the delta exceeds ½ sample; for over- and undershoots this might be altered to wait for a delta of 1 full sample). Since the number of samples/pixel is not always the same in x and y, calls to these functions are preceeded by setting up the correct pv, which in turn is used by these functions to determine the sample size and corresponding inheritance method. If there is ever a substantial need to use these inheritance functions manually, it would be helpful to replace the several inline ASM calls in the control program with appropriate higher level language constructs or syntax and update the control program compiler accordingly. In the process, the control program compiler should be able to make appropriate decisions as to how to set the pv in the code produced for the pre-program.

**VTT Talk**

The following list of TMT Commands now permits to override the projection and/or freedom vectors otherwise chosen by default by the compiler:

XAnchor(child)

YAnchor(child, cvt)

XDist(parent, child)

YDist(parent, child)

XLink(parent, child, cvt)

YLink(parent, child, cvt)

XShift(parent, child, …)

YShift(parent, child, …)

XInterpolate(parent0, child, … , parent1)

YInterpolate(parent0, child, … , parent1)

XIPAnchor(parent0, child, … , parent1)

YIPAnchor(parent0, child, … , parent1)

Any child in the above list gets to override the fv (as follows), and any parent, with the exception of the parent in the shift command (translates to SHPIX which does not respect the pv in the first place), and the parent1 in the interpolate commands (parent0 is allowed, and it doesn’t make sense to involve two different projection vectors).

The syntax for a parent or child (knot) to override the TT vector (ttv) fv or pv is as follows (else X or Y taken per command’s name)

knot no ttv override

knot > ttv in x-direction

knot ^ ttv in y-direction

knot > knot1 ttv on line from knot to knot1

knot ^ knot1 ttv perpendicular to line from knot to knot1

knot : knot0 > knot1 ttv on line from knot0 to knot1

knot : knot0 ^ knot1 ttv perpendicular to line from knot to knot1

Example:

XLink(2^3,12,74,>=)

YLink(1^0,12:3>2,72,>=)

The first XLink links from parent 2 to child 12 using cvt 74. In the process it sets the dual projection vector perpendicular to 2 and 3 (because of the 2^3), the freedom vector to X (because it is still an XLink, as opposed to a YLink, and rounds the distance, measured along the projection vector, per the rounding informative command previously used on child 12. In a typical scenario, this would be round-to-grid, hence the diagonal stroke weight gets rounded to grid in bi-level (black-and-white) and to 1/16 grid in ClearType. The second YLink links from parent 1 to child 12 using cvt 72. In the process it sets the dual projection vector perpendicular to 1 and 0 (because of the 1^0) and the freedom vector parallel to 3 and 2 (because of the :3>2).

Notes:

1. This is the only correct freedom vector in this scenario; a previous command already has linked child 12 perpendicular to 3 and 2, so the only direction of freedom is perpendicular to that vector, making it perpendicular to perpendicular, or parallel for short.
2. The Y in YLink, in this case, has become all but irrelevant. In fact, the code generated would be identical if one were to use XLink instead of YLink. This is of course perfectly legal because the two links use different projection vectors. The only reason one might prefer to make this a YLink is for the benefit of VTT’s graphical user interface, which is not yet smart enough to distinguish overridden freedom and projection vectors, but it can distinguish X from Y. The GUI will show two separate link arrows only if two separate “letter” are used, which may or may not help visualizing the more advanced code.
3. Having > and ^ permits to turn a YLink into an XLink and vice versa, which is not particularly meaningful. I didn’t want to merge XLink and YLink into simply Link, as I find XLink and YLink more readable for the simpler cases. On the other hand, there are some more complex commands (cf below) where there are two children involved, and there is a need for selecting separate freedom vectors in X and Y.
4. Caution: with the generality of these commands, it is possible to set the freedom vector perpendicular to the projection vector. Given the “duality” of the dual projection vector, it is not obvious how to test this at compile-time; i.e. the “original direction” of the pv is used to take some measurements, and eventually the “current direction” is used move the point. Most of the time, if the fv gets even close to the pv, TrueType gets numerically unstable, hence the current implementation tests the original direction of the dual projection vector against the freedom vector being within about ±3.58° (or scalar product < 1/16, much like flint)

The following are new TMT/VTT Talk commands, targeting two primary goals:

1. Re-interpret the rounding method to mean round-to-sample, instead of round-to-pixel, independently of the “hacks” in the rasterizer. This permits one set of hints to work well for a plurality of rendering environments; hence all the new commands are prefixed by “Res” (for rendering environment specific).
2. Add some higher-level commands, having a larger portion of the context, and therefore permitting to make smarter decisions. For instance, a YIPAnchor, followed by a YLink, likely accumulates rounding errors, especially on thin crossbars (“hair lines”), which can be avoided if the two are combined into a single, higher-level command.

