STSF User Guide

This guide is designed to introduce the STSF objects, and how and where they are used in order to create text output. The guide uses the XST Xserver extension API in it's examples.

STSF has six basic objects. Some or all of these objects will need to be set up before text can be drawn. The objects include:

XSTTypeEnv: This object is the root of all other objects and must be created before STSF can be used. The XSTTypeEnv object is responsible for maintaining the list of available fonts, drop-in objects such as scalers, and layout engines, font fallback policy, and global font fallbacks. Generally, only one XSTTypeEnv object will need to be required for the client and so it would normally be created during an initialization phase.

The following code snippet shows how to create an XSTTypeEnv object, and check that the connection to the STSF server was successful:

```
XSTTypeEnv env;
Display     *dpy;

env = XSTTypeEnvNew ( dpy );

if ( env == 0 ) {
        printf ( "Client could not connect to STSF\n" );
        exit ( 1 );
}
```

XSTText: This object maintains a copy of the clients text on the server. The object is responsible for maintaining text information such as the direction of flow, global metrics, local font fallbacks, justification, flush factor, and layout options. An XSTText object may represent a single line of text, all the way up to containing an entire document of text. As an example, a centered line of text will be required to be in a single XSTText object.

There are two ways to create an XSTText object. The first involves creating the object, and setting some of it's fields, while the second creates an empty object.

The following code snippet shows how to create an XSTText object, and how to manipulate some of its values:

```
XSTTypeEnv env;
XSTText text, text2;
char *str = some_string;
int str_len = XSTStringGetCharacters ( str, UTF7, strlen ( str ));
int oldstrlen = strlen ( str );
char *locale = setlocale ( NULL );
Display *dpy;

/*
 * Create text with string
 */
text = XSTTextNew ( dpy, env, (XSTString *)str, str_len, locale );

if ( text == 0 ) {
        printf ( "Text object failed to be created\n" );
        exit ( 1 );
}

/*
 * Create text, then set text
 */
text2 = XSTTextNewEmpty ( dpy, env );

if ( text2 == 0 ) {
        printf ( "Text object failed to be created\n" );
        exit ( 1 );
}

XSTTextSetText ( dpy, text2, (XSTString *)str, str_len, locale );

/*
 * Add text to end of string
 */
strcat ( str, some_additional_string );

added_len = XSTStringGetCharacters ( some_additonal_string,
                        UTF7, strlen ( some_additional_string ));

XSTTextUpdate ( dpy, text, ST_TC_INSERTED, str_len, added_len);
```

XSTLine: An XSTLine object defines a section of the XSTText object to be a single line of text. XSTLine objects are dependent on XSTText objects. This object maintains imposed metrics, caret information, and highlight data. Lines are the basic unit for rendering text.

Lines can be created in one of two manners. One method involves creating a line with a particular number of characters, while the other involves creating a line with a particular imposed length. The two lines creation methods can

be used together to create a line-breaking algorithm. The following is an example of using both methods for creating a simple line break algorithm:

```
Display *dpy;
XSTText text;
char *str = string_in_XSTText;
XSTLine line;
int start = 0, numchars = 0, chars_in_line;
double width = 8 * 72; /* 8 inches * 72 points per inch */

line = XSTLineNewForWidth ( dpy, text, start, width,
                                        &chars_in_line );

if ( line == 0 ) {
        printf ( "Line object was not successfully created\n" );
        exit ( 1 );
}

if ( str[chars_in_line] != ' ' ) { /* if we landed on a space, done */
        for ( i = chars_in_line - 1; i > 0; i-- )
                if ( str[i] == ' ' ) { /* find previous space */
                        numchars = i;
                        break;
                }

        if ( numchars != 0 ) { /* if space found, make new line */
                XSTLineDispose ( line );
                line = XSTLineNew ( dpy, text, start, numchars,
                                        &chars_in_line );
        }
}
```

XSTFontFamily/
XSTFont: XSTFontFamily and XSTFont values are contained in the XSTTypeEnv object. XSTFontFamilys contain a set of one or more XSTFonts. An example would be a FontFamily of Arial which contains Regular, Bold, Italic and Bold Italic Fonts. Fonts can be obtained either by first getting a FontFamily, then getting its associated Fonts, or by asking for the Fonts directly.

