Bitstream Font Fusion®
5.0a
Reference Guide

July 2009

BITSTREAM®

This document uses the following Bitstream Fonts:
Humanist 521 BT, Humanist 777 extra black, Bitstream Iowan Old Style™, Newspaper Pi, and Monospace 821.

Bitstream Inc., 245 First Street, Cambridge, MA 02142
Worldwide phone number: 617-497-6222
OEM Sales Phone number in the U.S. and Canada: 800-522-3668
7/10/09

Table of Contents

CHAPTER 4: FONT MANAGER API

CHAPTER 5: CACHE MANAGER API

Font Fusion Overview

1

Topics

- General Information
- Using the Font Fusion Core
- Using the Font Fusion Font Manager
- Using the Font Fusion Cache Manager
- Using the Font and Cache Managers

General Information

Font Fusion is Bitstream's premier font rendering subsystem, rendering high-quality characters in any format, at any resolution, on any platform or device. Font Fusion has been designed from the ground up to support consumer electronics devices, mobile phones, PDAs, set-top boxes, digital TVs, printers, graphics applications, and embedded systems. Delivering exceptional text rendering on all platforms and devices, Font Fusion also provides the best possible text output for TV of any solution on the market today. Font Fusion marks the convergence of the Bitstream TrueDoc® and T2K® rasterizers, available in an advanced, object-oriented architecture.

Architectural Overview

Font Fusion includes three components:

- Core Font Engine
- Font Manager (optional)
- Cache Manager (optional)

The Core Font Engine allows your application to render high-quality glyphs from various font formats. The Font Manager supports multiple fonts and font fragments simultaneously, while the Cache Manager boosts overall system performance by employing a high-speed cache to take advantage of memory resources.

Applications & Operating Systems Supported

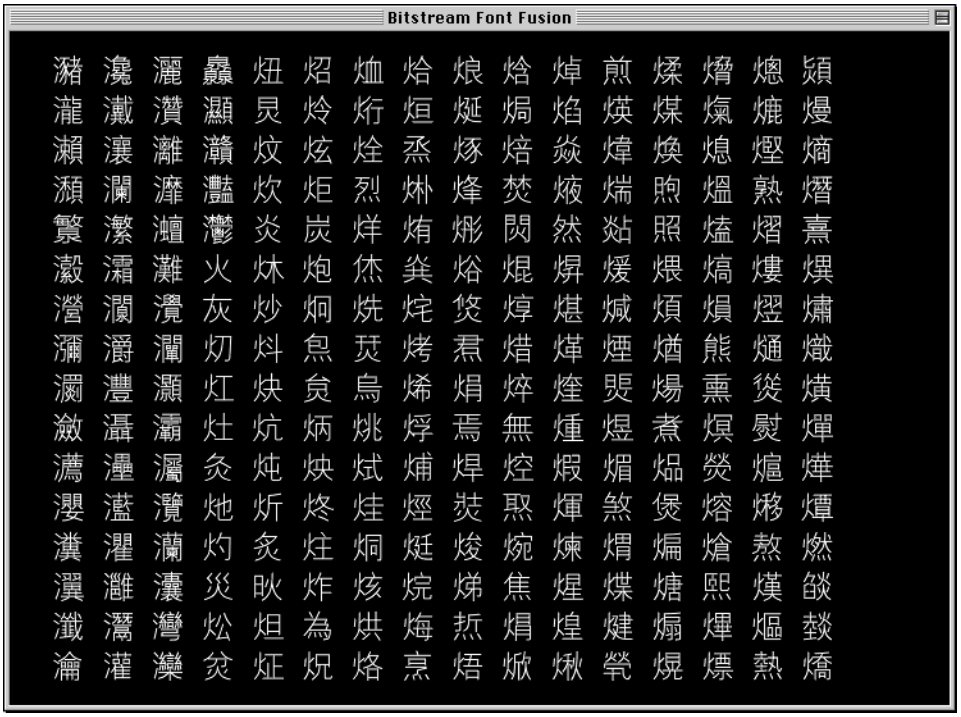
Font Fusion supports a multitude of applications and operating systems, including:

- Cross-platform applications
- Web (HTML) applications
- Macintosh® & Windows®
- Linux® & UNIX®
- Embedded operating systems
- Real time operating systems

Font Formats Supported

Font Fusion can render characters from all of the following font formats:

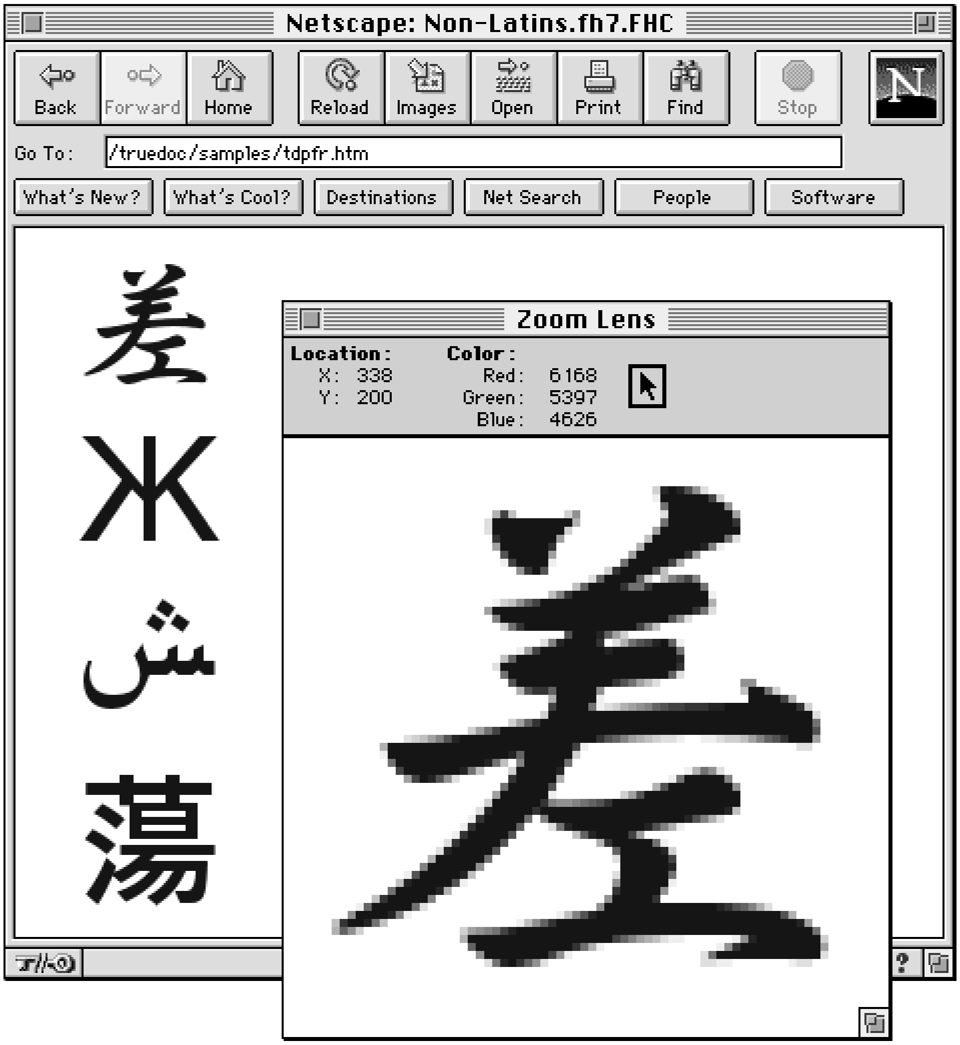
- Type 1
- TrueType®
- TrueType collections
- Microsoft® and Adobe® OpenType®
- Compact font format (CFF)/Type 2
- TrueDoc portable font resources (PFRs)
- Bitstream Speedo™
- T2K
- Embedded bitmaps (TrueType, TrueDoc, and T2K)
- Font Fusion Stroke (FFS)
- Dfont, Mac rfork font format
- PCL Encapsulated Outline (PCLeo)
- PCL Encapsulated Truetype Outline (PCLetto)
- Bitmap Distribution Format (BDF), Windows FNT Format



A traditional Chinese font, in FFS format, containing over 13,000 characters, occupies less than 0.5MB!

Multilingual Capabilities

Font Fusion can render any character shape (regardless of the complexity of the shape), is fully compatible with double-byte non-Latin fonts, and supports languages written vertically or right to left. Font Fusion can also render PFR (portable font resource) data stored locally or downloaded from other sources. This gives an interactive TV system, for example, the ability to support multiple languages without embedding large amounts of font data in a set-top box.

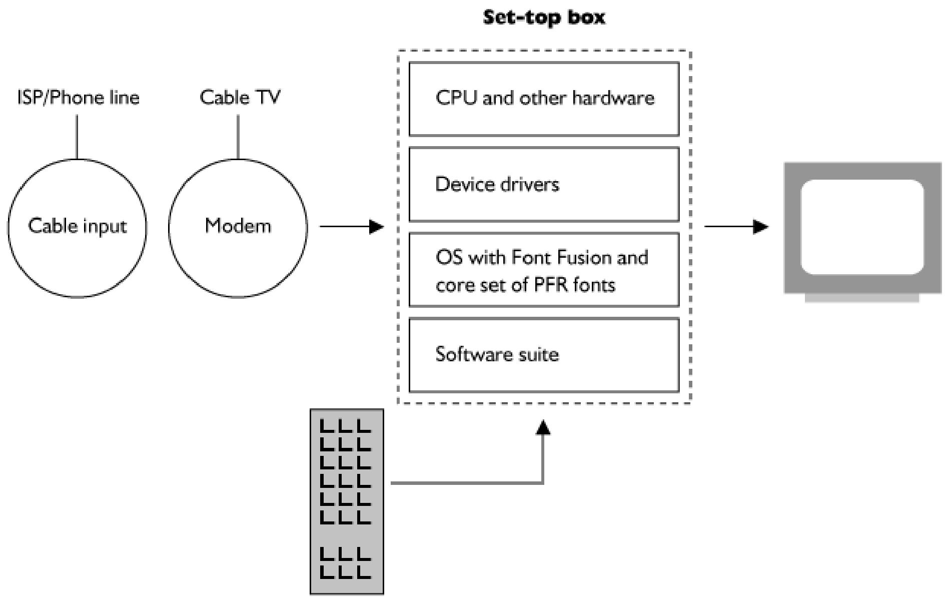


A sample of international characters recorded for a portable font resource (PFR) and regenerated on screen in an Internet browser. The zoomed-in box illustrates anti-aliasing capabilities. (Note the gray pixels at the edges of the character.)

Devices Supported

Bitstream Font Fusion can output characters to any bitmap device. In addition, Font Fusion can optimize the output quality for these devices:

- Color LCD displays
- Grayscale monitors
- Black-and-white monitors
- TV and high-definition TV (HDTV)
- Set-top boxes
- Continuous tone printers
- Embedded devices
- Internet appliances



Set-top box architecture for a Font Fusion-enhanced implementation. The Font Fusion font renderer and a core set of PFR fonts stored in ROM allow for the display of both static and dynamic text from a variety of sources, including multilingual documents.

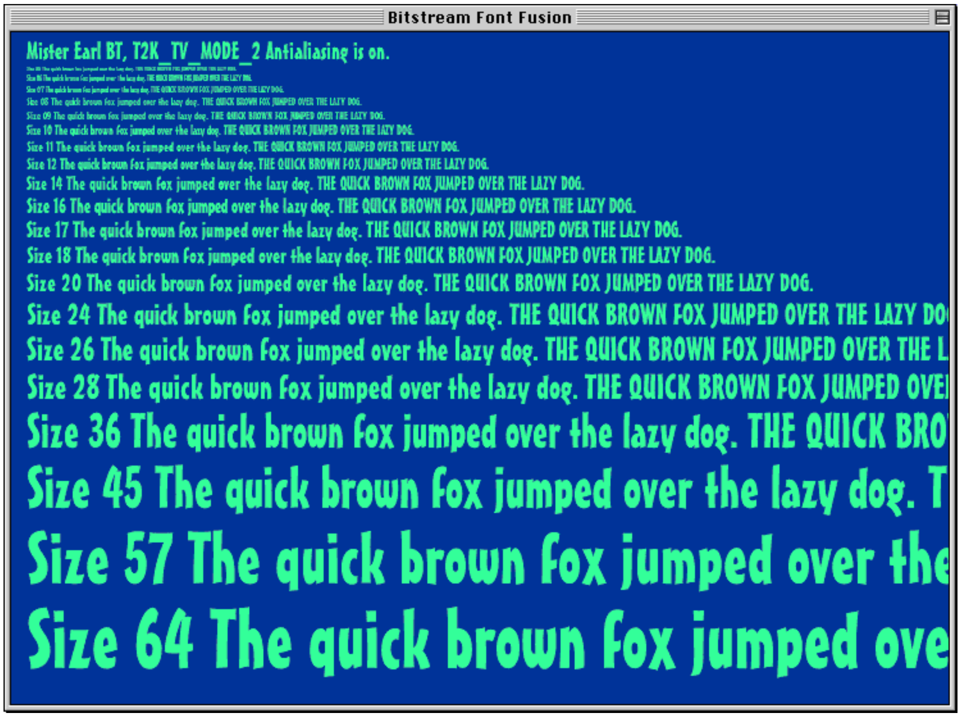
High-Quality Output

Font Fusion produces well-formed characters regardless of output size or resolution.

Font Fusion contains the following enhancements to ensure the highest quality output on a variety of devices:

- anti-aliasing
- filtering and other post-processing
- subpixel positioning

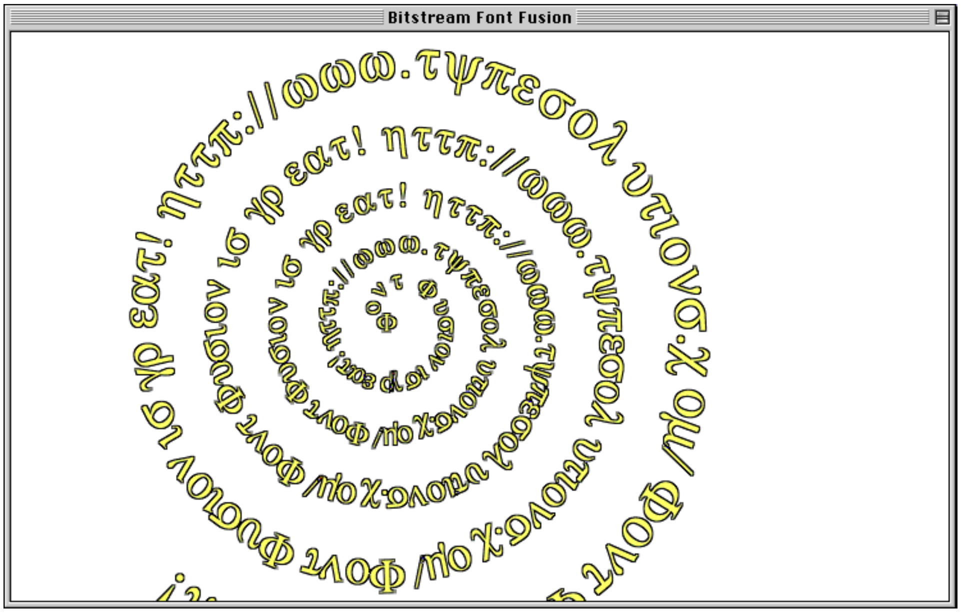
Font Fusion includes an **anti-aliasing** (sometimes called grayscaling) output module, which ensures smooth, well-defined character edges at all resolutions.



Mister Earl BT, T2K_TV_MODE_2 Antialiasing is on.
Size 05 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 06 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 07 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 08 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 09 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 10 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 11 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 12 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 14 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 16 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 17 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 18 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 20 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
Size 24 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DO
Size 26 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE L
Size 28 The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVE
Size 36 The quick brown fox jumped over the lazy dog. THE QUICK BRO
Size 45 The quick brown fox jumped over the lazy dog. T
Size 57 The quick brown fox jumped over the
Size 64 The quick brown fox jumped ove

The anti-aliasing technology developed by Bitstream for Font Fusion provides for layers of pixels of 128 to 256 shades of gray (or other colors) to soften the hard edges that often result when character outlines are fixed to a grid pattern.

Your application has the unique capability of supplying a filter function "plug-in" for **post-processing** of images that the Font Fusion Core produces. The Core creates bitmaps in either 1-bit or 8-bit depth (alpha values range from 0 to 126). You can apply Gaussian fuzz-filtering, smearing, colorizing, or even texture mapping to these bitmaps.

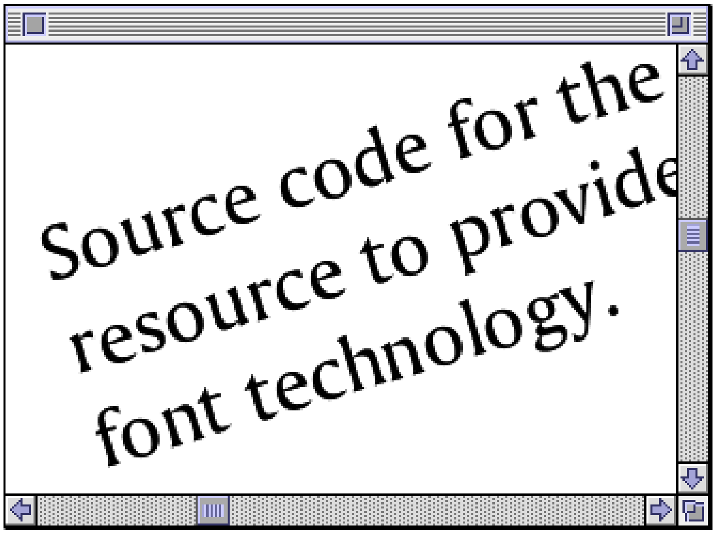


Using Font Fusion, you can take a single-color character and create a multiple-color character, complete with a border. The border is anti-aliased to the background, and the interior color is anti-aliased to the border.

With its **subpixel positioning** technology, Font Fusion can accurately adjust and control the placement of characters. Previously, outlines of character shapes were simply fitted to character grids in an output device (such as a laser printer or a computer screen). If a pixel fell within the outline of a character shape, the pixel was turned on. If not, the pixel was left off.

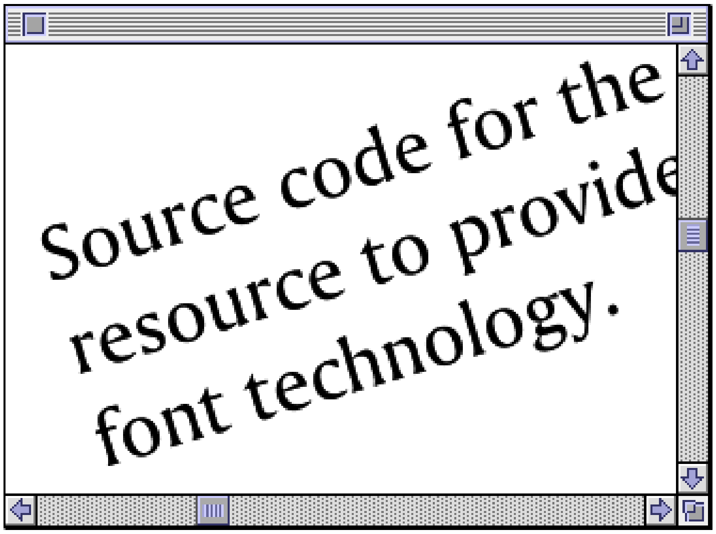
When combined with anti-aliasing (grayscale output module), Font Fusion can control the placement of characters down to 1/64 of a pixel in both x and y dimensions. This is done by first rendering the character in a 64-pixel grid, then down-sampling the glyph to a lower resolution. This can solve many problems that occur when composing text on a low-resolution device, such as a TV screen. This also allows for exceptional character spacing and quality when text is rotated or displayed on a slanted baseline.

NOTE: It is recommended that you use subpixel positioning only when rotating text.



Source code for the resource to provide font technology.

A common problem encountered in the display of text on low-resolution screens. Note that the baselines of the words are bouncing, and that certain characters (most notably the 'u' in 'Source,' the 'c' in code and the 'pr' in 'provide') have floated up. Also note the erratic spacing among the letters 'hnolog' in 'technology.'



Source code for the resource to provide font technology.

Font Fusion uses subpixel positioning to correct these problems. Subpixel positioning allows text to be placed more accurately. This sample shows the results when the text used in the previous picture has 1/16 pixel positioning applied. (Note that it corrects the baseline and spacing problems from the previous sample.)

Using the Font Fusion Core

Font Fusion is extremely small and fast. To achieve this, Bitstream abstracts the optional components—the Font and Cache Managers—from the Core.

The Core knows nothing about the Font Manager, and only enough about the Cache Manager to write data directly into the cache buffer, further increasing performance. In this design also, binding the Core to the Cache Manager takes place only at run-time.

The Core Engine is modelled on T2K, built by Sampo Kaasila, Bitstream Vice President of Research and Development. Sampo created T2K while working for Type Solutions, Inc., now a Bitstream company. Sampo has vast experience in font technology, as he was the lead engineer behind the TrueType technology while at Apple Computer, Inc.

If you are familiar with T2K, the Core continues to work exactly as it did before. The Font Manager and Cache Manager are strictly add-on layers. If you use both the Font Manager and the Cache Manager, the Cache Manager makes calls through the Font Manager to the Core only to manage multiple font fragments.

Bitstream designed both the Font and Cache Managers so that they are consistent with the Core in presenting an API to your application that uses C++.

Using the Font Fusion Font Manager

With the Font Manager, your application environment can install any number of font binary objects (input streams). These input streams, in the cases of TrueType Collections and TrueDoc PFRs, may contain more than one scalable typeface. They may also contain fragments of scalable typefaces that the Font Manager merges dynamically. This dynamic merger can take place across type technologies, as well.

Up to 64K input streams, 64K physical font fragments (the internal limit), 64K logical (merged) typefaces, and 64K dynamic fonts can exist in the Font Manager simultaneously if there is enough memory to support the Font Manager's internal data structures.

Once the Font Manager builds its internal data structures from installed input streams, your application can find out what merged typefaces are available and choose one to create a dynamic font cookie (a typeface, a transformation, and an optional algorithmic style), which you can then activate with the Core.

The Font Manager's **`RenderGlyph()`** function allows the Font Manager to merge multiple font fragments. It walks through each fragment of a merged typeface until the Core generates the character.

Merging Fonts Dynamically

Your application deals with three types of fonts when rendering characters.

Physical fonts, such as all the characters in Swiss 721™ or a subset of characters in Swiss 721, contain outline definitions in scalable form. They're the base font component of a font. They can include the entire font or a subset of it, i.e., a font fragment.

Logical fonts are merged "super" fonts made up of one or more physical font fragments. Physical and logical fonts are only concerned with outline font resources.

Dynamic fonts—such as Swiss 721, 10 point—are fonts with particular attributes that Font Fusion creates on the fly. Font Fusion renders characters using the outline data in the physical fonts.

NOTE: There is only one issue you need to note when you use both the Font and Cache Managers: the Cache Manager must make all requests for characters not already cached via the Cache Manager's **`RenderGlyph()`** function, **`FF_CM_RenderGlyph()`**. This allows you to support the dynamic merging of fonts.

Using the Font Fusion Cache Manager

With the Cache Manager, your application can use the Core to generate glyph images directly into the buffer of a high-speed cache mechanism.

Your application creates a new Cache Manager by calling the appropriate constructor. The only variable you pass to this function is the amount of memory that the cache will live in. To create characters, your application simply calls the Cache Manager to render the glyph. Data passed into this function include the font code, character code, subpixel positioning information, and the level of hinting you want.

In addition, you must pass in the current Scaler object your application holds. This scaler becomes a repository for all the information about the created bitmap as well as the image data itself.

The Cache Manager always appears to render the glyph, but first it looks in the cache buffer for a glyph matching the request. If it finds one, it makes it appear to your application that the Core has just rendered the glyph. In this way, your application can either call the Core directly for any glyphs, or call the Cache Manager to cache the glyphs. Your application's **`PrintChar()`**, or "BLT," of the glyph image is the same either way. The Core always delivers the image in the current Core scaler context that your application owns, either as the user of the Cache Manager by finding the image in the cache, or by creating the image, storing it in the cache, and exposing it to your application. Whichever method the Core uses, it is completely abstracted from you.

The only other major operation that your application can initiate is flushing the cache. All this function requires is a pointer to the cache that you want to flush.

NOTE: There is only one issue you need to note when you use both the Font and Cache Managers: the Cache Manager must make all requests for characters not already cached via the Cache Manager's **`RenderGlyph()`** function, **`FF_CM_RenderGlyph()`**. This allows you to support the dynamic merging of fonts.

Getting Started with the Font Fusion Core

Topics

- Getting started
- Allocating memory
- Assert statements
- Compile-time options
- Errors
- Font size options

Getting Started

When building Font Fusion into your application, you should first decide on the components that you will be integrating in addition to the core:

- The Cache Manager
- The Font Manager

The Font Fusion Core is required in any implementation. The demos can aid you in deciding what other components you need. You should first look at the demo that combines both the Cache Manager and Font Manager components. After looking at the demos, you should consider what options you need in your implementation, such as:

- Technologies (PFR, OTF, TTF, Speedo, CFF, Type 1, Stroke Based Fonts)
- Bitmap output format (monochrome, grayscale, LCD)
- Hinting (Native hinting, auto hinting, no hinting)

Once you have identified all the options that you want to build, for example, if you decided you need the Cache Manager, a TrueType font renderer, and auto hinting, you should type these project settings into the Size demo to see a fairly accurate estimate of what the size is going to be of your Font Fusion implementation.

Font Fusion is object-oriented, even though the actual implementation uses ANSI C. This means that you will be creating a number of objects when you use Font Fusion. All classes have a constructor and destructor. It is important that you call the proper destructor when you are done with a particular object.

What Files Should I Look at First?

We recommend that you first familiarize yourself with `t2k.h`. This file contains documentation, a coding example, and the actual Font Fusion API.

Second, you should look at `config.h`. Usually, this is the only file you need to edit. The file configures Font Fusion for your platform, enables or disables optional features. There are many features included in this file. Turn on only required features in order to minimize the size of the Font Fusion Core.

Run the Size demo with the version of `config.h` that you intend on using in order to get an idea of the size of your implementation.

How can I build the Font Fusion demos?

Font Fusion SDK includes a set of fully functional demos to illustrate how the Font Fusion APIs and individual components can be used. You can compile and run most of the demos on the Microsoft Windows/Macintosh operating systems as well as in GNU environments such as Linux.

Windows Build Environment

Each demo directory contains a Visual C++ project file (`.dsp`) that you can use to run the demos. You can build the demos collectively by executing the `AllWinDemo.dsp` file at `$(ROOT)/demo/All_Demo` or individually by opening the `*.dsp` file at the respective demo directory.

Mac Build Environment

Issue the "`make`" command at `$(ROOT)/demo/DEMO_NAME` to build the demo. To clean all the **`*.o`** and executable files, use the "`make clean`" command at `$(ROOT)/demo/DEMO_NAME`.

You can also run the Font Fusion demos by opening the `*.xcodeproj` file at `$(ROOT)/demo/DEMO_NAME.`

GNU Build Environment

To make the command line build convenient, the build procedure on the GNU platform is hierarchical and modular. Issue the "`make`" command at `$(ROOT)/demo/DEMO_NAME` to build the demo.

To clean all the **`*.o`** and executable files, use the "`make clean`" command at `$(ROOT)/demo/DEMO_NAME`.

The table below summarizes the functionality for each of the demos included:

DEMO_NAME	Description
`t2k_demo`	The Font Fusion core demo
`cm_demo`	Demo to illustrate Cache Manager implementation
`fm_demo`	Demo to illustrate Font Manager implementation

DEMO_NAME	Description
`fm_cm_demo`	Demo to illustrate the Cache Manager + Font Manager implementation
`size_demo`	Demo to estimate the size of the Font Fusion implementation
`pr_demo`	Demo to validate the working of PCL and SFS font
`otf_demo`	Demo to illustrate the 'vert' GSUB OpenType feature for vertical writing

What is the Best Way to Get Started?

First, configure `config.h`. Refer to the variables listed in "Compile-Time Options" on page 33 and enable or disable your options as needed.

Next, look at the coding example below. We recommend that you start "outside in," by creating the outermost object, followed by the objects it contains. See Chapter 7 for text flows of this process.

Creating and destroying a memory handle.

First create and destroy a `Memhandler` object. (This is the "outermost" object.)

```
/* Create the Memhandler object. */
tsiMemObject *mem = NULL;
mem= tsi_NewMemhandler( &errCode );
assert( errCode == 0 );

/* Destroy the Memhandler object. */
tsi_DeleteMemhandler( mem );
```

Creating an input stream.

Next, create an `InputStream` object.

```
/* Create the Memhandler and InputStream objects. */
tsiMemObject *mem;
unsigned char *data;
unsigned long length;
int *errCode;
tsiMemObject *mem = NULL;
InputStream *in = NULL;
mem= tsi_NewMemhandler( &errCode );
```

```
assert( errCode == 0 );
  in = New_InputStream3( mem, data, length, &errCode );
  assert( errCode == 0 );

/* Destroy the InputStream object. */
Delete_InputStream( in, &errCode  );

/* Destroy the Memhandler object. */
tsi_DeleteMemhandler( mem );
```

▸ Creating other objects.

Next, create the `T2K` scaler and `sfntClass` objects.

What are the Functions in Font Fusion?

Font Fusion functions are organized according to their roles, described in the sections below.

Core Functions:

The main core functions necessary to render a character:

- The **`tsiMemObject`** object handles all memory allocation, de-allocation, and re-allocation.
- The **`InputStream`** object provides a level of abstraction for the core by exposing certain methods used by Font Fusion to access the data. Font Fusion does not need to know if the data is in memory, on a disk, on a network, etc.
- The **`sfntClass`** is an internal class that represents a font.
- The **`T2K scaler`** object represents and instance of the font scaler. The font scaler's main task is to produce good looking bitmap images for characters at different sizes and transformations.
- The `T2K_NewTransformation()` allows you to set the transformation matrix and x and y resolutions when you render characters and strings.

Additional core functions include:

- **`ExtractPureT1FromPCType1()`** and **`ExtractPureT1FromMacPOSTResources()`**, which translate font data into formats which Font Fusion can process.

- **`FF_GetTTTablePointer()`**, which returns a pointer to a memory buffer containing a TrueType table.
- **`FF_GlyphExists()`**, which checks for the existence of characters in a font
- **`FF_NewColorTable()`**, which supports color combinations other than black and white.
- **`FF_PSNameToCharCode()`**, which translates PostScript character names to character code.
- **`Set_PlatformID`**, which defines the computer platform which will use the font.
- **`Set_PlatformSpecificID`**, which defines the specific character map within a TrueType font.
- **`FF_ForceCMAPChange()`**, which forces the changing of the TrueType character map.

Should I Use Public APIs Only?

You should only use functions visible in `t2k.h`. If you need to use something else, then let us know. Do not rely on any functions or methods outside of Font Fusion, because they may change from release to release.

Allocating Memory

This section discusses the following topics related to memory allocation.

- Using your own memory allocator and de-allocator with Font Fusion
- The `InputStream` object
- If you have a lot of fonts open and active at the same time
- The `tsiMemObject` object
- Using `tsiMemObject` per font

Using Your Own Memory Allocator and De-allocator with Font Fusion

`config.h` allows you to remap allocation, re-allocation, and deletion to anything you want. Refer to the first three defines in `config.h`, which are:

```
/*** #1 ***/
#define CLIENT_MALLOC( size )malloc( size )
/* #define CLIENT_MALLOC( size )AllocateTaggedMemory-
NilAllowed(n,"t2k") */

/*** #2 ***/
#define CLIENT_FREE( ptr )free( ptr )
/* #define CLIENT_FREE( ptr )FreeTaggedMemory(p,"t2k") */

/*** #3 ***/
#define CLIENT_REALLOC( ptr, newSize )realloc( ptr, newSize
)
/* #define CLIENT_REALLOC( ptr, newSize )ReallocateTagged-
MemoryNilAllowed(ptr, size, "t2k") */
```

The `config.h` file also contains the **`ENABLE_CLIENT_ALLOC`** compile conditional, which enables the allocation/re-allocation/de-allocation through the external memory managers. The `tsiMemObj` and cache manager APIs under this macro enable the client itself to handle all the memory allocation/re-allocation/de-allocation tasks.

The InputStream Object

The `InputStream` object provides a level of abstraction for the core. Basically, the `InputStream` object exposes certain methods that Font Fusion uses to access the data.

This means that Font Fusion does not need to know if the data is in memory, on a disk, on a network, etc. It also provides more robustness. For instance, the `InputStream` object checks for out-of-bounds read attempts.

This error checking, together with the abstraction `InputStream` provides, produces a more solid and robust design and therefore a better product.

The tsiMemObject Object

`tsiMemObject` is an object that handles all memory allocation, de-allocation, and re-allocation.

We do it this way, instead of making direct calls to the operating system, because `tsiMemObject` does a lot of error checking. For instance, `tsiMemObject` puts special markers both before and after allocated memory so that we can detect any attempts to write outside the allocated memory. It also detects any memory leaks and attempts to free already-freed memory.

In this way, `tsiMemObject` provides a solid foundation for the product.

Using One tsiMemObject per Font

We could have shared the `tsiMemObject` among all open fonts, but we get better performance using only one `tsiMemObject` per font.

The current implementation of `tsiMemObject` also has a maximum limit on the number of pointers it can allocate.

If You Have a lot of Fonts Open and

Active at the Same Time

This should not cause any problems. You should have one `tsiMemObject` per font.

You can choose to keep multiple fonts open simultaneously. You can also decide to use one or more Font Fusion scalers simultaneously. Or you could decide to only open one font at a time.

You can also use memory-based fonts when you create the `InputStream` class or you can use disk-based fonts.

These choices trade off memory use and speed. For instance, disk-based fonts do not require any memory allocation but take longer to access.

Assert Statements

Assert statements are in the code to detect and prevent programmer errors in pre-release and debug builds.

In a release build, you need to turn off assert statements in `config.h` to increase speed and to reduce the code size.

However, in debug builds, leave it on, to ensure that everything is working properly.

Optional: Redefining "Assert"

You have the option of redefining "assert" by adding two lines in the `config.h` file.

```
/*** #4 ***/
/* #undef assert  (line1) */
/* Just leave it for some clients, OR */
/* #define assert(cond) CLIENT_ASSERT( cond ), OR for a _FINAL_ build
_ALWAYS_ define as NULL for maximum speed */
/* #define assert(cond) NULL /*
#undef assert
#define assert(cond) NULL
*/
```

Compile-Time Options

The many features offered in Font Fusion can be enabled or disabled by modifying the `ff_user.h` include file or externally by setting the compile-time options to zero for OFF, or one for ON. For example, you can type `ENABLE_NATIVE_TT_HINTS=1` to turn on native Truetype hinting, or `ENABLE_NATIVE_TT_HINTS=0` to turn it off. The list of available compile-time options can be found in the config.h file. **All features are disabled by default except for the core code, `ENABLE_NATIVE_TT_HINTS`, `ENABLE_NON_ZERO_WINDING_RULE`, and `ENABLE_CACHING_EBLC`.** Do not use the option `ENABLE_T2KE`, as it is reserved for future development.