Following are the new TMT/VTT Talk commands in more detail.

1. ResXAnchor, ResYAnchor

Same meaning as XAnchor and YAnchor but added functionality of rendering environment specificity and overridable freedom vector.

2. ResXIPAnchor, ResYIPAnchor

Same meaning as XIPAnchor and YIPAnchor but added functionality of rendering environment specificity and overridable freedom and projection vector. Command will accept one child (interpolee) only.

3. ResXDist, ResXLink, ResYDist, ResYLink

Same meaning as XDist, … YLink but added functionality of rendering environment specificity and overridable freedom and projection vector. Unlike the originals, the Link variants always require a cvt (while the Dist variants never allow one).

4. ResXDDist, ResXDLink, ResYDDist, ResYDLink

Implements a pair of criss-crossed Dists or Links like so:

ResXDDist(parent0, child0, parent1, child1)

ResYDDist(parent0, child0, parent1, child1)

ResXDLink(parent0, child0, cvt0, parent1, child1, cvt1)

ResYDLink(parent0, child0, cvt0, parent1, child1, cvt1)

Projection vectors cannot be overridden, but both freedom vectors can. This command is useful to constrain e.g. the diagonal connecting the two horizontal bars of a “Z”. This scenario would traditionally lead to mutually dependant links (circular dependency) but can be solved with a bit of math and a square root (implemented in TrueType) if it is tackled at this higher level of abstraction. Notice that the two links may have different cvts; this permits to address tapering strokes.

5. ResXIPDist, ResXIPLink, ResYIPDist, ResYIPLink

Implement a combination of an IPAnchor with a Dist or Link to position a vertical or horizontal stroke between two parent points like so:

ResXIPDist<flag>(parend0, child0, child1, parent1)

ResYIPDist<flag>(parend0, child0, child1, parent1)

ResXIPLink<flag>(parend0, child0, child1, cvt, parent1)

ResYIPLink<flag>(parend0, child0, child1, cvt, parent1)

Neither parents nor children are allowed to override the freedom or projection vectors at this point. Notice that in this scenario the cvt is applied between two children. Imagine a traditional link but with the order of child -> parent being irrelevant. In fact, another way of looking at this kind of link is a “symmetric distance constraint”, as opposed to a traditional link, where a parent point is firmly anchored first, followed by anchoring a child point relative to the previous parent point (“asymmetric distance constraint”).

The <flag> instructs the command to use (present) or not to use (absent) some form of stroke position optimization (“tuning”, “sharpening”), like so:

ResXIPDist(…) /\* no optimization \*/

ResXIPDist||(…) /\* use optimization type specified in storage 12 or 13 \*/

ResXIPDist|<(…) /\* same as above, except if space gets “tight” bias positioning to the left \*/

ResXIPDist>|(…) /\* same as above, except if space gets “tight” bias positioning to the right \*/

Notice that for bi-level (black-and-white) optimization simply means to round to pixel grid.

Refer to the font program generated by the auto-hinter, function 84 to setup/change the default stroke optimization methods used. For an explanation of the stroke optimization methods implemented or to change/extend the implementation(s), refer to function 112.

6. ResXIPDDist, ResXIPDLink, ResYIPDDist, ResYIPDLink

Implements a generalization of ResXIPDist … ResYIPLink to two vertical or horizontal strokes, like so:

ResXIPDDist(parent0, child00, child01, child10, child11, parent1)

ResYIPDDist(parent0, child00, child01, child10, child11, parent1)

ResXIPDLink(parent0, child00, child01, cvt0, child10, child11, cvt1, parent1)

ResYIPDLink(parent0, child00, child01, cvt0, child10, child11, cvt1, parent1)

Neither parents nor children are allowed to override the freedom or projection vectors at this point. A typical usage of this command is any character that has two (main, or outer) vertical strokes, where both strokes are optimized (rounded or otherwise tuned), the stroke-to-stroke distance (or more loosely, the black-body-width) is approximated as faithfully to the original as possible, and the side-bearing space is budgeted as best as can be done, given that the positioning of both strokes will contribute to a total rounding error, and given that the side-bearing points are not altered.

