SDL_ttf

2 May 2005

Jonathan Atkins

Table of Contents

1 Overview

A Little Bit About Me

I am currently, as I write this document, a programmer for Raytheon. There I do all sorts of communications, network, GUI, and other general programming tasks in C/C++ on the Solaris and sometimes Linux Operating Systems. I've used SDL_ttf as one of the many methods of putting text on my SDL applications, and use it in my own SDL GUI code as well. While this document doesn't explain how and where to get fonts to use, it will explain how to use them with SDL_ttf.

Feel free to contact me: `jcatki@jcatki.no-ip.org`

The latest version of this library is available from:
SDL_ttf Homepage

I am also usually on IRC at irc.freenode.net in the #SDL channel as LIM

This is the README in the SDL_ttf source archive.

This library is a wrapper around the excellent FreeType 1.2 library, available at: Freetype Homepage

WARNING: There may be patent issues with using the FreeType library. Check the FreeType website for up-to-date details.

This library allows you to use TrueType fonts to render text in SDL applications.

To make the library, first install the FreeType library, then type 'make' to build the SDL truetype library and 'make all' to build the demo application.

Be careful when including fonts with your application, as many of them are copyrighted. The Microsoft fonts, for example, are not freely redistributable and even the free "web" fonts they provide are only redistributable in their special executable installer form (May 1998). There are plenty of freeware and shareware fonts available on the Internet though, and may suit your purposes.

Please see the file "COPYING" for license information for this library.

Enjoy! -Sam Lantinga `slouken@devolution.com` (5/1/98)

2 Getting Started

This assumes you have gotten SDL_ttf and installed it on your system. SDL_ttf has an INSTALL document in the source distribution to help you get it compiled and installed.

Generally, installation consists of:

```
./configure
make
make install
```

SDL_ttf supports loading fonts from TrueType font files, normally ending in .ttf, though some .fon files are also valid for use. Note that most fonts are copyrighted, check the license on the font before you use and redistribute.

Some free font sources are:

- Free UCS Outline Fonts
- Fonthead Design
- Bitstream Vera Fonts
- FreeUniFont
- 1001 Fonts
- Google!

You may also want to look at some demonstration code which may be downloaded from:
`http://jcatki.no-ip.org/SDL_ttf/`

2.1 Includes

To use SDL_ttf functions in a C/C++ source code file, you must use the SDL_ttf.h include file:

```
#include "SDL_ttf.h"
```

2.2 Compiling

To link with SDL_ttf you should use sdl-config to get the required SDL compilation options. After that, compiling with SDL_ttf is quite easy.
Note: Some systems may not have the SDL_ttf library and include file in the same place as the SDL library and includes are located, in that case you will need to add more -I and -L paths to these command lines.

Simple Example for compiling an object file:

```
cc -c `sdl-config --cflags` mysource.c
```

Simple Example for compiling an object file:

```
cc -o myprogram mysource.o `sdl-config --libs` -lSDL_ttf
```

Now `myprogram` is ready to run.

3 Functions

These are the functions in the SDL_ttf API.

3.1 General

These functions are useful, as they are the only/best ways to work with SDL_ttf.

3.1.1 TTF_Linked_Version

`const SDL_version *`**TTF_Linked_Version**`()`
`void `**TTF_VERSION**`(SDL_version *compile_version)`

This works similar to `SDL_Linked_Version` and **SDL_VERSION**.
Using these you can compare the runtime version to the version that you compiled with.

```
SDL_version compile_version, *link_version;
TTF_VERSION(&compile_version);
printf("compiled with SDL_ttf version: %d.%d.%d\n",
        compile_version.major,
        compile_version.minor,
        compile_version.patch);
link_version=TTF_Linked_Version();
printf("running with SDL_ttf version: %d.%d.%d\n",
        link_version->major,
        link_version->minor,
        link_version->patch);
```

See Also:
Section 3.1.2 [TTF_Init], page 9

3.1.2 TTF_Init

`int` **TTF_Init**`()`

Initialize the truetype font API.
This must be called before using other functions in this library, excepting `TTF_WasInit`.
SDL does not have to be initialized before this call.

Returns: 0 on success, -1 on errors

```
if(TTF_Init()==-1) {
    printf("TTF_Init: %s\n", TTF_GetError());
    exit(2);
}
```

See Also:
Section 3.1.4 [TTF_Quit], page 11, Section 3.1.3 [TTF_WasInit], page 10

3.1.3 TTF_WasInit

`int` **TTF_WasInit**`()`

Query the initilization status of the truetype font API.
You may, of course, use this before `TTF_Init` to avoid initilizing twice in a row. Or use this to determine if you need to call `TTF_Quit`.

```
if(!TTF_WasInit() && TTF_Init()==-1) {
    printf("TTF_Init: %s\n", TTF_GetError());
    exit(1);
}
```

See Also:
Section 3.1.2 [TTF_Init], page 9, Section 3.1.4 [TTF_Quit], page 11

3.1.4 TTF_Quit

`void` **TTF_Quit**`()`

Shutdown and cleanup the truetype font API.
After calling this the SDL_ttf functions should not be used, excepting `TTF_WasInit`. You may, of course, use `TTF_Init` to use the functionality again.

```
TTF_Quit();
// you could SDL_Quit(); here...or not.
```

See Also:
Section 3.1.2 [TTF_Init], page 9, Section 3.1.3 [TTF_WasInit], page 10

3.1.5 TTF_SetError

`void` **TTF_SetError**`(const char *`*fmt*`, ...)`

This is the same as `SDL_SetError`, which sets the error string which may be fetched with `TTF_GetError` (or `SDL_GetError`). This functions acts like printf, except that it is limited to **SDL_ERRBUFIZE**(1024) chars in length. It only accepts the following format types: `%s, %d, %f, %p`. No variations are supported, like `%.2f` would not work. For any more specifics read the SDL docs.

```
int myfunc(int i) {
    TTF_SetError("myfunc is not implemented! %d was passed in.",i);
    return(-1);
}
```

See Also:
Section 3.1.6 [TTF_GetError], page 13

3.1.6 TTF_GetError

`char *`**TTF_GetError**`()`

This is the same as `SDL_GetError`, which returns the last error set as a string which you may use to tell the user what happened when an error status has been returned from an SDL_ttf function call.

Returns: a char pointer (string) containing a human readable version or the reason for the last error that occured.

```
printf("Oh My Goodness, an error : %s", TTF_GetError());
```

See Also:
Section 3.1.5 [TTF_SetError], page 12

3.2 Management

These functions deal with loading and freeing a `TTF_Font`.

3.2.1 TTF_OpenFont

`TTF_Font *`**`TTF_OpenFont`**`(const char *`*`file`*`, int `*`ptsize`*`)`

file	File name to load font from.
ptsize	Point size (based on 72DPI) to load font as. This basically translates to pixel height.

Load *file* for use as a font, at *ptsize* size. This is actually `TTF_OpenFontIndex(`*`file`*`, `*`ptsize`*`, 0)`. This can load TTF and FON files.