Many features in Font Fusion increase ROM and RAM needs when used, so we recommend that you enable only the ones you intend to use, particularly if you need to conserve memory in your application. The table below lists approximate sizes for each feature. Use the Size demo to estimate the size of your Font Fusion implementation.

Core and Components	Description	Size
Font Fusion Core (minimum)	Includes the core with no other options specified.	35K
Font Fusion Core (default)	Includes the core, `ENABLE_NATIVE_TT_HINTS`, `ENABLE_MEM_VALIDATION`, and `ENABLE_NON_ZERO_WINDING_RULE`.	55K
Font Fusion (including everything)	Includes the core and all options ON	240K
Cache Manager	Size of the Cache Manager component	4K
Font Manager	Size of the Font Manager component	8K
Options: Allocating Memory and Assert Statements	**Description**	**Size**
`CLIENT_MALLOC`	See “Allocating Memory” on page 29.	N/A
`CLIENT_FREE`	See “Allocating Memory” on page 29.	N/A
`CLIENT_REALLOC`	See “Allocating Memory” on page 29.	N/A
`CLIENT_ASSERT`	See “Assert Statements” on page 32.	N/A

ENABLE_CLIENT_ALLOC	Enable this macro to allow having third party allocator/re-allocator/de-allocator methods to handle allocation, re-allocation, and de-allocation of memory for the `tsiMemObject` and cache manager.	N/A
ENABLE_MEM_VALIDATION	Ensures memory validation. This macro is ON by default.	3K
Options: Library Functions	**Description**	**Size**
CLIENT_STRLEN	Returns the length of the string. Use if the default string function, `strlen()` is not available or needs to be changed for the target platform.	N/A‘
CLIENT_STRCMP	Compares the strings. Use if the default string function, `strcmp()` is not available or needs to be changed for the target platform.	N/A
CLIENT_STRNCMP	Compares at most count characters of two strings. Use if the default string function, `strncmp()` is not available or needs to be changed for the target platform.	N/A
CLIENT_SETJMP	Saves the stack context/environment in `env` for later use by `CLIENT_LONGJMP`. Use if the standard library function, `setjmp()` is not available or needs to be changed for the target platform.	N/A
CLIENT_LONGJMP	Restores the environment saved by the last call of `CLIENT_LONGJMP` with the corresponding `env` argument. Use if the standard library function, `longjmp()` is not available or needs to be changed for the target platform.	N/A
CLIENT_JMPBUF	Specifies the buffer used by the `CLIENT_SETJMP` and `CLIENT_LONGJMP` routines to save and restore the program environment. Use if the standard buffer, `jmp_buf` is not available or needs to be changed for the target platform.	N/A
Options: Layout & Kerning	**Description**	**Size**

ENABLE_LINE_LAYOUT	Enable if you plan to use T2K_FindKernPairs() or T2K_MeasureTextInX().	1K
ENABLE_KERNING	Enable if you plan to use T2K_FindKernPairs() or T2K_MeasureTextInX().	.75K
LAYOUT_CACHE_SIZE somesize	This option speeds up **T2K_MeasureTextInX()**. Enable only if you also enable both ENABLE_LINE_LAYOUT and ENABLE KERNING, and you plan to use **T2K_MeasureTextInX()**. Note that Font Fusion consumes eight times somesize for the cache.	N/A
Options: Algorithmic Styles	**Description**	**Size**
ALGORITHMIC_STYLES	Use with **FF_New_sfntClass()** to enable algorithmic styles, such as emboldening.	1.5K
Options: Emboldening	**Description**	**Size**
ENABLE_2D_EMBOLD	Enables the 2-degree emboldening along both x-and y- direction.	<1K
ENABLE_CHECK_CONTOUR_DIRECTION	Enable this macro to check the contour direction for glyph outlines. Use only when the contour directions are not verified or known. Enabling this macro adds to the performance cost of the rendering engine, where the performance loss incurred is about 6 - 10%.	<2K
ENABLE_POSTHINT_ALGORITHMIC_STYLE	Enables emboldening algorithmic style post hinting.	<1K
Options: Font Support	**Description**	**Size**
ENABLE_T1	Enable if you need Type 1 font support.	16.1K
ROM_BASED_T1	Disable this option to save memory if Type 1 fonts are not in ROM (to disable, comment the option out or remove it).	N/A
ENABLE_MAC_T1	Enable if you have also defined ENABLE_T1 and you need Macintosh Type 1 font support for the Macintosh platform.	not available

`ENABLE_CFF`	Enable if you need Compact Font Format(CFF)/Type 2 font support.	19.7K
`ENABLE_OTF`	Enable if you need OpenType Font (OTF) format support. OTF is an extension of TrueType, with additional tables and the ability to support a CFF table. In order to get full OpenType font support you must enable OTF and CFF. An OpenType font should have the .otf file extension if it has CFF data that is going to be used, but if not, for backwards-compatibility with older versions of the Windows operating system, it should have a .ttf file extension.	.35K
`ENABLE_ORION`	Enable if you need to support entropy-encoded Font Fusion fonts (i.e., compact Kanji fonts, not stroke-based fonts).	N/A
`ENABLE_T2KS`	Enable if you need to support stroke-based fonts.	11.5K
`ENABLE_SPD`	Enable if you need Bitstream Speedo™ font support.	10.0K
`ENABLE_PFR`	Enable if you need TrueDoc® PFR (portable font resource) font support.	16.8K
`ENABLE_WINFNT`	Enable if you need Windows FNT font support.	15K
`ENABLE_SBIT`	Enable if you need embedded bitmap font support. Currently, Font Fusion supports embedded bitmaps in TrueType, Native T2K, and TrueDoc PFR formats.	11.9K
`ENABLE_SBITS_TRANSFORM`	Enable to support the transformation (as *scaling/obliquing*) of bitmap fonts.	N/A
`ENABLE_SBITS_COMPRESSION`	Enable if you need compressed CJK bitmap font support.	N/A
`ENABLE_BDF`	Enable if you need Bitmap Distribution Format (BDF) font support.	28K
`COMPRESSED_INPUT_STREAM`	Enable if you intend to use compressed fonts and need compressed font processing.	4K

ENABLE_MAC_RFORK	Enable if you intend to include Mac font suitcase (Dfont support).	< 1K
Options: OpenType Features	**Description**	**Size**
ENABLE_OPENTYPE_VERT	Enables the GSUB OpenType `vert` feature that replaces default glyph forms with variants adjusted for vertical writing. Core Panorama OTF modules are included under an optional package to provide OpenType processing support. The current implementation supports OpenType `vert` GSUB feature, but can be extended to support other OpenType features as well.	20K
Options: Optimized PFRs	**Description**	**Size**
ENABLE_2DEGREE_OPTIMIZED_PFR	Enables the support for enhanced and optimized 2-degree PFR format.	<1K
ENABLE_PSEUDOFONT_SUPPORT	Enables the pseudo-italic faces support, where the pseudo-italic faces are rendered from corresponding Regular or Bold typefaces at a fixed oblique angle.	<1K
Options: Font Mapping	**Description**	**Size**
ENABLE_T1_FORCE_ENCODING	Enable to include the ability to force Type 1 font mapping to Standard or ISOLatin1 encoding.	5K
Options: Unicode Support	**Description**	**Size**
ENABLE_UNICODE_32_SUPPORT	Enable to support 32-bit TrueType character maps (format 8, 10 and 12).	5K
Options: Anti-Aliasing	**Description**	**Size**
ENABLE_GASP_TABLE_SUPPORT	Enable if you plan to use **`T2K_GaspifyTheCmds()`**.	.5K
Options: "Seat-Belts" Mode	**Description**	**Size**

`USE_SEAT_BELTS`	This option is a more secure mode of operation for supporting TrueType fonts that don't conform perfectly to the TrueType Font Specification. It is enabled by default. You may disable the seat belts option by commenting it out or removing it.	5K
Options: SmartScale	**Description**	**Size**
`ENABLE_SMARTSCALE`	Enable if you need the SmartScale feature that “fits” the glyph within fixed screen parameters.	2K
Options: Fractional Size	**Description**	**Size**
`ENABLE_FRACTIONAL_SIZE`	Enables the fractional size support. The enhanced light-weight, processor-friendly hinting process in fractional mode provides crisp character output.	<1K
Options: Accurate Font Metrics	**Description**	**Size**
`ENABLE_EXTRA_PRECISION`	Enable this option to use a more precise and accurate value for advance width. If enabled Font Fusion calculates the advance width with float-point arithmetic.	<1K
Options: Filter Functions	**Description**	**Size**
`ENABLE_UNDERLINEFILTER`	Enable this option to use the **`T2K_CreateUnderlineCharacter()`** function described on page 104.	4K
`ENABLE_OUTLINEFILTER`	Enable this option to use the **`T2K_CreateOutlineCharacter()`** function described on page 100.	<1K
`ENABLE_MULTIPLE_FILTERS`	Enable this option to use the multiple filter functions described in “Using Multiple Filters” on page 99.	< 1K
Options: Hinting	**Description**	**Size**
`ENABLE_NATIVE_TT_HINTS`	Enable if you need native TrueType hint support. Leave this option on if you plan to use extended LCD modes. This is enabled by default.	19.4K

`ENABLE_NATIVE_T1_HINTS`	Enable if you need native Type 1 hint support, or if you plan to use Type 1 fonts with extended LCD modes.	22.4K
`ENABLE_AUTO_GRIDDING`	Enable if you want run-time, auto-hinting (gridding) support. Do not enable this option if you only need `TV_MODE`, which is ideal for TV if you use integer metrics and grayscale. Size for TrueType	24.2K
	Size for Type 1	23.8
`ENABLE_AUTO_GRIDDING_CORE`	Enable only if you plan to use LCD modes 2-4 or TV mode. This option is not needed for extended LCD modes. If you use LCD modes 3 or 4, you must also turn on `ENABLE_NATIVE_TT_HINTS` for TrueType fonts, or `ENABLE_NATIVE_T1_HINTS` for Type 1 fonts. For best-quality output on TVs or LCD devices, use `T2K_TV_MODE_2`. For best-quality output on LCD devices, use `T2K_LCD_MODE_4`.	see above
Options: Curve Conversion	**Description**	**Size**
`ENABLE_FF_CURVE_CONVERSION`	Enable if you want to use the function **`T2K_ConvertGlyphSplineType()`**, documented in “void T2K_ConvertGlyphSplineType(” on page 103.	5.32K
`ENABLE_STRKCONV`	Set the default to 32 lines. This improves the performance of the stroke font processor at 32 lines and below. At or near 32 lines (but not above 32 lines), anomalies in quality appear because of the high-speed method that Font Fusion uses.	3.5K
Options: For Scan Converter	**Description**	**Size**
`ENABLE_DROPOUT_ADAPTATION`	Enable this option to have the best fit bitmap data in subsequent render calls. It enables the dropout control adaptation in scan converter. This should be by default turned ON unless exact bitmaps and metrics are desired in all render calls without any dropout adaptation.	<1K

`ENABLE_MORE_TT_COMPATIBILITY`	Enable this option to get more pixel-for-pixel compatibility with industry-standard scan converters used to render TrueType fonts in Windows and on the Macintosh. Only define this option if rendering TrueType fonts on black-and-white, low-resolution devices.	1.6K
`MAKE_SC_ROWBYTES_A_4BYTE_MULTIPLE`	Enable this option to get the scan converter to pad all bitmap rows to four-byte multiples.	N/A
`REVERSE_SC_Y_ORDER`	Enable this option to force the y-axis to go up, not down, in bitmaps.	N/A
Options: Non-RAM, -ROM Fonts	**Description**	**Size**
`ENABLE_NON_RAM_STREAM`	Enable if you need non-RAM or non-ROM resident fonts. This allows you to leave fonts on a disk, a server, and so on.	5.1k
Options: Output Modes	**Description**	**Size**
`ENABLE_LCD_OPTION`	Enable if you need to use the standard LCD modes (2-4). **NOTE:** Since the extended LCD modes require a different command option to be turned on, your application can use both, if necessary.	4.1K
`ENABLE_EXTENDED_LCD_OPTION`	Enable if you need to use the extended LCD modes (horizontal or vertical RGB or BGR).	8K
Options: Printer Fonts	**Description**	**Size**
`ENABLE_PCL`	Enable this option to process scalable Intellifont® fonts that have been downloaded to a Hewlett-Packard® printer or printer emulation as encapsulated outlines. Bitstream's font reader for this format is included in the source modules **`pclread.c`** and **`pclread.h`**. When using `ENABLE_PCL`, you need to write a callback function, **`eo_get_char_data()`**. See Chapter 6 for instructions.	10K

ENABLE_PCLETTO	Enable this option to process scalable TrueType fonts that have been downloaded to a Hewlett-Packard printer or printer emulation as encapsulated outlines. No additional font reader module is required. When using ENABLE_PCLETTO, you need to write a callback function, **tt_get_char_data()**. See Chapter 6 for instructions.	.8K
Options: Caching Macro	**Description**	**Size**
ENABLE_CACHING_EBLC	Enable this option to cache the entire Embedded Bitmap Location (EBLC) table into memory during the FF_New_sfntClass() call.	< 1K
ENABLE_32BIT_CACHE_TAG	Enables the 32-bit font code for cache.	N/A
Options: Cache Size	**Description**	**Size**
ENABLE_COMMON_DEFGLYPH	Enable this option to cache one default missing glyph character rather than separate instances for each missing character requested.	< 1K
ENABLE_CACHE_COMPRESSION	Enable this option to turn ON run-length encoding compression in the cache manager. Use the **FF_CM_SetCompDecomp()** function to use your own compression algorithms for the cache manager. See page 180 for details.	< 1K
ENABLE_CACHE_RESIZE	Enable this option to use the **FF_CM_Class *FF_CM_SetCacheSize()** function, documented on page page 174.	< 1K
Options: Error Handling	**Description**	**Size**
ENABLE_CLIENT_ERROR	Enable to make Font Fusion call the OEM implemented method before emergency shutdown.	N/A

Errors

This section discusses the following topics related to errors.

- What happens when Font Fusion returns an error
- What to do if Font Fusion returns an error
- Font Fusion objects you need to restart if Font Fusion returns an error

What Happens When Font Fusion Returns an Error

When a fatal error is encountered, Font Fusion returns an error code appropriately and exits without performing any other work, except for any cleanup it needs to do. Font Fusion automatically frees up all memory and deletes all of its objects when it encounters an error. All references to Font Fusion objects become invalid; they can no longer be used.

Font Fusion requests user intervention after an unexpected fatal error occurs, as the user needs to restart all the objects that shared the same the `tsiMemObject`.

NOTE: Please note that Font Fusion does not rely on asserts for fatal errors that occur at run-time, instead returns an error code value for the error encountered. The user thereafter is responsible to start over when such an error condition is observed.

Enable the macro `ENABLE_CLIENT_ERROR` to make Font Fusion call the OEM implemented method before emergency shutdown is called and all the FF internal components are deleted.

The macro `CLIENT_ERROR` gets defined in `config.h` to a method `void ClientError(void* memObject, int errorCode)`, when `ENABLE_CLIENT_ERROR` is enabled. The parameters `memObject` and `errorCode` refer to `tsiMemHandler` object and the error value set by Font Fusion respectively.

You can also define `CLIENT_ERROR` to your own error handler of same signature as `ClientError` or, simply define `ClientError` method inside your code. This method will always be called before the emergency shutdown and can be helpful in development process to keep a track of the critical errors.

What to DO if Font Fusion Returns an Error

You need to set all Font Fusion references to `NULL`. Do not call any Font Fusion delete routines or similar routines.

Basically, you have to start from the beginning again—as if the Font Fusion object no longer exists.

Font Fusion Objects You Need to Restart if Font Fusion Returns an Error

You do not have to restart all Font Fusion objects—just all the objects that shared the same `tsiMemObject`.

You should have one `tsiMemObject` per font.

NOTE: See Chapter 8 for a list of error codes and descriptions.

Font Size options

If you application has limited space for fonts, you can use **compact** font formats, such as PFRs (portable font resources) or T2K fonts, or you can use **compressed** fonts.

Compact Font Formats

PFR (portable font resource) and T2K are two compact font formats offered by Bitstream. Any font can be converted into a PFR. Contact a Sales representative for more information.

About Our PFRs

Bitstream stores resident font sets in portable font resource (PFR) format, because it is much faster and more compact than other formats. The PFR is a Bézier-based, hinted format. A PFR stores only one copy of each character image in a typeface, no matter how many character sets reference that character. A PFR also eliminates images shared among two or more typefaces. This compact storage method saves RAM and ROM space.

Ultra Compact PFRs

Our ultra compact format creates a PFR that is approximately 35%-40% smaller than our standard PFR format. The advantage to using ultra compact fonts is the smaller size, however, the disadvantage is slower performance. When rendering characters at low resolutions, such as small point sizes on a screen display, you might notice slowdowns. However, you will not notice much of a slowdown when rendering characters at higher resolutions, such as on a 600 dpi printer.

Optimized OEM PFRs

The new optimized OEM PFRs from Bitstream employs a series of processes including bucketing, pseudo-italic faces and 2-degree curves to deliver compact size PFRs without degrading the engine performance.

- **Bucketing**: Fonts in PFR contain similar characters used by multiple faces. The process of keeping a single copy of these similar characters and removing them from other faces is termed as bucketing. Three different types of bucketing is done on the PFRs: Universal bucketing, italic bucketing and sans-serif bucketing.
- **Pseudo-italic Faces**: The pseudo-italic faces are the regular or bold faces, with a fixed oblique angle set to render the faux-italic style. To ensure the correct rendering of these pseudo-italic fonts, appropriate `lsb` correction is applied to these rendered faces. Enable the compile conditional `ENABLE_PSEUDOFONT_SUPPORT` to include the pseudo-italic font support.
- **2-degree curves**: These compact PFRs utilize 2 degree curves which are optimized to contain lesser number of outline point, where the reduction of points does not perturbs the quality of outlines. Enable the compile conditional `ENABLE_2DEGREE_OPTIMIZED_PFR` to include the 2-degree PFR support.

Compressed Font Formats

Any unencrypted font can be compressed (Type 1 PFB fonts cannot be compressed because they are encrypted). There are two components to the compression technology:

- Creation—performed by Bitstream. Contact a Sales representative for more information.
- Reading and rendering—performed by Font Fusion through the standard `InputStream` functions.

Font Fusion does not distinguish between compressed and uncompressed fonts. The input stream automatically detects and processes compressed font files. Enable this code with the `COMPRESSED_INPUT_STREAM` compile-time option.

Font compression is useful for developers who are creating applications for small embedded systems and consumer electronics devices that must conserve ROM and RAM space. PFR (portable font resource) and OTF/TTF (OpenType and TrueType fonts) yield the best compression results.

Font Format Compression Ratios

Font Type	Ratio
Regular PFRs (single)	35-54%
Regular PFRs (multiple)	22-38%
Compact PFRs (multiple)	9-24%
Optimized OEM PFRs	8-21%
OTFs/TTFs (single)	26-53%
FFS (Font Fusion stroke-based fonts)	8-34%

Variance in compression is due to the number of characters in the font and the complexity of the typeface design. Bitstream compresses fonts in blocks of 1-32K bytes. Larger blocks may yield better compression, but also may require more dynamic memory to uncompress. For example, 32K blocks would require at least a 32K buffer.

Font Fusion also cannot do non-RAM compressed fonts. Fonts must be loaded into RAM and uncompressed. This includes ROM-based and RAM-based (disk-based) fonts, as these fonts also need to be uncompressed in RAM. Additionally, there may be a speed degradation when using compressed fonts, which must be uncompressed before they can be rendered.

Font Fusion Core API

3

tsi Functions: Overview

- tsi_NewMemhandler()
- tsi_DeleteMemhandler()

The tsiMemObject Object

`tsiMemObject` is an object that handles all memory allocation, de-allocation, and re-allocation. We do it this way, instead of making direct calls to the operating system, because `tsiMemObject` does a lot of error checking. This creates a more stable product.

For instance, `tsiMemObject` puts special markers both before and after allocated memory so that we can detect any attempts to write outside the allocated memory. It also detects any memory leaks and attempts to free already-freed memory. In this way, `tsiMemObject` provides a solid foundation for the product.

Using One tsiMemObject per Font

We could have shared the `tsiMemObject` among all open fonts, but we get better performance using only one `tsiMemObject` per font.

The current implementation of `tsiMemObject` also has a maximum limit on the number of pointers it can allocate.

Creating and Destroying a Memory Handle

The following code exemplifies creating and destroying a `Memhandler` object. (This is the “outermost” object.)

```
/* Create the Memhandler object. */
tsiMemObject *mem = NULL;
mem= tsi_NewMemhandler( &errCode );
assert( errCode == 0 );
/* Destroy the Memhandler object. */
tsi_DeleteMemhandler( mem );
```

tsi Functions

tsiMemObject *tsi_NewMemhandler(

```
int *errCode,
tsi_ClientAllocMethod allocPtr,
tsi_ClientDeAllocMethod freePtr,
tsi_ClientReAllocMethod reallocPtr,
void * clientArgs)
```

Arguments

`errCode` is a pointer to the returned error code.

`allocPtr` is a function pointer of (`tsi_ClientAllocMethod`) type which is the allocation method used by the client.

`freePtr` is a function pointer of (`tsi_ClientDeAllocMethod`) type which is the de-allocation method used by the client.

`reallocPtr` is a function pointer of (`tsi_ClientReAllocMethod`) type which is the re-allocation method used by the client.

`clientArgs` is of (`void *`) type which contains the arguments used by the client.

Description

`tsi_NewMemhandler()` creates an object to handle all memory allocation. You can allow a third party to create an object to handle all memory allocation/re-allocation/de-allocation by enabling the `ENABLE_CLIENT_ALLOC` macro.

It returns a context pointer, which is `NULL` on failure.

Here is a prototype of the `tsi_ClientAllocMethod` method data type:

```
typedef void * (* tsi_ClientAllocMethod) (size_t size, void
* clientArgs)
```

The above method takes two parameters; size of the memory needed and the client arguments. The function returns a pointer to the allocated memory.

Here is a prototype of the `tsi_ClientDeAllocMethod` method data type:

```
typedef void (*  tsi_ClientDeAllocMethod) (void * mem, void
* clientArgs)
```

The above method takes two parameters; a pointer to the memory and client arguments.

Here is a prototype of the `tsi_ClientReAllocMethod` method data type:

```
typedef void * (*  tsi_ClientReAllocMethod) (void * oldMem,
size_t newSize, void * clientArgs)
```

The above method takes three parameters; an old memory pointer , new size of the memory needed, and the client arguments. The function returns a pointer to the new allocated memory.

NOTE: Please note that the **`allocPtr`**, **`freePtr`**, **`reallocPtr`**, and **`clientArgs`** parameters are only applicable when the macro **`ENABLE_CLIENT_ALLOC`** is ON.

void tsi_DeleteMemhandler(

`tsiMemObject *t)`

Arguments

`t` is a pointer to the `tsiMemObject`.

Description

`tsi_DeleteMemHandler()` destroys the memory object you created with **`tsi_NewMemhandler()`**.

InputStream Functions: Overview

- New_InputStream3()
- New_InputStream()
- New_NonRamInputStream()
- PF_READ_TO_RAM()
- Delete_InputStream()

The InputStream Object

The `InputStream` object provides a level of abstraction for the core by exposing certain methods that Font Fusion uses to access the data. This means that Font Fusion does not need to know if the data is in memory, on a disk, on a network, etc. It also provides more robustness. For instance, the `InputStream` object checks for out-of-bounds read attempts. This error checking, together with the abstraction `InputStream` provides, produces a more solid and robust design and therefore a better product.

Creating an Input Stream

The following code exemplifies creating and destroying `Memhandler` and `InputStream` objects.

```
/* Create the Memhandler and InputStream objects. */
tsiMemObject *mem;
unsigned char *data;
unsigned long length;
int *errCode;
tsiMemObject *mem = NULL;
InputStream *in = NULL;
mem= tsi_NewMemhandler( &errCode );
assert( errCode == 0 );
  in = New_InputStream3( mem, data, length, &errCode );
  assert( errCode == 0 );

/* Destroy the InputStream object. */
Delete_InputStream( in, &errCode  );

/* Destroy the Memhandler object. */
```

```
    tsi_DeleteMemhandler( mem );
```

If You Have a lot of Fonts Open and Active at the same Time

This should not cause any problems. You should have one `tsiMemObject` per font.

You can choose to keep multiple fonts open simultaneously. You can also decide to use one or more Font Fusion scalers simultaneously. Or you could decide to only open one font at a time.

You can also use memory-based fonts when you create the `InputStream` class or you can use disk-based fonts.

These choices trade off memory use and speed. For instance, disk-based fonts do not require any memory allocation but take longer to access.

Using Your Own Memory Allocator and De-allocator with Font Fusion

`config.h` allows you to remap allocation, re-allocation, and deletion to anything you want. Refer to the first three definitions in `config.h`.

InputStream Functions

InputStream *New_InputStream3(

```
   tsiMemObject *mem,
   unsigned char *data,
   unsigned long length,
   int *errCode)
```

Arguments

`mem` is a pointer to the `tsiMemObject`.

`data` is a pointer to your font data.

`length` is the length of the font data.

`errCode` is a pointer to the returned error code.

Description

`New_InputStream3()` is a pointer to an `InputStream` object, i.e., your font data. Use it if reading data from memory. Your application finds the memory for the font, and deletes the memory when it is done with the `InputStream`.

Recommendation: Use NewInputStream3()

You should normally use **`NewInputStream3()`**. This way, your application finds the memory for the font, and deletes the memory when it is done with the `InputStream`. Also note that if the font is in ROM, no allocation or de-allocation is involved, so this method is the only one that you can use.

InputStream *New_InputStream(

```
   tsiMemObject *mem,
   unsigned char *data,
   unsigned long length,
   int *errCode)
```

Arguments

`mem` is a pointer to the `tsiMemObject`.

`data` is a pointer to your font data.

`length` is the length of the font data.

`errCode` is a pointer to the returned error code.

Description

`New_InputStream()` is a pointer to an `InputStream` object, i.e., your font data. Use it if reading font data from memory.

NOTE: You should only use **`New_InputStream()`** if you used the class `tsiMemObject` to allocate the memory for the font data. Typically, only Font Fusion should be using the `tsiMemObject` class. In any case, if your application uses **`New_InputStream()`**, then Font Fusion deletes the memory that the font uses with a method in `tsiMemObject`.

InputStream *New_NonRamInputStream(

```
   tsiMemObject *mem,
   void *nonRamID,
   PF_READ_TO_RAM readFunc,
   unsigned long length,
   int *errCode)
```

Arguments

`mem` is a pointer to the `tsiMemObject`.

`nonRamID` is a pointer to help your application identify and find the right font. In the example in the section “void PF_READ_TO_RAM(” on page 57, it is a pointer to a file `(FILE *)`, but we defined it in the example as a pointer to a void

`(void *)` so that it can point to anything. We pass this pointer to your **`PF_READ_TO_RAM()`** function so that your application can locate the correct font.

`readFunc` is a pointer to a function described below.

`length` is the length of the font data.

`errCode` is a pointer to the returned error code.

Description

`New_NonRamInputStream()` is a pointer to an `InputStream` object, i.e., your font data. Use it if reading data from ROM, disk, or a remote server. Make sure you define `ENABLE_NON_RAM_STREAM`, as in:

```
#ifdef ENABLE_NON_RAM_STREAM
InputStream *New_NonRamInputStream( tsiMemObject *mem, void
*nonRamID,
   PF_READ_TO_RAM readFunc, unsigned long length, int
*errCode );
#endif
```

void PF_READ_TO_RAM(

```
void *id,
uint8 *dest_ram,
unsigned long offset,
long numBytes)
readFunc
```

Arguments

`id` is a pointer to an id.

`dest_ram` is a pointer to the memory where the function needs to write the font data.

`offset` is the offset in bytes from the beginning of the font data to the data we need to retrieve.

`numBytes` is the number of bytes we need to retrieve starting at the above offset.

`readFunc` is a pointer to a function for reading font data from ROM, disk, or a remote server. Use it with the **New_NonRamInputStream()** function.

Sample Code

```
#ifdef ENABLE_NON_RAM_STREAM
typedef int (*PF_READ_TO_RAM) ( void *id, uint8 *dest_ram,
unsigned long offset, long numBytes );
#endif

#ifdef JUST_AN_EXAMPLE_OF_PF_READ_TO_RAM
int ReadFileDataFunc( void *id, uint8 *dest_ram, unsigned
long offset, long numBytes )
{
   int error;
   size_t count;
   FILE *fp = (FILE *)id;
   assert( fp != NULL );
   /* A real version of this function should only, for
example,  * call fseek if there is a need */
   error   = fseek( fp, offset, SEEK_SET );
   assert( error == 0 );
   count   = fread( dest_ram, sizeof( char ), numBytes, fp
);
   assert( ferror(fp) == 0 && count == (size_t)numBytes );
   return (ferror(fp) == 0 && count == (size_t)numBytes ) ?
0 : -1;
}
#endif
```

void Delete_InputStream(

```
InputStream *t,
int *errCode)
```

Arguments

`t` is a pointer to the `InputStream` object.

`errCode` is a pointer to the returned error code.

Description

`Delete_InputStream()` destroys the `InputStream` object you created with **`New_InputStream()`**.

sfntClass Functions: Overview

- FF_New_sfntClass()
- FF_Delete_sfntClass()

The sfntClass Object

The `sfntClass` is an internal class that represents a font. All supported font formats share it.

ALGORITHMIC_STYLES

The compile-time option `ALGORITHMIC_STYLES` enables algorithmic styling.

The sixth parameter to **`FF_New_sfntClass()`**, `T2K_AlgStyleDescriptor *styling` is normally set to `NULL`. But if you enable `ALGORITHMIC_STYLES`, you can set it equal to an algorithmic style descriptor.

Here is an example using the algorithmic emboldening that Font Fusion provides.

```
style.StyleFunc= tsi_SHAPET_BOLD_GLYPH;
style.StyleMetricsFunc=tsi_SHAPET_BOLD_METRICS;
style.params[0] = 5L << 14;
sfnt0 = FF_New_sfntClass( mem, fontType, 0, in, NULL, &style,
&errCode );
```

You can also write your own outline-based style modifications and use them instead of the algorithmic emboldening that Font Fusion provides. Just model them after the code for algorithmic emboldening in `shapet.c`.

Baseline shift for 2D emboldened characters

On applying 2-degree emboldening style to glyph shapes, the outlines are spread along both x and y-directions. This results in uneven baselines that may produce visually disjoint glyphs. Font Fusion overcomes this uneven baseline problem by including the API `T2K_SetBaselineShift()` that takes into account this delta shift and returns the glyph outline after baseline corrections.

Baseline shifting for 2D post-hint emboldened characters is enabled by default.

T2K_SetBaselineShift(

```
T2K * t,
int bSet)
```

Arguments

`t` is the pointer to the `t2k` object itself.

`bSet` is the parameter to enable or disable the baseline shift option. Set the parameter to nonzero if you want to enable the baseline correction for posthint emboldened output.

Description

Call the `T2K_SetBaselineShift()` API after creating `t2k` object.

If the parameter `bSet` is set to a nonzero value in `T2K_SetBaselineShift()`, the function `tsi_SHAPET_BOLD_GLYPH()` returns the glyph outline after baseline correction.

sfntClass Functions

sfntClass *FF_New_sfntClass(

```
tsiMemObject *mem,
short fontType,
long fontNum,
InputStream *in1,
InputStream *in2,
T2K_AlgStyleDescription styling,
int *errCode)
```

Arguments

`mem` is a pointer to the `tsiMemObject`.

`fontType` denotes the type of font. You can set the `fontType` to the font types displayed in the table below, as defined in `truetype.h`. Or you can call **`FF_FontTypeFromStream()`** to automatically set it based on automatic sniffing of the input stream data. This function is defined in `t2k.h`.

fontType	Description
FONT_TYPE_1	Use for Type 1 fonts.
FONT_TYPE_2	Use for Type 2/CFF fonts.
FONT_TYPE_TT_OR_T2K	Use for TrueType or T2K fonts, such as Font Fusion.
FONT_TYPE_PFR	Use for TrueDoc portable font resources (PFR)
FONT_TYPE_SPD	Use for Bitstream Speedo fonts.
FONT_TYPE_OTF	Use for OpenType fonts, both TrueType/OpenType (TTF/OTF) and Adobe compressed (CFF/OTF) font format data.

`fontNum` is the logical font number.

`in1` is a pointer to the primary `InputStream` object.

`in2` is a pointer to the secondary `InputStream` object.
This is usually == `NULL`.

`styling` is a pointer to a function that modifies the outlines algorithmically. This is normally == `NULL`.

`errCode` is a pointer to the returned error code.

Description

`FF_New_sfntClass()` is a pointer to a new Font Fusion font (`sfntClass`) object that this function creates.

`void FF_Delete_sfntClass(`

```
sfntClass *t,
int *errCode)
```

Arguments

`t` is a pointer to the `sfntClass` object.

`errCode` is a pointer to the returned error code.

Description

`FF_Delete_sfntClass()` destroys the Font Fusion font (`sfntClass`) object you created with **`FF_New_sfntClass()`**.

PlatformID Functions: Overview

- **`Set_PlatformID()`**
- **`Set_PlatformSpecificID()`**

The PlatformID

The platform identifier code defines the computer platform which will use the font. The table below shows a list of Platform IDs:

Platform ID	Platform Name	Description
0	Unicode	Unicode version.
1	Macintosh	Script Manager code.
2	reserved	Reserved, not currently used.
3	Microsoft	Microsoft encoding.

Setting the Platform and Platform-Specific ID

Here are two optional functions to set the preferred platform and platform-specific ID.

```
/* Invoke right after NewT2K(), t is of type (T2K *) */
#define Set_PlatformID( t, ID ) ((t)->font->prefered-
PlatformID = (ID))
#define Set_PlatformSpecificID( t, ID ) ((t)->font-
>preferedPlatformSpecificID = (ID))
```

Mapping Table to Use with TrueType and Native T2K Fonts

Use the functions **`Set_PlatformID( scaler, ID )`** and **`Set_PlatformSpecificID( scaler, ID )`** with TrueType and native T2K fonts.

To use the Unicode mapping that Windows uses, include these arguments:

```
Set_PlatformID( scaler, 3 )
Set_PlatformSpecificID( scaler, 1 )
```

You can insert the code right after the **`NewT2K()`** constructor.

Getting the Font Name

To get the font name, call **`T2K_SetNameString()`** after you call **`Set_PlatformID()`** and **`Set_PlatformSpecificID()`**.

The **`T2K_SetNameString()`** function sets the values for the public fields `nameString8` or `nameString16` in the T2K object(structure).

To use Microsoft Unicode mapping and names, include these arguments:

```
/* Use 3,1 to pick Microsoft Unicode character mapping */
Set_PlatformID( scaler, 3 );
Set_PlatformSpecificID( scaler, 1 );
/* Pick American English and the full font name */
T2K_SetNameString( scaler, 0x0409, 4 );
```

PlatformID Functions

Set_PlatformID(

```
T2K *t2kScaler,
uint16 ID)
```

Arguments

`t2kScaler` is a pointer to the `T2K` scaler object, an instance of the font scaler.