This particular implementation represents a set of compromises that correspond to the usage scenario of today’s fonts. Applications and operating systems are modeled to be linear (i.e. text does not re-flow at different resolutions). An alternative set of compromises could be implemented, especially if the actual usage scenario of a font could be communicated in some way or other to the hinting code implementing the particular strategy. For instance, today the hinting code can inquire the rasterizer about the rendering environment downstream (black-and-white, grey-scaled, assorted variants of ClearType, …) but not about whether the font is going to be used in a WYSIWYG environment (hence use an “outside-in” constraint strategy), or in an environment of intelligent adaptive layout (hence use a “left-to-right” or “right-to-left” constraint strategy).

Notice that the two links may have different cvts; this permits to address fonts with a non-trivial stroke weight contrast (e.g. the two vertical strokes of an uppercase “U” are not necessarily the same width, hence likely require different cvts).

A potential variant of this set of commands could implement the cases where the character does not have e.g. two vertical strokes, but is “wider” than a single stroke, such as a lc “r” (single stroke plus some arch extending to the right) or a lc “j” (single stroke plus some tail extending to the left).

Generalizations to three (such as lc “m”) or more strokes (stroke “ladders” such as in EA glyphs) would seem like a logical next step but both exacerbates the “outside-in” vs “left-to-right” dilemma and—in the case of EA fonts—may not be the alternatives upon which to make a decision as to which strategy to use. I believe to remember from Eichii Kono that correct proportions of “components” within an EA glyph may take priority over regular spacing of several strokes within one such “component.”

7. ResXIPDDDist, ResXIPDDLink, ResYIPDDDist, ResYIPDDLink

Implements a generalization of ResXIPDist … ResYIPLink to a diagonal stroke, like so:

ResXIPDDDist(grandParent0, parent0, child0, parent1, child1, grandParent1)

ResYIPDDDist(grandParent0, parent0, child0, parent1, child1, grandParent1)

ResXIPDDLink(grandParent0, parent0, child0, cvt0, parent1, child1, cvt1, grandParent1)

ResYIPDDLink(grandParent0, parent0, child0, cvt0, parent1, child1, cvt1, grandParent1)

Currently, only the children may override the freedom vector. It is assumed that the interpolation pretends to be done either in X or in Y, that as a result of the interpolation the parent points are placed, and that this is followed by placing the child points. In practice, this requires some thought as to the order in which commands are used, but typically is feasible. For instance, a pair of ResXIPDDLink can be used to control position and weight of the outer diagonal strokes on an uppercase “A”, even an italic one, by choosing the side-bearings to be the grand-parents, the “outer” corners to be the parents, and the “inner” corners of the strokes to be the children. Since the children can override the freedom vectors, this permits to solve the “inktrap” at the inside of the junction of the two diagonal strokes.

Note that the above assumes simple inktraps where two “inner” corners are one and the same point, such as in Verdana uppercase “A.” For inktraps using to separate control points, such as Corbel uppercase “A,” a pair of shift instructions can take care of the inktrap. At some point the auto-hinter will generate a piece of code like this:

ResXLink(2^3,15,55,>=)

XShift(15,8)

ResYLink(1^0,8:3>2,55,>=)

YShift(8,15:3>2)

Notice the pair of shifts; they treat the pair of control points 8 and 15 representing the inktrap as if they were a rigid unit. Whatever “happens” to 15 is duplicated on 8, and vice versa, probably corresponding to the original intent of the shift instruction.

8. ResIIPDDist, ResIIPDLink

Implements a generalization of ResXIPDist … ResYIPLink to an italic stroke, like so:

ResIIPDDist(grandParent0, parent0, child0, parent1, child1, grandParent1)

ResIIPDLink(grandParent0, parent0, child0, cvt0, parent1, child1, cvt1, grandParent1)

Currently, none of the control points may override the freedom or projection vector. The projection vector is perpendicular to the italic angle, and the children are assumed to have already been constrained in Y direction, hence their freedom vector can safely be chosen to be in X direction.

9. Smooth/

Smooth has been upgraded recently to accept the ‘/’ much like the existing link and interpolate commands can choose the vectors to follow the italic angle. Interpolation in Y translates to an IUP[Y], while for the IUP[X] part the compiler will emit repeated calls to support functions 134 and 135. These support functions essentially implement an interpolation of multiple children by specifying a pair of parents only, and assuming that any point inbetween is to be interpolated. One function implements the simple case where a contour’s start is not within that range, the other requires the start and end points to be specified. Both identify the potential of two control points having the same position on the pv, which unfortunately required some fixes in the rasterizer and fstrace, see “#ifndef VTT\_PRO\_SP\_YAA\_AUTO\_COM // BeatS BUG FIX” in interp.c and fstrace.c.