Returns: a pointer to the font as a `TTF_Font`. **NULL** is returned on errors.

```
// load font.ttf at size 16 into font
TTF_Font *font;
font=TTF_OpenFont("font.ttf", 16);
if(!font) {
    printf("TTF_OpenFont: %s\n", TTF_GetError());
    // handle error
}
```

See Also:
Section 3.2.3 [TTF_OpenFontIndex], page 17, Section 3.2.2 [TTF_OpenFontRW], page 16, Section 3.2.5 [TTF_CloseFont], page 19

3.2.2 TTF_OpenFontRW

`TTF_Font *`**TTF_OpenFontRW**`(SDL_RWops *`*`src`*`, int `*`freesrc`*`, int `*`ptsize`*`)`

src	The source SDL_RWops as a pointer. The font is loaded from this.
freesrc	A non-zero value mean is will automatically close/free the *src* for you.
ptsize	Point size (based on 72DPI) to load font as. This basically translates to pixel height.

Load *src* for use as a font, at *ptsize* size. This is actually `TTF_OpenFontIndexRW(`*`src`*`, `*`freesrc`*`, `*`ptsize`*`, 0)` This can load TTF and FON formats. Using `SDL_RWops` is not covered here, but they enable you to load from almost any source.

NOTE: *src* is not checked for **NULL**, so be careful.

Returns: a pointer to the font as a `TTF_Font`. **NULL** is returned on errors.

```
// load font.ttf at size 16 into font
TTF_Font *font;
font=TTF_OpenFontRW(SDL_RWFromFile("font.ttf"), 1, 16);
if(!font) {
    printf("TTF_OpenFontRW: %s\n", TTF_GetError());
    // handle error
}
```

Note that this is unsafe because we don't check the validity of the `SDL_RWFromFile`'s returned pointer.

See Also:
Section 3.2.4 [TTF_OpenFontIndexRW], page 18, Section 3.2.1 [TTF_OpenFont], page 15, Section 3.2.5 [TTF_CloseFont], page 19

3.2.3 TTF_OpenFontIndex

`TTF_Font *`**TTF_OpenFontIndex**`(const char *`*`file`*`, int `*`ptsize`*`, long `*`index`*`)`

file	File name to load font from.
ptsize	Point size (based on 72DPI) to load font as. This basically translates to pixel height.
index	choose a font face from a multiple font face containing file. The first face is always index 0.

Load *file*, face *index*, for use as a font, at *ptsize* size. This is actually `TTF_OpenFontIndexRW(SDL_RWFromFile(`*`file`*`), `*`ptsize`*`, `*`index`*`)`, but checks that the RWops it creates is not **NULL**. This can load TTF and FON files.

Returns: a pointer to the font as a `TTF_Font`. **NULL** is returned on errors.

```
// load font.ttf, face 0, at size 16 into font
TTF_Font *font;
font=TTF_OpenFontIndex("font.ttf", 16, 0);
if(!font) {
    printf("TTF_OpenFontIndex: %s\n", TTF_GetError());
    // handle error
}
```

See Also:
Section 3.2.4 [TTF_OpenFontIndexRW], page 18, Section 3.2.1 [TTF_OpenFont], page 15, Section 3.2.5 [TTF_CloseFont], page 19

3.2.4 TTF_OpenFontIndexRW

`TTF_Font` ***TTF_OpenFontIndexRW***`(SDL_RWops *`*`src`*`, int `*`freesrc`*`, int `*`ptsize`*`, long `*`index`*`)`

src	The source SDL_RWops as a pointer. The font is loaded from this.
freesrc	A non-zero value mean is will automatically close/free the *src* for you.
ptsize	Point size (based on 72DPI) to load font as. This basically translates to pixel height.
index	choose a font face from a multiple font face containing file. The first face is always index 0.

Load *src*, face *index*, for use as a font, at *ptsize* size. This can load TTF and FON formats. Using `SDL_RWops` is not covered here, but they enable you to load from almost any source.

NOTE: *src* is not checked for **NULL**, so be careful.

Returns: a pointer to the font as a `TTF_Font`. **NULL** is returned on errors.

```
// load font.ttf, face 0, at size 16 into font
TTF_Font *font;
font=TTF_OpenFontRW(SDL_RWFromFile("font.ttf"), 1, 16, 0);
if(!font) {
    printf("TTF_OpenFontIndexRW: %s\n", TTF_GetError());
    // handle error
}
```

Note that this is unsafe because we don't check the validity of the `SDL_RWFromFile`'s returned pointer.

See Also:
Section 3.2.3 [TTF_OpenFontIndex], page 17, Section 3.2.2 [TTF_OpenFontRW], page 16, Section 3.2.5 [TTF_CloseFont], page 19

3.2.5 TTF_CloseFont

`void` **TTF_CloseFont**`(TTF_Font *`*font*`)`

font Pointer to the TTF_Font to free.

Free the memory used by *font*, and free *font* itself as well. Do not use *font* after this without loading a new font to it.

```
// free the font
// TTF_Font *font;
TTF_CloseFont(font);
font=NULL; // to be safe...
```

See Also:
Section 3.2.1 [TTF_OpenFont], page 15, Section 3.2.2 [TTF_OpenFontRW], page 16, Section 3.2.3 [TTF_OpenFontIndex], page 17, Section 3.2.4 [TTF_OpenFontIndexRW], page 18

3.3 Attributes

These functions deal with `TTF_Font`, and global, attributes.

See the end of Section 3.3.12 [TTF_GlyphMetrics], page 32 for info on how the metrics work.

3.3.1 TTF_ByteSwappedUNICODE

`void` **TTF_ByteSwappedUNICODE**`(int` *`swapped`*`)`

swapped	if non-zero then UNICODE data is byte swapped relative to the CPU's native endianess. if zero, then do not swap UNICODE data, use the CPU's native endianess.

This function tells SDL_ttf whether UNICODE (Uint16 per character) text is generally byteswapped. A **UNICODE_BOM_NATIVE** or **UNICODE_BOM_SWAPPED** character in a string will temporarily override this setting for the remainder of that string, however this setting will be restored for the next one. The default mode is non-swapped, native endianess of the CPU.

```
// Turn on byte swapping for UNICODE text
TTF_ByteSwappedUNICODE(1);
```

See Also:
Chapter 5 [Defines], page 53

3.3.2 TTF_GetFontStyle

`int` **TTF_GetFontStyle**`(TTF_Font *`*`font`*`)`

font	The loaded font to get the style of

Get the rendering style of the loaded *font*.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: The style as a bitmask composed of the following masks:
TTF_STYLE_BOLD
TTF_STYLE_ITALIC
TTF_STYLE_UNDERLINE
If no style is set then **TTF_STYLE_NORMAL** is returned.

```
// get the loaded font's style
//TTF_Font *font;
int style;
style=TTF_GetFontStyle(font);
printf("The font style is:");
if(style==TTF_STYLE_NORMAL)
    printf(" normal");
else {
    if(style&TTF_STYLE_BOLD)
        printf(" bold");
    if(style&TTF_STYLE_ITALIC)
        printf(" italic");
    if(style&TTF_STYLE_UNDERLINE)
        printf(" underline");
}
printf("\n");
```

See Also:
Section 3.3.3 [TTF_SetFontStyle], page 23,
Chapter 5 [Defines], page 53

3.3.3 TTF_SetFontStyle

`void` **TTF_SetFontStyle**`(TTF_Font *`*`font`*`, int `*`style`*`)`

font	The loaded font to get the style of
style	A bitmask of the desired style composed from the **TTF_STYLE_*** defined values

Set the rendering style of the loaded *font*.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

NOTE: This will flush the internal cache of previously rendered glyphs, even if there is no change in style, so it may be best to check the current style using *TTF_GetFontStyle* first.