`ID` is the platform ID for accessing character map tables. It is the same as what Microsoft's TrueType documentation specifies. See "Mapping Table to Use with TrueType and Native T2K Fonts" on page 65 for more information.

Description

`Set_PlatformID()` sets the platform ID for accessing character map (`cmap`) tables in TrueType and native T2K fonts. The default platform ID is the first one in the font, usually platform ID 1, platform specific ID 0. Call this function to change to another platform ID.

Set_PlatformSpecificID(

```
T2K *t2kScaler,
uint16 ID)
```

Arguments

`t2kScaler` is a pointer to the `T2K` scaler object, an instance of the font scaler.

`ID` is the platform-specific ID for accessing character map tables. It is the same as what Microsoft's TrueType documentation specifies. See "Mapping Table to Use with TrueType and Native T2K Fonts" on page 65 for an example.

Description

`Set_PlatformSpecificID()` sets the platform-specific ID for accessing `cmap` tables in TrueType and native T2K fonts. The default platform ID is the first one in the font, usually platform ID 1, platform specific ID 0. Call this function to change to another platform-specific ID.

T2K Functions: Overview

The following functions are part of the Font Fusion T2K core:

- DeleteT2K()
- FF_T2K_Core_FilterReference()
- NewT2K()
- T2K_ConvertGlyphSplineType()
- T2K_CreateUnderlineCharacter()
- T2K_FindKernPairs()
- T2K_FontSbitsAreEnabled()
- T2K_FontSbitsExists()
- T2K_GaspifyTheCmds()
- T2K_GetBytesConsumed()
- T2K_GetGlyphIndex()
- T2K_GetIdealLineWidth()
- T2K_GlyphSbitsExists()
- T2K_LayoutString()
- T2K_MeasureTextInX()
- T2K_MultipleFilter()
- T2K_MultipleFilter_Add()
- T2K_MultipleFilter_Delete()
- T2K_MultipleFilter_Init()
- T2K_NewTransformation()
- T2K_PurgeMemory()
- T2K_RenderGlyph()
- T2K_SetNameString()
- T2K_TransformXFunits()
- T2K_TransformYFunits()

The T2K Scaler Object

The T2K scaler object represents an instance of the font scaler. The main task of the font scaler is to produce good looking bitmap images for characters at different sizes and transformations, such as rotated characters. To use it, you first need to set the transformation with **`T2K_NewTransformation()`**. Basically, you specify:

- the `x` and `y` resolutions
- a `2*2` transformation matrix (includes the point size)
- a true or false setting to enable or disable embedded bitmaps

Then you call **`T2K_RenderGlyph()`** to actually get outline or bitmap data. After you are done with the output data, you need to call **`T2K_PurgeMemory()`** to free up memory.

Modifying the Transformation Matrix

You can modify the transformation matrix to make algorithmic italics (obliquing) or rotate characters.

Making Algorithmic Italics (Obliquing Text)

Before calling **`T2K_NewTransformation()`**, set the transformation matrix this way:

```
trans.t00 = ONE16Dot16 * size;
trans.t01 = ONE16Dot16 * sin(italic_angle) * size;
trans.t10 = 0;
trans.t11 = ONE16Dot16 * size;
```

`size` is a number, such as 16.

`italic_angle` is a number, such as 12.0 degrees. In this case, `ONE16Dot16 * sin( 12.0 )` is 13626.

Examples of Transformations

Typically, you have the following:

```
trans.t00 = size;
trans.t01 = 0;
trans.t10 = 0;
trans.t11 = size;
```

`size` is a fractional number in `16.16` format.

To condense the text in the `x`-direction to 80 percent, use the following. We do not promote condensing text, since there are condensed fonts designed that way, but the following example shows you how to do it.

```
trans.t00 = util_FixMul( size, 8*0x10000/10 );
trans.t01 = 0;
trans.t10 = 0;
trans.t11 = size;
```

`size` is a fractional value in `16.16` format.

To stretch the text in the `y`-direction to 125 percent, use the following. We do not promote extending text, but the following example shows you how to do it.

```
trans.t00 = size;
trans.t01 = 0;
trans.t10 = 0;
trans.t11 = util_FixMul( size, 125*0x10000/100 );
```

`size` is a fractional value in `16.16` format.

Rotating Text

To rotate the text at an angle, alpha-measured **clockwise** from the `x`-axis, use the following. If the angle is in the first quadrant, it is negative.

```
trans.t00 = util_FixMul( size , cosvalue );
trans.t01 = util_FixMul( size , sinvalue );
trans.t10 = util_FixMul( size , -sinvalue );
trans.t11 = util_FixMul( size , cosvalue );
```

`size` is a fractional value in `16.16` format. The `cosvalue` and `sinvalue` above are `cos(angle)` and `sin(angle)` in `16.16` format.

T2K_RenderGlyph(): Getting Bitmap and Outline Output

This section discusses the following topics related to monochrome and grayscale output, as well as output for a TV.

- Grayscale
- The difference between `T2K_GRID_FIT` and `T2K_NAT_GRID_FIT`
- `T2K_TV_MODE`
- The Difference Between T2K_TV_MODE_2 and T2K_TV_MODE
- Driving color LCD displays

Grayscale

For best quality, use grayscale whenever possible.

Getting Grayscale or Monochrome Output

To get grayscale or monochrome output, set the fifth argument, `greyScaleLevel`, in **`T2K_RenderGlyph()`**.

For monochrome output, set `greyScaleLevel` to `BLACK_AND_WHITE_BITMAP`.

Scalar -> baseAddr 8 pixels per byte.

For grayscale output, set `greyScaleLevel` to `GREY_SCALE_BITMAP_HIGH_QUALITY`.

Scalar -> baseAddr 1 pixel per byte.

NOTE: Do Not Edit T2K_BLACK_VALUE and T2K_WHITE_VALUE to Get a Different Range of Grayscale Values

You should not edit anything in `t2k.h`. The values are there so that you can put an “assert” statement in your code to automatically detect whether or not Bitstream changes the values in the future.

The Difference Between T2K_GRID_FIT and T2K_NAT_GRID_FIT

`T2K_GRID_FIT` allows you to apply run-time auto-hinting/gridding to your character images. Note that this slows down Font Fusion. `T2K_NAT_GRID_FIT` is much faster.

T2K_TV_MODE

T2K_TV_MODE improves character rendering when you do **not** use T2K_GRID_FIT (run-time auto-hinting/gridding) or T2K_NAT_GRID_FIT, but you do use integer metrics and grayscale (e.g., for a TV screen).

T2K_TV_MODE adjusts the white space around the character and also ensures that you get left-to-right grayscale symmetry for simple characters.

If you use T2K_TV_MODE, turn off T2K_GRID_FIT and T2K_NAT_GRID_FIT.

You should use T2K_TV_MODE when the following conditions are true:

- You do not want to use T2K_GRID_FIT (run-time auto-hinting/gridding) or T2K_NAT_GRID_FIT.
- You are using grayscale.
- You are using integer metrics (only one version of each character per size), not fractional positioning with fractional metrics.

However, note that for Latin fonts, T2K_TV_MODE_2 normally looks better than T2K_TV_MODE.

Improving Output on an Interlaced TV Device

To improve output on an interlaced TV, first turn off grid-fitting. This gives you an image with smoother transitions. This also speeds up Font Fusion.

Next, turn on T2K_TV_MODE if you use integer metrics. Also, do **not** use fractional pixel positioning to improve the quality, because this will affect the left-to-right symmetry of characters.

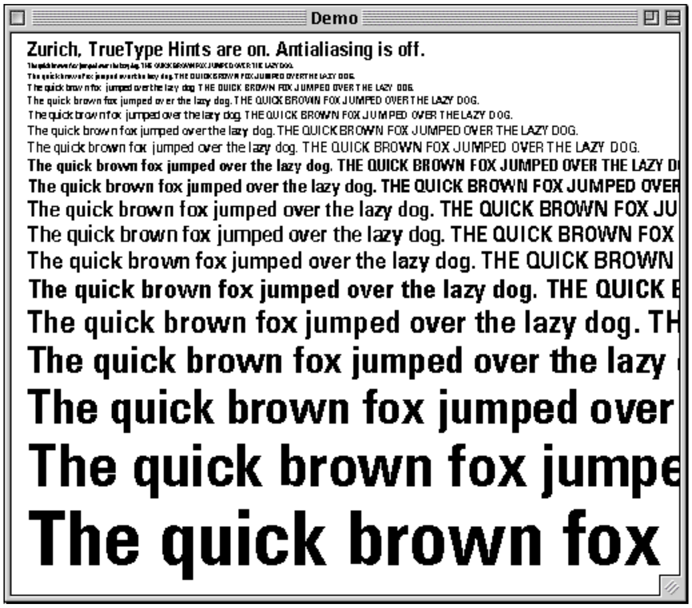
Then experiment with a simple filter to make the image more blurry. A simple 3*3 convolution is probably sufficient. You should probably average more in the y-direction than in the x-direction so as to avoid the interlacing flicker. But your hardware may have this capability already built into it.

The Difference Between T2K_TV_MODE_2 and T2K_TV_MODE

The _2 modifier in T2K_TV_MODE_2 activates a lightweight “y”hint strategy that improves the quality of the output. For instance, it makes symmetrical the anti-aliased bitmap pattern on the top and bottom of a lowercase “o”.

We recommend using T2K_TV_MODE_2 to get the best-quality output on TV devices and one of the extended LCD modes to get the best-quality output on

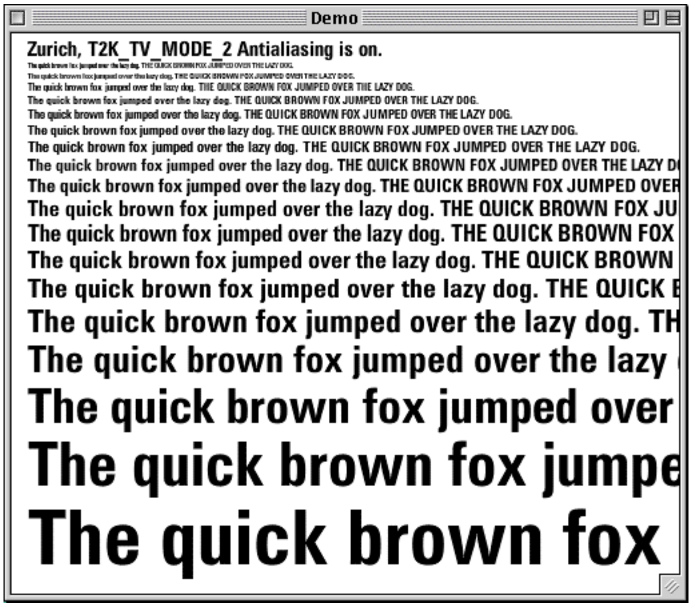
LCD devices, such as LCD TVs. For example, compare the figures below, which show no anti-aliasing and T2K_TV_MODE_2. For optimal quality, the regular TV modes already make characters left-to-right symmetrical.



Demo

Zurich, TrueType Hints are on. Antialiasing is off.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY D
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JU
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX
The quick brown fox jumped over the lazy dog. THE QUICK BROWN
The quick brown fox jumped over the lazy dog. THE QUICK B
The quick brown fox jumped over the lazy dog. TH
The quick brown fox jumped over the lazy
The quick brown fox jumped over
The quick brown fox jumpe
The quick brown fox

No anti-aliasing.



Demo

Zurich, T2K_TV_MODE_2 Antialiasing is on.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER THE LAZY D
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JUMPED OVER
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX JU
The quick brown fox jumped over the lazy dog. THE QUICK BROWN FOX
The quick brown fox jumped over the lazy dog. THE QUICK BROWN
The quick brown fox jumped over the lazy dog. THE QUICK B
The quick brown fox jumped over the lazy dog. TH
The quick brown fox jumped over the lazy
The quick brown fox jumped over
The quick brown fox jumpe
The quick brown fox

T2K_TV_MODE_2, anti-aliasing is on.

LCD Modes

Font Fusion features two types of LCD modes, standard and extended. The standard and extended LCD modes work independently from one another, so you can run both at the same time if your application requires it. We recommend using extended LCD modes to get the best-quality output on LCD devices, especially color LCD devices (color LCDs must have the RGB color orientation).

In the LCD mode, an indexed bitmap is returned. Font Fusion uses the index to find the right color in the lookup table provided. You can recompute the color lookup table for any foreground and background color. Typically there is a black foreground color and a white background color.

Standard LCD Modes

For optimal quality, standard LCD modes make characters left-to-right symmetrical. The standard LCD modes are described below.

- `T2K_LCD_MODE`, which is the basic LCD mode.
- `T2K_LCD_MODE_2`, which activates a lightweight "y" hint strategy that improves the quality of the output. For instance, it makes the anti-aliased bitmap pattern symmetrical on the top and bottom of a lowercase "o".
- `T2K_LCD_MODE_3`, which relies on hints automatically generated at run-time.
- `T2K_LCD_MODE_4`, which uses native hints. Native hinting makes this mode faster than mode 3.

Extended LCD Modes

There are four extended LCD modes for the different orientations of LCD screens:

- Horizontal RGB (`T2K_EXT_LCD_H_RGB`)
- Horizontal BGR (`T2K_EXT_LCD_H_BGR`)
- Vertical RGB (`T2K_EXT_LCD_V_RGB`)
- Vertical BGR (`T2K_EXT_LCD_V_BGR`)

To use an LCD Mode

1 For standard LCD mode, turn on the ENABLE_LCD_OPTION macro in config.h. For extended LCD modes, also turn on the ENABLE_EXTENDED_LCD_OPTION macro in config.h.

2 Specify the desired “cmd” argument for either standard or extended LCD mode.

3 Be sure that the ENABLE_NATIVE_TT_HINTS macro is on (it is on by default).

4 Turn on the ENABLE_NATIVE_T1_HINTS macro if you are using Type 1 fonts.

Driving Color LCD Displays

1 Use **FF_NewColorTable** to get the RGB colors for LCD display. These colors will be indexed by any bitmap produced by T2K. If your platform is using a Color Lookup Table, you will need to set these colors in that table. This is how you extract the actual RGB colors from the T2K color table:

```
ff_ColorTableType *pColorTable;
```

For black text on white Set Rb = Gb = Bb = 0xff, and Rf = Gf = Bf = 0.

```
pColorTable = FF_NewColorTable( mem, Rb, Gb, Bb, Rf, Gf, Bf );
```

The call above returns the color for all the indices in the bitmap. pColorTable->N will contain the specified number of elements in the array, and pColorTable->ARGB[0] contains the first ARGB value.

```
ARGB = pColorTable->ARGB[ byte index from the bitmap ];
B = (ARGB & 0xff); ARGB >>= 8;
G = (ARGB & 0xff); ARGB >>= 8;
R = (ARGB & 0xff);
```

When this is complete, free up the color-table, but do not call this per-character for speed reasons.

```
FF_DeleteColorTable( mem, pColorTable);
```

2 Do not invoke either **FF_SetBitRange255()** or **FF_SetRemapTable()**. If you need to shift the range, we recommend using a filter function.

3 Invoke **T2K_NewTransformation()** to set the transformation matrix.

NOTE: For standard LCD modes, set the xRes to three times the yRes, since LCD screens contain three times as many colored pixels in the x direction as in the y direction. Setting these resolutions in this way tells T2K we have a non-square aspect ratio where the x resolution is three times higher than the y resolution. You do not need to do this for the extended modes as these modes handle the conversion internally.

4 In the `cmd` parameter, turn on the bit flag to the LCD mode option you want to use, for example, `T2K_EXT_LCD_H_RGB` to use the most common extended LCD mode.

5 In the greyScalelevel parameter to the **`T2K_RenderGlyph()`** function, set `GREY_SCALE_BITMAP_HIGH_QUALITY` in the `greyScalelevel` parameter .

You now have an indexed color bitmap. When you draw the bitmap you need to take into account that the bitmap contains indices for the colored pixels.

T2K_RenderGlyph(): Hinting

This section discusses the following topics related to hinting.

- When to turn hinting on and off
- Turning run-time hinting on and off
- Supporting native TrueType hinting in Font Fusion

When to Turn Hinting on or off

For monochrome output, we recommend that you turn hinting on.

For a high-quality display device, such as a computer monitor, we also recommend that you turn hinting on.

For a low-quality display device, such as a TV monitor, you should turn hinting off.

Turn Run-Time Hinting on or off

You control hinting through the sixth argument, `T2K_GRID_FIT`, of **`T2K_RenderGlyph()`**.

You enable it by turning on the `T2K_GRID_FIT` bit.

You disable it by turning off the `T2K_GRID_FIT` bit.

Supporting Native TrueType Hinting in Font Fusion

You need these additional `.c` and `.h` files:

- `fnt.c`
- `fnt.h`
- `t2ktt.c`
- `t2ktt.h`

You also need to `#define ENABLE_NATIVE_TT_HINTS` in `config.h`.

In addition, you need to turn on `T2K_NAT_GRID_FIT` (native grid-fitting) in the `cmd` argument to **`T2K_RenderGlyph()`**.

T2K_RenderGlyph(): Rendering Characters and Strings

This section discusses the following topics related to rendering characters and character strings.

- Accessing bitmap data after calling **`T2KRenderGlyph()`**
- Outline winding direction
- Leaving `USE_NON_ZERO_WINDING_RULE` on
- When making any white-space character, **`T2K_RenderGlyph()`** returns `NULL` for `baseAddr`
- Getting outline spline data
- Handling third- degree Bézier curves when `t2k->glyph->curveType == 2`
- Making a colored, bordered character

Accessing Bitmap Data After Calling T2K_RenderGlyph()

You access bitmap data through public fields in the `T2K` class. To begin, define the following fields:

```
/* Begin bitmap data */
long width, height;
F26Dot6 fTop26Dot6, fLeft26Dot6;
long rowBytes;
unsigned char *baseAddr;
/* unsigned char baseAddr[N], N=t->rowBytes * t->height */
uint32 *baseARGB;
/* End bitmap data */
```

`baseAddr` is either a bit array or a byte array. `baseARGB` is a 32-bit array (`ARGB`). Note that the color filter border uses `baseARGB`, but baseARGB is also envisioned for future use for outputting color fonts.

Outline Winding Direction

The outline winding direction matters because the run-time-hinting process uses this information to determine where the black and white areas are.

For TrueType and native T2K fonts, the black (inside) area should be on the right if you follow a contour in the direction of increasing point numbers.

For Type 1 fonts, the black (inside) area should be on the left if you follow a contour in the direction of increasing point numbers.

TrueType and native T2K outlines need to use the correct winding direction. Type 1 outlines should also use the correct winding direction, which is the opposite of TrueType and native T2K outlines.

Leaving USE_NON_ZERO_WINDING_RULE on

The `USE_NON_ZERO_WINDING_RULE` option determines what kind of fill rule the Font Fusion scan converter uses. We recommend that you leave this setting on.

This enables a non-zero winding rule. Otherwise, the scan converter uses an even-odd filling rule. For example, the even-odd filling rule turns an area where two strokes overlap—which is rare—into white, but the non-zero winding rule keeps such areas black.

For an embedded system where the fonts are well built and you do not have overlapping strokes, you get a small increase in speed—probably less than 1%—by disabling this.

When Making any White-Space Character, T2K_RenderGlyph() Returns NULL for baseAddr

You do not have to check for the existence of white-space characters—i.e., any glyph without pixels—and advance the `x` position accordingly, because there is no bitmap to draw! Just check for `baseAddr == NULL` instead.

You also need to check for the following:

```
baseARGB == NULL && baseAddr == NULL
```

Getting Outline Spline Data

To get outline spline data, turn on the `T2K_RETURN_OUTLINES` option for the `cmd` argument of **`T2K_RenderGlyph()`**. This sets the public field `glyph` in Font Fusion.

```
/*** Begin outline data */
GlyphClass *glyph;
/*** End outline data */
```

`GlyphClass` includes public fields with the outline data.

Here are the relevant fields in `GlyphClass`:

```
 /* For curveType, use 2 for TrueType (second-degree B-spline)
outlines, and use 3 for Type 1 (third-degree Bézier) */
short curveType;
/* Number of contours in the character */
short contourCount;
/* Number of points in the characters, plus zero for sidebearing
points */
short pointCount;
/* sp[contourCount] start points */
int16 *sp;
 /* ep[contourCount] end points */
int16 *ep;
 /* oox[pointCount] unscaled unhinted points. Add two extra points
for lsb, and rsb */
int16 *oox;
/* ooy[pointCount] unscaled unhinted points. Set y to zero for the
two extra points */
/* Do NOT include the two extra points in sp[], ep[], contourCount,
and do NOT include the two extra points in pointCount */
int16 *ooy;
/* onCurve[pointCount] indicates if a point is on or off the curve.
It should be true or false */
uint8 *onCurve;
/* The actual points in device coordinates */
F26Dot6 *x, *y;
```

The character is made out of `contourCount` contours.

The outline coordinates are stored in `F26Dot6` format in the `x` and `y` arrays (six fractional bits).

Each contour starts with the point number `sp[contour]`, and ends with point number `ep[contour]`.

`sp[0]` should typically be zero. The letter A typically has 2 contours, B has 3, C has 1, etc. The contours are self closing.

Each point is either an “on” or “off” curve point. A third-degree Bézier is on, off, off, on. A second-degree parabola is on, off, on. So a particular point `n` is described by `x[n]`, `y[n]`, `onCurve[n]`.

Note that a second-degree b-spline curve allows many consecutive “off” curve points.

Handling Third-Degree Beziér Curves when t2k->glyph->curveType == 2

First, make sure you have turned on the `T2K_RETURN_OUTLINES` bit flag to **`T2K_Render_Glyph()`** and that you are using the `t2k->glyph` structure.

We also encourage you to see if your current code can handle the second-degree curves directly, since you can render them more quickly than third-degree curves.

If not, this is how you go between them:

You need to find all straight lines and parabolas. The function **`Make2ndDegreeEdgeList()`** from `t2ksc.c` shows you how to do that.

Now each parabola, which is equal to a second-degree Beziér curve, is described by the points `A,B,C`. Each third-degree Beziér curve is described by points `P1,P2,P3,P4`. Map points `[A,B,C]` to `[P1,P2,P3,P4]` as follows:

```
P1 = A;
P2 = (2B + A)/3;
P3 = (2B + C)/3;
P4 = C;
```

Making a Colored, Bordered Character

Invoke the filter function **`T2K_CreateBorderedCharacter()`**, enabled in the `t2kextra.c` file. See “How to write a filter function” on page 90 for more information. After this you can find the 32-bit colored, bordered character in `t2kscaler->baseARGB`. The format is `ARGB`, 8 bits each.

T2K_RenderGlyph(): Sample Code for Rendering Characters and Strings

This section discusses the following topics related to rendering characters and character strings.

- Drawing a character
- Drawing a character string using **`MyDrawCharExample()`**
- Sample code for evaluating outline data

Drawing a Character

Here is an example. Note that this example is for drawing characters on the Macintosh and has not been optimized at all for performance.

```
/* Simple example that shows how to get bitmap data from the T2K
scaler object. The example assumes you are using a screen-
coordinate system where the upper left position is 0,0. */

static void MyDrawCharExample( T2K *scaler, int x, int y  )
{
  uint16 left, right, top, bottom;
  unsigned short R, G, B, alpha;
  uint32 *baseARGB = NULL;
  int xi, yi, xd;
  char *p;

  p = (char *)scaler->baseAddr;
#ifdef ENABLE_T2KE
  baseARGB = scaler->baseARGB;
#endif

  left = 0 + x;
  top = 0 + y;
  right= scaler->width  + x;
  bottom= scaler->height + y;

  if ( baseARGB == NULL && p == NULL ) return; /*****/
  assert( T2K_BLACK_VALUE == 126 );

  MoveTo( x, y );

  for ( yi = top; yi < bottom; yi++ ) {
    for ( xi = left; xi < right; xi++ ) {
      xd = xi - left;
#ifdef USE_COLOR
```

```
    if ( baseARGB != NULL ) {
      /* Extract alpha */
      alpha = baseARGB[xd] >> 24;
      /* Extract Red */
      R = (baseARGB[xd] >> 16) & 0xff;
      /* Extract Green */
      G = (baseARGB[xd] >>  8) & 0xff;
      /* Extract Blue */
      B = (baseARGB[xd] >>  0) & 0xff;
    } else {
      alpha = p[xd];
      /* Map [0-126] to [0,255] */
      alpha = alpha + alpha + (alpha>>5);

      /* Set to Black */
      R = G = B = 0;
    }

    if ( alpha ) {
      /* RGBColor contains 16 bit color info for R,G,B each */
      RGBColor colorA, colorB;

      /* Get the background color */
      GetCPixel( xi, yi, &colorB );
      /* Map to 0-256 */
      alpha++;
      /* newAlpha = old_alpha + (1.0-old_alpha) * alpha */
      /* Blend foreground and background colors */
      R = (((long)(256-alpha) * (colorB.red>> 8) + alpha * R )>>8);
      G = (((long)(256-alpha) * (colorB.green>> 8) + alpha * G )>>8);
      B = (((long)(256-alpha) * (colorB.blue>> 8) + alpha * B )>>8);

      assert( R >= 0 && R <= 255 );

      /* Map 8 bit data to 16 bit data */
      colorA.red= R << 8;
      colorA.green= G << 8;
      colorA.blue= B << 8;
      /* Set the foreground color/paint to colorA */
      RGBForeColor( &colorA );
      /* Paint pixel xi, yi with colorA */
      MoveTo( xi, yi );
      LineTo( xi, yi );
    }
  #else
    /* Paint pixel xi, yi */
    if ( p[ xd>>3] & (0x80 >> (xd&7)) ) {
      MoveTo( xi, yi );
      LineTo( xi, yi );
      }
    #endif
    }
      /* Advance to the next row */
      p += scaler->rowBytes;
      if ( baseARGB != NULL ) {
        baseARGB += scaler->rowBytes;
      }
     }
    }
```

Drawing a String Using MyDrawCharExample()

Here is an example, using the **`MyDrawCharExample()`**, discussed previously.

```
F16Dot16 x, y;

x = y = 12 << 16;
while (characters to draw..)
  /* Render the character */
  T2K_RenderGlyph( scaler, charCode, 0, 0,
    GREY_SCALE_BITMAP_HIGH_QUALITY, T2K_SCAN_CONVERT,  &errCode );
  assert( errCode == 0 );
  /* Now draw the character */
  MyDrawCharExample( scaler, ((x + 0x8000)>> 16) +
    (scaler->fLeft26Dot6 >> 6), ((y + 0x8000)>> 16) -
    (scaler->fTop26Dot6 >>6)  );
  x += scaler->xAdvanceWidth16Dot16;/* advance the pen forward */
  /* Free up memory */
  T2K_PurgeMemory( scaler, 1, &errCode );
  assert( errCode == 0 );
}
```

Sample Code for Evaluating Outline Data

To see how Font Fusion breaks down the outlines into straight lines and parabolas (or third-degree Beziér curves), look at the following in `t2ksc.c`:

- **`Make2ndDegreeEdgeList()`** for parabolas
- **`Make3rdDegreeEdgeList()`** for third-degree Beziér curves

Once you have a parabola (described by three points: A,B, and C), you describe it in parametric form as follows:

```
(1-t)*(1-t)*A + 2 *t *(1-t)*B + t*t*C
```

Once you have a third-degree Beziér curve (described by four points: A,B,C,D), you describe it in parametric form as follows:

```
(1-t)*(1-t)*(1-t)*A + 3*(1-t)*(1-t)*t*B + 3*(1-t)*t*t * C + t*t*t *
D
```

In both cases, `t` starts as being equal to zero at point A, and then it goes to one by the last point.

Metrics

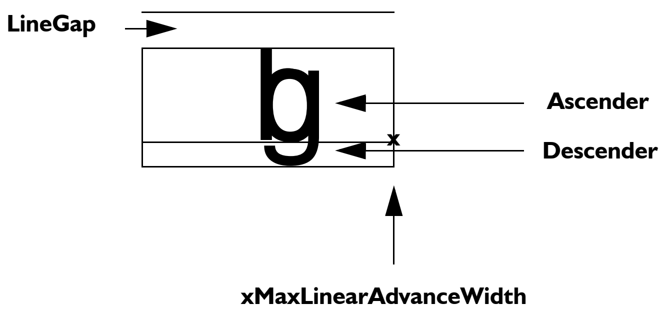
This section discusses the following topics related to font or glyph metrics.

- Font-wide metrics
- Glyph-specific metrics
- Measuring the widths and other metrics of strings

Font-Wide Metrics

The font-wide metrics are the combined metrics for all the characters in a font. These are useful for such activities as pagination, setting line spacing, pre-allocating memory, or setting the minimum number of characters on a line.

Here is an example of the letters "b" and "g" superimposed, showing font-wide metrics.



Here is the font-wide horizontal metrics data:

```
int       horizontalFontMetricsAreValid;
F16Dot16  xAscender,yAscender;
F16Dot16  xDescender,yDescender;
F16Dot16  xLineGap,yLineGap;
F16Dot16  xMaxLinearAdvanceWidth, yMaxLinearAdvanceWidth;
F16Dot16  caretDx, caretDy; /* [0,K] for vertical */
F16Dot16  xUnderlinePosition, yUnderlinePosition;
/* 0 if unknown */
F16Dot16  xUnderlineThickness, yUnderlineThickness;
/* 0 if unknown */
```

Here is the font-wide vertical metrics data:

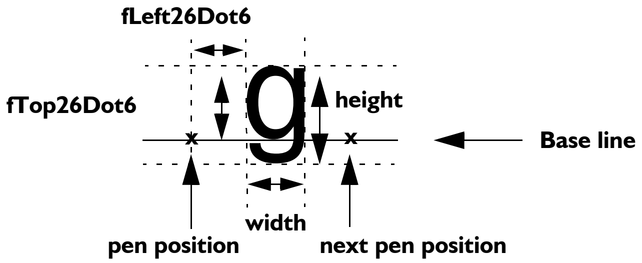
```
int       verticalFontMetricsAreValid;
F16Dot16  vert_xAscender,vert_yAscender;
F16Dot16  vert_xDescender,vert_yDescender;
F16Dot16  vert_xLineGap,vert_yLineGap;
```

```
    F16Dot16  vert_xMaxLinearAdvanceWidth,
vert_yMaxLinearAdvanceWidth;
    F16Dot16  vert_caretDx,vert_caretDy;
    /* [0,K] for vertical */
```

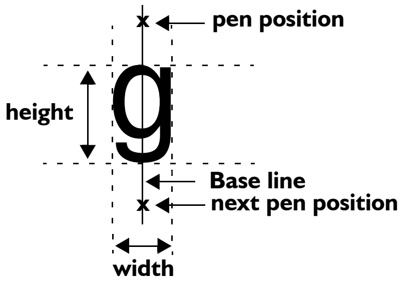
The Ascender is the distance above the baseline, and the Descender is the distance below the baseline. Both have x and y components. For left-to-right horizontal text, the x component is zero. It is non-zero for rotated text.

Glyph-Specific Metrics

Below is a lowercase letter "g" showing horizontal/vertical glyph-specific metrics respectively.



Horizontal metrics for the lowercase character 'g'.



Vertical metrics for the lowercase character 'g'.

- `height` is a 32-bit integer specifying the number of scan lines.
- `width` is a 32-bit integer specifying the number of pixels.

Here is the glyph specific horizontal metrics data for positioning the top left corner of the bitmap:

```
    int       horizontalMetricsAreValid;
    F16Dot16  xAdvanceWidth16Dot16, yAdvanceWidth16Dot16;
    F16Dot16  xLinearAdvanceWidth16Dot16,
yLinearAdvanceWidth16Dot16;
```

```
F26Dot6   fTop26Dot6, fLeft26Dot6;
```

Use `LinearAdvanceWidth` values for non-orthogonal rotations (e.g., 45 degrees).

Here is the glyph specific vertical metrics data for positioning the top left corner of the bitmap:

```
int       verticalMetricsAreValid;
F16Dot16  vert_xAdvanceWidth16Dot16, vert_yAdvanceWidth16Dot16;
F16Dot16  vert_xLinearAdvanceWidth16Dot16,
          vert_yLinearAdvanceWidth16Dot16;
F26Dot6   vert_fTop26Dot6, vert_fLeft26Dot6;
```

So for horizontal text, put the top left corner of the bitmap at the following:

```
[((x + 0x8000)>> 16) + (scaler->fLeft26Dot6 >> 6),
  ((y + 0x8000)>> 16) - (scaler->fTop26Dot6 >>6)]
```

Then advance the pen as follows:

```
x += scaler->xAdvanceWidth16Dot16;
y += scaler->yAdvanceWidth16Dot16;
```

And for vertical text, put the top left corner of the bitmap at the following:

```
[((x + 0x8000)>> 16) + (scaler->vert_fLeft26Dot6 >> 6),
  ((y + 0x8000)>> 16) - (scaler->vert_fTop26Dot6 >>6)]
```

Then advance the pen:

```
x += scaler->vert_xAdvanceWidth16Dot16;
y += scaler->vert_yAdvanceWidth16Dot16;
```

Measuring the Widths and Other Metrics of Strings (such as X11's XTextWidth, and XTextExtent)

In `t2k.h`, the closest method to **`XTextWidth()`** is **`T2K_MeasureTextInX()`**. It measures the linear, unhinted width. It cannot measure the hinted width without actually rendering the characters.

Font Fusion also has two sample functions called **`T2K_GetIdealLineWidth()`** and **`T2K_LayoutString()`**. They can help you lay out an entire line so that the total width is the ideal linear width, while still using run-time, hinted, individual characters and metrics.

Using Filters

Your application has the unique capability of supplying a filter function "plug-in" to perform post-processing on images that the Font Fusion Core produces. These fundamental images can experience post-processing in a filter function in the form of Gaussian fuzz-filtering, smearing, colorizing, texture mapping, underline, strikethrough, or a multiple of these processes. This section discusses using one filter function at a time. See "Using Multiple Filters" on page 99 for details on how to use multiple filters together. The following topics are discussed:

- Sample filter functions
- Using the filter function
- How to write a filter function
- Basic rules for writing a filter function plug-in
- Removing or changing filters

Your application can define or create functions doing just about anything to the source bitmap image, and have that new image appear to emerge from the **`RenderGlyph()`** function of the Core.

This is extremely useful and convenient under normal circumstances, but it is vital to performance when you use a bitmap cache, such as the Font Fusion Cache Manager. The filtered images can then be at the ready in the Cache Manager, avoiding the costly filtering process when Font Fusion renders the character another time. This is a huge performance benefit to your application.

Sample Filter Functions

The following sample filter functions can be enabled by uncommenting the appropriate defines from the top of the `t2kextra.h` file.

- `T2K_TV_Effects()`

  This filter implements the edge effects required by the FCC. See "Support for FCC Standards for Closed Captioning Compliance" on page 94 for more information about this filter.

  Uncomment the line below in t2kextra.h to implement this filter:

  ```
  /* #define ENABLE_T2K_TV_EFFECTS */
  ```
- `T2K_CreateBorderedCharacter()`

  This filter creates a border around a character. See "How to write a filter function" on page 90 for a description of how to create this filter.