The following code block shows how to get all of the FontFamilies, a particular Family name, its subfonts, and their typeface names:

NOTE: This code also gets the width, weight, and style of the font which can be used provide additional information about the particular font to an end-user.

```
Display *dpy;
```

```
XSTTypeEnv env;
XSTFontFamily *ff;
XSTFont *fonts;
int num_ff, num_fonts, char_count;
utf8 *ff_name, *f_name;
char *locale = setlocale ( NULL );

/*
 * Set the language the font names are to be returned in.
 */
XSTLocale loc, gen_loc = XSTTypeEnvGetLocale ( env, locale );
loc = gen_loc;

ff = XSTTypeEnvFindAllFontFamilies ( dpy, env, &num_ff );

for ( i = 0; i < num_ff; i++ ) {
        if ( ff[i] == 0 ) { /* check for font family validity */
                printf ( "This font family is not valid\n" );
                exit ( 1 );
        }

        ff_name = (utf8 *) XSTFontFamilyGetUnicodeName ( dpy,
                        env, ff[i], UTF8, &loc, &char_count );
        printf ( "Family Name: %s\n", ff_name );

        fonts = XSTFontFamilyGetFonts ( dpy, env, ff[i],
                                                &num_fonts );

        for ( j = 0; j < num_fonts; j++ ) {
                XSTFontWeightClass weight;
                XSTFontWidthClass width;
                XSTFontStyleClass style;

                if ( fonts[j] == 0 ) {
                        printf ( "This font is not valid\n" );
                        exit ( 1 );
                }
                loc = gen_loc; /* reset the locale as it may change */

                f_name = (utf8 *)
                        XSTFontGetUnicodeTypefaceName ( dpy,
                                env, fonts[j], UTF8, &loc,
                                &char_count, &weight, &width,
                                &style );
                printf ( "Sub Font Name: %s\n", f_name );
        }
}
```

XSTStyle: XSTStyle objects cover the XSTText objects characters. There needs to be at least one XSTStyle object per XSTText object. These objects maintain information about the font to use, its size, any underlines, strikethroughs, scaler, layout engine, effects, baselines, and kerning options.

XSTStyle objects can be created either empty, or with some default values. The following example shows how to create a style, set some values, and assign it to cover a portion of an XSTText object:

```
Display *dpy;
XSTText text;
XSTStyle style;
int start = 0;
int length = 10;
XSTFont font;
double size = 12.0;

style = XSTStyleNewDefault ( dpy, env );

if ( style == 0 ) {
        printf ("Style failed on creation\n" );
        exit ( 1 );
}

XSTStyleSetFont ( dpy, style, ST_SM_FONT | ST_SM_SIZE, font,
                        size, 0, 0 );

XSTTextSetStyle ( dpy, text, style, start, length );
```

There are three different approaches that an application can use in setting styles. Take the following example where we have a section of text where part is represented with a different font, some is underlined, and some bold. The list of different styles required is shown as well.

```
|----------------------text----------------------|
           |---font 2--------------|
         |--------bold----------|                |--bold--|
                  |--underline-|

|-s1--|s2|-s3-|----s4----|--s3--|s5|---s1---|---s2---|s1-|
```

s1 = font1
s2 = font1, bold
s3 = font2, bold
s4 = font2, bold, underline
s5 = font2

Now suppose that we want to add an italic area of text. This would require making new styles:

```
|-s1--|s2|-s3-|----s4----|--s3--|s5|---s1---|---s2---|s1-|
                                  |------italic------|

|-s1--|s2|-s3-|----s4----|s3|-s6-|s5|---s8---|s7|--s2--|s1-|
```

s5 = font2, italic
s6 = font2, bold, italic
s7 = font1, bold, italic
s8 = font1, italic

The first, brute force method would be to create a unique style for all possible styles that can be used with the text. In this case, we would need 8 different styles as shown below:

st1 = font1	st2 = font2
st3 = font1, bold	st4 = font2, bold
st5 = font1, underline	st6 = font2, underline
st7 = font1, bold, underline	st8 = font2, bold, underline

Upon adding the italic, we would need to create the following additional 8 styles:

st9 = font1, italic	st10 = font2, italic
st11 = font1, bold, italic	st12 = font2, bold, italic
st13 = font1, underline, italic	st14 = font2, underline, italic
st15 = font1, bold, underline, italic	st16 = font2, bold, uline, italic

We could then assign the new s6, s7, and s8 to equal st12, st11, and st9

respectively. The existing s5 would change to equal st10. While this does allow simple setting of the styles within the text, and that no style is duplicated, the number of styles grows exponentially, and the likelyhood of using all or even a large portion of the allocated styles is minimal.

The second method would be to create only those styles necessary for the new italic to be put in place. To accomplish this, we need to take the following steps:

Since the italic splits s3, we need to copy s3 and create s6.
Since the italic splits s2, we need to copy s2 and create s7.
Since the italic fully covers s5, we need to change s5s contents.
Since the italic does not fully cover s1, we need to copy s1 and create s9.
Now we need to perform an XSTTextSetStyle for s2, s3, s5, s6, s7, and s8.