NOTE: I've seen that combining TTF_STYLE_UNDERLINE with anything can cause a segfault, other combinations may also do this. Some brave soul may find the cause of this and fix it...

```
// set the loaded font's style to bold italics
//TTF_Font *font;
TTF_SetFontStyle(font, TTF_STYLE_BOLD|TTF_STYLE_ITALIC);

// render some text in bold italics...

// set the loaded font's style back to normal
TTF_SetFontStyle(font, TTF_STYLE_NORMAL);
```

See Also:
Section 3.3.2 [TTF_GetFontStyle], page 22,
Chapter 5 [Defines], page 53

3.3.4 TTF_FontHeight

`int` **TTF_FontHeight**`(TTF_Font *`*`font`*`)`

font	The loaded font to get the max height of

Get the maximum pixel height of all glyphs of the loaded *font*. You may use this height for rendering text as close together vertically as possible, though adding at least one pixel height to it will space it so they can't touch. Remember that SDL_ttf doesn't handle multiline printing, so you are responsible for line spacing, see the *TTF_FontLineSkip* as well.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: The maximum pixel height of all glyphs in the font.

```
// get the loaded font's max height
//TTF_Font *font;
printf("The font max height is: %d\n", TTF_FontHeight(font));
```

See Also:
Section 3.3.5 [TTF_FontAscent], page 25,
Section 3.3.6 [TTF_FontDescent], page 26,
Section 3.3.7 [TTF_FontLineSkip], page 27,
Section 3.3.12 [TTF_GlyphMetrics], page 32

3.3.5 TTF_FontAscent

`int` **TTF_FontAscent** `(TTF_Font *font)`

font The loaded font to get the ascent (height above baseline) of

Get the maximum pixel ascent of all glyphs of the loaded *font*. This can also be interpreted as the distance from the top of the font to the baseline.
It could be used when drawing an individual glyph relative to a top point, by combining it with the glyph's maxy metric to resolve the top of the rectangle used when blitting the glyph on the screen.

```
rect.y = top + TTF_FontAscent(font) - glyph_metric.maxy;
```

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: The maximum pixel ascent of all glyphs in the font.

```
// get the loaded font's max ascent
//TTF_Font *font;

printf("The font ascent is: %d\n", TTF_FontAscent(font));
```

See Also:
Section 3.3.4 [TTF_FontHeight], page 24,
Section 3.3.6 [TTF_FontDescent], page 26,
Section 3.3.7 [TTF_FontLineSkip], page 27,
Section 3.3.12 [TTF_GlyphMetrics], page 32

3.3.6 TTF_FontDescent

`int` **TTF_FontDescent** `(TTF_Font *font)`

font The loaded font to get the descent (height below baseline) of

Get the maximum pixel descent of all glyphs of the loaded *font*. This can also be interpreted as the distance from the baseline to the bottom of the font.
It could be used when drawing an individual glyph relative to a bottom point, by combining it with the glyph's maxy metric to resolve the top of the rectangle used when blitting the glyph on the screen.

```
rect.y = bottom - TTF_FontDescent(font) - glyph_metric.maxy;
```

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: The maximum pixel height of all glyphs in the font.

```
// get the loaded font's max descent
//TTF_Font *font;

printf("The font descent is: %d\n", TTF_FontDescent(font));
```

See Also:
Section 3.3.4 [TTF_FontHeight], page 24,
Section 3.3.5 [TTF_FontAscent], page 25,
Section 3.3.7 [TTF_FontLineSkip], page 27,
Section 3.3.12 [TTF_GlyphMetrics], page 32

3.3.7 TTF_FontLineSkip

`int` **TTF_FontLineSkip**`(TTF_Font *font)`

font	The loaded font to get the line skip height of

Get the reccomended pixel height of a rendered line of text of the loaded *font*. This is usually larger than the *TTF_FontHeight* of the *font*.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: The maximum pixel height of all glyphs in the font.

```
// get the loaded font's max descent
//TTF_Font *font;

printf("The font descent is: %d\n", TTF_FontDescent(font));
```

See Also:
Section 3.3.4 [TTF_FontHeight], page 24,
Section 3.3.5 [TTF_FontAscent], page 25,
Section 3.3.6 [TTF_FontDescent], page 26,
Section 3.3.12 [TTF_GlyphMetrics], page 32

3.3.8 TTF_FontFaces

long **TTF_FontFaces**(TTF_Font **font*)

font	The loaded font to get the number of available faces from.

Get the number of faces ("sub-fonts") available in the loaded *font*. This is a count of the number of specific fonts (based on size and style and ???) contained in the font itself. It seems to be a useless fact to know, since it can't be applied in any other SDL_ttf functions.
NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: The number of faces in the *font*.

```
// get the loaded font's number of faces
//TTF_Font *font;

printf("The number of faces in the font is: %ld\n", TTF_FontFaces(font));
```

See Also:
Section 3.3.9 [TTF_FontFaceIsFixedWidth], page 29,
Section 3.3.10 [TTF_FontFaceFamilyName], page 30,
Section 3.3.11 [TTF_FontFaceStyleName], page 31

3.3.9 TTF_FontFaceIsFixedWidth

`int` **TTF_FontFaceIsFixedWidth**`(TTF_Font *`*font*`)`

font The loaded font to get the fixed width status of.

Test if the current font face of the loaded *font* is a fixed width font. Fixed width fonts are monospace, meaning every character that exists in the font is the same width, thus you can assume that a rendered string's width is going to be the result of a simple calculation:
`glyph_width * string_length`
NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: >0 if *font* is a fixed width font. 0 if not a fixed width font.

```
// get the loaded font's face fixed status
//TTF_Font *font;

if(TTF_FontFaceIsFixedWidth(font))
    printf("The font is fixed width.\n");
else
    printf("The font is not fixed width.\n");
```

See Also:
Section 3.3.8 [TTF_FontFaces], page 28,
Section 3.3.10 [TTF_FontFaceFamilyName], page 30,
Section 3.3.11 [TTF_FontFaceStyleName], page 31,
Section 3.3.12 [TTF_GlyphMetrics], page 32

3.3.10 TTF_FontFaceFamilyName

`char *` **TTF_FontFaceFamilyName**`(TTF_Font *`*font*`)`

font The loaded font to get the current face family name of.

Get the current font face family name from the loaded *font*. This function may return a **NULL** pointer, in which case the information is not available.
NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: The current family name of of the face of the *font*, or **NULL** perhaps.

```
// get the loaded font's face name
//TTF_Font *font;

char *familyname=TTF_FontFaceFamilyName(font);
if(familyname)
    printf("The family name of the face in the font is: %s\n", familyname);
```

See Also:
Section 3.3.8 [TTF_FontFaces], page 28,
Section 3.3.9 [TTF_FontFaceIsFixedWidth], page 29,
Section 3.3.11 [TTF_FontFaceStyleName], page 31

3.3.11 TTF_FontFaceStyleName

`char * `**TTF_FontFaceStyleName**`(TTF_Font *`*font*`)`

font	The loaded font to get the current face style name of.