  Uncomment the line below in t2kextra.h to implement this filter:

```
/* #define ENABLE_COLORBORDERS */
```

- `T2K_FlickerFilterExample()`

  This filter implements a flicker filter for TV screens. See “Flicker Filter Example” on page 96 for a description of this filter function.

  Uncomment the line below in t2kextra.h to implement this filter:

  ```
  /* #define ENABLE_FLICKER_FILTER */
  ```

Using Filter Functions

- **If you are using the Font Fusion Core without the Cache Manager**, you plug in the filter function through the `setter` macro, **`FF_Set_T2K_Core_FilterReference()`**, defined in `t2k.h`. See page 102 for details.
- **If you are using the Font Fusion Cache Manager**, you plug in the filter function through the Cache Manager interface, **`FF_CM_SetFilter()`**. See page 179 for more information about this function.

  The Cache Manager simply “wires up” the filter specification to the Core before asking the Core to render a glyph. But it also keeps track of the `FilterTag` your application defines. It stores this filter tag with the resulting image in the cache, so that you can use and store up to 256 different filters in the cache at one time.

How to write a filter function

You can write a filter function to do just about anything you can think of. The filter function itself can link together a series of subfilters. You can also use the filter function to convert the Core images into whatever bitmap format your graphics device requires, which will boost your system performance once the images are in the cache. There are several basic rules for writing a filter function plug in:

- Allocate destination memory from the right place.
- Find the source image in the `T2K` class.
- Leave the destination image in the right place in the `T2K` class.
- Update glyph metrics that the filter affects.
- Dispose of the source image memory properly.
- Ensure that the `T2K` class is informed about properly disposing of the destination memory.

All these rules are followed closely in a sample filter function in `t2kextra.c` called **`T2K_CreateBorderedCharacter()`**. Please refer to this example to better understand the filter function specification. We will describe the basic rules in greater detail, making direct references to this working example.

Here is a prototype of the `FF_T2K_FilterFuncPtr` data type:

```
typedef void *(*FF_T2K_FilterFuncPtr)( void *t2k, void
*filterParamsPtr);
```

As you can see, it has a simple interface, taking only two parameters. These are a pointer to a `T2K` class and a pointer to a parameter block. The parameter block is usually a data structure you design for that particular filter.

The **`T2K_CreateBorderedCharacter()`** function does make use of the second argument, a pointer to a filter function parameter block. This block is defined and agreed upon by your application and the filter function. It can be anything you need it to be, or nothing at all.

`T2K_CreateBorderedCharacter()` points a `T2K_BorderfilterParams` pointer at the void pointer, and reads parameters for how thick the border is, what color it is, and what color the core of the character is. Then it goes about creating a 32-bit depth, multi-colored image from the original image. It does this by first creating a smeared, fuzzy-border, colored background of the image. Then with less smearing, it paints the same image a little smaller on top in the requested core color. While doing this, it follows the basic rules.

Basic rules for writing a filter function plug-in:

1 **Allocate destination memory from the right place.**

If the Cache is active, the filter function asks the Cache Manager for the destination memory. Otherwise, it calls `tsi_AllocMem` for the amount it needs. It pays attention to setting the `internal_baseARGB flag` properly, so that the Core knows how to purge memory after it renders the character.

The source memory is pointed to in the `baseAddr` field, and the input data is at 8-bit depth. The filter function deepens the color depth and acquires destination memory at 32 bits-per-pixel and sets all `T2K` class fields properly.

On input, the filter function finds this state:

`T2K Class`	State
`internal_baseAddr`	true
`baseAddr`	set and valid, allocated by the Core and not by the Cache Manager
`internal_baseARGB`	false
`baseARGB`	`NULL`

The filter function example allocates the destination memory it needs. If it is successful getting cache memory, it sets the flag for the `baseARGB` memory, `internal_baseARGB`, to FALSE. If it cannot get cache memory, it sets `internal_baseARGB` to TRUE.

2 **Find the source image in the T2K class.**

The **`T2K_CreateBorderedCharacter()`** function finds the source memory in the `baseAddr` field, and its dimensions that are described in `t->height` and `t->width` fields. The number of bytes per row is in `t->rowBytes`.

So the filter function can easily walk through the source image and “paint” the new, colored image based on the source information. As stated before, this filter takes two walks through the image, one fuzzier than the other, and each time it “paints” a different color on the destination.

3 **Leave the destination image in the right place in the T2K class.**

The **`T2K_CreateBorderedCharacter()`** function is leaving the destination image behind, so to speak, in the `T2K` class, and cleaning up the source image memory. This sample leaves the image in the 32-bit pointer field `baseARGB`.

4 **Update glyph metrics that the filter affects.**

The other key thing this function does, since it expands the image metrics somewhat in the smearing, is that it properly describes the expansion of the image bounding box by also changing the following values:

```
t->rowBytes
t->width
t->height
t->xAdvanceWidth16Dot16
t->xLinearAdvanceWidth16Dot16
t->fTop26Dot6
t->vert_fTop26Dot6
```

They are all affected by the delta `x` and delta `y` parameters in its private parameter block: the things that control the amount of smear and hence the thickness of the actual "border" around the character. Your filter may or may not affect all of these particular glyph metrics.

5 **Dispose of the source image memory properly.**

This function found the source memory at the `baseAddr` field. If the flag `internal_baseAddr` is TRUE, that means the Core acquired the memory by calling **`tsi_FastAllocMem ()`**, so it calls **`tsi_FastDeAllocN()`**. If `internal_baseAddr` is FALSE, that means the Core got the memory from the Cache. This should never happen. Cache memory should always be left alone! Never try to de-allocate cache memory, because you will crash your system!

6 **Ensure that the T2K class is informed about properly disposing of the destination memory.**

The following table shows the state of the affected `T2K` class members after this filter function completes:

`T2K Class`	State
`internal_baseAddr`	false
`baseAddr`	`NULL`, and disposed of by this filter function
`internal_baseARGB`	true if from **`tsi_AllocMem( )`**, false if from cache memory
`baseARGB`	set and valid

When your application calls **`T2K_PurgeMemory()`**, the Core can intelligently handle the cleanup.

Removing or Changing Filters

The system only maintains the last filter set, therefore, if you want to turn the filter off, set the filter function pointer to `NULL`. If you are using the cache manager, you can do this with the following call:

```
FF_CM_SetFilter(theCache, 0, NULL, NULL);
```

If you are not using the Cache Manager, set the filter function pointer to null with the following call:

```
FF_Set_T2K_Core_FilterReference(t2k, NULL, NULL );
```

Both of these examples also set the parameter's pointer to `NULL` for the sake of neatness. As stated above, Font Fusion only maintains the last filter set, so you can switch to another filter by simply setting a new function pointer or parameter's-block pointer combination.

Support for FCC Standards for Closed Captioning Compliance

Font Fusion supports font requirements mandated by the Federal Communications Commission (FCC) set forth by the Electronics Industries Alliance (EIA) for closed captioning display on both analog and digital television systems.

Font Fusion includes the ability to display text using the following six character edge effects:

- embossed (to support the specified "raised" effect)
- engraved ("depressed")
- outline ("uniform")
- drop shadow left
- drop shadow right
- or none

Edge effects can be implemented by using the `ENABLE_T2K_TV_EFFECTS` plug-in filter, defined in the t2kextra.h file, and the `T2K_TV_Effects()` function. It is recommended that you use `T2K_RenderGlyph()` with grayscale and in `T2K_TV_MODE_2` for best results, or you can use native hints combined with grayscale. The `T2K_TV_Effects()` filter function is described in detail below.

void T2K_TV_Effects(

```
T2K *t,
void *params)
```

Arguments

- `t` is a pointer to the T2K scaler object
- `params` is a void pointer that points to T2K_TVFilterParams and
- `myParams->selector` selects the FCC styles:
  - `0 – none`
  - `1 – raised`
  - `2 – depressed`
  - `3 – uniform(outline)`
  - `4 – drop-shadow-left`
  - `5 – drop-shadow-right`

Description

This is a plug in filter which implements the 6 FCC character edge effects. It is recommended that you use `T2K_RenderGlyph()` with `GrayScale` and in `T2K_TV_MODE_2` for best results. Alternatively you could also try native hints combined with `GrayScale`.

NOTE: If you are using the cache manager, you can use all plug-in filters by using `FF_CM_SetFilter()`. See “void FF_CM_SetFilter(” on page 179 for assistance.

T2K_TVFilterParams Structure

```
typedef struct {
   uint8 greyScaleLevel;
      /* greyScaleLevel used */
   uint8 selector;
      /* FCC style  : 0 none, 1 raised, 2 depressed,
                      3 uniform(outline), 4 drop-shadow-left,
                      5 drop-shadow-right */
   int32 dX, dY;
      /* dX, dY is the thickness of the border. (Should be 1 or 2 )
      */
   uint32 R,G,B;
      /* R,G,B, is the color of the character.
        (All values should be between 0 and 255) */
   uint32 borderR, borderG, borderB;
      /* borderR, borderG, borderB is the color of the border.
        (All values should be between 0 and 255) */
   uint32 sunnyBorderR, sunnyBorderG, sunnyBorderB;
      /* This is the sunny side of the border. Only used for raised
```

```
           and depressed styles
          (All values should be between 0 and 255)
         */

} T2K_TVFilterParams;
```

Flicker Filter Example

This is intended to be an example that can be tweaked further on the actual deployment hardware.

It implements this filter:

```
1 4 1
2 4 2
1 4 1
```

OR this filter if `FAST_FLICKER_FILTER` is defined

```
0 4 0
2 4 2
0 4 0
```

If you can not tell the difference then use the fast version. This should be tweaked for the deployment hardware. A piece of advice is that you should not blur the image more than necessary.

```
void T2K_FlickerFilterExample( T2K *t, void *params )
===> params is unused and should be set to NULL.
     t is the pointer to the T2K scaler object
```

Using Glow Filter

The glow filter adds a neon-like glow effect to the glyphs, making it appear as light source is being shined up from underneath the character.

`T2K_CreateGlowCharacter()` function should be used to create a glow effect on a character. Macro `ENABLE_GLOWFILTER` should be turned ON to use this API.

`void T2K_CreateGlowCharacter(`

```
T2K *t,
void *params)
```

Arguments

`t` is a pointer to the `T2K` scaler object.

`params` is the address of the structure type `T2K_GlowFilerParams.`

Description

The `T2K_CreateGlowCharacter()` API creates a glow effect around an antialiased character. This function can be invoked right after `T2K_RenderGlyph()`.

For best results, use `T2K_RenderGlyph()` with grayscale antialiasing mode.

`T2K_GlowFilerParams` is a structure of the following type:

```
typedef struct {
   uint8 greyScaleLevel;
   int32 spread;
   int32 glow;
   uint32 Red;
   uint32 Green;
   uint32 Blue;
   uint32 glowR;
   uint32 glowG;
   uint32 glowB;
} T2K_GlowFilerParams;
```

In the structure `T2K_GlowFilerParams`,

`greyScaleLevel` refers to the monochrome or grayscale mode

`spread` refers to the range of the glow, which can vary from 2 to 10.

`glow` is the intensity of the glow.

`Red` is the fore color of the character, the value should be between 0 and 255.

`Green` is the fore color of the character, the value should be between 0 and 255.

`Blue` is the fore color of the character, the value should be between 0 and 255.

`glowR` is the color of the glow, the value should be between 0 and 255.

`glowG` is the color of the glow, the value should be between 0 and 255.

`glowB` is the color of the glow, the value should be between 0 and 255.

Using Multiple Filters

You can combine multiple filters through the use of the **`T2K_MultipleFilter()`** function. Turn on the `ENABLE_MULTIPLE_FILTERS` compile-time option in order to use these functions. Create the multiple filter as described below, then apply it in the same way as you would apply any other filter. Delete a filter in the multiple filter combination with the **`T2K_MultipleFilter_Delete()`** function.

NOTE: The multiple filter function still caches bitmaps even if you do not have any filters defined.

To create a multiple filter

1 Define the multiple filter parameter structure:

```
T2K_MultipleFilterParams MultiFilterParams;
```

2 Initialize the multiple filter:

```
T2K_MultipleFilter_Init(&T2K, (void*)&MultiFilterParams);
```

3 Add the first filter function:

```
T2K_MultipleFilter_Add(&MultiFilterParams,
                       filterTag,
                       filterIndex,
                       (FF_T2K_FIlterFuncPtr)Filter_1,
                       Filter_1_params,
                       &errCode );
```

4 Add the second filter function:

```
T2K_MultipleFilter_Add(&MultiFilterParams,
                       filterTag,
                       filterIndex,
                       (FF_T2K_FIlterFuncPtr)Filter_2,
                       Filter_2_params,
                       &errCode );
```

5 Continue adding the filter functions you require.

NOTE: Remember that the order you enter the functions is not the order in which they are applied. The filterIndex sets the order of how multiple filters are applied. For example, if the underscore filter is followed by the bold filter, you will embolden the underscore, while if you reverse this order, the underscore will not be emboldened.

Creating Outlines

You can create outlines of a character through the use of **`T2K_CreateOutlineCharacter()`** function. Turn ON the `ENABLE_OUTLINEFILTER` compile-time option in order to use this filter function.

The **`T2K_CreateOutlineCharacter()`** filter function is capable of creating outlines in Monochrome and Grayscale mode, whereas **`T2K_CreateBorderedCharacter()`** creates a colored border and works in `baseARGB` mode.

void T2K_CreateOutlineCharacter(

```
T2K *t,
void *params)
```

Arguments

`t` is a pointer to the T2K scaler object.

`params` is a void pointer that points at T2K_OutlineFilterParams.

Description

Creates an outlined character. This function can be invoked right after `T2K_RenderGlyph()`.

Getting the Font Name

To get the font name, call **`T2K_SetNameString()`** after you call **`Set_PlatformID()`** and **`Set_PlatformSpecificID()`**.

It sets the public field `nameString8` or `nameString16` in the Font Fusion object (structure).

To use Microsoft Unicode mapping and names, include these arguments:

```
/* Use 3,1 to pick Microsoft Unicode character mapping */
```

```
Set_PlatformID( scaler, 3 );
Set_PlatformSpecificID( scaler, 1 );
/* Pick American English and the full font name */
T2K_SetNameString( scaler, 0x0409, 4 );
```

Enabling “sbits”

“Sbits” are embedded bitmaps. Font Fusion supports modification of the source sbits through the transformation matrix when you call one of the **`RenderGlyph()`** functions. This enables your application to resize, oblique, mirror, or rotate embedded bitmaps.

▶ To enable sbits

1 Set the `ENABLE_SBITS` compile-time option in `config.h` to enable the code necessary to process embedded bitmaps.

2 For `T2K_NewTransformation(enableSbits)`, set `enableSbits` to `true`.

Use the following functions to query whether you enabled the sbits or whether sbits exist in your fonts.

- **`T2K_FontSbitsAreEnabled()`** is a query method to check whether or not you enabled the “sbits”.
- **`T2K_FontSbitsExists()`** is a query method to check whether or not a TrueType, native T2K, or PFR font contains any “sbits.”
- **`T2K_GlyphSbitsExists()`** is a query method to check if a particular glyph exists in sbit format for the current point size in a TrueType or native T2K font. If you need to use `characterCode`, then map it to `glyphIndex` by using **`T2K_GetGlyphIndex()`** first.

Use the following macros to get the extended sbits support:

- `ENABLE_SBITS_TRANSFORM` to enable the transformation (as *scaling/ obliquing*) of bitmap fonts.
- `ENABLE_SBITS_COMPRESSION` if you need compressed CJK bitmap font support.

T2K Functions

void DeleteT2K(

```
   T2K *t,
   int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object you created when calling **`NewT2K()`**.

`errCode` is a pointer to the returned error code.

Description

`DeleteT2K()` deletes the `T2K` object you previously created.

void FF_Set_T2K_Core_FilterReference(

```
   T2K *theCache,
   FF_T2K_FilterFuncPtr funcptr,
   void *params)
```

Arguments

`theCache` is a pointer to the cache class returned from FF_CM_New().

`funcptr` is a function pointer to the actual filter function.

`params` is a pointer to an optional parameter block for the filter function.

Description

`FF_Set_T2K_Core_FilterReference()` sets parameters related to filtering. These parameters are stored and applied to new characters that Font Fusion creates.

T2K *NewT2K(

```
tsiMemObject *mem,
sfntClass *font,
int *errCode)
```

Arguments

`mem` is a pointer to the `tsiMemObject`.

`font` is a pointer to the `sfntClass` (font) object.

`errCode` is a pointer to the returned error code.

Description

`NewT2K()` is a pointer to a new `T2K` object that this function creates.

void T2K_ConvertGlyphSplineType(

```
T2K *t,
short curveTypeOut,
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`curveTypeOut` specifies the kind of curve you want to convert the outlines to at run time. Valid values, as follows, return glyphs made up of:

1	First-degree poly-lines (straight-line segments)
2	Second-degree quadratic B-splines (parabolas and straight-line segments)
3	Third-degree cubic Bézier curves (cubics and straight-line segments)

`errCode` is a pointer to the returned `errorCode`.

Description

`T2K_ConvertGlyphSplineType()` allows you to access outline data in the outline format you need, no matter what format the original outline font was in. If the outline format of the original font is different from the requested format, Font Fusion does a run-time curve conversion of the outlines. You can request the outlines as:

- first degree poly-lines
- second-degree quadratic B-splines
- third-degree cubic Béziers

This function creates a glyph with outlines of `curveTypeOut`, independent of the original curve type.

Call this function after **`T2K_RenderGlyph()`**, but before you call **`T2K_PurgeMemory()`**. Make sure you set the `T2K_RETURN_OUTLINES` bit in the function **`T2K_RenderGlyph()`**.

Note that you also have to define the compile-time option `ENABLE_FF_CURVE_CONVERSION`.

void T2K_CreateUnderlineCharacter(

```
T2K *t,
void *params)
```

Arguments

`t` is a pointer to the `T2K` scaler object.

`params` is a void pointer that points to the **`T2K_UnderLineFilterParams`** structure, described below.

Description

This filter function operates as two filters in one, depending on the `isUnderline` value set in the structure. Set this value to `true` to create an underline, or `false` to create a strikethrough. For gray and monochrome displays, set `mode` to `0`. If you are using extended LCD modes, send the LCD mode. See “Extended LCD Modes” on page 74 for a list of these modes.

The **`T2K_UnderLineFilterParams()`** structure is expressed by the code below.

```
typedef struct {
   uint8 greyScaleLevel; /* greyScaleLevel used */
   uint8 isUnderline;     /* non zero if underline is needed
                             else strikethrough */
   #ifdef ENABLE_EXTENDED_LCD_OPTION
      /* 0 for gray and monochrome else for LCD mode */
      uint16 mode;        /* If ext LCD modes are used then
the
                             modes used should be passed in
                             this member variable */
   #endif /* ENABLE_EXTENDED_LCD_OPTION */
} T2K_UnderLineFilterParams;
```

`T2K_KernPair *T2K_FindKernPairs(`

```
T2K *t,
uint16 *baseSet,
int baseLength,
uint16 charCode,
int *pairCountPtr)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`baseSet` is a pointer to a `baseLength` number of 16-bit-wide character codes.

`baseLength` is the number of 16-bit-wide character codes.

`charCode` is the character code.

`pairCountPtr` is a pointer to the number of kerning pairs found between the character with the `charCode` combined with itself and all the members of `baseSet`.

Description

Call this function to return a pointer to **`T2K_KernPair()`** with `*pairCountPtr` entries. The entries consist of all kern pairs between (1) the character with the character code combined with itself and (2) all the members of `baseSet`. (A character should only appear once in `baseSet`.)

Your application must de-allocate the pointer if it is not equal to `NULL`:

```
tsi_DeAllocMem( t->mem, pointer )
```

T2K_KernPair structure and #ifdef statements

```
#ifdef ENABLE_KERNING
typedef struct {
uint16 left; /* left character code */
uint16 right; /* right character code */
int16 xKern; /* value in FUnits */
int16 yKern; /* value in FUnits */
} T2K_KernPair;
#endif /* ENABLE_KERNING */

#ifdef ENABLE_LINE_LAYOUT

#ifdef LINEAR_LAYOUT_EXAMPLE
```

char T2K_FontSbitsAreEnabled(

`T2K *t)`

Arguments

`t` is a pointer to the `T2K` object itself.

Description

This is a macro that checks whether or not you enabled “sbits” (embedded bitmaps).

char T2K_FontSbitsExists(

`T2K *t)`

Arguments

`t` is a pointer to the `T2K` object itself.

Description

This is a macro that checks whether or not a TrueType, native T2K, or PFR font contains any “sbits” (embedded bitmaps).

void T2K_GaspifyTheCmds(

```
T2K *t
greyScaleLevelPtr
cmdInPtr)
```

Arguments

`t` is a pointer to the T2K object itself.

`greyScaleLevelPtr` is a pointer to the level of anti-aliasing you want to apply to the `T2K_RenderGlyph()`, `FF_FM_RenderGlyph()` or `FF_CM_RenderGlyph()` functions.

`cmdInPtr` is a pointer to the command argument you intend to pass to the `T2K_RenderGlyph()`, `FF_CM_RenderGlyph()`, or `FF_FM_RenderGlyph()` functions.

Description

Modifies the greyScaleLevel and cmdIn parameters for `T2K_RenderGlyph` according to the wishes of the Grid-fitting and Scan-Conversion Procedure (GASP) Table table if one exists. This is dependent on if `ENABLE_GASP_TABLE_SUPPORT` is defined.

See http://www.microsoft.com/OpenType/otspec/GASP.HTM or http://partners.adobe.com/asn/developer/opentype/gasp.html for more information on GASP tables.

Example

```
T2K_GaspifyTheCmds( scaler, &greyScaleLevel, &cmd );
T2K_RenderGlyph( scaler, charCode, 0, 0, greyScaleLevel,
cmd, &errCode );
```

int T2K_GetBytesConsumed(

```
T2K *t)
```

Arguments

`t` is a pointer to the `T2K` object itself.

Description

Some TrueType fonts support mixed 1 and 2 byte streams using the format 2 character map. This macro returns the number of bytes of the input character code that were used to resolve to a glyph index.

uint16 T2K_GetGlyphIndex(

```
T2K *t,
uint16 charCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`charCode` is the character code.

Description

Call this function to get the position of the character image in a font, given the character code. The glyph index is simply a number from 0 to `n`-1, assuming the font contains `n` number of glyphs:
`(N = T2K_GetNumGlyphsInFont( scaler );)`

void T2K_GetIdealLineWidth(

```
T2K *t,
const T2KCharInfo cArr[],
long lineWidth[],
T2KLayout out[])
```

Arguments

`t` is a pointer to the `T2K` object itself.

`cArr[]` is an array containing one entry per character. Each entry is of type `T2KCharInfo`, described below.

`lineWidth[]` is an array filled in by **`T2K_GetIdealLineWidth()`** for later use by **`T2K_LayoutString()`**. It contains the ideal linearly-scaled width for the entire string (taking kerning into account).

`out` contains data that **`T2K_LayoutString()`** needs later.

Description

Call this function before **`T2K_LayoutString()`**.

Use both functions to first get the line length you want, and then lay out a character string along it.

The desired line length is the sum of the fractional, advance width of all the characters. However, one problem is that hinted characters—such as you find in TrueType fonts—produce integer advance widths, and this integer advance width can in some cases be far removed from an ideal, linearly-scaled advance width.

The integer width makes the text look better, but then you may be faced with the opposite goal of WYSIWYG. For WYSIWYG, the line lengths of text have to scale linearly. **`T2K_LayoutString()`** achieves this goal by primarily putting the distortion to the integer advance widths, which is caused by the overall linear line width goal, into the space characters on the line.

`T2K_GetIdealLineWidth()` fills in for each character the integer, non-linear advance widths.

Then in the final step, when your application calls **`T2K_GetIdealLineWidth()`**, the function simply modifies the non-linear metrics in the `out` parameter so that the total line width becomes equal to the linearly-scaled line width. In this way, the linearly- scaled WYSIWYG is maintained between two devices, for instance, the screen and printer.

T2KCharInfo

Before calling **`T2K_GetIdealLineWidth()`**, your application needs to fill in all the fields in the `cArr` array. There is one entry per character. Each entry is of type `T2KCharInfo`. See the code below for an example:

```
typedef struct {
   /* input */
   uint16 charCode;
   uint16 glyphIndex;
   F16Dot16 AdvanceWidth16Dot16[ T2K_NUM_INDECES ];
   F16Dot16 LinearAdvanceWidth16Dot16[ T2K_NUM_INDECES ];
   F26Dot6 Corner[ T2K_NUM_INDECES ]; /* fLeft26Dot6,
fTop26Dot6 */
   long Dimension[ T2K_NUM_INDECES ]; /* width, height */
} T2KCharInfo;
```

Arguments

`charCode` is the character code.

`glyphIndex` is the glyph index.

`AdvanceWidth16Dot16` is the gridded/hinted advance width in 16.16 pixels.

`LinearAdvanceWidth16Dot16` is the linearly-scaled advance width in 16.16 pixels.

`Corner` is equal to the T2K fields `fLeft26Dot6`, `fTop26Dot6` the scan-conversion process returns when Font Fusion renders glyphs.

`Dimension` is equal to the T2K bitmap width and height the scan-conversion process returns when Font Fusion renders glyphs.

int T2K_GlyphSbitsExists(

```
T2K *t)
uint16 glyphIndex,
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`glyphIndex` is the position of the character image in a TrueType, native T2K, or PFR font.

`errCode` is a pointer to the returned `errorCode`.

Description

Call this function to check whether or not a particular glyph exists in `sbit` format for the current point size in a TrueType or native T2K font. If you need to use `characterCode`, then map it to `glyphIndex` by using **`T2K_GetGlyphIndex()`** first.

void T2K_LayoutString(

```
const T2KCharInfo cArr[],
long lineWidth[],
T2KLayout out[])
```

Arguments

`cArr[]` is an array containing one entry per character. Each entry is of type `T2KCharInfo`, described in the previous section.

`lineWidth[]` is an array filled in by **`T2K_GetIdealLineWidth()`**. It contains the ideal linearly-scaled width for the entire string (taking kerning into account).

`out` contains data that **`T2K_GetIdealLineWidth()`** provides.

Description

Call this function after **`T2K_GetIdealLineWidth()`**. See the previous function for more information.

uint32 T2K_MeasureTextInX(

```
T2K *t,
const uint16 *text,
int16 *xKernValuesInFUnits,
uint32 numChars)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`text` is a pointer to a 16-bit-wide character string. Note that `text` has to be at least `numChars` long.

`xKernValuesInFUnits` is a pointer to an array of kerning values. Note that `xKernValuesInFUnits` has to be at least `numChars` long.

`numChars` is the number of characters in the `text` character string.

Description

Call this function to get the total pixel width of a character string and compute its kerning values. This function returns a `uint32` value that is the total length of the string in integer pixel units.

void T2K_MultipleFilter(

```
T2K *t,
void *params)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`params` is a void pointer that points to the **`T2K_MultipleFilterParams`** structure, described below.

Description

This function makes a character glyph pass through all the filters that are registered to it. The **`T2K_MultipleFilterParams`** structure is expressed by the code below.

```
typedef struct {
/* The array of filter method pointers */
FF_T2K_FilterFuncPtr activeFilters[ T2K_NUM_MULTIPLE_FILTERS
];
/* The array of filter arguments */
void *parameters[ T2K_NUM_MULTIPLE_FILTERS ];
/* Current number of registered filters */
int16 numOfFilters;
/* Current filter tag */
uint8 filterTag;
} T2K_MultipleFilterParams;
```

See “Using Multiple Filters” on page 99 for details.

void T2K_MultipleFilter_Add(

```
T2K_MultipleFilterParams *filterParams,
uint8 filterTag,
uint8 index,
FF_T2K_FilterFuncPtr filterToAdd,
void *Params,
int *errCode);
```

Arguments

`filterParams` is a pointer to the **`T2K_MultipleFilterParams`** structure.

`filterTag` is the filter that should be `OR`ed with the current filter.

`index` is the index at which the filter will be set. This value also decides the order at which the filter will be applied.

`filterToAdd` is a pointer to the filter method that has to be added.

`Params` is a pointer the arguments that have to be passed when this particular filter is applied.

`errCode` will be non-zero when reporting exceptional conditions.

Description

Adds a new filter to the multiple filter class. The new `filterTag` is returned if the addition of the new filter is successful.

void T2K_MultipleFilter_Delete(

```
   T2K_MultipleFilterParams *filterParams,
   uint8 filterTag,
   uint8 index,
   int *errCode)
```

Arguments

`filterParams` is a pointer to the **`T2K_MultipleFilterParams`** structure.

`filterTag` is the filter that should be `OR`ed with the current filter.

`index` is the index at which the filter will be deleted.

`filterToAdd` is a pointer to the filter method that has to be added.

`errCode` will be non-zero when reporting exceptional conditions.

Description

Deletes the existing filter from the multiple filter class. If the function successfully deletes the filter, it returns the new `filterTag`.

void T2K_MultipleFilter_Init(

```
   T2K_MultipleFilterParams *filterParams)
```

Arguments

`filterParams` is a pointer to the **`T2K_MultipleFilterParams`** structure.

Description

Initializes the multiple filter class.

void T2K_NewTransformation(

```
T2K *t
int doSetUpNow
long xRes
long yRes
T2K_TRANS_MATRIX *trans
int enableSbits
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object.

`doSetUpNow` determines if Font Fusion needs to do some setup work now or later. Recommended setting is `== true`.

`xRes` and `yRes` represent the resolution of the output device in dots per inch. For example, a Windows screen device uses 96 for `xRes` and `yRes`.

`trans` is a pointer to the transformation matrix. Basically, you specify:

- the `x` and `y` resolutions
- a `2*2` transformation matrix (includes the point size)
- a true or false setting to enable or disable embedded bitmaps

Then you call **`T2K_RenderGlyph()`** to actually get outline or bitmap data. After you are done with the output data, you need to call **`T2K_PurgeMemory()`** to free up memory.

`enableSbits` enables embedded bitmaps if they exist.

`errCode` is a pointer to the returned error code.

Description

`T2K_NewTransformation()` allows you to set the transformation matrix and `x` and `y` resolutions when you render characters and strings. It informs the `T2K` object about the current transformation and size.

`void T2K_PurgeMemory(`

```
T2K *t,
int level,
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

Normally, set `level` = 1.

`errCode` is a pointer to the returned `errorCode`.

Description

Call the function **`T2K_PurgeMemory()`** after you are done with the output data from **`T2K_RenderGlyph()`**.

Also, set the following compile-time option:

```
#define MAX_PURGE_LEVEL 2
```

`void T2K_RenderGlyph(`

```
T2K *t,
long code,
int8 xFracPenDelta,
int8 yFracPenDelta,
uint8 greyScaleLevel,
uint16 cmd,
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`code` usually specifies the code for the character you want to render. However, if you want to use the glyph index instead, then set the `T2K_CODE_IS_GINDEX` bit in the `cmd` argument of **`T2K_RenderGlyph()`**. The glyph index is simply a number from 0 to `n`-1, assuming the font contains `n` number of glyphs:
`(N = T2K_GetNumGlyphsInFont( scaler );)`

`xFracPenDelta` and `yFracPenDelta` are normally set to zero. You can use them with non-zero values if you are also using fractional character positioning.

`greyScaleLevel` describes the level of anti-aliasing you want to apply. See the section below.

`cmd` describes to Font Fusion what to do with various `bitflags`. See the section below. You can also refer to comments that define the `bitflags` in `t2k.h`.

`errCode` is a pointer to the returned `errorCode`.

Description

`T2K_RenderGlyph()` allows you to create a character image.

Bits for the greyScaleLevel field

```
#define BLACK_AND_WHITE_BITMAP              0
#define GREY_SCALE_BITMAP_LOW_QUALITY       1
#define GREY_SCALE_BITMAP_MEDIUM_QUALITY    2
#define GREY_SCALE_BITMAP_HIGH_QUALITY      3 /* Recommended for grey-scale */
#define GREY_SCALE_BITMAP_HIGHER_QUALITY    4
#define GREY_SCALE_BITMAP_EXTREME_QUALITY   5 /* Slowest */

/* When doing grey-scale the scan-      converter returns values in the range
T2K_WHITE_VALUE -- T2K_BLACK_VALUE */

#define T2K_BLACK_VALUE                     126
#define T2K_WHITE_VALUE                     0

/* The Caller HAS to deallocate outlines && t->baseAddr with
T2K_PurgeMemory( t, 1 ) */

/* fracPenDelta should be between 0 and 63, 0 represents the normal pixel
alignment,
   16 represents a quarter pixel offset to the right,
   32 represents a half pixel offset of the character to the right,
   and -16 represents a quarter/4 pixel shift to the left. */

/* For Normal integer character positioning set fracPenDelta == 0 */
/* IPenPos = Trunc( fracPenPos );  FracPenDelta = fPenPos - IPenPos */
/* The bitmap data is relative to  IPenPos, NOT fracPenPos */

/*
 * The T2K call to render a character.
 *
 * t: is a pointer to the T2K object itself.
 * code: If the bit T2K_CODE_IS_GINDEX is set in 'cmd' it is the glyphIndex,
otherwise it is the characterCode for the character we are rendering.
```

```
 * xFracPenDelta, yFracPenDelta: Normally set to zero. Can be used with non-zero
values if fractional character positioning is used.
 * cmd: Describes to T2K what is to be done with various bitflags. See commetns
that define the bitflags in T2K.H.
 * errCode: is a pointer to the returned errorCode.
 *
 */
```

Bits for the cmd field

```
#define T2K_GRID_FIT                            0x0001
#define T2K_SCAN_CONVERT                        0x0002
#define T2K_RETURN_OUTLINES                     0x0004
#define T2K_CODE_IS_GINDEX                      0x0008 /* Otherwise it is the
                                                characterCode */
#define T2K_USE_FRAC_PEN                        0x0010
#define T2K_SKIP_SCAN_BM                        0x0020 /* Everything works as
                                                normal, however we do _not_
                                                generate the actual bitmap */
#define T2K_TV_MODE                             0x0040 /* Ideal for TV if you
                                                use integer metrics, and
                                                gray-scale (please turn off
                                                T2K_GRID_FIT) */
#define T2K_NAT_GRID_FIT                        0x0080 /* Enables native TrueType
                                                          hint/gridding support */
#define T2K_LCD_MODE                            0x0100 /* Ideal for LCD screens */
#define T2K_Y_ALIGN                             0x0200 /* Ideal with T2K_LCD_MODE,
                                                          and T2K_TV_MODE */
#define T2K_TV_MODE_2                           ( T2K_TV_MODE  | T2K_Y_ALIGN )
#define T2K_LCD_MODE_2                          ( T2K_LCD_MODE | T2K_Y_ALIGN )
#define T2K_LCD_MODE_3                          ( T2K_LCD_MODE | T2K_GRID_FIT )
#define T2K_LCD_MODE_4                          ( T2K_LCD_MODE | T2K_NAT_GRID_FIT )

#define EXT_HOR_RGB  0x1000  /* This means R G B */
#define EXT_HOR_BGR  0x2000  /* This means B G R */
#define EXT_VER_RGB  0x4000  /* This means R
                                           G
                                           B */
#define EXT_VER_BGR  0x8000  /* This means B
                                           G
                                           R */
#define T2K_EXT_LCD_H_RGB ( T2K_LCD_MODE | T2K_NAT_GRID_FIT | EXT_HOR_RGB )
                                        /* This means R G B */
#define T2K_EXT_LCD_H_BGR ( T2K_LCD_MODE | T2K_NAT_GRID_FIT | EXT_HOR_BGR )
                                        /* This means B G R */
#define T2K_EXT_LCD_V_RGB ( T2K_LCD_MODE | T2K_NAT_GRID_FIT | EXT_VER_RGB )
                                        /* This means R
                                                      G
                                                      B */
#define T2K_EXT_LCD_V_BGR ( T2K_LCD_MODE | T2K_NAT_GRID_FIT | EXT_VER_BGR )
                                        /* This means B
                                                      G
                                                      R */
```

```
#define T2K_VERTICAL                    0x0400 /* Causes vertical rotation
                                                  and positioning */
#define T2K_VERT_SUB                    0x0800 /* Enables use of vertical
                                                  substitution characters
                                                  from the VT data
                                                  segment */
```

void T2K_SetNameString(

```
T2K *t,
uint16 languageID,
uint16 nameID)
```

Arguments

`t` is a pointer to the `T2K` object itself.