As you can see the list of steps necessary for this procedure can be complex as many sets of coverage must be compared, and many functions must be called to create new styles, change styles, and set their coverage regions. Additionally, we would need to check and see if any of the newly created styles, s6, s7, and s8 already existed. For instance, had the italic style been another bold style, s5, s6, s7 and s8 would all be the same as s2.

The third method would be to relinquish control to STSF. To do this, create an empty style and set the italic coverage, then use the following command to have STSF create all of the necessary styles internally:

XSTTextOverwriteStyle (dpy, text, style, start, length);

GC: The GC holds many of the graphics functions that are associated with STSF. Through the GC, the client application has access to the transformation matrix, the output type, colors, alpha range, and output mode.

The following code snippets show how to set the output type and mode, set alpha ranges, and perform transformations:

```
Display *dpy;
GC gc;
XSTAlphaStruct alpha;
XSTMatrix mat;

XSTGCSetOutputFormat ( dpy, gc, ST_DEVICE_RASTER,
                ST_OM_RASTER_MONO |
                ST_OF_SUPPRESS_FRACTIONAL_METRICS );

alpha.min_alpha = 0.0;
alpha.max_alpha = 0.5; /* 50% transparent */

/*
 * Change the highlight alpha to be semi transparent.
 */
XSTGCSetAlphaRange ( dpy, gc, mHighlightAlpha, NULL,
                                &alpha, NULL, NULL );

mat.xx = 1.0;
mat.xy = 0.0;
mat.yx = 0.0;
mat.yy = 1.0;
mat.tx = tx;
mat.ty = ty;

XSTGCSetMatrix ( dpy, gc, &mat );
if ( shear_x != 0.0 || shear_y != 0.0 )
        XSTGCShear ( dpy, gc, shear_x, shear_y );
if ( rotate_theta != 0.0 )
        XSTGCRotate ( dpy, gc, rotate_theta );
if ( scale_x != 1.0 || scale_y != 1.0 )
        XSTGCScale ( dpy, gc, scale_x, scale_y );
```

Now that we have covered the basic objects, and how to use some of the features of each, we will look at some of the procedures to display text on the screen. The following code shows a sample loop for displaying the contents of an XSTText in anti-aliased mode:

```
Display *dpy;
Drawable d;
GC gc;
XSTLine *lines;
int num_lines;

XSTGCSetOutputFormat ( dpy, gc, ST_DEVICE_RASTER,
                                ST_OM_RASTER_GRAYSCALE );

for ( i = 0; i < num_lines; i++ ) {
        XSTTrapezoid trap;
        XSTRectangle rect;
        double space;

        trap = XSTLineMeasureText ( dpy, gc, lines[i] );
        rect = XSTConvertGetBoundingRectangle ( &trap );
        free ( trap );
        met = XSTStyleGetDesignMetrics ( dpy, style );
        space = rect->by - rect->ay + met->leading;
        free ( rect );
        XSTLineRender ( dpy, d, gc, lines[i] );
        XSTGCTranslate ( dpy, gc, 0, space );
}
```

XSTGlyphVector:

The glyph vector is an array of positioned glyphs that the application sends to STSF for displaying. Use of this object within the STSF framework requires some form of layout software on the client side. The purpose of this object is to allow applications to utilize standalone layout engines such as Pango that are not available internally to the STSF architecture, or as a plug-in to the framework. This object is very fast as no layout is required within STSF, and only the rendering pipeline is used. Styled Glyphs are associated with an XSTStyle object. In this case, the glyph vector rendering code ignores underlines, highlights, and strikethroughs. These must be drawn by the client application.

The following code snippet shows how to use the Glyph Vector code.

```
Display *dpy;
Drawable d;
GC gc;
XSTGlyphVector vector;
XSTStyledGlyph *glyphs;
int num_glyphs, line_start[], line_length[];
double X_position_of_glyph[], Y_position_of_glyph[];
STGlyph glyph_id[];
XSTStyle default_XSTstyle;

/*
 * Create the Styled glyphs. These are glyph id and position, along with
 * the associated XSTStyle object.
 */
for ( i = 0; i < num_glyphs; i++ )
{
        glyphs[i].glyph = glyph_id[i];
        glyphs[i].style = default_XSTstyle;
        glyphs[i].pos.x = X_position_of_glyph[i];
        glyphs[i].pos.y = Y_position_of_glyph[i];
}

vector = XSTGlyphVectorNew ( dpy, env, glyphs, count );

 if ( vector == 0 ) {
        printf ("GlyphVector failed on creation\n" );
        exit ( 1 );
}

/*
 * Draw the glyph vector in terms of individual lines.
 */
for ( i = 0; i < numlines; i++ )
{
        XSTGlyphVectorRender ( dpy, d, gc, vector, line_start[i],
                                        line_length[i] );
}
```