Get the current font face style name from the loaded *font*. This function may return a **NULL** pointer, in which case the information is not available.
NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: The current style name of of the face of the *font*, or **NULL** perhaps.

```
// get the loaded font's face style name
//TTF_Font *font;

char *stylename=TTF_FontFaceStyleName(font);
if(stylename)
    printf("The name of the face in the font is: %s\n", stylename);
```

See Also:
Section 3.3.8 [TTF_FontFaces], page 28,
Section 3.3.9 [TTF_FontFaceIsFixedWidth], page 29,
Section 3.3.10 [TTF_FontFaceFamilyName], page 30

3.3.12 TTF_GlyphMetrics

`int` **TTF_GlyphMetrics**`(TTF_Font *`*font*`, Uint16` *ch*`, int *`*minx*`, int *`*maxx*`, int *`*miny*`, int *`*maxy*`, int *`*advance*`)`

font The loaded font from which to get the glyph metrics of *ch* from

ch the UNICODE char of which to get the glyph metrics for

minx poiter to int to store the retuned minimum X offset into, or **NULL** when no return value desired.

maxx poiter to int to store the retuned maximum X offset into, or **NULL** when no return value desired.

miny poiter to int to store the retuned minimum Y offset into, or **NULL** when no return value desired.

maxy poiter to int to store the retuned maximum Y offset into, or **NULL** when no return value desired.

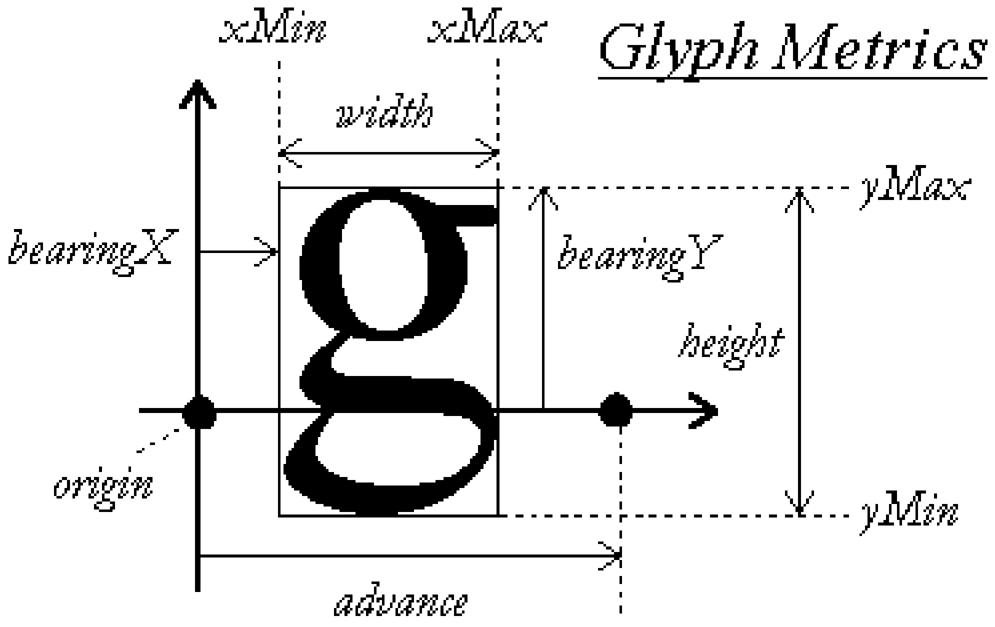
advance poiter to int to store the retuned advance offset into, or **NULL** when no return value desired.

Get desired glyph metrics of the UNICODE chargiven in *ch* from the loaded *font*.
NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: 0 on success, with all non-**NULL** parameters set to the glyph metric as appropriate. -1 on errors, such as when the glyph named by *ch* does not exist in the font.

```
// get the glyph metric for the letter 'g' in a loaded font
//TTF_Font *font;
int minx,maxx,miny,maxy,advance;
if(TTF_GlyphMetrics(font,'g',&minx,&maxx,&miny,&maxy,&advance)==-1)
    printf("%s\n",TTF_GetError());
else {
    printf("minx    : %d\n",minx);
    printf("maxx    : %d\n",maxx);
    printf("miny    : %d\n",miny);
    printf("maxy    : %d\n",maxy);
    printf("advance : %d\n",advance);
}
```

This digram shows the relationships between the values:



Here's how the numbers look:

```
TTF_FontHeight       : 33
TTF_FontAscent       : 26
TTF_FontDescent      : -6
TTF_FontLineSkip     : 33
TTF_GlyphMetrics('g'):
        minx=0
        maxx=21
        miny=0
        maxy=21
        advance=24
```

We see from the Line Skip that each line of text is 33 pixels high, including spacing.
The Ascent-Descent=32, so there seems to be one pixel worth of space minimum between lines.

Let's say we want to draw the surface of glyph 'g' (retrived via Section 3.4.4 [TTF_RenderGlyph_Solid], page 42 or a similar function), at coordinates (X,Y) for the top left corner of the desired location. Here's the math using glyph metrics:

```
//SDL_Surface *glyph,*screen;
SDL_Rect rect;
int minx,miny,advance;
TTF_GlyphMetrics(font,'g',&minx,NULL,&miny,NULL,&advance);
rect.x=X+minx;
rect.y=Y+miny;
SDL_BlitSurface(glyph,NULL,screen,&rect);
X+=advance;
```

Let's say we want to draw the same glyph at coordinates (X,Y) for the origin (on the baseline) of the desired location. Here's the math using glyph metrics:

```
//TTF_Font *font;
//SDL_Surface *glyph,*screen;
SDL_Rect rect;
int minx,miny,advance;
TTF_GlyphMetrics(font,'g',&minx,NULL,&miny,NULL,&advance);
rect.x=X+minx;
rect.y=Y-TTF_FontAscent(font)+miny;
SDL_BlitSurface(glyph,NULL,screen,&rect);
X+=advance;
```

NOTE: These examples assume that 'g' *is* present in the font!

See the web page at The FreeType2 Documentation Tutorial for more.

Any glyph based rendering calculations will not result in accurate kerning between adjacent glyphs. (see Chapter 6 [Glossary], page 54)

See Also:
Section 3.3.4 [TTF_FontHeight], page 24,
Section 3.3.5 [TTF_FontAscent], page 25,
Section 3.3.6 [TTF_FontDescent], page 26,
Section 3.3.7 [TTF_FontLineSkip], page 27,
Section 3.3.13 [TTF_SizeText], page 35,
Section 3.3.14 [TTF_SizeUTF8], page 36,
Section 3.3.15 [TTF_SizeUNICODE], page 37,

3.3.13 TTF_SizeText

`int` **TTF_SizeText** `(TTF_Font *`*font*`, const char *`*text*`, int *`*w*`, int *`*h*`)`

font	The loaded font to use to calculate the size of the string with.
text	The LATIN1 null terminated string to size up.
w	pointer to int in which to fill the text width, or **NULL** for no desired return value.
h	pointer to int in which to fill the text height, or **NULL** for no desired return value.