Set `t->nameString8` or `t->nameString16` depending on whether or not the name is encoded as a byte string or as a 16-bit Unicode string.

`languageID` and `nameID` are the same as what Microsoft's TrueType documentation specifies for the name table.

Description

Call this function after **`Set_PlatformID()`** and **`Set_PlatformSpecificID()`**.

void T2K_TransformXFunits(

```
T2K *t,
short xValueInFUnits,
F16Dot16 *x,
F16Dot16 *y)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`xValueInFUnits` is the `x` font unit value. It is a measurement (e.g., a kerning value) in FUnits (font units). There are 1000 FUnits for the em in Type 1 fonts and, typically, 2048 FUnits for the em in TrueType fonts. So, for example, in a

Type 1 font, if you have a distance representing 7% of the em, then your application passes in `0.07*1000 = 70` as the value in FUnits.

`x` is a pointer to the `x` fractional pixel value, i.e., a value in `16.16` format.

`y` is a pointer to the `y` fractional pixel value, i.e., a value in `16.16` format.

Description

Call this function to transform `xInFUnits` into 16.16 `x` and `y` values.

Font Fusion stores outlines and outline metrics in font units or "FUnits." Type 1 fonts have 1000 FUnits per em and TrueType fonts typically have 2048 FUnits per em.

When we render characters, we produce output in a pixel space. This function simply maps measurements in FUnits into this output pixel domain. Note that the results are in 16.16 format. This means we have 16 integer bits and 16 fractional bits. So, in binary, 1.5 pixels would be represented as

```
00000000 00000001 10000000 00000000
```

and in hex it would be

```
0x00011000
```

A typical use is that you have a kerning value in FUnits. But before you can draw a character or string in pixel space, you use this function to map the FUnit value into the pixel domain. Note that the mapping is size- and transformation-dependent.

void T2K_TransformYFunits(

```
T2K *t,
short yValueInFUnits,
F16Dot16 *x,
F16Dot16 *y)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`yValueInFUnits` is the `y` font unit value. It is a measurement (e.g., a kerning value) in FUnits (font units). There are 1000 FUnits for the em in Type 1 fonts and, typically, 2048 FUnits for the em in TrueType fonts. So, for example, in a Type 1 font, if you have a distance representing 7% of the em, then your application passes in `0.07*1000 = 70` as the value in FUnits.

`x` is a pointer to the `x` fractional pixel value, i.e., a value in `16.16` format.

`y` is a pointer to the `y` fractional pixel value, i.e., a value in `16.16` format.

Description

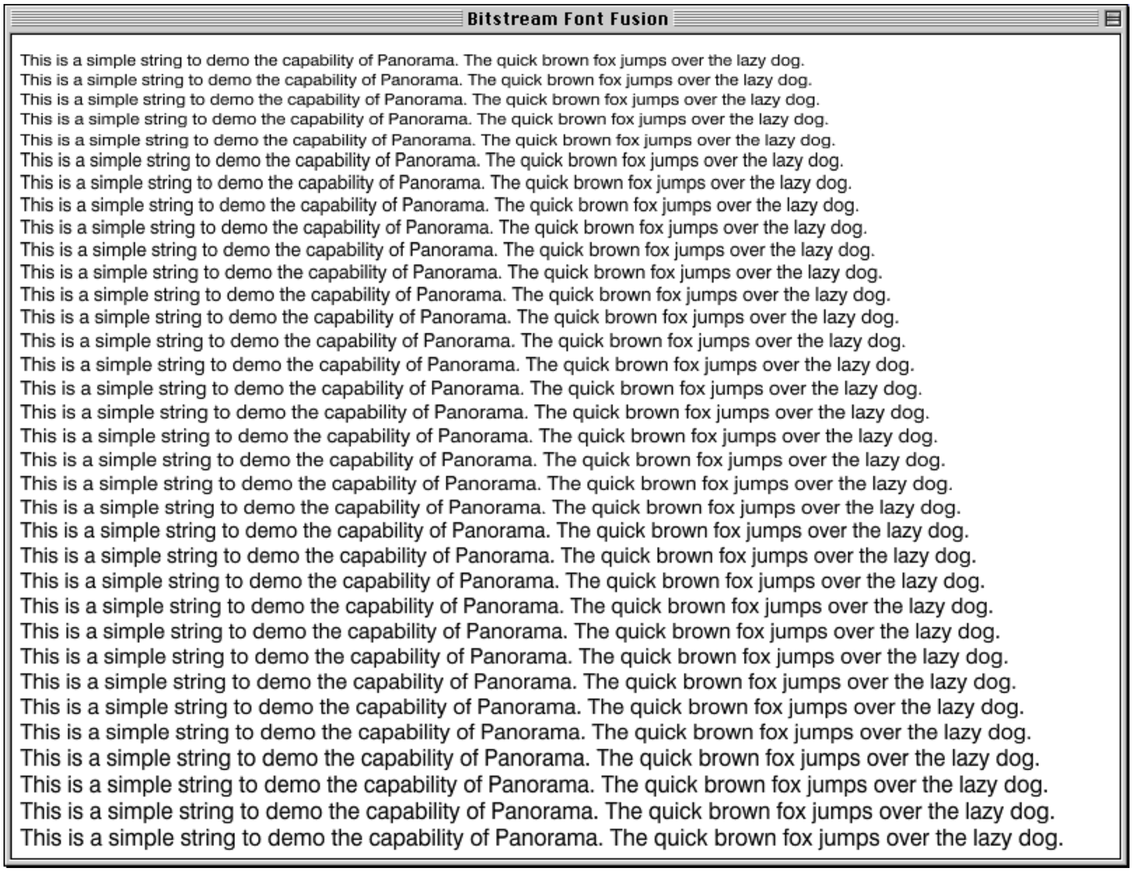
Call this function to transform `yInFUnits` into 16.16 `x` and `y` values. See the previous function for more information.

Functions for Fractional Sizing

Font Fusion supports fractional sizing of characters with a high precision level. The enhanced light-weight, processor-friendly Y-direction hinting process in fractional mode provides crisp character output.

The compile conditional `ENABLE_FRACTIONAL_SIZE` should be turned ON to include the fractional size support.

The figure below shows the fractional sizing applied to the text:



```
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
This is a simple string to demo the capability of Panorama. The quick brown fox jumps over the lazy dog.
```

Fractional size in Font Fusion with enabled Y-align hinting settings

T2K_SetFracSizeMode(

```
t2k,
bSet)
```

Arguments

`t` is the pointer to the `t2k` object itself.

`bSet` is the parameter to enable or disable the fractional size option. Set the parameter to nonzero if you want to enable the fractional size option and 0 to disable.

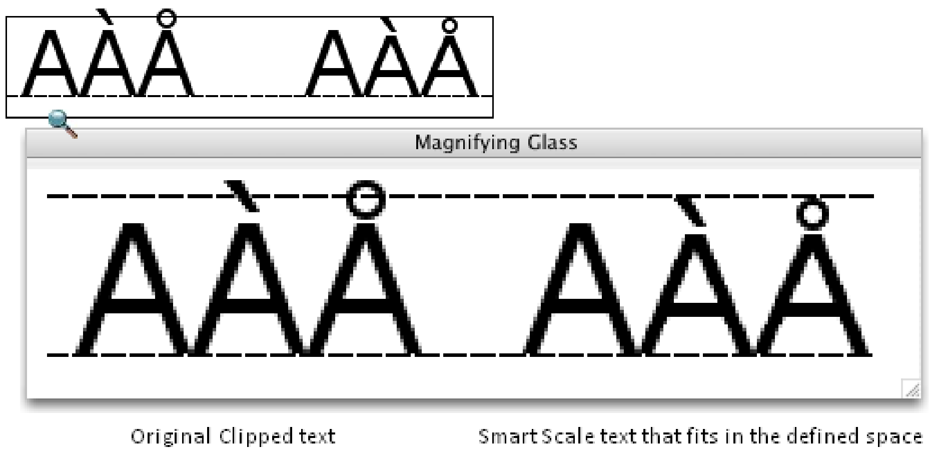
Description

The method `T2K_SetFracSizeMode()` sets the fractional size mode of scaler. The API `T2K_NewTranformation` should be called after setting this property.

Functions for SmartScale

Mobile devices have fixed screen parameters that force the characters to fit within the set height restrictions. This results in clipping of character portions that extend beyond the bounding box. Font Fusion overcomes this clipping problem by including the SmartScale mechanism that scales the characters such that full characters is visible within fixed display region. The technology ensures that the scaled characters are in proportion to the other characters in the font.

Few characters in a font may extend beyond the set bounding box limit. Examples of such characters include composite glyphs or glyph with extended attributes, such as accent acute, grave, and other diacritical marks or characters with typographic descenders as in lowercase g, p, or y. The SmartScale technology with Font Fusion intelligently scales these characters to fit within the bounding box without losing on readability and legibility.



Normal and zoomed-in view of original and smart scale text rendered using Font Fusion. Smart scaling regulates the adjustment of characters that extend beyond the set height parameters and may get clipped when rendered on small screen devices

The SmartScale technology works in all three rendering modes, monochrome, grayscale, and LCD mode. The mechanism adds slight performance overhead for the candidate glyphs, which is negligible in case the cache manager is used, since once the scaled glyph is rendered, it is also stored in the cache for next time reference.

T2K_SetSmartScale(

```
T2K * t,
int bSet)
```

Arguments

`t` is the pointer to the `t2k` object itself.

`bSet` is the parameter to enable or disable the SmartScale option. Set the parameter to nonzero if you want to enable the SmartScale option and 0 to disable.

Description

Call the `T2K_SetSmartScale()` API after creating `t2k` object. The t2k object can be created using:

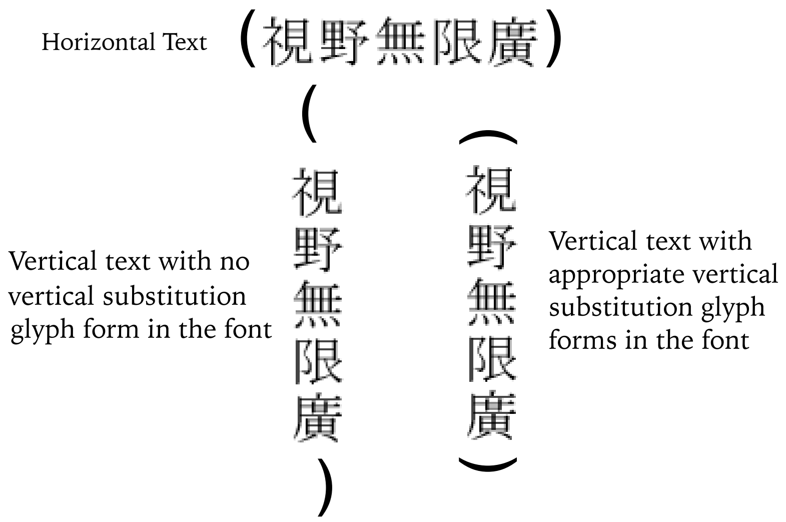
```
t2k = NewT2K(tsiMemObject *mem, sfntClass *font, int
*errCode)
```

If the parameter `bSet` is set to a nonzero value in `T2K_SetSmartScale()`, the function `T2K_RenderGlyph()` renders a smart-scaled glyph. Macro `ENABLE_SMARTSCALE` should be turned ON to use this API.

Functions for Vertical Writing

Proper rendering of far East Asian scripts including kanji variants and proportional Chinese-Japanese-Korean (CJK) requires vertical processing support. Vertical processing includes both translation/rotation as well as substitution of appropriate glyph variant (with changed orientation) for vertical writing.

The figure below shows how the vertical substitution works. When the text is rendered vertically without turning on vertical substitution (left side of the figure), the parentheses appear incorrectly. However, when vertical substitution is turned on (right side of the figure) the parentheses appear in the appropriate orientation.


Horizontal Text
(視野無限廣)
Vertical text with no vertical substitution glyph form in the font
(視野無限廣)
(視野無限廣)
Vertical text with appropriate vertical substitution glyph forms in the font

Vertical Substitution forms in a font

To enable the vertical processing, the following `RenderGlyph()` `cmd` options are used:

`cmd`	Description
`T2K_VERTICAL`	Causes vertical rotation and positioning
`T2K_VERT_SUB`	Specifies the use of Vertical substitution characters from the VT data segment (also used for GSUB support in TrueType fonts)

Functions for Translating Font Data

- ExtractPureT1FromPCType1()
- ExtractPureT1FromMacPOSTResources()

unsigned char *ExtractPureT1FromPCType1(

```
unsigned char *src,
unsigned long *length,
int *errCode)
```

Arguments

`src` is a pointer to the data to transform (the .PFB data read into memory).

`length` is a pointer to the length of the data. The value that this function returns is the length of the resulting font file data.

`errCode` is a pointer to the returned `errorCode`.

Description

Call this function to translate PC .PFB font file data into a pure, non-segmented form, which Font Fusion can then process.

char *ExtractPureT1FromMacPOSTResources(

```
tsiMemObject *mem,
short refNum,
unsigned long *length)
```

Arguments

`mem` is a pointer to the `tsiMemObject`.

`refnum` is the resource reference number for the font.

`length` is a pointer to the length of the data. The value that this function returns is the length of the resulting font file data.

Description

Call this function to translate Macintosh Type 1 font file data into a pure, non-segmented form which Font Fusion can then process.

Functions to Force Type1 Encoding

- `T2K_GetT1Encoding`
- `T2K_ForceT1Encoding`

T2K_GetT1Encoding(

```
T2K *t)
```

Arguments

`t` is the pointer to the `T2K` object itself.

Description

The function returns the current Type1 encoding type used to encode the font. Macro `ENABLE_T1_FORCE_ENCODING` should be turned ON to use this API.

Returns

Returns the current encoding type of Font Fusion being currently used to encode the Type1 font.

The `FF_T1_Encoding` enumeration is of the following type:

```
typedef enum {
   FF_T1_ADOBE_STANDARD_ENCODING,
   FF_T1_ISO_LATIN1_ENCODING,
   FF_T1_MAC_ENCODING,
   FF_T1_CUSTOM_ENCODING,
   FF_T1_DEFAULT_ENCODING = 0xffff,
}FF_T1_Encoding;
```

T2K_ForceT1Encoding(

```
T2K *t,
FF_T1_Encoding encType)
```

Arguments

`t` is the pointer to the `T2K` object

`encType` refers to the `FF_T1_Encoding` enumeration that lists the different font encoding types.

Description

The `T2K_ForceT1Encoding()` API forces the encoding on Type1 font. The method should be called only once during the lifetime of the font. Forcing an encoding can slow down Font Fusion performance.

Macro `ENABLE_T1_FORCE_ENCODING` should be turned ON to use this API.

Returns

Returns 0 if it fails to force the encoding else 1.

Functions For Use With Stroke-Based Fonts

- T2K_GetNumAxes()
- T2K_SetCoordinate()

int32 T2K_GetNumAxes(

```
T2K *t)
```

Arguments

`t` is a pointer to the T2K object itself.

Description

Returns the number of axes of the font specified. The return value will be one only for Stroke-Based fonts. This is useful when using an algorithmic bold. If it is a Stroke-Based font, use the `T2K_SetCoordinate()` function below to set an algorithmic bold. Both of these functions are used to control the weight of Stroke-Based font output.

void T2K_SetCoordinate(

```
T2K *t
int32 n
F16Dot16 value)
```

Arguments

`t` is a pointer to the T2K object itself.

`int32 n` indicates which axis on which to apply the `F16Dot16 value`.

`F16Dot16 value` is an emboldening amount specified as shown below.

Description

The “int32 n” in the T2K_SetCoordinate()call indicates which axis to apply the “value” to. Since stroke fonts only have 1 axis this value must be 0 for stroke fonts. The “value” is an emboldening amount specified as follows:

0x1000 (lightest) -> 0x8000 (50% = normal = default) -> 0x10000 (darkest)

Additional Functions

- FF_GetTTTablePointer( )
- FF_GlyphExists ( )
- ff_ColorTableType *FF_NewColorTable( )
- FF_PSNameToCharCode( )
- FF_SetBitRange255( )

uint8 *FF_GetTTTablePointer(

```
T2K *t
long tag
unsigned char **ppTbl
size_t *bufSize
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`tag` is a 4-byte identifier of the table, for example, ‘`cmap`’.

`ppTbl` is the address of a character pointer. This function will allocate this pointer by calling `CLIENT_MALLOC`.

`bufSize` is a pointer to return the size of the table.

`errCode` is a pointer to the returned `errorCode`.

Description

This function returns a pointer to a memory buffer containing any arbitrary TrueType table.

int FF_GlyphExists(

```
T2K *t
long code
uint16 cmd
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`code` is a character code.

`cmd` describes to Font Fusion what to do with various `bitflags`.

`errCode` is a pointer to the returned `errorCode`.

Description

This function checks for the existence of characters in a font. This makes it possible to test more certainly for glyph existence in font formats like Speedo, TrueDoc, Type1 and CFF fonts. The function returns true if the glyph exists, false otherwise.

void FF_ForceCMAPChange(

```
T2K *t
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`errCode` is a pointer to the returned `errorCode`.

Description

This function forces the unloading of the current TrueType character map (cmap) and loads the cmap * currently selected by **Set_PlatformID(scaler, ID)**, and **Set_PlatformSpecificID(scaler, ID)**.

This function presumes you have already used the set platform macros. The values set by those macros will be ignored if a character forced the loading of the TrueType cmap, unless you call this function.

ff_ColorTableType *FF_NewColorTable(

```
tsiMemObject *mem
uint16 Rb
uint16 Gb
uint16 Bb
uint16 Rf
uint16 Gf
uint16 Bf)
```

Arguments

`mem` is a pointer to the tsiMemObject.

`Rb` is the 8-bit **red** component of the background color.

`Gb` is the 8-bit **green** component of the background color.

`Bb` is the 8-bit **blue** component of the background color.

`Rf` is the 8-bit **red** component of the foreground color.

`Gf` is the 8-bit **green** component of the foreground color.

`Bf` is the 8-bit **blue** component of the foreground color.

Description

The `ff_ColorTableType` constructor supports color combinations other than black against white. The foreground and background color values need to be in the range 0-255.

Here are step by step instructions:

1. Use **`FF_NewColorTable`** to get the RGB colors for LCD display. These colors will be indexed by any bitmap produced by T2K. If your platform is using a Color Lookup Table, you will need to set these colors in that table. This is how you extract the actual RGB colors from the T2K color table:

```
ff_ColorTableType *pColorTable;
/* for black text on white Set Rb = Gb = Bb = 0xff,
 * and Rf = Gf = Bf = 0. */
pColorTable = FF_NewColorTable( mem, Rb, Gb, Bb, Rf, Gf, Bf );
/* For all the indices in the bitmap you get the color by doing
this.*/
/* pColorTable->N will contain # elements in the array */
/* pColorTable->ARGB[0] contains the first ARGB value */
ARGB = pColorTable->ARGB[ byte index from the bitmap ];
B = (ARGB & 0xff); ARGB >>= 8;
G = (ARGB & 0xff); ARGB >>= 8;
R = (ARGB & 0xff);
/* When done free up the color-table, but please do not call this
per
*  character for speed reasons. */
FF_DeleteColorTable( mem, pColorTable);
```

2 Do not invoke either **`FF_SetBitRange255()`** or **`FF_SetRemapTable()`**. If you need to shift the range, we recommend using a filter function.

3 Invoke **`T2K_NewTransformation()`** to set the transformation matrix.

NOTE: For standard LCD modes, set the `xRes` to 3 times the `yRes`, since LCD screens contain three times as many colored pixels in the x direction as in the y direction. Setting these resolutions in this way tells T2K we have a non-square aspect ratio where the x resolution is three times higher than the y resolution. You do not need to do this for the extended modes as these modes handle the conversion internally.

4 In the cmd parameter, turn on the bit flag to the LCD mode option you want to use, for example, `T2K_EXT_LCD_H_RGB` to use the most common extended LCD mode.

5 In the greyScalelevel parameter to the **`T2K_RenderGlyph()`** function, set `GREY_SCALE_BITMAP_HIGH_QUALITY` in the `greyScalelevel` parameter .

You now have an indexed color bitmap. When you draw the bitmap you need to take into account that the bitmap contains indices for the colored pixels.

void FF_ModifyColorTable(

```
register ff_ColorTableType *t,
uint16 Rb,
uint16 Gb,
uint16 Bb,
uint16 Rf,
uint16 Gf,
uint16 Bf)
```

Arguments

`t` is a pointer to an existing Color Table to be modified.

`Rb` is the Red background value.

`Gb` is the Green background value.

`Bb` is the Blue background value.

`Rf` is the Red foreground value.

`Gf` is the Green foreground value.

`Bf` is the Blue foreground value.

Description

This routine modifies the existing colorTable (pointed to by argument “`t`”) with the specified foreground and background values.

int FF_PSNameToCharCode(

```
T2K *t
char *PSName
int *errCode)
```

Arguments

`t` is a pointer to the `T2K` object itself.

`PSName` is a pointer to a PostScript® glyph name.

`errCode` is a pointer to the returned `errorCode`.

Description

This function converts a PostScript font name to a character code. This works for Type 1 and CFF/Type 2 fonts only.

int FF_SetBitRange255(

```
T2K *t
boolean value)
```

Arguments

`t` is a pointer to the T2K object itself.

`true` sets the gray scale native range from the default (0 to 126) to 0 to 255.

`false` sets the gray scale native range back to the default of 0 to 126.

Description

Font Fusion returns anti-aliased text within a range of 0 to 126, which allows for 127 shades of gray. Some systems require 256 shades of gray. These systems can now set the boolean value to false to return 256 shades of gray. It is faster to use the gray scale native range of 0 to 126. This function is optional and can be called any time before T2K_RenderGlyph.

Sample Code

Macintosh

This is a Macintosh code example of linear text layout using kerning.

```
totalWidth = T2K_MeasureTextInX( scaler, string16, kern,
numChars);
for ( i = 0;  (charCode = string16[i]) != 0; i++ ) {
F16Dot16 xKern, yKern;

/* Create a character */
T2K_RenderGlyph( scaler, charCode, 0, 0,
BLACK_AND_WHITE_BITMAP, T2K_GRID_FIT | T2K_RETURN_OUTLINES
| T2K_SCAN_CONVERT, &errCode );
assert( errCode == 0 );
T2K_TransformXFunits( scaler, kern[i], &xKern, &yKern );

bm->baseAddr = (char *)scaler->baseAddr;
bm->rowBytes = scaler->rowBytes;
bm->bounds.left = 0;
bm->bounds.top= 0;
bm->bounds.right= scaler->width;
bm->bounds.bottom= scaler->height;

MyDrawChar( graf, x + ( (scaler-
>fLeft26Dot6+(xKern>>10))>>6), y - (scaler-
>fTop26Dot6+(yKern>>10)>>6), bm );
/* We keep x as 32.16 */
x16Dot16 += scaler->xLinearAdvanceWidth16Dot16 + xKern; x +=
x16Dot16>>16; x16Dot16 &= 0x0000ffff;
/* Free up memory */
T2K_PurgeMemory( scaler, 1, &errCode );
assert( errCode == 0 );
}
```

T2K Scaler

This shows a “pseudo code” example for how to use the T2K scaler.

```
/* First configure T2K, please see "CONFIG.H" !!! */

  tsiMemObject *mem = NULL;
  InputStream *in = NULL;
  sfntClass *font = NULL;
  T2K *scaler = NULL;
  int errCode;
  T2K_TRANS_MATRIX trans;
  T2K_AlgStyleDescriptor style;


  /* Create a Memhandler object. Use ONE per font. */
  mem= tsi_NewMemhandler( &errCode );
  assert( errCode == 0 );
    /* Point data1 at the font data */
    If ( TYPE 1 ) {
      if ( PC Type 1 ) {
        /* Only call for .pfb files and NOT for .pfa files. */
        data1 = ExtractPureT1FromPCType1( data1, &size1, &errCode );
        /* data1 is not allocated just munged by this call ! */
      } else if ( Mac Type 1 ) {
        short refNum = OpenResFile( pascalName ); /* Open the resource with
some Mac call */
        data1 = (unsigned char *)ExtractPureT1FromMacPOSTResources( mem,
refNum, &size1 );
        CloseResFile( refNum ); /* Close the resource file with some Mac call
*/
        /* data1 IS allocated by the T2kMemory layer! */
      }
    }
    /* Please make sure you use the right New_InputStream call depending on who
allocated data1,
       and depending on if the font is in ROM/RAM or on the disk/server etc. */
    /* Create an InputStream object for the font data */
    in = New_InputStream( mem, data1, size1, &errCode ); /* if data allocated by
the T2kMemory layer */
    assert( errCode == 0 );
    **** OR ****
    in = New_InputStream3( mem, data1, size1, &errCode ); /* otherwise do this
if you allocated the data  */
    **** OR *****
    /* Allows you to leave the font on the disk, or remote server for instance
(!) */
    in = New_NonRamInputStream( mem, fpID, ReadFileDataFunc, length, &errCode
);

    assert( errCode == 0 );
      /* Create an sfntClass object. (No algorithmic styling) */
```

```
       short fontType = FONT_TYPE_TT_OR_T2K; /* Or, set equal to FONT_TYPE_1 for
type 1, FONT_TYPE_2 for CFF fonts */
       font = New_sfntClass( mem, fontType, in, NULL, &errCode );
       **** OR ****
       /* alternatively do this for formats that support multiple logical fonts
within one file */
       font = FF_New_sfntClass( mem, fontType, logicalFontNumber, in, NULL,
NULL, &errCode );

       /* Or if you wish to use algorithmic styling do this instead
        * T2K_AlgStyleDescriptor style;
        *
        * style.StyleFunc = tsi_SHAPET_BOLD_GLYPH;
        * style.StyleMetricsFunc=tsi_SHAPET_BOLD_METRICS;
        * style.params[0] = 5L << 14; (* 1.25 *)
        * font = New_sfntClass( mem, fontType, in, &style, &errCode );
        */
       assert( errCode == 0 );
         /* Create a T2K font scaler object.  */
         scaler = NewT2K( font->mem, font, &errCode );
         assert( errCode == 0 );
           /* 12 point */
           trans.t00 = ONE16Dot16 * 12;
           trans.t01 = 0;
           trans.t10 = 0;
           trans.t11 = ONE16Dot16 * 12;
           /* Set the transformation */
           T2K_NewTransformation( scaler, true, 72, 72, &trans, true, &errCode
);
           assert( errCode == 0 );
           loop {
             /* Create a character */
             T2K_RenderGlyph( scaler, charCode, 0, 0, BLACK_AND_WHITE_BITMAP,
T2K_GRID_FIT | T2K_RETURN_OUTLINES  | T2K_SCAN_CONVERT, &errCode );
             assert( errCode == 0 );
             /* Now draw the char */
             /* Free up memory */
             T2K_PurgeMemory( scaler, 1, &errCode );
             assert( errCode == 0 );
           }
         /* Destroy the T2K font scaler object. */
         DeleteT2K( scaler, &errCode );
         assert( errCode == 0 );
       /* Destroy the sfntClass object. */
       FF_Delete_sfntClass( font, &errCode );
       assert( errCode == 0 );
     /* Destroy the InputStream object. */
     Delete_InputStream( in, &errCode  );
     assert( errCode == 0 );
   /* Destroy the Memhandler object. */
   tsi_DeleteMemhandler( mem );
```

Font Manager API

4

Getting Started with the Font Manager

This section answers the following questions:

- Why use the Font Manager?
- What files should I look at first?
- How do I use the Font Manager?
- Why do Font Fusion, the Font Manager, and the Cache Manager have a **`RenderGlyph`** function?
- How does the Cache Manager know if the Font Manager should render a glyph? What's the configuration requirement for me to make these work together?
- How many fonts can I handle at once?
- Are there any other configuration parameters for the Font Manager?
- Why does the **`FF_FM_AddTypefaceStream()`** function take two stream arguments?
- Why does **`FF_FM_CreateFont()`** include a `flushCache` parameter?
- Do you have a coding example?

Why Use the Font Manager?

The Font Manager hides some complexities of using the Font Fusion Core. So it can make your life a bit easier. But mostly the Font Manager allows you to have more than one outline font “registered” and ready to use with the Font Fusion Core. It saves you from building up your own code to handle multiple fonts.

Your application can get information about each font input stream that Font Fusion “installs,” which is also convenient.

But one of the most important reasons for using the Font Manager is that across all font technologies, the Font Manager can dynamically merge font fragments of the same font. This supports dynamic downloading of font fragments of large character set fonts, such as Chinese, Japanese, or Korean (CJK) fonts.

It can also support adding characters to a font by adding small font fragments with the new characters. Characters in newer fragments override the same characters in earlier fragments. This means that, if your system embeds fonts in ROM, you can load a new font fragment along with the old font at run time.

What Files Should I Look at First?

First, you need to familiarize yourself with `t2k.h`. This file contains documentation, a coding example, and the actual Font Fusion API.

Second, you need to look at `config.h`. This file is the only file you normally need to edit. The file configures Font Fusion for your platform, it enables or disables optional features, and it allows you to build debuggable or non-debuggable versions. The file itself contains more information. Turn off features you do not need in order to minimize the size of the Font Fusion Core.

Third, you need to look at `ff_fm.h`. This file describes the API of the Font Fusion Font Manager.

How Do I Use the Font Manager?

Follow the steps below to use the Font Manager in your application:

1. Create the Font Manager class.

2. Configure the Font Manager at run-time, setting the platform ID, specific ID, language ID, name ID (which all affect only font names you'll receive in font properties or in font enumeration), and the horizontal and vertical resolution.

   NOTE: These configurations are optional, since the Font Manager has viable usable values for North American Latin fonts and presumes a 72 dpi by 72 dpi resolution.

3. You can then add font streams to the Font Manager at any time. Refer to the `t2k.h` file, or the “InputStream Functions: Overview” on page 53 for details on creating input streams.

4. After adding streams, you can call the font enumeration function, **`FF_FM_EnumTypefaces()`**, to find out what fonts are available and how many there are.

5. From the available fonts, you create strikes. When you create a strike, you are given a token representing the strike, which you use to select it.

6 Once you select a strike, you can render characters from it.

Font Fusion places all output in the `T2K` class just as if you were using the Font Fusion Core. See `t2k.h` for more information on using the Font Fusion Core.

With the Font Manager, the details of creating a `T2K` class and an `sfntClass` are managed by the Font Manager, making your life simpler. However, you still need to create input streams.

Why Do Font Fusion, the Font Manager, and the Cache Manager Have a RenderGlyph() Function?

We designed them that way so they could be independent of each other and work together.

The real **`RenderGlyph`** work is always done in the Font Fusion Core. If you are using the Cache Manager, **`FF_CM_RenderGlyph()`** first checks the cache for the glyph or calls another module to render the glyph and store it in the cache. It uses either the Font Manager **`RenderGlyph()`** function or it calls the Font Fusion Core.

The Font Manager **`RenderGlyph()`** function looks for the requested glyph from among the font fragments of the font, and then calls the **`T2K_RenderGlyph()`** function.

How Does the Cache Manager Know if the Font Manager Should Render a Glyph? What's the Configuration Requirement for Me to Make These Work Together?

There is no configuration requirement! You just build the Font or Cache Manager and use each at run time.

If you “register” a font with the Font Manager, when you create and select a strike, the Font Manager stamps or marks itself in the `T2K` class to let the Cache Manager know it is present.

If you are using the Cache Manager, the Cache Manager will respect this stamp, which consists of enough information for the Cache Manager to use the Font Manager’s API.

How Many Fonts Can I Handle at Once?

We designed the Font Manager to handle up to 64K InputStreams, 64K physical fonts, 64K logical fonts, and 128 “Strikes.”

Input streams are similar to (and often are) font files. Within some of these streams (such as TrueType Collections and TrueDoc PFRs), there can be more than one physical font.

Logical fonts are merged “super” fonts made up of one or more physical font fragments. Physical and logical fonts are only concerned with outline font resources.

A “strike” is an instance of an outline resource that Font Fusion scales and transforms to a particular size, aspect ratio, italic angle, etc. The default limit on the number of strikes is 128, but you can override the this limit by redefining `FF_FM_MAX_DYNAMIC_FONTS`.

Are There Any Other Configuration Parameters for the Font Manager?

No.

Why Does FF_FM_AddTypefaceStream() Take

Two Stream Arguments?

Some font technologies have metrics information in a second file. For example, Adobe® ships `.afm` files containing font metrics, including kerning information. Use the second stream argument for these auxiliary metrics files or streams. For other technologies or if you are not using kerning, pass `NULL` for the second stream parameter.

Why Does FF_FM_CreateFont() Include a flushCache Parameter?

This parameter is there to signal applications using the Cache Manager that the cache needs to be flushed. There is only one reason this would ever happen: when the Font Manager re-uses a font code from a previously deleted strike, it signals the cache to flush to purge glyphs that may be stranded in the cache from the deleted font.

If you are not using a Cache Manager, you can always ignore this parameter's value.

Is There a Coding Example?

Yes. There is a coding example at the end of this chapter.

Functions for Creating, Configuring, and Deleting the Font Manager

- FF_FM_New ( )
- FF_FM_AddTypefaceStream ( )
- FF_FM_SetPlatformID ( )
- FF_FM_SetPlatformSpecificID ( )
- FF_FM_SetLanguageID ( )
- FF_FM_SetNameID ( )
- FF_FM_Delete ( )

FF_FM_Class *FF_FM_New(

```
int *errCode)
```

Arguments

`errCode` is a pointer to the returned error code.

Description

`FF_FM_New()` creates a new instance of the Font Fusion Font Manager.

It returns a context pointer, which is `NULL` on failure. The `errCode` parameter is clear on success.

Possible error codes:

`T2K_ERR_MEM_MALLOC_FAILED`

void FF_FM_AddTypefaceStream(

```
FF_FM_Class *pFM,
InputStream StreamA,
InputStream StreamB,
int *errCode)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`StreamA` is the font data stream.

`StreamB` is a stream for additional font information. Some font technologies have metrics information in a second file. For example, Adobe ships `.afm` files containing font metrics, including kerning information. Use the second stream argument for these auxiliary metrics files or streams. For other technologies or if you are not using kerning, pass `NULL` for the second stream parameter.

`errCode` is a pointer to the returned error code.

Description

`FF_FM_AddTypefaceStream()` installs an outline font-resource stream to the Font Manager context. The `errCode` parameter returns 0 on success, or an error code on failure.

Possible error codes:

```
T2K_ERR_MEM_MALLOC_FAILED
T2K_ERR_MEM_REALLOC_FAILED
```

void FF_FM_SetPlatformID(

```
FF_FM_Class * pFM,
uint16 platformID)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`platformID` is the platform ID for accessing character map tables. It is the same as what Microsoft's TrueType documentation specifies.

Description

`FF_FM_SetPlatformID()` sets the platform ID for accessing `cmap` (character map) tables in TrueType and native T2K fonts. The default platform ID is 3 (Microsoft). Call this function to change to another platform ID.

In the Font Manager, this affects the settings stored in the `T2K` scaler created in **`FF_FM_SelectFont()`**.

For other font technologies, the Font Manager ignores this setting.

void FF_FM_SetPlatformSpecificID(

```
FF_FM_Class * pFM,
uint16 platformSpecificID)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`platformSpecificID` is the platform-specific ID for accessing character map tables. It is the same as what Microsoft's TrueType documentation specifies.

Description

`FF_FM_SetPlatformSpecificID()` sets the platform-specific ID for accessing `cmap` tables in TrueType and native T2K fonts. The default platform-specific ID is 1 (Unicode). Call this function to change to another platform-specific ID.

In the Font Manager, this affects the settings stored in the `T2K` scaler created in **`FF_FM_SelectFont()`**.

For other font technologies, the Font Manager ignores this setting.

void FF_FM_SetLanguageID(

```
FF_FM_Class * pFM,
uint16 languageID)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`languageID` is the language ID for accessing name tables. It is the same as what Microsoft's TrueType documentation specifies.

Description

`FF_FM_SetLanguageID()` sets the language ID for accessing name tables in TrueType and native T2K fonts. The default language ID is `0x0409` (Microsoft, American English). Call this function to change to another language ID for name tables.

In the Font Manager, this affects the name strings that the `enumTypefaceCallBack` argument of the **`FF_FM_EnumTypefaces()`** function returns. Call this function as needed before **`FF_FM_AddTypefaceStream()`**.

For other font technologies, Font Fusion ignores this setting.

void FF_FM_SetNameID(

```
FF_FM_Class * pFM,
uint16 nameID)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`nameID` is the name ID for accessing name tables. It is the same as what Microsoft's TrueType documentation specifies.

Description

`FF_FM_SetNameID()` sets the name ID for accessing name tables in TrueType and native T2K fonts. The default name ID is 4 (full font name). Call this function to change to another name ID.

In the Font Manager, this affects the name strings that the `enumTypefaceCallBack` argument of the **`FF_FM_EnumTypefaces()`** function returns. Call this function as needed before **`FF_FM_AddTypefaceStream()`**.

For other font technologies, Font Fusion ignores this setting.

`void FF_FM_Delete(`

```
FF_FM_Class * pFM,
int *errCode)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`errCode` is a pointer to the returned error code.

Description

`FF_FM_Delete()` deletes the Font Fusion Font Manager context and cleans up memory.

Function for Installing Fonts and Getting Font Information

- FF_FM_EnumTypefaces ( )

enumTypefaceCallback() Function

The **`enumTypefaceCallback()`** function is a callback function that you write. The **`FF_FM_EnumTypefaces()`** API function calls the callback once for each font installed in the Font Manager context.

Regarding the `faceName` parameters of the callback, one or the other of these parameters is not `NULL`. The non-`NULL` parameter points to the name of the font. The `faceName8` parameter points to a null-terminated, eight-bit (one-byte) string. The `faceName16` parameter points to a null-terminated, sixteen-bit (two-byte) string.

int FF_FM_EnumTypefaces(

```
FF_FM_Class * pFM,
int enumTypefaceCallBack(uint8 *faceName8,
uint16 *faceName16))
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`enumTypefaceCallBack` is a function you write that the Font Manager calls for each logical font in the internal list order that the Font Manager keeps. It allows your application to find out how many fonts are actually available and what their names are.

Description

`FF_FM_EnumTypefaces()` enumerates the available logical fonts that the Font Manager finds among the outline font resources. At least two of the supported type technologies allow multiple logical fonts in each outline resource: TrueType Collections and TrueDoc PFRs.

Once you install the outline resources, your application can use this function to determine what logical fonts are available.

This function returns whatever the **`enumTypefaceCallBack()`** function returns the last time **`FF_FM_EnumTypefaces()`** calls it.

The Font Manager calls the **`enumTypefaceCallBack()`** function once for each logical font in the internal list order that the Font Manager keeps, or until the **`enumTypefaceCallBack()`** function returns a non-zero value.

Functions for Creating and Using Fonts

- FF_FM_CreateFont ( )
- FF_FM_SetXYResolution ( )
- FF_FM_SelectFont ( )
- FF_FM_DeleteFont ( )
- FF_FM_RenderGlyph ( )

uint16 FF_FM_CreateFont(

```
FF_FM_Class * pFM,
uint16 index,
boolean *flushCache,
T2K_TRANS_MATRIX *trans,
T2K_AlgStyleDescriptor *styling,
int *errCode)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`index` is the logical font index of the installed font. This means that when you call **`FF_FM_EnumTypefaces()`**, the first time it calls the callback function it gives you information about logical font 0, the second time about logical font 1, and so on. So once you have enumerated the fonts, you will know which index you want.

`flushCache` is a pointer to a TRUE or FALSE setting. It is TRUE if creating the font requires Font Fusion to flush the cache. In this event, existing valid tokens of previously created fonts remain valid. The cache flush is rarely needed to clean out stranded glyphs from fonts that your application already deleted. If you are not using a Cache Manager, you can always ignore this parameter's value.

`trans` is a pointer to the current transformation matrix. Basically, you specify:

- the `x` and `y` resolutions
- a `2*2` transformation matrix (including the point size)
- a true or false setting to enable or disable embedded bitmaps

After you are done with the output data, you call **`FF_FM_DeleteFont()`** to delete the font you created with this function.

`styling` is a pointer to a function that modifies the outlines algorithmically. This is normally == `NULL`. The compile-time option `ALGORITHMIC_STYLES` enables algorithmic styling. If you enable `ALGORITHMIC_STYLES`, you can set it equal to an algorithmic style descriptor. Here is an example using the algorithmic emboldening that Font Fusion provides.

```
style.StyleFunc= tsi_SHAPET_BOLD_GLYPH;
style.StyleMetricsFunc=tsi_SHAPET_BOLD_METRICS;
style.params[0] = 5L << 14;
sfnt0 = FF_New_sfntClass( mem, fontType, 0, in, NULL, &style,
&errCode );
```

You can also write your own outline-based style modifications and use them instead of the algorithmic emboldening that Font Fusion provides. Just model them after the code for algorithmic emboldening in `shapet.c`.

`errCode` is a pointer to the returned error code.

Description

`FF_FM_CreateFont()` creates a font instance (an outline resource plus a transformation) and returns a font code for your application to use to select a font. Note, however, that this functions does not select the font.

This function also allows you to set the transformation matrix and `x` and `y` resolutions when you render characters and strings. It informs the `T2K` object about the current transformation and size.

The font code is valid if the `errCode` parameter is not set.

This function also sets `flushCache` to TRUE if creating the font requires Font Fusion to flush the cache. In this event, existing valid tokens of previously created fonts remain valid. The cache flush is rarely needed to clean out stranded glyphs from fonts that your application already deleted.

Possible error returns (in `errCode`):

```
FF_FM_ERR_FONT_OFLO_ERR
FF_FM_ERR_BAD_INDEX
T2K_ERR_MEM_MALLOC_FAILED
```

How Do I Calculate Character Sizes

Character size is best expressed by the following formula:

Size = # of lines = point size/72 Points Per Inch (PPI) * resolution.

For example,

Size = 100 lines = 24pt/72PPI * 300 Dots Per Inch (DPI).

Examples of Transformations

In the following examples, `size` is a fractional number in `16.16` format.

Typically, you have the following:

```
trans.t00 = size;
trans.t01 = 0;
trans.t10 = 0;
trans.t11 = size;
```

To condense the text in the `x`-direction to 80 percent, use the following. We do not promote condensing text, since there are condensed fonts designed that way, but the following example shows you how to do it.

```
trans.t00 = util_FixMul( size, 8*0x10000/10 );
trans.t01 = 0;
trans.t10 = 0;
trans.t11 = size;
```

To stretch the text in the `y`-direction to 125 percent, use the following. We do not promote extending text, but the following example shows you how to do it.

```
trans.t00 = size;
trans.t01 = 0;
trans.t10 = 0;
trans.t11 = util_FixMul( size, 125*0x10000/100 );
```

To rotate the text at an angle, alpha-measured **clockwise** from the `x`-axis, use the following. If the angle is in the first quadrant, it is negative.

```
trans.t00 = util_FixMul( size , cosvalue );
trans.t01 = util_FixMul( size , sinvalue );
trans.t10 = util_FixMul( size , -sinvalue );
trans.t11 = util_FixMul( size , cosvalue );
```

In the example above, the `cosvalue` and `sinvalue` above are `cos(angle)` and `sin(angle)` in `16.16` format.

void * FF_FM_SetXYResolution(

```
FF_FM_Class * pFM,
long xRes,
long yRes)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`xRes` is the horizontal resolution for the output display in dots-per-inch (dpi). The default is 72 dpi.

`yRes` is the vertical resolution in dots-per-inch (dpi). The default is 72 dpi.

Description

`FF_FM_SetXYResolution()` sets the internal, default dots-per-inch (dpi) for the display resolution through the Font Manager.

The internal default is a horizontal resolution of 72 dpi, and a vertical resolution of 72 dpi. Using this function, your application can change the output resolution specification before calling **`FF_FM_SelectFont()`**.

T2K * FF_FM_SelectFont(

```
FF_FM_Class * pFM,
uint16 fontCode,
int *errCode)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`fontCode` is the font code for a font instance you previously created with a call to **`FF_FM_CreateFont()`**.

`errCode` is a pointer to the returned error code.

Description

`FF_FM_SelectFont()` selects a previously created font instance. The selected font is now active in the `T2K` scaler object and ready for Font Fusion to render glyphs.

The `errCode` parameter is set on failure, indicating that the `T2K` scaler object that this function returned is invalid. Possible error code returns (in `errCode`):

```
T2K_ERR_MEM_IS_NULL
FF_FM_ERR_BAD_FONTCODE
```

void FF_FM_DeleteFont(

```
FF_FM_Class * pFM,
uint16 fontCode,
int *errCode)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`fontCode` is the font code for a font instance you previously created with a call to **`FF_FM_CreateFont()`**.

`errCode` is a pointer to the returned error code.

Description

`FF_FM_DeleteFont()` deletes a previously created font instance. The font instance represented by `fontCode` becomes invalid and images from it may be stranded in the cache, requiring you to flush the cache.

This function returns nothing. It sets `errCode` to 0 on success, or it returns:

```
FF_FM_ERR_FONT_CODE_ERR
```

void FF_FM_RenderGlyph(

```
FF_FM_Class * pFM,
uint16 fontCode,
T2K **pScaler,
long code,
int8 xFracPenDelta,
int8 yFracPenDelta,
uint8 greyScaleLevel,
uint16 cmd,
int *errCode)
```

Arguments

`pFM` is a pointer to the current Font Manager context.

`fontCode` is the font code for a font instance you previously created with a call to **`FF_FM_CreateFont()`**.

`pScaler` is a pointer to a pointer to the current `T2K` scaler object.

`xFracPenDelta` and `yFracPenDelta` are normally set to zero. You can use them with non-zero values if you are also using fractional character positioning.

`greyScaleLevel` describes the level of anti-aliasing you want to apply. For more information, see the section “Bits for the `greyScaleLevel` argument” described in the **`T2K_RenderGlyph()`** function in Chapter 3.

`cmd` describes to Font Fusion what to do with various `bitflags`. For more information, see the section “Bits for the `cmd` argument” described in the **`T2K_RenderGlyph()`** function in Chapter 3.

`errCode` is a pointer to the returned error code.

Description

`FF_FM_RenderGlyph()` renders a glyph from a font instance active in the `T2K` scaler object.

It uses the `fontCode` parameter to reference internal `FF_FM` data structures to walk through “merged” font fragments until Font Fusion successfully renders the glyph or depletes the fragment list. This function may alter what `pScaler` points to as it is making each glyph. Possible error codes include error set by **`setjmp()`**. Refer to the **`T2K_RenderGlyph()`** function in Chapter 3.

Sample Code

The following is a simple example program using the Font Manager and the Font Fusion Core.

```
  /* First configure T2K, please see "CONFIG.H" !!! */
  /* compile and link with
             Font Fusion Core sources,
             Font Manager sources
             (and optionally Cache Manager sources)
             and standard 'C' libraries
  */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "t2k.h"
#include "cachemgr.h"
#include "ff_fm.h"

#define ALL_BLACK_AND_WHITE1/* change to 0 for greyscale */
#define CACHE_SIZE(32*1024)
#define USE_CACHE0/* change to 1 for testing use with Cache Manager */

/* Function Prototypes */
int main(int argc, char* argv[]);
static int enumFontCB(uint16 listIndex, uint8 *faceName8, uint16 *faceName16);
static void DoPrintFontProperties(T2K *aScaler);
static void print16Dot16(F16Dot16 aCode);
static void PrintChar( T2K *scaler );

/* PROGRAM CODE: */
/*
 * Main Program: Example of using the Font Fusion Font Manager (and Cache
Manager)
 */
int main(int argc, char* argv[])
{
/* LOCALS: */
int errCode = 0;
FF_FM_Class *pFMGlobals;
#if USE_CACHE
FF_CM_Class *theCache;
uint8 filterTag = 0;/* anywhere from 0...255 */
#endif
char flushCache;
uint16 fontCode, faceIndex;
T2K_TRANS_MATRIX trans;
T2K_AlgStyleDescriptor styleDesc, *stylePtr;
T2K *aScaler;
int testSize, jj;
long charCode;
```

```
#if ALL_BLACK_AND_WHITE
uint8 greyScaleLevel = BLACK_AND_WHITE_BITMAP;
uint16 cmd = T2K_NAT_GRID_FIT | T2K_GRID_FIT | T2K_SCAN_CONVERT;
#else
uint8 greyScaleLevel = GREY_SCALE_BITMAP_HIGH_QUALITY;
uint16 cmd = T2K_TV_MODE  | T2K_SCAN_CONVERT;
#endif
tsiMemObject *mem = NULL;
char *fName = "TT0003M_.TTF";
FILE *fpID;
unsigned long length, count;
unsigned char *data = NULL;
InputStream *InputStreamA = NULL, *InputStreamB = NULL;
char success = false;/* we will set to true if we create an input stream OK */

  /* CODE BEGINS: */
  UNUSED (argc);
  UNUSED (argv);
  printf("Hello World from Font Fusion Font Manager Demo Program!\n");

  /* Create a Memhandler object. Use ONE for up to 507 (?) streams. */
  mem= tsi_NewMemhandler( &errCode );
  assert( errCode == 0 );
  /* load a disk file into memory and make an input stream:*/
  if (mem != NULL && !errCode)
  {
    /* this is always deeply nested, eh? */
    fpID = fopen(fName, "rb");
    assert( fpID != NULL );
    if (fpID)
    {
      errCode = fseek( fpID, 0L, SEEK_END );
      assert( errCode == 0 );
      if (!errCode)
      {
        length = (unsigned long)ftell( fpID );
        assert( ferror(fpID) == 0 );
        if ( ferror(fpID) == 0 )
        {
          errCode = fseek( fpID, 0L, SEEK_SET );  /* rewind */
          assert( errCode == 0 );
          if (!errCode)
          {
            data = (unsigned char *)CLIENT_MALLOC( sizeof( char ) * length );
            assert( data != NULL );
            if (data)
            {
              count = fread( data, sizeof( char ), length, fpID );
              assert(ferror(fpID) == 0 && count == length );
              if (ferror(fpID) == 0 && count == length )
              {
                errCode = fclose( fpID );
                assert( errCode == 0 );
```

```
                    /* Please make sure you use the right New_InputStream call 
depending on who allocated data1,
                       and depending on if the font is in ROM/RAM or on the disk/
server etc. */
                    /* Create an InputStream object for the font data */
                    InputStreamA = New_InputStream3( mem, data, length, &errCode 
);
                    assert( errCode == 0 );
                    success = true;
                  }
                  else printf("Failed fread() of file!\n");
                }
                else printf("Failed allocating data buffer size = %ld, for 
file!\n", length);
              }
              else printf("Failed rewinding file with fseek()!\n");
            }
            else printf("Failed getting size of file with ftell()!\n");
          }
          else printf("Failed fseek() to end of file!\n");
        }
        else printf("Failed opening file: %s with fopen()!\n", fName);
      }
      else printf("Failed getting new mem handler with tsi_NewMemhandler()!\n");
      InputStreamB = NULL;

      if (success)
      {
#if USE_CACHE
        /* Create a new Cache Manager to play around with. */
        theCache = FF_CM_New(CACHE_SIZE, &errCode);
        assert( errCode == 0 );
        if (theCache)
        {
#endif
          pFMGlobals = FF_FM_New(&errCode);
          if (pFMGlobals)
          {
            /* configure the way we like it: */
            FF_FM_SetPlatformID(pFMGlobals,3);
            FF_FM_SetPlatformSpecificID(pFMGlobals,1);
            FF_FM_SetLanguageID(pFMGlobals,0x0409);
            FF_FM_SetNameID(pFMGlobals,4);
            FF_FM_SetXYResolution(pFMGlobals, 72, 72);
            /* Add an input stream pack to the Font Manager */
            FF_FM_AddTypefaceStream(pFMGlobals,
                                 InputStreamA,
                                 InputStreamB,
                                 &errCode);
            assert (errCode == 0);

            /* TESTING: FF_FM_EnumTypefaces() */
            FF_FM_EnumTypefaces (pFMGlobals, enumFontCB);
```

```
         /* TESTING: FF_FM_CreateFont() */
         stylePtr = NULL; /* we are not using the styleDesc variable at all */
#ifdef ALGORITHMIC_STYLES
         styleDesc.StyleFunc= tsi_SHAPET_BOLD_GLYPH;
         styleDesc.StyleMetricsFunc=tsi_SHAPET_BOLD_METRICS;
#else
         styleDesc.StyleFunc= NULL;
         styleDesc.StyleMetricsFunc=NULL;
#endif
         styleDesc.params[0] = 5L << 14;

         testSize = 24;/* 24 lines per em */
         faceIndex = 0;
         trans.t00 = ONE16Dot16 * testSize;
         trans.t01 = 0;
         trans.t10 = 0;
         trans.t11 = ONE16Dot16 * testSize;
         fontCode = FF_FM_CreateFont(
                     pFMGlobals,
                     faceIndex,
                     &flushCache,
                     (T2K_TRANS_MATRIX *)&trans,
                     stylePtr,
                     &errCode);
         assert(errCode == 0);
#if USE_CACHE
         if (flushCache) /* will not happen with just one font created */
           FF_CM_Flush(theCache, &errCode);
         assert(errCode == 0);
         filterTag = 0;
         FF_CM_SetFilter(theCache,
                     filterTag,
                     NULL,
                     NULL);/* all characters from now on will be coded with this
tag */
#endif

         aScaler = FF_FM_SelectFont(pFMGlobals,
                       fontCode,
                       &errCode);
         if (aScaler && !errCode)
         {
           /* print font properties */
           T2K_SetNameString( aScaler, 0x409, 4 ); /* see if we can get a font
name, too */
           DoPrintFontProperties(aScaler);

           printf("'A - Z' test\n");
           charCode = 'A';
           for (jj = 0; jj < 26; jj++, charCode++)
           {
             printf("\n***Here comes the %c ****\n", (char)charCode);
#if USE_CACHE
             FF_CM_RenderGlyph(theCache,fontCode,
                        &aScaler, charCode,
                        0, 0,
                        greyScaleLevel, cmd, &errCode);
#else
                 FF_FM_RenderGlyph(pFMGlobals,fontCode,
                        &aScaler, charCode,
                        0, 0,
                        greyScaleLevel, cmd, &errCode);
#endif
                 assert( errCode == 0 );
                 /* Now draw the char */
                 PrintChar( aScaler );
                 /* Free up memory */
                 T2K_PurgeMemory( aScaler, 1, &errCode );
                 assert( errCode == 0 );
#if USE_CACHE
                 /* render same char, expect to get from cache: */
                 printf("\n***Here comes the %c ****\n", (char)charCode);
                 FF_CM_RenderGlyph(theCache,fontCode,
                        &aScaler, charCode,
                        0, 0,
                        greyScaleLevel, cmd, &errCode);
                 assert( errCode == 0 );
                 /* Now draw the char */
                 PrintChar( aScaler );
                 /* Free up memory */
                 T2K_PurgeMemory( aScaler, 1,
                             &errCode );
                 assert( errCode == 0 );
#endif
               }
             }

             FF_FM_Delete(pFMGlobals, &errCode);
             assert (errCode == 0);
           }
           else printf("Unable to initialize the Font Manager, errCode = %d!\n",
errCode);
#if USE_CACHE
           FF_CM_Delete(theCache, &errCode);
           assert (errCode == 0);
         }
         else printf("Unable to initialize the Cache Manager, errCode = %d!\n",
errCode);
#endif
   } /* if success */
   else
     errCode = 1;/* InputStream initialization failed */

   /* clean up */
   if (InputStreamA)
   {
     Delete_InputStream( InputStreamA, &errCode  );
     assert (errCode == 0);
```

```
  }

  if (data)
    CLIENT_FREE(data);

  if (mem)/* Destroy the Memhandler object. */
    tsi_DeleteMemhandler( mem );

  return errCode;
}

/*
 * Enumerate Fonts Callback Function
 */
static int enumFontCB(uint16 listIndex, uint8 *faceName8, uint16 *faceName16)
{
  int retCode = 0;
  printf("Logical font index %d ", (int)listIndex);
  if (faceName8)
    printf("%s\n", faceName8);
  else if (faceName16)
  {
    char s[64];
    int ii = 0;
    while (faceName16[ii])
      s[ii] = (char)(faceName16[ii++]); /* dumb: truncate 16 bit chars to 8 bit
chars */
    s[ii] = 0;
    printf("%s\n", s);
  }
  else
    printf("\n");
  return retCode;
}

/*
 * Print/Display Character Function
 */
static void PrintChar( T2K *scaler )
{
int y, x, k, w, h;
char c;
  w = scaler->width;
  assert( w <= scaler->rowBytes * 8 );
  h = scaler->height;

  /* printf("w = %d, h = %d\n", w, h ); */
  k = 0;
  for ( y = 0; y < h; y++ )
  {
    for ( x = 0; x < w; x++ )
    {
      if (scaler->rowBytes == w)
```

```
        {   /* greyscale, byte walk, divide values by 12, map to digits and clamp
> 9 to '@' */
          c =
            (char)((scaler->baseAddr[ k + x ] ) ?
                scaler->baseAddr[ k + x ]/12 + '0' : '.');
          if (c > '9')
            c = '@';
        }
        else/* BLACK_AND_WHITE, fancy bit walk, off = '.' and on = '@' */
          c =
            (char)((scaler->baseAddr[ k + (x>>3) ] & (0x80 >> (x&7)) ) ?
                '@' : '.');
        printf("%c", c );
      }
      printf("\n");
      k += scaler->rowBytes;
    }
    if (scaler->embeddedBitmapWasUsed)
      printf("Bitmap was embedded BITMAP!\n");
    else
      printf("Bitmap generated from OUTLINE.\n");
}

/* utility functions */

/*
 * Print 16.16 Function
 */
static void print16Dot16(F16Dot16 aCode)
{
int hiWord, loWord;
  hiWord = (aCode & 0xffff0000) >> 16;
  loWord = aCode & 0x0000ffff;
  printf("0x%x.", hiWord);
  printf("%4x\n", loWord);
}

/*
 * Print Font Properties Function
 */
static void DoPrintFontProperties(T2K *aScaler)
{
char s[64];
int ii;
  printf("Font Properties:\n");
  if (aScaler->nameString8)
    printf("Font name: %s\n", aScaler->nameString8);
  if (aScaler->nameString16)
    {
    for (ii = 0; aScaler->nameString16[ii]; ii++)
      s[ii] = (char)aScaler->nameString16[ii];
    s[ii] = 0;
    printf("Font name: %s\n", s);
    }
```

```
printf("# logical fonts inside: %d\n", (int)aScaler->numberOfLogicalFonts);
if (aScaler->horizontalFontMetricsAreValid)
{
  /*** Begin font wide HORIZONTAL Metrics data */
  printf("xAscender = ");
  print16Dot16(aScaler->xAscender);
  printf("yAscender = ");
  print16Dot16(aScaler->yAscender);

  printf("xDescender = ");
  print16Dot16(aScaler->xDescender);
  printf("yDescender = ");
  print16Dot16(aScaler->yDescender);

  printf("xLineGap = ");
  print16Dot16(aScaler->xLineGap);
  printf("yLineGap = ");
  print16Dot16(aScaler->yLineGap);

  printf("xMaxLinearAdvanceWidth = ");
  print16Dot16(aScaler->xMaxLinearAdvanceWidth);
  printf("yMaxLinearAdvanceWidth = ");
  print16Dot16(aScaler->yMaxLinearAdvanceWidth);

  printf("caretDx = ");
  print16Dot16(aScaler->caretDx);
  printf("caretDy = ");
  print16Dot16(aScaler->caretDy);

  printf("xUnderlinePosition = ");
  print16Dot16(aScaler->xUnderlinePosition);
  printf("yUnderlinePosition = ");
  print16Dot16(aScaler->yUnderlinePosition);

  printf("xUnderlineThickness = ");
  print16Dot16(aScaler->xUnderlineThickness);
  printf("yUnderlineThickness = ");
  print16Dot16(aScaler->yUnderlineThickness);
  /*** End font wide HORIZONTAL Metrics data */
}

if (aScaler->verticalFontMetricsAreValid)
{
  /*** Begin font wide VERTICAL Metrics data */
  printf("vert_xAscender = ");
  print16Dot16(aScaler->vert_xAscender);
  printf("vert_yAscender = ");
  print16Dot16(aScaler->vert_yAscender);

  printf("vert_xDescender = ");
  print16Dot16(aScaler->vert_xDescender);
  printf("vert_yDescender = ");
  print16Dot16(aScaler->vert_yDescender);
```

```
        printf("vert_xLineGap = ");
        print16Dot16(aScaler->vert_xLineGap);
        printf("vert_yLineGap = ");
        print16Dot16(aScaler->vert_yLineGap);

        printf("vert_xMaxLinearAdvanceWidth = ");
        print16Dot16(aScaler->vert_xMaxLinearAdvanceWidth);
        printf("vert_yMaxLinearAdvanceWidth = ");
        print16Dot16(aScaler->vert_yMaxLinearAdvanceWidth);

        printf("vert_caretDx = ");
        print16Dot16(aScaler->vert_caretDx);
        printf("vert_caretDy = ");
        print16Dot16(aScaler->vert_caretDy);

      }
    }
```

Cache Manager API

5

- Getting Started with the Cache Manager
- Functions for Creating and Deleting the Cache Manager
- Functions for Working with the Cache Manager
- Sample Code

Getting Started with the Cache Manager

This section discusses the following topics:

- Overview of the Cache Manager
- If you want to write your own Cache Manager
- Why do Font Fusion, the Font Manager, and the Cache Manager have a RenderGlyph()function?

Overview of the Cache Manager

The Cache Manager allows your application to make glyphs (bitmap character images) and put them into a cache. It is very simple and straightforward to use. The functions for creating and deleting the Cache Manager allow you to turn on the cache, allocate memory for it, use it, and delete it when you are finished with it.

The functions for working with the Cache Manager allow you to render glyphs into it, empty it, and resize it. In addition, you can set filter parameters if you want to apply special effects to characters, for example, drop shadows, cross-hatch fills, and others. See “Using Filters” on page 89 for details.

Enable specific compile-time options to enhance the capabilities of the cache, as described below.

- Turn on `ENABLE_CACHE_RESIZE` compile-time option to enable dynamic resizing of the cache, then use the `FF_CM_Class *FF_CM_SetCacheSize()` function to set the size.
- Turn on the `ENABLE_CACHE_COMPRESSION` compile-time option to compress and decompress the cache. By default, this is done using run-length encoding, but you can specify different compression and decompression algorithms to use by registering them with the `FF_CM_SetCompDecomp()` function.
- Turn on additional filter functions using the `ENABLE_UNDERLINEFILTER` and `ENABLE_MULTIPLE_FILTERS` compile-time options.
- Turn on the `ENABLE_COMMON_DEFGLYPH` compile-time option to optimize the size of the cache by storing the default missing glyph once instead of for every character that needs it.

If You Want to Write Your Own Cache Manager

If you want to write your own cache mechanism, please contact Bitstream for technical support: 617-497-6222. Bitstream has made it very easy for you to integrate cache mechanisms to Font Fusion.

Why Do Font Fusion, the Font Manager, and the Cache Manager Have a RenderGlyph() Function?

We designed these functions so they could be independent of each other and work together.

The real **`RenderGlyph`** work is always done in the Font Fusion Core. If you are using the Cache Manager, **`FF_CM_RenderGlyph()`** first checks the cache for the glyph or calls another module to render the glyph and store it in the cache. It uses either the Font Manager **`RenderGlyph`** function or it calls the Font Fusion Core.

The Font Manager **`RenderGlyph`** function, **`FF_FM_RenderGlyph()`**, looks for the requested glyph from among the font fragments of the font, and then calls the **`T2K_RenderGlyph()`** function.

NOTE: Use the **`FF_CM_GlyphInCache()`** function if you only want to check the cache for the glyph.

Functions for Creating and Deleting the Cache Manager

- FF_CM_New ( )
- FF_CM_Delete ( )

FF_CM_Class *FF_CM_New(

```
long sizeofCache,
int *errCode,
tsi_ClientAllocMethod allocPtr,
tsi_ClientDeAllocMethod freePtr,
tsi_ClientReAllocMethod reallocPtr,
void * clientArgs)
```

Arguments

`long` is the size of the cache, including `FF_CM_Class`.

`errCode` is a pointer to the returned error code.

`allocPtr` is a function pointer of (`tsi_ClientAllocMethod`) type which is the allocation method used by the client.

`freePtr` is a function pointer of (`tsi_ClientDeAllocMethod`) type which is the de-allocation method used by the client.

`reallocPtr` is a function pointer of (`tsi_ClientReAllocMethod`) type which is the re-allocation method used by the client.

`clientArgs` is of (`void *`) type which contains the arguments used by the client.

Description

`FF_CM_New()` is a pointer to a function that creates a new instance of the Font Fusion Cache Manager. If the `ENABLE_CLIENT_ALLOC` macro is ON, **`FF_CM_New()`**is a pointer to a function that allows a third party to create a new instance of the Font Fusion Cache Manager. In this case all the allocation/re-allocation/de-allocation of memory is handled by the client itself with the help of the arguments explained above.

It returns a context pointer, which is `NULL` on failure.

To learn more about the code type for the typedefs, See "tsiMemObject *tsi_NewMemhandler(" on page 51

NOTE: Please note that the **`allocPtr`**, **`freePtr`**, **`reallocPtr`**, and **`clientArgs`** parameters are only applicable when the macro **`ENABLE_CLIENT_ALLOC`** is ON.

void FF_CM_Delete(

```
FF_CM_Class *theCache,
int *errCode)
```

Arguments

`theCache` is a pointer to the cache class returned from **`FF_CM_New()`**.

`errCode` is a pointer to the returned error code.

Description

`FF_CM_Delete()` destroys the cache manager context and frees the memory used to hold the cache.

Functions for Working with the Cache Manager

Use the following functions to work with the cache manager. These functions are described below.

- FF_CM_RenderGlyph ( )
- FF_CM_GlyphInCache( )
- FF_CM_Flush ( )
- FF_CM_SetFilter ( )
- FF_CM_Class *FF_CM_SetCacheSize ( )
- FF_CM_SetCompDecomp ( )

void FF_CM_RenderGlyph(

```
FF_CM_Class *theCache,
uint16 font_code,
T2K **theScaler,
long char_code,
int8 xFracPenDelta,
int8 yFracPenDelta,
uint8 greyScaleLevel,
uint16 cmd,
int *errCode)
```

Arguments

`theCache` is a pointer to the cache class returned from **`FF_CM_New()`**.

`font_code` is an integer code that identifies the font or a font instance you previously created.

`theScaler` is a pointer to the current `T2K` scaler object in which the font is current.

`char_code` is the character code to render.

`xFracPenDelta` and `yFracPenDelta` are normally set to zero. You can use them with non-zero values (0 to 63) if you are also using fractional (subpixel) character positioning.

`greyScaleLevel` describes the level of anti-aliasing you want to apply. See "Bits for the greyScaleLevel field" on page 117 for more information.

`cmd` describes to Font Fusion what to do with various `bitflags`. See "Bits for the cmd field" on page 118 for more information.

`errCode` is a pointer to the returned error code.

Description

`FF_CM_RenderGlyph()` renders a glyph from a font instance active in the `T2K` scaler object.

This function searches the cache based on the input parameters. If it finds the character in the cache, then it updates the appropriate fields in the `T2K` scaler object. If the character is not in the cache, then the Cache Manager begins the sequence to create the character and place it in the cache. Font Fusion then updates the scaler.

Parameters passed into the function include the scaler that is to be updated as well as a pointer to the cache.

The `char_code` and `font_code` refer to the desired character in a particular font. The `FracPenDelta` fields are required by the core. They have valid values from 0 to 63 and refer to subpixel positioning. The `greyScaleLevel` is another field needed by the core. It refers to what output mode the core should operate in. The `cmd` is used to set the desired level of hinting in the core.

int FF_CM_GlyphInCache(

```
FF_CM_Class *theCache,
uint16 font_code,
T2K **theScaler,
long char_code,
int8 xFracPenDelta,
int8 yFracPenDelta,
uint8 greyScaleLevel,
uint16 cmd,
int *errCode)
```

Arguments

`theCache` is a pointer to the cache class returned from **`FF_CM_New()`**.

`font_code` is an integer code that identifies the font or a font instance you previously created.

`theScaler` is a pointer to the current `T2K` scaler object in which the font is current.

`char_code` is the character code to render.

`xFracPenDelta` and `yFracPenDelta` are normally set to zero. You can use them with non-zero values if you are also using fractional (subpixel) character positioning.

`greyScaleLevel` describes the level of anti-aliasing you want to apply. See "Bits for the greyScaleLevel field" on page 117 for more information.

`cmd` describes to Font Fusion what to do with various `bitflags`. See "Bits for the cmd field" on page 118 for more information.

`errCode` is a pointer to the returned error code.