Calculate the resulting surface size of the LATIN1 encoded *text* rendered using *font*. No actual rendering it done, however correct kerning is done to getthe actual width. The height returned in *w* is the same as you can get using Section 3.3.4 [TTF_FontHeight], page 24.
NOTE: Passing a **NULL** *font* into this function will cause a segfault. **NOTE**: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: 0 on success with the ints pointed to by *w* and *h* set as appropriate, if they are not **NULL**. -1 is returned on errors, such as a glyph in the string not being found.

```
// get the width and height of a string as it would be rendered in a loaded font
//TTF_Font *font;
int w,h;
if(TTF_SizeText(font,"Hello World!",&w,&h)) {
    // perhaps print the current TTF_GetError(), the string can’t be rendered...
} else {
    printf("width=%d height=%d\n",w,h);
}
```

See Also:
Section 3.3.14 [TTF_SizeUTF8], page 36,
Section 3.3.15 [TTF_SizeUNICODE], page 37,
Section 3.4.1 [TTF_RenderText_Solid], page 39,
Section 3.4.5 [TTF_RenderText_Shaded], page 43,
Section 3.4.9 [TTF_RenderText_Blended], page 47

3.3.14 TTF_SizeUTF8

`int` **TTF_SizeUTF8**`(TTF_Font *`*font*`, const char *`*text*`, int *`*w*`, int *`*h*`)`

font	The loaded font to use to calculate the size of the string with.
text	The UTF8 null terminated string to size up.
w	pointer to int in which to fill the text width, or **NULL** for no desired return value.
h	pointer to int in which to fill the text height, or **NULL** for no desired return value.

Calculate the resulting surface size of the UTF8 encoded *text* rendered using *font*. No actual rendering it done, however correct kerning is done to getthe actual width. The height returned in *w* is the same as you can get using Section 3.3.4 [TTF_FontHeight], page 24.
NOTE: Passing a **NULL** *font* into this function will cause a segfault. **NOTE**: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: 0 on success with the ints pointed to by *w* and *h* set as appropriate, if they are not **NULL**. -1 is returned on errors, such as a glyph in the string not being found.

Note that this example uses the same text as in the LATIN1 example, that is because plain ASCII is UTF8 compatible.

```
// get the width and height of a string as it would be rendered in a loaded font
//TTF_Font *font;
int w,h;
if(TTF_SizeUTF8(font,"Hello World!",&w,&h)) {
    // perhaps print the current TTF_GetError(), the string can't be rendered...
} else {
    printf("width=%d height=%d\n",w,h);
}
```

See Also:
Section 3.3.13 [TTF_SizeText], page 35,
Section 3.3.15 [TTF_SizeUNICODE], page 37,
Section 3.4.2 [TTF_RenderUTF8_Solid], page 40,
Section 3.4.6 [TTF_RenderUTF8_Shaded], page 44,
Section 3.4.10 [TTF_RenderUTF8_Blended], page 48

3.3.15 TTF_SizeUNICODE

`int` **TTF_SizeUNICODE**`(TTF_Font *`*font*`, const Unit16 *`*text*`, int *`*w*`, int *`*h*`)`

font	The loaded font to use to calculate the size of the string with.
text	The UNICODE null terminated string to size up.
w	pointer to int in which to fill the text width, or **NULL** for no desired return value.
h	pointer to int in which to fill the text height, or **NULL** for no desired return value.

Calculate the resulting surface size of the UNICODE encoded *text* rendered using *font*. No actual rendering it done, however correct kerning is done to getthe actual width. The height returned in *w* is the same as you can get using Section 3.3.4 [TTF_FontHeight], page 24.
NOTE: Passing a **NULL** *font* into this function will cause a segfault. **NOTE**: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: 0 on success with the ints pointed to by *w* and *h* set as appropriate, if they are not **NULL**. -1 is returned on errors, such as a glyph in the string not being found.

```
// get the width and height of a string as it would be rendered in a loaded font
//TTF_Font *font;
int w,h;
Uint16 text[]={'H','e','l','l','o',' ',
               'W','o','r','l','d','!'};
if(TTF_SizeUNICODE(font,text,&w,&h)) {
    // perhaps print the current TTF_GetError(), the string can't be rendered...
} else {
    printf("width=%d height=%d\n",w,h);
}
```

See Also:
Section 3.3.13 [TTF_SizeText], page 35,
Section 3.3.14 [TTF_SizeUTF8], page 36,
Section 3.4.3 [TTF_RenderUNICODE_Solid], page 41,
Section 3.4.7 [TTF_RenderUNICODE_Shaded], page 45,
Section 3.4.11 [TTF_RenderUNICODE_Blended], page 49

3.4 Render

These functions render text using a `TTF_Font`.
There are three modes of rendering:

Solid **Quick and Dirty**
Create an 8-bit palettized surface and render the given text at fast quality with the given font and color. The 0 pixel value is the colorkey, giving a transparent background, and the 1 pixel value is set to the text color. The colormap is set to have the desired foreground color at index 1, this allows you to change the color without having to render the text again. Colormap index 0 is of course not drawn, since it is the colorkey, and thus transparent, though it's actual color is 255 minus each RGB component of the foreground. This is the fastest rendering speed of all the rendering modes. This results in no box around the text, but the text is not as smooth. The resulting surface should blit faster than the Blended one. Use this mode for FPS and other fast changing updating text displays.

Shaded **Slow and Nice, but with a Solid Box**
Create an 8-bit palettized surface and render the given text at high quality with the given font and colors. The 0 pixel value is background, while other pixels have varying degrees of the foreground color from the background color. This results in a box of the background color around the text in the foreground color. The text is antialiased. This will render slower than Solid, but in about the same time as Blended mode. The resulting surface should blit as fast as Solid, once it is made. Use this when you need nice text, and can live with a box...

Blended **Slow Slow Slow, but Ultra Nice over another image**
Create a 32-bit ARGB surface and render the given text at high quality, using alpha blending to dither the font with the given color. This results in a surface with alpha transparency, so you don't have a solid colored box around the text. The text is antialiased. This will render slower than Solid, but in about the same time as Shaded mode. The resulting surface will blit slower than if you had used Solid or Shaded. Use this when you want high quality, and the text isn't changing too fast.

3.4.1 TTF_RenderText_Solid

`SDL_Surface *`**TTF_RenderText_Solid**`(TTF_Font *`*`font`*`, const char *`*`text`*`, SDL_Color` *`fg`*`)`

font	Font to render the text with. A **NULL** pointer is not checked.
text	The LATIN1 null terminated string to render.
fg	The color to render the text in. This becomes colormap index 1.

Render the LATIN1 encoded *text* using *font* with *fg* color onto a new surface, using the *Solid* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

```
// Render some text in solid black to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *text_surface;
if(!(text_surface=TTF_RenderText_Solid(font,"Hello World!",color))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.13 [TTF_SizeText], page 35,
Section 3.4.2 [TTF_RenderUTF8_Solid], page 40,
Section 3.4.3 [TTF_RenderUNICODE_Solid], page 41,
Section 3.4.4 [TTF_RenderGlyph_Solid], page 42,
Section 3.4.5 [TTF_RenderText_Shaded], page 43,
Section 3.4.9 [TTF_RenderText_Blended], page 47

3.4.2 TTF_RenderUTF8_Solid

`SDL_Surface *`**TTF_RenderUTF8_Solid**`(TTF_Font *`*font*`, const char *`*text*`, SDL_Color` *fg*`)`

font	Font to render the text with. A **NULL** pointer is not checked.
text	The UTF8 null terminated string to render.
fg	The color to render the text in. This becomes colormap index 1.

Render the UTF8 encoded *text* using *font* with *fg* color onto a new surface, using the *Solid* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

Note that this example uses the same text as in the LATIN1 example, that is because plain ASCII is UTF8 compatible.

```
// Render some UTF8 text in solid black to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *text_surface;
if(!(text_surface=TTF_RenderUTF8_Solid(font,"Hello World!",color))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.14 [TTF_SizeUTF8], page 36,
Section 3.4.1 [TTF_RenderText_Solid], page 39,
Section 3.4.3 [TTF_RenderUNICODE_Solid], page 41,
Section 3.4.4 [TTF_RenderGlyph_Solid], page 42,
Section 3.4.6 [TTF_RenderUTF8_Shaded], page 44,
Section 3.4.10 [TTF_RenderUTF8_Blended], page 48

3.4.3 TTF_RenderUNICODE_Solid

`SDL_Surface` ***TTF_RenderUNICODE_Solid**`(TTF_Font *`*font*`, const Uint16 *`*text*`, SDL_Color` *fg*`)`

font Font to render the text with. A **NULL** pointer is not checked.

text The UNICODE null terminated string to render.

fg The color to render the text in. This becomes colormap index 1.

Render the UNICODE encoded *text* using *font* with *fg* color onto a new surface, using the *Solid* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

```
// Render some UNICODE text in solid black to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *text_surface;
Uint16 text[]={'H','e','l','l','o',' ',
               'W','o','r','l','d','!'};
if(!(text_surface=TTF_RenderUNICODE_Solid(font,text,color))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.15 [TTF_SizeUNICODE], page 37,
Section 3.4.1 [TTF_RenderText_Solid], page 39,
Section 3.4.2 [TTF_RenderUTF8_Solid], page 40,
Section 3.4.4 [TTF_RenderGlyph_Solid], page 42,
Section 3.4.7 [TTF_RenderUNICODE_Shaded], page 45,
Section 3.4.11 [TTF_RenderUNICODE_Blended], page 49

3.4.4 TTF_RenderGlyph_Solid

`SDL_Surface *`**TTF_RenderGlyph_Solid**`(TTF_Font *`*`font`*`, Uint16 `*`ch`*`, SDL_Color `*`fg`*`)`

font	Font to render the glyph with. A **NULL** pointer is not checked.
text	The UNICODE character to render.
fg	The color to render the glyph in. This becomes colormap index 1.

Render the glyph for the UNICODE *ch* using *font* with *fg* color onto a new surface, using the *Solid* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors, such as when the glyph not available in the font.

```
// Render and cache all printable ASCII characters in solid black
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *glyph_cache[128-20];
Uint16 ch;
for(ch=20; ch<128; ++ch)
    glyph_cache[ch-20]=TTF_RenderGlyph_Solid(font,ch,color);
```

Combined with a cache of the glyph metrics (minx, miny, and advance), you might make a fast text rendering routine that prints directly to the screen, but with inaccurate kerning. (see Chapter 6 [Glossary], page 54)

See Also:
Section 3.4.8 [TTF_RenderGlyph_Shaded], page 46,
Section 3.4.12 [TTF_RenderGlyph_Blended], page 50,
Section 3.4.1 [TTF_RenderText_Solid], page 39,
Section 3.4.2 [TTF_RenderUTF8_Solid], page 40,
Section 3.4.4 [TTF_RenderGlyph_Solid], page 42,
Section 3.3.12 [TTF_GlyphMetrics], page 32

3.4.5 TTF_RenderText_Shaded

`SDL_Surface *`**`TTF_RenderText_Shaded`**`(TTF_Font *`*`font`*`, const char *`*`text`*`, SDL_Color `*`fg`*`, SDL_Color `*`bg`*`)`

font	Font to render the text with. A **NULL** pointer is not checked.
text	The LATIN1 null terminated string to render.
fg	The color to render the text in. This becomes colormap index 1.
bg	The color to render the background box in. This becomes colormap index 0.

Render the LATIN1 encoded *text* using *font* with *fg* color onto a new surface filled with the *bg* color, using the *Shaded* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

```
// Render some text in shaded black on white to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0}, bgcolor={0xff,0xff,0xff};
SDL_Surface *text_surface;
if(!(text_surface=TTF_RenderText_Shaded(font,"Hello World!",color,bgcolor))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.13 [TTF_SizeText], page 35,
Section 3.4.6 [TTF_RenderUTF8_Shaded], page 44,
Section 3.4.7 [TTF_RenderUNICODE_Shaded], page 45,
Section 3.4.8 [TTF_RenderGlyph_Shaded], page 46,
Section 3.4.1 [TTF_RenderText_Solid], page 39,
Section 3.4.9 [TTF_RenderText_Blended], page 47

3.4.6 TTF_RenderUTF8_Shaded

`SDL_Surface *`**`TTF_RenderUTF8_Shaded`**`(TTF_Font *`*`font`*`, const char *`*`text`*`, SDL_Color `*`fg`*`, SDL_Color `*`bg`*`)`

font	Font to render the text with. A **NULL** pointer is not checked.
text	The UTF8 null terminated string to render.
fg	The color to render the text in. This becomes colormap index 1.
bg	The color to render the background box in. This becomes colormap index 0.

Render the UTF8 encoded *text* using *font* with *fg* color onto a new surface filled with the *bg* color, using the *Shaded* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

Note that this example uses the same text as in the LATIN1 example, that is because plain ASCII is UTF8 compatible.

```
// Render some UTF8 text in shaded black on white to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0}, bgcolor={0xff,0xff,0xff};
SDL_Surface *text_surface;
if(!(text_surface=TTF_RenderUTF8_Shaded(font,"Hello World!",color,bgcolor))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.14 [TTF_SizeUTF8], page 36,
Section 3.4.5 [TTF_RenderText_Shaded], page 43,
Section 3.4.7 [TTF_RenderUNICODE_Shaded], page 45,
Section 3.4.8 [TTF_RenderGlyph_Shaded], page 46,
Section 3.4.2 [TTF_RenderUTF8_Solid], page 40,
Section 3.4.10 [TTF_RenderUTF8_Blended], page 48

3.4.7 TTF_RenderUNICODE_Shaded

`SDL_Surface` ***TTF_RenderUNICODE_Shaded**`(TTF_Font` *font*`, const Uint16` *text*`, SDL_Color` *fg*`, SDL_Color` *bg*`)`

font	Font to render the text with. A **NULL** pointer is not checked.
text	The UNICODE null terminated string to render.
fg	The color to render the text in. This becomes colormap index 1.
bg	The color to render the background box in. This becomes colormap index 0.

Render the UNICODE encoded *text* using *font* with *fg* color onto a new surface filled with the *bg* color, using the *Shaded* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