Description

The Cache Manager Query Glyph function searches the cache based on the input parameters. If the character is found in the cache, the function returns true. If the character is not in the cache, it returns false.

If the glyph is found in the cache, the scaler structure is populated with the bitmap information so that they glyph can be rendered immediately, eliminating the need to subsequently call FF_CM_RenderFlyph(). When the Scaler structure is populated in this way, the `T2K_RETURN_OUTLINES` flag is set in the cmd parameter, it will definitely be ignored. Only the bitmap information is delivered giving a performance advantage to the calling unit.

NOTE: This function returns a non-zero value if the character is in the cache. The function returns a zero value if the character is not in the cache.

Possible Error Codes

NONE

void FF_CM_Flush(

```
FF_CM_Class *theCache,
int *errCode)
```

Arguments

`theCache` is a pointer to the cache class returned from **`FF_CM_New()`**.

`errCode` is a pointer to the returned error code.

Description

`FF_CM_Flush()` re-initializes the cache.

void FF_CM_SetFilter(

```
FF_CM_Class *theCache,
uint16 FilterTag,
FF_T2K_FilterFuncPtr BitmapFilter,
void *filterParamsPtr)
```

Arguments

`theCache` is a pointer to the cache class returned from **`FF_CM_New()`**.

`FilterTag` is a numeric tag for identifying filtering that **`theFilterFunc()`** does. For example 0 might indicate a drop shadow filter, 1 a cross-hatch fill, and so on.

`BitmapFilter` is a function pointer to the actual filter function.

`filterParamsPtr` is a pointer to an optional parameter block for the filter function.

Description

`FF_CM_SetFilter()` sets parameters related to filtering. These parameters are stored and applied to new characters that Font Fusion creates. The `FilterTag` is a component of the search criteria for finding characters in the cache. It distinguishes one glyph from another, for example, one with no filter versus one with a drop shadow applied. The filter function is described in “Using Filters” on page 89.

`FF_CM_Class *FF_CM_SetCacheSize(`

```
FF_CM_Class *theOldCache,
int32 newCacheSize,
int *errCode)
```

Arguments

`theOldCache` is a pointer to the old cache.

`newCacheSize` is the new cache size.

`errCode` is a pointer to the returned error code.

Description

Use this function to set the size of the cache.

`void FF_CM_SetCompDecomp(`

```
FF_CM_Class *theCache,
FF_CM_CompressionPtr compression ,
FF_CM_DecompressionPtr decompression)
```

Arguments

`theCache` is a pointer to the cache class returned from **`FF_CM_New()`**.

`compression` is a pointer to register the compression method to use.

`decompression` is a pointer to register the decompression method to use.

Description

If you enable cache compression the system automatically uses the built in run-length encoding (rle) compression/decompression algorithm. This requires no additional API calls. If you want a different form of compression you can code any form of compression or decompression necessary and use the **FF_CM_SetCompDecomp()** routine to register these methods for use by the system.

Sample Code

The following is a simple example program using the Cache Manager and Font Fusion Core.

```
/* To see Cache Manager instrumentation, add this line to CONFIG.H or cachemgr.h:
*/
/* #define CM_DEBUG1*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "T2K.H"
#include "cachemgr.h"

/*
 *  Constants
 */
#define CACHE_SIZE20000

/*
 *  Prototypes
 */
int main(void);
static void PrintChar( T2K *scaler );


/*
 *  main: where it all happens, mainly!
 */
int main(void)
{
  /* locals needed for the Cache Manager */
  FF_CM_Class *theCache;
  int    errCode;
  uint16 font_code = 0; /* only one font, always fontCode 0 */
  uint8  filterTag = 0;  /* only one filter: none */
  /* locals needed for general T2K Core usage */
  FILE   *fpID = NULL;
  unsigned long length, count;
  unsigned char *data;
  T2K_TRANS_MATRIX trans;
  unsigned short charCode;
  char   *string = "AABBCCabcdefghijklmnopqrsabcdefghijklmnopqrsabcdef-
ghijklmnopqrs";
  int      i;
  tsiMemObject*mem = NULL;
  InputStream *in = NULL;
  sfntClass *font = NULL;
  T2K    *scaler = NULL;
  char   *fName = "TT0003M_.TTF";
  short  fontType = FONT_TYPE_TT_OR_T2K;
  int      fontSize = 24;
  uint8  cmd = T2K_RETURN_OUTLINES | T2K_NAT_GRID_FIT  | T2K_SCAN_CONVERT;
  uint8  greyScaleLevel = BLACK_AND_WHITE_BITMAP;

  printf("\n\n\n");
  printf("**********    ******     **       **\n");
  printf("    **      **     **    **     **\n");
  printf("    **              **   **   **\n");
  printf("    **              **   ** **\n");
  printf("    **             **    ****\n");
  printf("    **            **     ** **\n");
  printf("    **          **       **   **\n");
  printf("    **        **         **     **\n");
  printf("    **      **********   **       **\n\n\n");
  printf ("Hello World, this is a simple Font Fusion Example,\n\n");
  printf ("showing use of the Cache Manager with T2K Core,\n\n");
  printf ("with just printf statements for output,\n\n");
  printf ("from www.bitstream.com !\n\n");

  // Create a new Cache Manager to play around with.
  theCache = FF_CM_New(CACHE_SIZE, &errCode);
  assert( errCode == 0 );

     /* configure Cache filterTag for all the characters we will make */
     FF_CM_SetFilter(theCache,
              filterTag,
              NULL,
              NULL);
     /* Create the Memhandler object. */
     mem= tsi_NewMemhandler( &errCode );
     assert( errCode == 0 );

     /* Open the font. */
     fpID= fopen(fName, "rb"); assert( fpID != NULL );
     errCode= fseek( fpID, OL, SEEK_END ); assert( errCode == 0 );
     length= (unsigned long)ftell( fpID ); assert( ferror(fpID) == 0 );
     errCode= fseek( fpID, OL, SEEK_SET ); assert( errCode == 0 ); /* rewind */

     /* Read the font into memory. */
     data= (unsigned char *)malloc( sizeof( char ) * length ); assert( data !=
NULL );
     count= fread( data, sizeof( char ), length, fpID ); assert( ferror(fpID) ==
0 && count == length );
     errCode= fclose( fpID ); assert( errCode == 0 );
     /* in = New_NonRamInputStream( mem, fpID, ReadFileDataFunc, length,
&errCode  ); */


       /* Create the InputStream object, with data already in memory */
       in = New_InputStream3( mem, data, length, &errCode ); /* */
       assert( errCode == 0 );

          /* Create an sfntClass object*/
          font = New_sfntClass( mem, fontType, in, NULL, &errCode );
          assert( errCode == 0 );

            /* Create a T2K font scaler object.  */
            scaler = NewT2K( font->mem, font, &errCode );
            assert( errCode == 0 );
              /* 12 point */
              trans.t00 = ONE16Dot16 * fontSize;
              trans.t01 = 0;
              trans.t10 = 0;
              trans.t11 = ONE16Dot16 * fontSize;
              /* Set the transformation */
              T2K_NewTransformation( scaler, true, 72, 72, &trans, true,
&errCode );
              assert( errCode == 0 );

              for ( i = 0; (charCode = string[i]) != 0; i++ ) {
                /* Create a character */
                printf("\n\n***Here comes the %c ****\n\n", (char)charCode);
                FF_CM_RenderGlyph(theCache,font_code,
                        &scaler, charCode,
                        0, 0,
                        greyScaleLevel, cmd, &errCode);
                assert( errCode == 0 );
                /* Now draw the char */
                PrintChar( scaler );
                /* Free up memory */
                T2K_PurgeMemory( scaler, 1, &errCode );
                assert( errCode == 0 );
              }


            /* Destroy the T2K font scaler object. */
            DeleteT2K( scaler, &errCode );
            assert( errCode == 0 );


          /* Destroy the sfntClass object. */
          Delete_sfntClass( font, &errCode );

        /* Destroy the InputStream object. */
        Delete_InputStream( in, &errCode  );

      free( data );
      /* Destroy the Memhandler object. */
      tsi_DeleteMemhandler( mem );

    FF_CM_Delete(theCache, &errCode);

    return 0;
}

/*
 * Print/Display Character Function
 */
static void PrintChar( T2K *scaler )
{
int y, x, k, w, h;
char c;
  w = scaler->width;
  assert( w <= scaler->rowBytes * 8 );
  h = scaler->height;

  /* printf("w = %d, h = %d\n", w, h ); */
  k = 0;
  for ( y = 0; y < h; y++ )
  {
    for ( x = 0; x < w; x++ )
    {
      if (scaler->rowBytes == w)
      {    /* greyscale, byte walk, divide values by 12, map to digits and clamp
> '9' to '@' */
        c =
          (char)((scaler->baseAddr[ k + x ] ) ?
              scaler->baseAddr[ k + x ]/12 + '0' : '.');
        if (c > '9')
          c = '@';
      }
      else/* BLACK_AND_WHITE, fancy bit walk, off = '.' and on = '@' */
        c =
          (char)((scaler->baseAddr[ k + (x>>3) ] & (0x80 >> (x&7)) ) ?
              '@' : '.');
      printf("%c", c );
    }
    printf("\n");
    k += scaler->rowBytes;
  }
}
```

6 Font Fusion API for Printer Developers

Topics

General Information

If you are developing a Hewlett-Packard printer or printer emulation, you need to enable compile-time options and write callback functions described in this appendix.

Compile-Time Options

Use the following compile-time options if you need to process scalable Intellifont outlines.

Options: Printer Fonts	Description
`ENABLE_PCL`	Enable this option to process scalable Intellifont fonts that have been downloaded to a Hewlett-Packard printer or printer emulation as encapsulated outlines. We have supplied a font reader for this format with two new source modules, `pclread.c` and `pclread.h`. When using `ENABLE_PCL`, you need to write a callback function, `eo_get_char_data()`, documented in this appendix.
`ENABLE_PCLETTO`	Enable this option to process TrueType fonts that have been downloaded to a Hewlett-Packard printer or printer emulation as encapsulated outlines. No additional font reader module is required. When using `ENABLE_PCLETTO`, you need to write a callback function, `tt_get_char_data()`, documented in this appendix.

Font Types

There are two additional font types that you can specify if you are a printer developer using the Font Fusion API. These are listed in the table below:

`fontType`	**Description**
`FONT_TYPE_PCL`	Use with PCL encapsulated outlines (PCLeo) on HP printer emulations
`FONT_TYPE_PCLETTO`	Use with TrueType encapsulated outlines (PCLetto) on HP printer emulations

Callback Functions

- eo_get_char_data()
- tt_get_char_data()

int eo_get_char_data(

```
long cCode,
uint8 cmd,
uint8 **pCharData,
uint16 *dataSize,
uint16 *charCode,
int16 *gIndex)
```

Arguments

`cCode` is the character code or glyph index you are requesting.

`cmd` is equal to zero (`==0`) if `cCode` is a character code; it is equal to `T2K_CODE_IS_GINDEX` if `cCode` is a glyph index value.

`pCharData` returns a pointer to the glyph data for the character.

`dataSize` returns the size of the glyph data.

`charCode` returns the character code you are requesting.

`gIndex` returns the glyph index of the character code, or it is the same as `cCode` if `cmd==T2K_CODE_IS_GINDEX`.

Description

Write the callback function **`eo_get_char_data()`** to get a pointer to an outline character string from your application for a scalable Intellifont font.

You also need to define the compile-time option `ENABLE_PCL`.

int tt_get_char_data(

```
long cCode,
uint8 cmd,
uint8 **pCharData,
uint16 *dataSize,
int16 *gIndex,
HPXL_MetricsInfo_t *metricsInfo)
```

Arguments

`cCode` is the character code or glyph index you are requesting.

`cmd` is equal to zero (==`0`) if `cCode` is a character code; it is equal to `T2K_CODE_IS_GINDEX` if `cCode` is a glyph index value.

`pCharData` returns a pointer to the glyph data for the character.

`dataSize` returns the size of the glyph data.

`gIndex` returns the glyph index of the character code, or it is the same as `cCode` if `cmd==T2K_CODE_IS_GINDEX`.

`metricsInfo` provides the application-level set width and left or top side bearing information, described below.

Description

Write the callback function **`tt_get_char_data()`** to get a pointer to an outline character string from your application for a scalable TrueType font.

You also need to define the compile-time option `ENABLE_PCLETTO`.

metricsInfo

The HPXL_MetricsInfo_t structure allows your application to return metrics information about the requested character if it has received that information in the downloaded material. This is consistent with the Hewlett-Packard XL implementation for downloads of Asian font. Here is a specification of that structure:

```
#ifdef ENABLE_PCLETTO
typedef struct
{
   uint8 lsbSet;
   uint16 lsb;
   uint8 awSet;
   uint16 aw;
   uint8 tsbSet;
   uint16 tsb;
}HPXL_MetricsInfo_t;
#endif
```

Currently this structure is only supported for PCLeTTo fonts, consistent with the XL specification, which only applies to downloaded TrueType (PCLeTTo) fonts.

The callback function **tt_get_char_data()** must flag which values are valid in the structure it returns by setting each of the Set fields to true, in order for Font Fusion to respect the corresponding value. Therefore, if the application has a left side-bearing for the requested character, it sets the lsbSet argument to true and sets the lsb member to the side-bearing value.

All units for these arguments are in design units.

Format 16 Font Header Support

Format 16 Font Header is an extended PCL soft font header structure which increases the data segment size fields to long (from short) and includes optional data segments for supporting gallery characters (*GC*), vertical translation (*VT*), vertical rotation (*VR*) and vertical exclusion (*VE*).

Font Header Format 16 (Universal Font Header) is identical in structure to Format 15 (Universal Scalable Font Header), with the size field for data segments increased from 16 bits to 32 bits, and a new fontType (fontType 3) added to signify “large” fonts. A “large” font is a bound font with character codes that are not limited to 8-bit values. Some font data segments in “large” fonts can exceed 65535 bytes.

The compile conditional **`ENABLE_PCLETTO`** should be turned ON to enable the Format 16 Font Header support in Font Fusion.

When Font Fusion encounters a PCLetto font with the fontType set to “3”, all segmented data blocks which follow the header structure are assumed to have 4-byte size fields as opposed to the default two-bytes for non-“large” fonts.

The support of the Format 16 Font Header primarily involves support for the following segmented data blocks:

- Galley Character (*GC*): Indicates the character code for the missing glyph based on ranges of character codes. If a missing character falls into one of the ranges in this table the associated missing glyph character is used instead of the default (glyph index 0).
- Vertical Exclusion (*VE*): If vertical writing is specified (`T2K_VERTICAL` and/ or `T2K_VERT_SUB`) and the requested character code is found in this table the character will not have any special vertical processing done to it.
- Vertical Translation (*VT*): Identifies the vertical substitute glyphs. If vertical substitution is specified (`T2K_VERT_SUB`) and the requested glyph index exists in this table the associated “Vertical” glyph index is rendered instead.
- Vertical Rotation (*VR*): If vertical writing is specified (`T2K_VERTICAL`) and the requested glyph has not been substituted it will be rotated counter-clockwise 90° and positioned according to the users specification.

FF_SetVertPos(

```
sfntClass*,
uint8 value)
```

Arguments

`sfntClass` refers to the `sfntClass` (font) object.

`value` defines the vertical positioning of the font.

Description

Use the `FF_SetVertPos()` API to set the `sfntClass->vert_pos` element as follows:

Value	Description
0	OFF
1	Positions the character middle right against the origin
2	Positions the character top right against the origin.

** All other values are ignored.

Font Information Table

The Font Information Table is the glue that binds together the files comprising a particular resident font set.

FIT File Structure

For each font in the resident font set, the Font Information Table contains these fields, as shown in Figure 14-1:

- the font alias name
- an 88-byte field describing font attributes (for PCL emulations)
- the page description language being emulated (PCL, PostScript, and/or TrueType)
- the address in memory where you will store the font
- a next-search encoding value that, when decoded, gives you the next index into the Font Information Table where you can find a support typeface for dropout (missing) characters

Record 1 (First font in RFS)	**Font alias name field**	**PCL font attribute**	**PCL emulation field**	**Resident font address field**	**Next-search encoding value field**	**Required resolution field**
. . .	. . .	. . .	. . .	. . .	. . .	. . .
Record 2 (First font in RFS)	**Font alias name field**	**PCL font attribute**	**PCL emulation field**	**Resident font address field**	**Next-search encoding value field**	**Required resolution field**

Font Alias Name

Typically, page description languages request fonts in one of two ways. One is by name; the second is by attribute.

The FIT lets you translate a font name that an application requests into the name of the identical font in your Bitstream Resident Font Set. It also lets you match the attributes of a requested font with an equivalent font in the Resident Font Set.

The font alias name depends on the page description language you support. If you support PostScript, the font alias name for Times is Bitstream Dutch 801 SWA (a Bitstream Speedo typeface set width adjusted to Adobe widths); for Helvetica, it is Bitstream Swiss 721 SWA.

If you support Windows 3.1 GDI (TrueType), the font alias name for Times New Roman is Bitstream Dutch 801 SWM (a Bitstream Speedo typeface set width adjusted to Monotype widths); for Arial, it is Bitstream Swiss 721 SWM.

If, on the other hand, you support PCL, the FIT also includes an 88-byte field describing Hewlett-Packard PCL-compatible font attributes.

Font Name Requested	Bitstream Equivalent
Helvetica Bold	Swiss 721 Bold SWA (SWA=set-width adjusted to Adobe widths)
Arial Bold	Swiss 721 Bold SWM (SWM=set-width adjusted to Monotype widths)
Univers Bold	Swiss 721 Bold SWC (SWC=set-width adjusted to Agfa, formerly known as Compugraphic, widths)

PCL Font Attributes

This 88-byte field lets you match the attributes of a requested PCLeo font with an equivalent font in the resident font set. The attributes are described below.

Bytes in the PCL Font Attributes Field

All values in the PCL font attributes field are described below. They reflect the values as they pertain to Bitstream fonts in the HP LaserJet IIIsi-compatible resident font set.

Value	Font Type
0	7-bit (96 characters: characters 32-127 allowed)
1	8-bit (192 characters: characters 32-127 and 160-255 allowed)
2	PC-8 (256 characters: all characters can be printed; but characters 0, 7-15, and 27 must be printed in transparency mode)
10	Scalable

The table below shows the 88-byte PCL font attributes field of FIT.

Byte	MSB (Most Significant Byte)	LSB (Least Significant Byte)
0	Font Descriptor Size (88)	
2	Descriptor Format	Font Type
4	Style MSB	
6	Baseline Distance	
8	Cell Width	
10	Cell Height	
12	Orientation	Spacing
14	Symbol Set	
16	Pitch (Default HMI)	
18	Height	
20	xHeight	
22	Width Type	Style LSB
24	Stroke Weight	Typeface LSB
26	Typeface MSB	Serif Style
28	Quality	Placement
30	Underline Distance	Underline Height

Byte	MSB (Most Significant Byte)	LSB (Least Significant Byte)
32	Reserved1	
34	Reserved2	
36	Reserved3	
38	Number of Outlines	
40	Pitch Extended	Height Extended
42	Cap Height	
44-47	Font Number	
48-63	Font Name	
64	Scale Factor	
66	Master X Resolution	
68	Master Y Resolution	
70	Master Underline Position	
72	Master Underline Height	
74	LRE Threshold	
76	Italic Angle	
78-81	Character Complement MSW	
82-85	Character Complement LSW	
86	Data Size	

Font Descriptor Size

2-byte unsigned integer. This value is always 88, although your printer or application ignores it. The value is most useful in identifying the type of header (that is, a PCLeo "unbound" header) that follows.

Descriptor Format

Unsigned byte. This value is always twelve.

Font Type

Unsigned byte. This value is always 10 for fonts in the resident font set. Note that this value is appropriate for the font's character set.

Style MSB

2-byte unsigned integer. Combine the Style MSB (most significant byte, byte 4) with the Style LSB (least significant byte, byte 23) to come up with the "Style." Calculate the style using this formula:

```
Style = Posture + (4 * Width) + (32 * Structure)
```

This is the binary structure of the "Style":

X	Reserved	Structure	Width	Posture

where the values for posture, width, and structure are determined as follows:

Value	Bits 0-1, Posture
0	Upright
1	Italic
2	Alternate Italic
3	Reserved

Value	Bits 2-4, Width (multiply the value by 4)
0	Normal
1	Condensed
2	Compressed or Extra Condensed
3	Extra Compressed
4	Ultra Compressed

Value	Bits 2-4, Width (multiply the value by 4)
5	Reserved
6	Extended or Expanded
7	Extra Extended or Extra Expanded

Value	Bits 5-9, Structure (multiply the value by 32)
0	Solid
1	Outline
2	Inline
3	Contour
4	Solid with Shadow
5	Outline with Shadow
6	Inline with Shadow
7	Contour with Shadow
8-11	Patterned
12-15	Patterned with Shadow
16	Reverse
17	Inverse in Open Border
18-31	Reserved

Baseline Distance

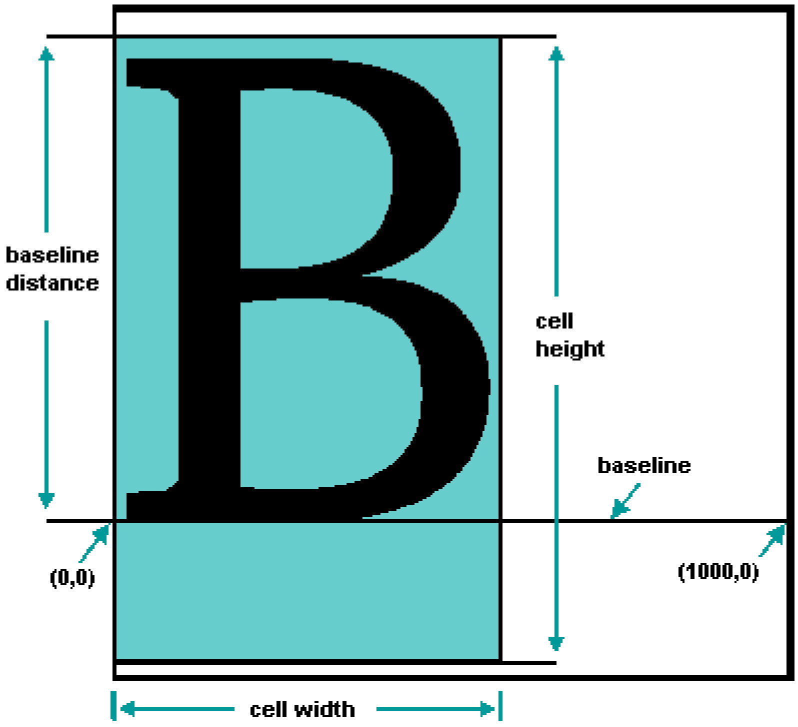
2-byte unsigned integer. The distance from the top of the cell to the baseline in ORUs, or outline resolution units (1/1000 of an em). See Figure 14-4. Your PCL emulator ignores this field.

Cell Width

2-byte unsigned integer. The width of the font wide bounding box, in ORUs. (The fontwide bounding box is the minimum rectangle that can enclose the largest character or compound character, such as a fraction, in the font). Note that the cell width can be larger than the em square; the cell width can range from 1 to 65535. Your PCL emulator ignores this field.

Cell Height

2-byte unsigned integer. The height of the fontwide bounding box, in ORUs. Note that the cell height can be larger than the em square; the cell height can range from 1 to 65535. Your PCL emulator ignores this field.



Capital "B" within the em square of a Bitstream font

Orientation

Unsigned byte. Always zero. Your PCL emulator ignores this field.

Spacing

Boolean. Zero specifies fixed spacing; one specifies proportional spacing.

Symbol Set

2-byte unsigned integer. Always zero.

NOTE: The Character Complement values (bytes 78-85) indicate symbol set compatibility.

Pitch

2-byte unsigned integer. The default pitch used for monospaced and proportional fonts, in ORUs. (The default HMI, or Horizontal Motion Index, is equal to this pitch value. The default HMI is the set width of a space character for the font and is used if you cannot generate a character image from the current font.)

The default set width for monospaced fonts is determined by the pitch value.(The set width is the width of any monospaced character plus its side bearings.)

Height

2-byte unsigned integer. This value is always 96.

xHeight

2-byte unsigned integer. The height from the baseline to the top of the lowercase "x," in ORUs.

Width Type

Signed byte. The relative width of the font, as follows. Note that your PCL emulator ignores this field.

Value	Width Type
-5	Ultra Compressed
-4	Extra Compressed
-3	Compressed or Extra Condensed
-2	Condensed
0	Normal
2	Expanded
3	Extra Expanded

NOTE: Use bits 2-4 of the Style MSB value (byte 4) to determine the width type for the current font.

Style LSB

Unsigned byte. The least significant byte of the "Style." See bits 0-1 (Posture) of the Style MSB (byte 4) for details.

Stroke Weight

Signed byte. The relative weight (thickness) of the strokes in a range from -7 to 7, where these values have the following stroke weights:

Value	Stroke Weight
-7	Ultra Thin
-6	Extra Thin
-5	Thin
-4	Extra Light
-3	Light
-2	Demi Light
-1	Semi Light
0	Medium, Book, or Text
1	Semi Bold
2	Demi Bold
3	Bold
4	Extra Bold
5	Black
5	Extra Black
7	Ultra Black

Typeface LSB

Unsigned byte. Indicates Typeface Family, as explained below.

Typeface MSB

Unsigned byte. The value for the Typeface Family field (bits 0 through 8) is represented in the Typeface LSB. The value takes into account the typeface only, not the style (e.g., roman, italic, bold, bold italic). For example, a value of 52 indicates Univers (equivalent to Bitstream Swiss 721).

The value for the Version field (bits 9 and 10) is always 0.

The value for the Vendor field (bits 11 through 14) is either 4 (for a Bitstream font that has the word "SWC" in the font name) or 2 (for a Bitstream font that does not have the word "SWC" in the font name). The font name value is at bytes 48-63.

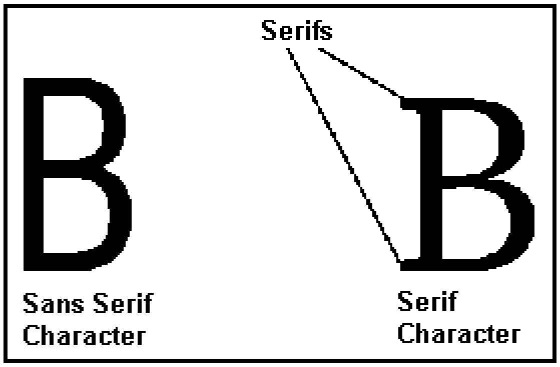
Combined, the total Version and Vendor value is either 32 (for a Bitstream font that has the word "SWC" in the font name) or 16 (for a Bitstream font that does not have the word "SWC" in the font name).

This is the binary structure of the "Typeface," where the most significant bit (bit 15) of the MSB is zero:

```
15     14           10      8             0
|0     |Vendor     |Version |Typeface Family |
Typeface MSB       Typeface LSB
```

Serif Style

Unsigned byte. Although HP has designated several serif style definitions, this field is most useful in distinguishing between serif and sanserif styles. The figure below shows an example of sanserif and serif styles.



Sans serif capital "B" and serif capital "B"

This field specifies a serif or sanserif style. The upper two bits (6 and 7) are used to determine the serif style of typeface-insensitive characters to complement the font. These are the serif style values for the lower six bits (0 through 5), followed by the values for the upper two bits:

Value	Serif Style, Lower Six Bits
0	Sanserif, Square
1	Sanserif, Round
2	Serif, Line
3	Serif, Triangle
4	Serif, Swath
5	Serif, Block
6	Serif, Bracket
7	Rounded, Bracket
8	Flair Serif, Modified Sans
9	Script, Nonconnecting
10	Script, Joining
11	Script, Calligraphic
12	Script, Broken Letter

Value	Serif Style, Upper Two Bits
64	Sanserif
128	Serif
192	Reserved

NOTE: Reserve other numbers for future use.

Quality

Unsigned byte. The quality of the font, as follows:

Value	Quality
0	Data processing
1	Near letter quality
2	Letter Quality

Your PCL emulator ignores this field.

Placement

Signed byte. Specifies the position of character patterns relative to the baseline. Your PCL emulator ignores this field.

Underline Distance

Signed byte. This value is always zero for PCLeo fonts. See Master Underline Position, bytes 70 and 71, for more relevant information.

Old Underline Height

Unsigned byte. This value is always zero for PCLeo fonts. See Master Underline Height, bytes 72 and 73, for more relevant information.

Reserved1

2-byte unsigned integer. This value is reserved.

Reserved2

2-byte unsigned integer. This value is reserved.

Reserved3

2-byte unsigned integer. This value is reserved.

Number of Outlines

2-byte unsigned integer. This value is reserved.

Pitch Extended

Unsigned byte. This value is always zero.

Height Extended

Unsigned byte. This value is always zero.

Cap Height

2-byte unsigned integer. The distance from the baseline to the cap line (the top of an unaccented, upper-case letter, such as the "H"), in ORUs.

Font Number

4-byte unsigned integer. The font ID number that Bitstream assigns to the font. Your PCL emulator ignores this field.

Font Name

16-character string. The short font name, which can be up to 16 characters long. It does not include the style (e.g., roman, italic, bold, bold italic).

Scale Factor

2-byte unsigned integer. It indicates the number of Outline Resolution Units (ORUs) of the characters and This field can take variable values. The outline resolution of the OEM PFRs is 256.

Master X Resolution

2-byte unsigned integer. It indicates the pixel resolution, in the x scan direction, at which the font was designed. This field can take variable values. Normally, there are 256 ORUs along the x axis of an OEM PFR.

Master Y Resolution

2-byte unsigned integer. It indicates the pixel resolution, in the y scan direction, at which the font was designed. This field can take variable values. Normally, there are 256 ORUs along the y axis of an OEM PFR.

Master Underline Position

2-byte signed integer. The distance from the baseline to the center of the underline, in ORUs.

Master Underline Height

2-byte unsigned integer. The thickness of the underline, in ORUs.

LRE Threshold

2-byte unsigned integer. This value is always zero.

Italic Angle

2-byte unsigned integer. The tangent of the italic angle multiplied by 2 15 (2 to the 15th power). This value is zero for upright fonts.

Character Complement MSW

4-byte unsigned integer. Character complement, most significant word.

Character Complement LSW

4-byte unsigned integer. Character complement, least significant word.

The Character Complement MSW and LSW are each 32-bit values that indicate the character set that the current font is compatible with.

Each bit identifies a collection of character sets that the font is compatible with; each bit is interpreted independently.

Here are some examples of individually defined bits:

Bit	Value	Compatibility with Character Set Collection
63	0	Font is compatible with Basic Latin collection (for such character sets as ISO 8859/1 Latin 1)
	1	Font is not compatible with this collection
62	0	Font is compatible with East European Lain collection (for such character sets as ISO 8859/2 Latin 2)
	1	Font is not compatible with this collection
61	0	Font is compatible with Turkish collection (for such character sets as ISO 8859/9 Latin 5)
	1	Font is not compatible with this collection
34	0	Font is compatible with Math collection (for such character sets as Math 8)
	1	Font is not compatible with this collection
33	0	Font is compatible with Semi-graphic collection (for such character sets as PC-8 D/N)
	1	Font is not compatible with this collection
32	0	Font is compatible with Dingbats collection (for character sets such as ITC Zapf Dingbats series 100, 200, etc.)
	1	Font is not compatible with this collection

Here are some examples of complete Character Complement MSW and LSW values:

Hex Value	Compatibility
0000000000000000	Default; font is compatible with every character set collection
7FFFFFFFFFFFFFFF	Font is compatible with the Basic Latin character sets only
FFFFFFFEFFFFFFFF	Font is compatible with ITC Zapf Dingbats character sets only

Data Size

2-byte unsigned integer. This value is always zero.

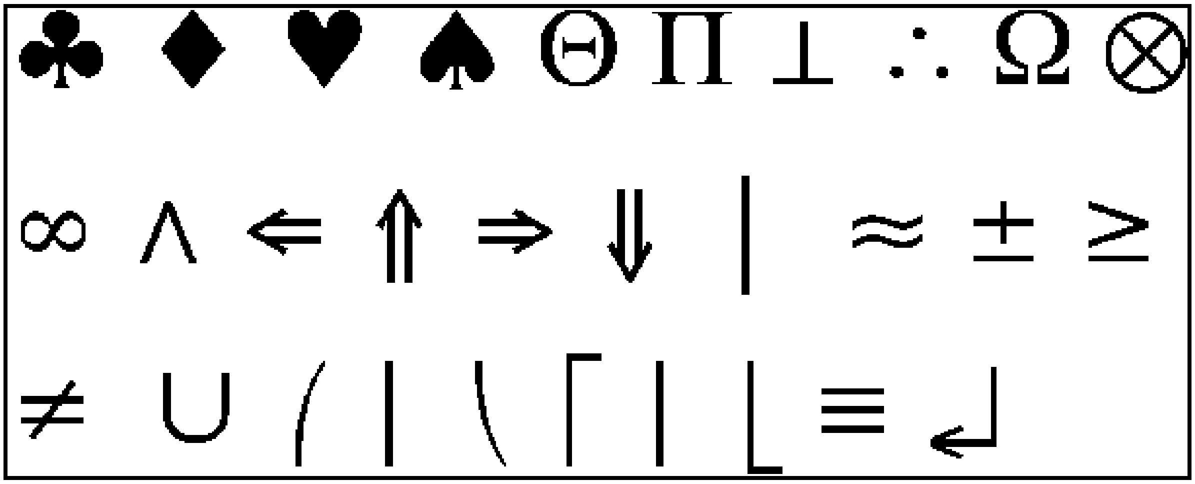
Page Description Language Being Emulated for the Resident Font Set

The value for this field is either `pdlPCL` (for PCL resident font set emulations), `pdlPostScript` (for PostScript resident font set emulations), `pdlGDI` (for TrueType resident font set emulations), or `pdlSupport` (for resident auxiliary and support fonts).

This field helps you include or exclude resident fonts for matches by font name or font attributes. It also helps you distinguish the ORUs (outline resolution units), or design units, being used for the resident font set without having to look up the ORU value in a data structure.

Providing Substitute Support for Missing Characters

Typically, support characters—such as line draw and border pieces, mathematical signs, symbols, etc., that fall into a broad class of special or requested characters—are taken from supplemental fonts. Supplemental characters augment characters in standard character sets.



Examples of support characters

If a font does not contain a support character, the FIT contains a field that allows you to look for the dropout character in another font. This field gives you the next index in the Font Information Table where you can find a support typeface for dropout (missing) characters.

The "next-search" encoding value field in the FIT lets you search for a character index in the next relative or absolute place. A relative jump occurs within the same FIT; for example, from a bold italic to a bold font, or from a serif to a sanserif font. You then search for the same character index in the relative font.

Using FIT at Run-Time

You can burn binary FIT images into ROM or load them from disk at run-time. Your application should maintain a dynamic FIT structure. Or your application should maintain a FIT structure large enough for the maximum possible number of fonts a user can load. This includes font cartridge combinations that a user can insert into your printer, and, depending on your implementation, downloaded font entries. You must add to or delete from the FIT structure during run-time as a user downloads fonts and adds and removes cartridges.