```
// Render some UNICODE text in shaded black on white to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0}, bgcolor={0xff,0xff,0xff};
SDL_Surface *text_surface;
Uint16 text[]={'H','e','l','l','o',' ',
               'W','o','r','l','d','!'};
if(!(text_surface=TTF_RenderUNICODE_Shaded(font,text,color,bgcolor))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.15 [TTF_SizeUNICODE], page 37,
Section 3.4.5 [TTF_RenderText_Shaded], page 43,
Section 3.4.6 [TTF_RenderUTF8_Shaded], page 44,
Section 3.4.8 [TTF_RenderGlyph_Shaded], page 46,
Section 3.4.3 [TTF_RenderUNICODE_Solid], page 41,
Section 3.4.11 [TTF_RenderUNICODE_Blended], page 49

3.4.8 TTF_RenderGlyph_Shaded

`SDL_Surface *`**TTF_RenderGlyph_Shaded**`(TTF_Font *`*font*`, Uint16` *ch*`, SDL_Color` *fg*`, SDL_Color` *bg*`)`

font	Font to render the glyph with. A **NULL** pointer is not checked.
text	The UNICODE character to render.
fg	The color to render the glyph in. This becomes colormap index 1.
bg	The color to render the background box in. This becomes colormap index 0.

Render the glyph for the UNICODE *ch* using *font* with *fg* color onto a new surface filled with the *bg* color, using the *Shaded* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors, such as when the glyph not available in the font.

```
// Render and cache all printable ASCII characters in shaded black on white
//SDL_Surface *screen;
SDL_Color color={0,0,0}, bgcolor={0xff,0xff,0xff};
SDL_Surface *glyph_cache[128-20];
Uint16 ch;
for(ch=20; ch<128; ++ch)
    glyph_cache[ch-20]=TTF_RenderGlyph_Shaded(font,ch,color,bgcolor);
```

Combined with a cache of the glyph metrics (minx, miny, and advance), you might make a fast text rendering routine that prints directly to the screen, but with inaccurate kerning. (see Chapter 6 [Glossary], page 54)

See Also:
Section 3.4.4 [TTF_RenderGlyph_Solid], page 42,
Section 3.4.12 [TTF_RenderGlyph_Blended], page 50,
Section 3.4.5 [TTF_RenderText_Shaded], page 43,
Section 3.4.6 [TTF_RenderUTF8_Shaded], page 44,
Section 3.4.8 [TTF_RenderGlyph_Shaded], page 46,
Section 3.3.12 [TTF_GlyphMetrics], page 32

3.4.9 TTF_RenderText_Blended

`SDL_Surface *`**TTF_RenderText_Blended**`(TTF_Font *`*font*`, const char *`*text*`, SDL_Color` *fg*`)`

font	Font to render the text with. A **NULL** pointer is not checked.
text	The LATIN1 null terminated string to render.
fg	The color to render the text in. Pixels are blended between transparent and this color to draw the sntialiased glyphs.

Render the LATIN1 encoded *text* using *font* with *fg* color onto a new surface, using the *Blended* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

```
// Render some text in blended black to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *text_surface;
if(!(text_surface=TTF_RenderText_Blended(font,"Hello World!",color))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.13 [TTF_SizeText], page 35,
Section 3.4.10 [TTF_RenderUTF8_Blended], page 48,
Section 3.4.11 [TTF_RenderUNICODE_Blended], page 49,
Section 3.4.12 [TTF_RenderGlyph_Blended], page 50,
Section 3.4.1 [TTF_RenderText_Solid], page 39,
Section 3.4.5 [TTF_RenderText_Shaded], page 43

3.4.10 TTF_RenderUTF8_Blended

SDL_Surface ***TTF_RenderUTF8_Blended**(TTF_Font **font*, const char **text*, SDL_Color *fg*)

font	Font to render the text with. A **NULL** pointer is not checked.
text	The UTF8 null terminated string to render.
fg	The color to render the text in. Pixels are blended between transparent and this color to draw the sntialiased glyphs.

Render the UTF8 encoded *text* using *font* with *fg* color onto a new surface, using the *Blended* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

Note that this example uses the same text as in the LATIN1 example, that is because plain ASCII is UTF8 compatible.

```
// Render some UTF8 text in blended black to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *text_surface;
if(!(text_surface=TTF_RenderUTF8_Blended(font,"Hello World!",color))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.14 [TTF_SizeUTF8], page 36,
Section 3.4.9 [TTF_RenderText_Blended], page 47,
Section 3.4.11 [TTF_RenderUNICODE_Blended], page 49,
Section 3.4.12 [TTF_RenderGlyph_Blended], page 50,
Section 3.4.2 [TTF_RenderUTF8_Solid], page 40,
Section 3.4.6 [TTF_RenderUTF8_Shaded], page 44

3.4.11 TTF_RenderUNICODE_Blended

`SDL_Surface *`**TTF_RenderUNICODE_Blended**`(TTF_Font *`*font*`, const Uint16 *`*text*`, SDL_Color `*fg*`)`

font	Font to render the text with. A **NULL** pointer is not checked.
text	The UNICODE null terminated string to render.
fg	The color to render the text in. Pixels are blended between transparent and this color to draw the sntialiased glyphs.

Render the UNICODE encoded *text* using *font* with *fg* color onto a new surface, using the *Blended* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.
NOTE: Passing a **NULL** *text* into this function will result in undefined behavior.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors.

```
// Render some UNICODE text in blended black to a new surface
// then blit to the upper left of the screen
// then free the text surface
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *text_surface;
Uint16 text[]={'H','e','l','l','o',' ',
               'W','o','r','l','d','!'};
if(!(text_surface=TTF_RenderUNICODE_Blended(font,text,color))) {
    //handle error here, perhaps print TTF_GetError at least
} else {
    SDL_BlitSurface(text_surface,NULL,screen,NULL);
    //perhaps we can reuse it, but I assume not for simplicity.
    SDL_FreeSurface(text_surface);
}
```

See Also:
Section 3.3.15 [TTF_SizeUNICODE], page 37,
Section 3.4.9 [TTF_RenderText_Blended], page 47,
Section 3.4.10 [TTF_RenderUTF8_Blended], page 48,
Section 3.4.12 [TTF_RenderGlyph_Blended], page 50,
Section 3.4.3 [TTF_RenderUNICODE_Solid], page 41,
Section 3.4.7 [TTF_RenderUNICODE_Shaded], page 45

3.4.12 TTF_RenderGlyph_Blended

`SDL_Surface *`**TTF_RenderGlyph_Blended**`(TTF_Font *`*`font`*`, Uint16` *`ch`*`, SDL_Color` *`fg`*`)`

font	Font to render the glyph with. A **NULL** pointer is not checked.
text	The UNICODE character to render.
fg	The color to render the glyph in. Pixels are blended between transparent and this color to draw the sntialiased glyph.

Render the glyph for the UNICODE *ch* using *font* with *fg* color onto a new surface, using the *Blended* mode (see Section 3.4 [Render], page 38). The caller (you!) is responsible for freeing any returned surface.

NOTE: Passing a **NULL** *font* into this function will cause a segfault.

Returns: a pointer to a new SDL_Surface. **NULL** is returned on errors, such as when the glyph not available in the font.

```
// Render and cache all printable ASCII characters in blended black
//SDL_Surface *screen;
SDL_Color color={0,0,0};
SDL_Surface *glyph_cache[128-20];
Uint16 ch;
for(ch=20; ch<128; ++ch)
    glyph_cache[ch-20]=TTF_RenderGlyph_Blended(font,ch,color);
```

Combined with a cache of the glyph metrics (minx, miny, and advance), you might make a fast text rendering routine that prints directly to the screen, but with inaccurate kerning. (see Chapter 6 [Glossary], page 54)

See Also:
Section 3.4.4 [TTF_RenderGlyph_Solid], page 42,
Section 3.4.8 [TTF_RenderGlyph_Shaded], page 46,
Section 3.4.9 [TTF_RenderText_Blended], page 47,
Section 3.4.10 [TTF_RenderUTF8_Blended], page 48,
Section 3.4.12 [TTF_RenderGlyph_Blended], page 50,
Section 3.3.12 [TTF_GlyphMetrics], page 32

4 Types

These types are defined and used by the SDL_ttf API.

4.1 TTF_Font

```
typedef struct _TTF_Font TTF_Font;
```

The opaque holder of a loaded font. You should always be using a pointer of this type, as in `TTF_Font*`, and not just plain `TTF_Font`. This stores the font data in a struct that is exposed only by using the API functions to get information. You should not try to access the struct data directly, since the struct may change in different versions of the API, and thus your program would be unreliable.

See Also:
Section 3.2 [Management], page 14

5 Defines

TTF_MAJOR_VERSION
: `2`
SDL_ttf library major number at compilation time

TTF_MINOR_VERSION
: `0`
SDL_ttf library minor number at compilation time

TTF_PATCHLEVEL
: `7`
SDL_ttf library patch level at compilation time

UNICODE_BOM_NATIVE
: `0xFEFF`
This allows you to switch byte-order of UNICODE text data to native order, meaning the mode of your CPU. This is meant to be used in a UNICODE string that you are using with the SDL_ttf API.

UNICODE_BOM_SWAPPED
: `0xFFFE`
This allows you to switch byte-order of UNICODE text data to swapped order, meaning the reversed mode of your CPU. So if your CPU is LSB, then the data will be interpretted as MSB. This is meant to be used in a UNICODE string that you are using with the SDL_ttf API.

TTF_STYLE_NORMAL
: `0x00`
Used to indicate regular, normal, plain rendering style.

TTF_STYLE_BOLD
: `0x01`
Used to indicate bold rendering style. This is used a bitmask along with other styles.

TTF_STYLE_ITALIC
: `0x02`
Used to indicate italicized rendering style. This is used a bitmask along with other styles.

TTF_STYLE_UNDERLINE
: `0x04`
Used to indicate underlined rendering style. This is used a bitmask along with other styles.

See Also:
Chapter 6 [Glossary], page 54,

6 Glossary

Byte Order This all has to do with how data larger than a byte is actually stored in memory. The CPU expects 16bit and 32bit, and larger, data to be ordered in one of the two ways listed below. SDL has macros which you can use to detemine which endian your program will be using.

Big-Endian(MSB) means the most significant byte comes first in storage. Sparc and Motorola 68k based chips are MSB ordered.
(SDL defines this as **SDL_BYTEORDER==SDL_BIG_ENDIAN**)

Little-Endian(LSB) is stored in the opposite order, with the least significant byte first in memory. Intel and AMD are two LSB machines.
(SDL defines this as **SDL_BYTEORDER==SDL_LIL_ENDIAN**)

LATIN1 Latin-1 is an extension of ASCII, where ASCII only defines characters 0 through 127. Latin-1 continues and adds more common international symbols to define through character 255.

ISO 8859-1 (Latin-1) Unicode Table (pdf)

0080 C1 Controls and Latin-1 Supplement 00FF

	008	009	00A	00B	00C	00D	00E	00F
0	XXX 0080	DCS 0090	NB SP 00A0	° 00B0	À 00C0	Ð 00D0	à 00E0	ð 00F0
1	XXX 0081	PU1 0091	¡ 00A1	± 00B1	Á 00C1	Ñ 00D1	á 00E1	ñ 00F1
2	BPH 0082	PU2 0092	¢ 00A2	² 00B2	Â 00C2	Ò 00D2	â 00E2	ò 00F2
3	NBH 0083	STS 0093	£ 00A3	³ 00B3	Ã 00C3	Ó 00D3	ã 00E3	ó 00F3
4	IND 0084	CCH 0094	¤ 00A4	´ 00B4	Ä 00C4	Ô 00D4	ä 00E4	ô 00F4
5	NEL 0085	MW 0095	¥ 00A5	µ 00B5	Å 00C5	Õ 00D5	å 00E5	õ 00F5
6	SSA 0086	SPA 0096	¦ 00A6	¶ 00B6	Æ 00C6	Ö 00D6	æ 00E6	ö 00F6
7	ESA 0087	EPA 0097	§ 00A7	· 00B7	Ç 00C7	× 00D7	ç 00E7	÷ 00F7
8	HTS 0088	SOS 0098	¨ 00A8	¸ 00B8	È 00C8	Ø 00D8	è 00E8	ø 00F8
9	HTJ 0089	XXX 0099	© 00A9	¹ 00B9	É 00C9	Ù 00D9	é 00E9	ù 00F9
A	VTS 008A	SCI 009A	ª 00AA	º 00BA	Ê 00CA	Ú 00DA	ê 00EA	ú 00FA
B	PLD 008B	CSI 009B	« 00AB	» 00BB	Ë 00CB	Û 00DB	ë 00EB	û 00FB
C	PLU 008C	ST 009C	¬ 00AC	¼ 00BC	Ì 00CC	Ü 00DC	ì 00EC	ü 00FC
D	RI 008D	OSC 009D	SHY 00AD	½ 00BD	Í 00CD	Ý 00DD	í 00ED	ý 00FD
E	SS2 008E	PM 009E	® 00AE	¾ 00BE	Î 00CE	Þ 00DE	î 00EE	þ 00FE
F	SS3 008F	APC 009F	¯ 00AF	¿ 00BF	Ï 00CF	ß 00DF	ï 00EF	ÿ 00FF

Kerning Kerning is the process of spacing adjacent characters apart depending on the actual two adjacent characters. This allows some characters to be closer to each other than others. When kerning is not used, such as when using the glyph metrics advance value, the characters will be spaced out at a constant size that accomodates all pairs of adjacent characters. This would be the maximum space between characters needed. There's currently no method to retrieve the kerning for a pair of characters from SDL_ttf, However correct kerning will be applied when a string of text is rendered instead of individual glyphs.

Index

(Index is nonexistent)