Your application selects fonts by walking through the run-time FIT structure; comparing font selection criteria to the attributes found in the `pclHdr` member; and scoring the criteria by the rules defined in the *PCL5 Developer's Guide,* sections 5-3 and 5-4. When walking through the FIT structure to select a primary font, your application should ignore entries whose `pdlType` member is flagged as `pdlSupport`. Your application may need to distinguish other `pdlTypes`, such as PostScript fonts from the other primary fonts, but you may never need to distinguish the `pdlGDI` fonts from the `pdlPCL` fonts.

Two Different Support-Font Structures

Bitstream has developed a more verbose support-font network to precisely emulate the HP standard. It also makes finding the correct support-font entry point into the FIT logic more comprehensible. "Entry point" in this context refers to the font in the FIT where your code begins pursuing the FIT's logic encoded in the `nextSearchEncode` field.

Current Font-Support Network

LaserJet 4 resident fonts are divided into several logical groups. These are (1) Intellifont Proportional, (2) Intellifont Fixed Pitch 5291, (3) Intellifont Fixed Pitch 4409, (4) Unicode, and (5) PostScript.

The Spacing member of the PCL header, if equal to zero, identifies a fixed-pitch font. Each group except group 5 has its own support-font set. Here is the support-font set for group 1.

font3158	font3159	font3160	font3161
(Serif Medium)	(Serif Bold)	(Sans medium)	(Sans Bold)

font3162

(Universal)

Following the FIT's `nextSearchEncode` member for a primary resident font reveals that it goes to the top layer first, selecting one of these four support fonts, and from there to the bottom layer. The top-level decision involving a downloaded font in this group is simple: does the primary (Intellifont proportional) font have serifs and is it bold? Here is the support-font set for group 2.

font3229	font3230
(Fix 5291 Medium)	(Fix 5291 Bold)

font3227

(Fix 5291 Universal)

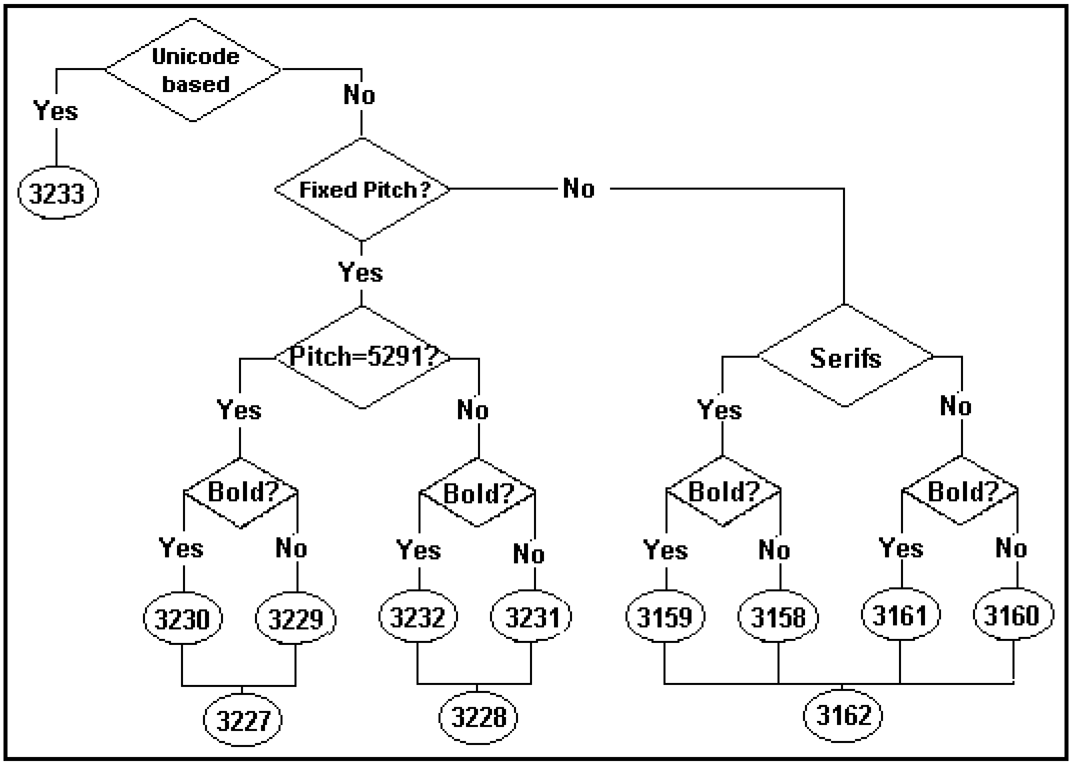
Again, the top-level decision involving a downloaded font in this group is simple: is the downloaded font, with a pitch of 5291, bold or not?

The exact same thing is true for group 3, except that the pitch is 4409.

```
  font3231          font3232

  (Fix 4409 Medium)    (Fix 4409 Bold)

         font3228

         (Fix 4409 Universal)
```

Group 4's support-font set is limited to Universal characters only, so there is only one font, ID = 3233. Your application can make no other decisions once it determines the Unicode basis of the font.

Group 5 has no support fonts; resident fonts in that set are bucketed within font families only. The support system defined above supports the primary resident fonts. If a downloaded font is the current font, and there are missing characters, you should simply follow the same logic. A decision tree for a downloaded font might appear as shown in the figure below. The terminal integers shown in figure are the font IDs of resident support fonts. The first font ID you encounter in the path is the point of entry into the FIT logic. Follow the `nextSearchEncode` path to complete the support-font path.



Decision tree for downloaded font in current support-font network.

Legacy FIT Usage

The legacy support-font system differs in two ways from that used in the current product. First, Bitstream embedded support characters for groups 2, 3, and 4 into certain primary resident fonts in those three groups. Second, the support set for groups 1, 2, and 3 was more compact. This required the path to deviate, as defined in the nextSearchEncode member of the FIT structure, from the sanserif to the serif supplements.

The main issue with the legacy FITs was entering the FIT logic for the Unicode fonts (group 4) and the Fixed Pitch fonts (groups 2 and 3).

For the Unicode fonts, the answer is easy. All the universal characters are in the Times New Roman emulator font (ID = 3234).

For Fixed Pitch = 5291, bold support started at Courier Bold (ID = 3221), roman support at Courier Medium (ID = 3220). When the pitch was 4409, bold support started at Letter Gothic Bold (ID = 3225), roman support at Letter Gothic Regular (ID = 3224). From those entry points, the FIT directs the remainder of the support path.

Text Flows

7

Overview

The text flows below illustrate the typical functions called by an application using Font Fusion. Each function has an `errCode` associated with it. If there is an error, Font Fusion deletes all of its objects. In the case of an error, you need to restart the objects that shared the same `tsiMemObject`.

Font Fusion Core

```
tsi_NewMemhandler(&errCode)
if (errCode) goto ERROR;
    New_InputStream3(..., &errCode)
    if (errCode) goto ERROR;
        New_sfntClass(..., &errCode)
        if (errCode) goto ERROR;
            NewT2k(..., &errCode)
            if (errCode) goto ERROR;
                T2K_NewTransformation(..., &errCode)
                if (errCode) goto ERROR;
                T2K_RenderGlyph(..., &errCode)
                if (errCode) goto ERROR;
                T2K_PurgeMemory( ..., &errCode )
                if (errCode) goto ERROR;
           DeleteT2K(..., &errCode)
            if (errCode) goto ERROR;
        Delete_sfntClass(..., &errCode)
        if (errCode) goto ERROR;
    Delete_InputStream(..., &errCode)
    if (errCode) goto ERROR;
tsi_DeleteMemhandler()

:ERROR
```

Font Manager

```
tsi_NewMemhandler(&errCode);
if (errCode) goto ERROR;
    FF_FM_New(&errCode);
    if (errCode) goto ERROR;
        New_InputStream3(..., &errCode)
        if (errCode) goto ERROR;
            FF_FM_AddTypefaceStream(..., &errCode)
            if (errCode) goto ERROR;
                FF_FM_CreateFont(..., &errCode)
                if (errCode) goto ERROR;
                    FF_FM_SelectFont(..., &errCode)
                    if (errCode) goto ERROR;
                    FF_FM_RenderGlyph(..., &errCode)
                    if (errCode) goto ERROR;
                    T2K_PurgeMemory( ..., &errCode )
                    if (errCode) goto ERROR;
                FF_FM_DeleteFont(..., &errCode)
                if (errCode) goto ERROR;
        FF_FM_Delete(pFMGlobals, &errCode)
        if (errCode) goto ERROR;
    Delete_InputStream(..., &errCode)
    if (errCode) goto ERROR;
tsi_DeleteMemhandler()

:ERROR
```

Cache Manager

```
FF_CM_New(CACHE_SIZE, &errCode)
if (errCode) goto ERROR;
    tsi_NewMemhandler(&errCode)
    if (errCode) goto ERROR;
        New_InputStream3(..., &errCode)
        if (errCode) goto ERROR;
            New_sfntClass(..., &errCode)
            if (errCode) goto ERROR;
                NewT2k(..., &errCode)
                if (errCode) goto ERROR;
                    T2K_NewTransformation(..., &errCode)
                    if (errCode) goto ERROR;
                    FF_CM_RenderGlyph(..., &errCode)
                    if (errCode) goto ERROR;
                    T2K_PurgeMemory( ..., &errCode )
                    if (errCode) goto ERROR;
                DeleteT2K(..., &errCode)
                if (errCode) goto ERROR;
            Delete_sfntClass(..., &errCode)
            if (errCode) goto ERROR;
        Delete_InputStream(..., &errCode)
        if (errCode) goto ERROR;
    tsi_DeleteMemhandler()
FF_CM_Delete(..., &errCode)

:ERROR
```

Font Manager and Cache Manager

```
tsi_NewMemhandler(&errCode);
if (errCode) goto ERROR;
    New_InputStream3(..., &errCode)
    if (errCode) goto ERROR;
        FF_CM_New(CACHE_SIZE, &errCode)
        if (errCode) goto ERROR;
            FF_FM_New(&errCode);
            if (errCode) goto ERROR;
               FF_FM_AddTypefaceStream(..., &errCode)
                if (errCode) goto ERROR;
                    FF_FM_CreateFont(..., &errCode)
                    if (errCode) goto ERROR;
                        FF_FM_SelectFont(..., &errCode)
                        if (errCode) goto ERROR;
                        FF_CM_RenderGlyph(..., &errCode)
                        if (errCode) goto ERROR;
                        T2K_PurgeMemory( ..., &errCode )
                        if (errCode) goto ERROR;
                    FF_FM_DeleteFont(..., &errCode);
                    if (errCode) goto ERROR;
            FF_FM_Delete(pFMGlobals, &errCode)
            if (errCode) goto ERROR;
        FF_CM_Delete(theCache, &errCode)
        if (errCode) goto ERROR;
    Delete_InputStream(..., &errCode)
    if (errCode) goto ERROR;
tsi_DeleteMemhandler()

:ERROR
```

Error Codes

8

- Font Fusion Core
- Font Manager
- Cache Manager

Font Fusion Core Error Codes.

The table below displays the error codes which can be received for the Font Fusion core.

Error Code	Mnemonic
10000	T2K_ERR_MEM_IS_NULL
10001	T2K_ERR_TRANS_IS_NULL
10002	T2K_ERR_RES_IS_NOT_POS
10003	T2K_ERR_BAD_GRAY_CMD
10004	T2K_ERR_BAD_FRAC_PEN
10005	T2K_ERR_GOT_NULL_GLYPH
10006	T2K_ERR_TOO_MANY_POINTS
10007	T2K_ERR_BAD_T2K_STAMP
10008	T2K_ERR_MEM_MALLOC_FAILED
10009	T2K_ERR_BAD_MEM_STAMP
10010	T2K_ERR_MEM_LEAK
10011	T2K_ERR_NULL_MEM
10012	T2K_ERR_MEM_TOO_MANY_PTRS
10013	T2K_ERR_BAD_PTR_COUNT
10014	T2K_ERR_MEM_REALLOC_FAILED
10015	T2K_ERR_MEM_BAD_PTR
10016	T2K_ERR_MEM_INVALID_PTR
10017	T2K_ERR_MEM_BAD_LOGIC
10018	T2K_ERR_INTERNAL_LOGIC
10019	T2K_ERR_USE_PAST_DEATH

Error Code	Mnemonic
10020	T2K_ERR_NEG_MEM_REQUEST
10021	T2K_BAD_CMAP
10022	T2K_UNKNOWN_CFF_VERSION
10023	T2K_MAXPOINTS_TOO_LOW
10024	T2K_EXT_IO_CALLBACK_ERR
10025	T2K_BAD_FONT

Font And Cache Manager Error Codes

The table below displays the error codes which can be received for the Font Manager.

Error Code	Mnemonic
20000	FF_FM_ERR_BAD_INDEX
20001	FF_FM_ERR_BAD_FONTCODE
20002	FF_FM_ERR_CREATE_FONT_OFLO_ERR
20003	FF_FM_ERR_FONT_CODE_ERR
20004	FF_FM_ERR_UNKNOWN_FONT_TYPE

The Cache Manager has no specific error codes.

Font Fusion FAQ

This chapter tries to give you hints and backgrounds for the most frequently asked questions as how to start, what files to look at, and a few problems with rules of thumb to get over them. The chapter also lists some performance tuning tips to ensure optimal performance and efficient memory use when you use Font Fusion.

Performance Tuning Tips

Font Fusion is designed to be fast, small, and customizable. Below listed are some points that describe how to maximize performance and minimize the memory consumptions.

Please note that performance and storage may vary, depending on device, operating system, and data.

- Outlines should not be requested from `T2K_RenderGlyph()` when they are not needed. Since the outlines are not cached, doing so substantially degrades the performance.
- For the best quality and end-user experience you should use grayscale antialiasing mode whenever possible.
- T2K run-time hinting degrades the performance. Keep in mind the following:
  - For monochrome output - turn it ON.
  - For a high quality display device such as a computer monitor - turn it ON.
  - For a device such as a TV monitor - turn it OFF.
- In the release build, turn OFF asserts in "`CONFIG.H`" to increase speed and to reduce the code size.
- For an embedded system where the fonts are well build and you do not have overlapping strokes, disable `USE_NON_ZERO_WINDING_RULE` in `config.h` as you will get a small (probably less than 1%) speed-up by disabling this.
- Employ one `tsiMemObject` per font as a slight performance advantage is observed.
- Turn ON auto-gridding only if it is required. Auto-gridding is not recommended for general use.
- Do not use Font Manager in normal conditions. It should be employed only when number of font streams are needed at a single instance of time.
- Do not call Font Fusion functions more often than needed.
- The include file names are totally configurable in Font Fusion. In case of header file name conflicts as `dtypes.h`, please rename the file and change the corresponding macros in `ffinclude.h`. If need arises to rename the `config.h` header file, please define the macro `FF_FFCONFIG_HEADER` to the desired file name in the build system.

FAQ

Q 1: What files do I need to look at initially?

A 1: First you need to familiarize yourself with the file "T2K.H"."T2K.H" contains documentation, a coding example and the actual T2K API.

Second you need to look at "CONFIG.H". "CONFIG.H" is the only file you normally need to edit. The file configures T2K for your platform, and it enables or disables optional features, and it allows you to build debug or non-debug versions. The file itself contains more information. Turn OFF features you do not need in order to minimize the size of the T2K font engine.

Q 2: What is the basic principle for the usage of T2K API?

A 2: The basic idea is that T2K was designed to be object oriented, even though the actual implementation is only using ANSI C. This means that you will be creating a number of objects when you use T2K.

All classes have a constructor and destructor. It is important that you call the proper destructor when you are done with a particular object.

Q 3: What is the best way of getting T2K going on a new platform?

A 3: First configure "CONFIG.H" and then look at the coding example T2K_DOCUMENTATION_CODING_EXAMPLE given in T2K.H. We recommend that you start "outside in".

First create and destroy a "Memhandler" object:

```
tsiMemObject *mem = NULL;
  mem = tsi_NewMemhandler( &errCode );
  assert( errCode == 0 );
  /* Destroy the Memhandler object. */
  tsi_DeleteMemhandler( mem );
```

Next create an InputStream object.

```
  tsiMemObject *mem = NULL;
  InputStream *in = NULL;
  mem = tsi_NewMemhandler( &errCode );
  assert( errCode == 0 );
    /* otherwise do this if you allocated the data */
    in = New_InputStream3( mem, data1, size1, &errCode );
    assert( errCode == 0 );
    /* Destroy the InputStream object. */
    Delete_InputStream( in, &errCode  );
  /* Destroy the Memhandler object. */
  tsi_DeleteMemhandler( mem );
```

Next you would create the "`sfntClass`" object and then finally the "`T2K`" scaler object.

Q 4: What do I do next with the T2K scaler object?

A 4: Follow the steps listed below:

A Set the "transformation" with the `T2K_NewTransformation()` call/ method. Essentially, you specify the pointsize, x and y resolution, and a 2*2 transformation matrix, and true/false if you want embedded bitmaps to be enabled.

B Call `T2K_RenderGlyph()` to get bitmap and/or outline data.

C Call `T2K_PurgeMemory` to free up memory.

Q 5: To get bitmap - I call `T2K_RenderGlyph()`, but where do I get access to the bitmap data?

A 5: You get access to the bitmap data through public fields in the `T2K` class/ structure. Find the following fields:

```
/* Begin bitmap data */
long width, height;
F26Dot6 fTop26Dot6, fLeft26Dot6;
long rowBytes;
unsigned char *baseAddr; /* unsigned char baseAddr[N], N = t-
>rowBytes * t->height */
uint32 *baseARGB;
/* End bitmap data */
```

`baseAddr` is either a bit-array, or a byte array.

`baseARGB` is a 32 bit array (ARGB) used for color bitmaps.

Q 6: Can you give me a simple/naive example of how to actually draw a character?

A 6: Simple naive example on how to get bitmap data from the `T2K` scaler object. The example assumes a screen-coordinate system where the top leftmost position on the screen is 0,0.

```
static void MyDrawCharExample( T2K *scaler, int x, int y )
{
  uint16 left, right, top, bottom;
  unsigned short R, G, B, alpha;
  uint32 *baseARGB = NULL;
  int xi, yi, xd;
  char *p;

  p = (char *)scaler->baseAddr;
  baseARGB = scaler->baseARGB;

  left= 0 + x;
  top = 0 + y;
  right= scaler->width  + x;
```

```
    bottom= scaler->height + y;


    if ( baseARGB == NULL && p == NULL )
      return; /*****/

    assert( T2K_BLACK_VALUE == 126 );

    MoveTo( x, y );

    for ( yi = top; yi < bottom; yi++ ) {
      for ( xi = left; xi < right; xi++ ) {
        xd = xi - left;

#ifdef USE_COLOR
        if ( baseARGB != NULL ) {
          alpha = baseARGB[xd] >> 24;/* Extract alpha */
          R = (baseARGB[xd] >> 16) & 0xff;/* Extract Red */
          G = (baseARGB[xd] >>  8) & 0xff;/* Extract Green */
          B = (baseARGB[xd] >>  0) & 0xff;/* Extract Blue */
        } else {
          alpha = p[xd];
          alpha = alpha + alpha + (alpha>>5); /* map [0-126] to
[0,255] */
          R = G = B = 0;/* Set to Black */
        }

        if ( alpha ) {
          RGBColor colorA, colorB;/* RGBColor contains 16 bit color
info for R,G,B each */

          GetCPixel( xi, yi, &colorB );/* Get the background color */
          alpha++; /* map to 0-256 */
          /* newAlpha = old_alpha + (1.0-old_alpha) * alpha */
          /* Blend foreground and background colors */
          R = (((long)(256-alpha) * (colorB.red>> 8) + alpha * R
)>>8);
          G = (((long)(256-alpha) * (colorB.green>> 8) + alpha * G
)>>8);
          B = (((long)(256-alpha) * (colorB.blue>> 8) + alpha * B
)>>8);

          assert( R >= 0 && R <= 255 );

          colorA.red= R << 8;/* Map 8 bit data to 16 bit data */
          colorA.green= G << 8;
          colorA.blue= B << 8;
          RGBForeColor( &colorA );/* Set the foreground color/paint
to colorA */
          MoveTo( xi, yi );/* Paint pixel xi, yi with colorA */
          LineTo( xi, yi );
        }
#else
        /* Paint pixel xi, yi */
        if ( p[ xd>>3] & (0x80 >> (xd&7)) ) {
          MoveTo( xi, yi );
          LineTo( xi, yi );
        }
#endif
```

```
        }

        /* Advance to the next row */
        p += scaler->rowBytes;
        if ( baseARGB != NULL ) {
          baseARGB += scaler->rowBytes;
        }
      }
    }
```

Q 7: How do I draw a string with the above MyDrawCharExample()?

A 7: Pseudo code on how to draw a string with the MyDrawCharExample() referred to in Question 6

```
  F16Dot16 x, y;
  x = y = 12 << 16;

  while (characters to draw..)
    /* Render the character */
    T2K_RenderGlyph( scaler, charCode, 0, 0,
           GREY_SCALE_BITMAP_HIGH_QUALITY,
           T2K_SCAN_CONVERT, &errCode );
    assert( errCode == 0 );
    /* Now draw the character */
    MyDrawCharExample( scaler, ((x + 0x8000)>> 16) + (scaler-
>fLeft26Dot6 >> 6),
             ((y + 0x8000)>> 16) - (scaler->fTop26Dot6 >>6) );
    x += scaler->xAdvanceWidth16Dot16;/* advance the pen forward */
    /* Free up memory */
    T2K_PurgeMemory( scaler, 1, &errCode );
    assert( errCode == 0 );
  }
```

Q 8: How do I get grey-scale or monochrome output?

A 8: For monochrome output set 5th input parameter to `T2K_RenderGlyph` called `greyScaleLevel` equal to `BLACK_AND_WHITE_BITMAP`.

To get grey-scale output set 5th input parameter to `T2K_RenderGlyph` called `greyScaleLevel` equal to `GREY_SCALE_BITMAP_HIGH_QUALITY`.

Q 9: How do I turn the T2K run-time hinting ON or OFF?

A 9: T2K run-time hinting is enabled by turning ON the `T2K_GRID_FIT` bit in the 6th input parameter to `T2K_RenderGlyph` called cmd.

It is disabled by turning OFF the `T2K_GRID_FIT` bit in the 6th input parameter to `T2K_RenderGlyph` called cmd.

Q 10: I am using an interlacing TV as the output device. How do I make it look good?

A 10: First turn OFF grid-fitting to get an image with less sharp transitions. (This also speeds up `T2K`). Then turn ON `T2K_TV_MODE` if you use integer metrics and do not use fractional positioning to improve the quality.

Then you most likely also want to experiment with a simple filter to make the image more blurry. A simple 3*3 convolution is probably sufficient. You should probably "average" more in the y-direction than the x-direction to avoid the interlacing flicker. Alternatively your hardware may have already have this built in.

Q 11: A T2K call/method returned an error. What should the code do?

A 11: `T2K` automatically deletes all of its objects when it hits an error. This means that all references to `T2K` objects become invalid, and can no longer be used.

Q 12: I noticed that there are a lot of asserts in the code. Why?

A 12: Asserts are in the code to detect/prevent programmer errors. They are only there to catch programmer errors.

In a release build you need to turn OFF asserts in "`CONFIG.H`" to increase speed and to reduce the code size. However in debug builds please leave it ON, to ensure that everything is working as intended.

Q 13: How do I decide what mapping table (character set) to use in a TrueType or T2K font?

A 13: Use `Set_PlatformID(scaler, ID)`, and `Set_PlatformSpecificID(scaler, ID)`

To use the Unicode mapping used by Windows do the following:

```
Set_PlatformID( scaler, 3 );
Set_PlatformSpecificID( scaler, 1 );
```

You can insert the code right after the `NewT2K` constructor.

Q 14: Can you explain the second parameter called code to `T2K_RenderGlyph`?

A 14: This normally specifies the character code for the character you wish to render.

However if you wish to use the glyph index instead then set the `T2K_CODE_IS_GINDEX` bit in the 6th input parameter to `T2K_RenderGlyph` called cmd. The glyph index is simply a number from 0 to N-1, assuming the font contains N glyphs.

```
(N = T2K_GetNumGlyphsInFont( scaler );)
```

Q 15: Does the outline winding direction matter in T2K?

A 15: Yes, Postscript outlines should use the correct winding direction and TrueType and T2K outlines also need to use the correct winding direction which is actually the opposite of the Postscript direction. It matters because the run-time-hinting process use this information to figure out where the black and white areas are. For TrueType and T2K the direction should be such that if you follow a contour in the direction of increasing point numbers then the black (inside) area should be on your right.

Q 16: I need my fonts to be as small as possible. What should I do?

A 16: You need to contact Type Solutions to have them translated to the T2K format or to apply font compression.

Q 17: Can you explain `ALGORITHMIC_STYLES` in `config.h`?

A 17: Use `ALGORITHMIC_STYLES` to enable algorithmic styling.

The 6th parameter to `FF_New_sfntClass();` `T2K_AlgStyleDescriptor *styling` is normally set to NULL, but if `ALGORITHMIC_STYLES` is enabled you can set it equal to an algorithmic style descriptor. Here is an example using the algorithmic bolding provided by T2K.

```
style.StyleFunc=tsi_SHAPET_BOLD_GLYPH;
style.StyleMetricsFunc=tsi_SHAPET_BOLD_METRICS;
style.params[0]=5L << 14;
sfnt0 = FF_New_sfntClass( mem, fontType, 0, in, NULL, &style,
&errCode );
```

You can also write your own outline based style modifications and use them instead of the algorithmic bolding provided by T2K. Just model them after the Type Solutions code for algorithmic bolding in "`SHAPET.c`".

Q 18: What is the story with `USE_NON_ZERO_WINDING_RULE` in `config.h`?

A 18: The recommended setting is to leave it ON. This determines what kind of fill rule the T2K scan-converter will use. This enables a non-zero winding rule, otherwise the scan-converter will use an even-odd filling rule. For example the even-odd filling rule will turn an area where two strokes overlap (rare) into white, but the non zero winding rule will keep such areas black. For an embedded system where the fonts are well build and you do not have overlapping strokes you will get a small (probably less than 1%) speed-up by disabling this.

Q 19: In the internal part of the options, there is an option called `SAMPO_TESTING_T2K (!)` which should be disabled. The only thing it

does is to enable `ENABLE_WRITE` and `ENABLE_PRINTF`. But those two options are enabled unconditionally at the top of "don't touch" part. What is going on?

A 19: You should leave it OFF, since it is in the "don't touch" part. When `T2K` is built as a font engine `T2K_SCALER` is defined (Not when built as a translator) `ifdef T2K_SCALER` actually does an `undef ENABLE_PRINTF` and an `undef ENABLE_WRITE`. Since the `T2K` font engine never does disk writes and does not use `printf` statements.

However, when the code is undergoing testing at Type Solutions the `T2K` font engine may write to disk for logging purposes and use `printf` statements.

Q 20: Are the functions used in the `T2K_DOCUMENTATION_CODING_EXAMPLE` part of `T2K.h` the only public APIs? Are there any other functions that might be useful to know?

A 20: Yes, but the idea is that you should only use functions/methods visible in `T2K.H`.

For instance `T2K_MeasureTextInX()` may be useful for quickly determining lengths of text string. If you find a need to use something else then let us know and if it makes sense we may bring out a public way to do it.

Do not rely on any function/methods outside of `T2K` since they may change from release to release.

Q 21: I found that when a string contains a space, `T2K_RenderGlyph` returns NULL `baseAddr`. Why is that? Do I have to check the existence of space characters and advance x position accordingly?

A 21: Since there is no bitmap to draw, T2K returns NULL `baseAddr`. Do not check for space, just check for `baseAddr == NULL` instead.

In future with `T2KE`, check for (`baseARGB == NULL && baseAddr == NULL`)

Q 22: Are there functions for measuring widths and other metrics of strings (such as `X11s XTextWidth`, and `XTextExtent`)?

A 22: `T2K_MeasureTextInX` in `T2K.H` is equivalent to `XTextWidth` in `X11`. (It measures the linear un-hinted width, it can not measure the hinted width without actually rendering the characters).

There is no equivalent to `X11s XTExtExtent`, since this sort of function typically would need to be implemented on top of the font-bit-map cache. `T2K` does not cache the output data, so `T2K` clients typically implement a cache on top of `T2K` so that the second time a character is requested it can come directly from the cache without invoking T2K. This sort of cache makes things go fast. String wide functions should be implemented so that they request information from the cache, to avoid T2K having to recompute everything several times.

T2K also has two early experimental functions called `T2K_GetIdealLineWidth()`, `T2K_LayoutString()`.

They can help you to layout an entire line so that the total width is the ideal linear width, while still using run-time hinted individual characters and metrics. At least is attempts to do this by mostly putting the nonlinearities into the space characters between the words.

Q 23: Can I edit T2K_BLACK_VALUE, and T2K_WHITE_VALUE so that I get a different range for the grey-scale?

A 23: No, you should not edit anything in `T2K.H`. They are there so that you can put in an assert in your code and can automatically detect if they are ever changed by Type Solutions in the future.

Q 24: What does T2K_TV_MODE do?

A 24: It improves the results for situations where you do ***not*** use `T2K_GRID_FIT` (T2K hinting) and you do use gray-scale such as the TV-screen and you do use integer metrics.

It compensates for the integralization of the advance width by adjusting the white space around the character ***and*** additionally it also ensures that we get left right symmetry in the gray-scale for simple symmetrical characters.

If you use `T2K_TV_MODE` then turn OFF `T2K_GRID_FIT`.

You should use `T2K_TV_MODE` when the following three conditions are true

- You do not wish to use `T2K_GRID_FIT` (T2K hinting).
- You are using gray-scale.
- You are using integer metrics and not fractional positioning with fractional metrics. (Only one version of each character per size).

Q 25: How do I get the type face name?

A 25: You basically call `T2K_SetNameString()` (see below), at some point after you have called `Set_PlatformID()`, and `Set_PlatformSpecificID()`.

It sets the public fields `nameString8` or `nameString16` in the T2K `object(structure)`.

To use Microsoft Unicode mapping and names you can use:

```
Set_PlatformID( scaler, 3 );
```

```
Set_PlatformSpecificID( scaler, 1 );/* 3,1 Picks Microsoft Unicode
character mapping. */
T2K_SetNameString( scaler, 0x0409, 4 );/* Picks American English &
the full font name */
```

Q 26: How do I get native TrueType hint support in T2K?

A 26: Follow the steps listed below:

A You require two additional .c and two .h files. (`fnt.c/.h` and `t2ktt.c/.h`)

B `#define ENABLE_NATIVE_TT_HINTS` in `config.h`.

C Then turn on `T2K_NAT_GRID_FIT` (T2K-native-grid-fitting) in the cmd parameter to `T2K_RenderGlyph()`.

Q 27: How do I slant the the text (make algorithmic italics)?

A 27: Set the transformation matrix this may:

```
trans.t00 = ONE16Dot16 * size;
trans.t01 = ONE16Dot16 * sin( italic_angle) * size;
trans.t10 = 0;
trans.t11 = ONE16Dot16 * size;
```

before calling `T2K_NewTransformation()`.

`Size` is a number such as 16.

`italic_angle` is a number such as 12.0 degrees. In this case `ONE16Dot16 * sin( 12.0)` would be 13626.

Q 28: I need the actual outline spline data. How do I get it?

A 28: Turn on the `T2K_RETURN_OUTLINES` to the cmd parameter to `T2K_RenderGlyph()`. This will set the public field glyph in `T2K`.

```
  /*** Begin outline data */
  GlyphClass *glyph;
  /*** End outline data */
  GlyphClass has public fields with the outline data.
  Here are the relevant fields in GlyphClass:
  shortcurveType;/* 2 for TrueType (2nd degree B-spline) outlines
and 3 for Type 1 (3rd Degree Bezier) */
  shortcontourCount;/* number of contours in the character */
  shortpointCount;/* number of points in the characters + 0 for the
sidebearing points */
  int16*sp;/* sp[contourCount] Start points */
  int16*ep;/* ep[contourCount] End points */
  int16*oox;/* oox[pointCount] Unscaled Unhinted Points, add two
extra points for lsb, and rsb */
  int16*ooy;/* ooy[pointCount] Unscaled Unhinted Points, set y to
zero for the two extra points */
         /* Do NOT include the two extra points in sp[], ep[],
contourCount */
         /* Do NOT include the two extra points in pointCount */
  uint8 *onCurve;/* onCurve[pointCount] indicates if a point is on
or off the curve, it should be true or false */
```

```
  F26Dot6 *x, *y;/* The actual points in device coordinates. */
```

The character is made out of '`contourCount`' contours.

The outline coordinates are stored in `F26Dot6` format in the x and y arrays. (6 fractional bits)

Each contour starts with the point number `sp[contour]`, and ends with with point number `ep[contour]`.

`sp[0]` should typically be zero. The letter A typically has 2 contours, B has 3, C has 1 etc.

The contours are self closing. Each point is either an "ON" or "OFF" curve point.

A third degree Bezier is ON, OFF, OFF ON.

A 2nd degree parabola is ON, OFF, ON.

So a particular point n is described by x[n], y[n], onCurve[n].

NOTE: Please note that 2nd degree B-spline allow many consecutive OFF curve points.

Q 29: I do not suppose you have any sample code which actually goes through the process of getting and evaluating the outline data?

A 29: To see how `T2K` breaks down the outlines into parabolas (or 3rd degree beziers) and straight lines you can look at:

`Make2ndDegreeEdgeList()` for parabolas and

`Make3rdDegreeEdgeList()` for 3rd degree beziers inside `T2KSC.c`.

Once you have a parabola (described by three points A,B,C) it is described in parametric form by

```
(1-t)*(1-t)*A + 2 *t *(1-t)*B + t*t*C
```

Once you have a 3rd degree Bezier (described by 4 points A,B,C,D) it is described in parametric form by

```
(1-t)*(1-t)*(1-t)*A + 3*(1-t)*(1-t)*t*B + 3*(1-t)*t*t * C+ t*t*t *
D
```

In both cases t starts being equal to zero at point A and then it goes to one by the last point.

Q 30: Can you explain the glyph specific metrics in more detail ?

A 30: If you have regular left to right text, follow the horizontal text example given on the next page:

Here is a lower case letter "g" showing the meanings of glyph specific metrics.

```
       t2k->fTop26Dot6 (26.6) This is the vertical line below, from base line to
the top of the bitmap.
         <----> t2k->fLeft26Dot6 (26.6)
          ^    ***********    ^ t2k->height (integer number of scanlines) (32.0)
           |   *            *  |
           |   *            *  |
           |   *            *  |
           |   *            *  |
           |   *            *  |
---------0 v----************---|---0-----------------
         ^                  *  |   ^
pen pos- |                  *  |   |- next pen position.
                            *  |
               *            *  |
                ***********    v

              <------------->   t2k->width (integer number of pixels) (32.0)
         <------------------------->   t2k->xAdvanceWidth16Dot16 (16.16)
```

So for horizontal text put the left top corner of the bitmap at

```
[((x + 0x8000)>> 16) + (scaler->fLeft26Dot6 >> 6), ((y + 0x8000)>>
16) - (scaler->fTop26Dot6 >>6)]
```

then advance the pen:

```
x += scaler->xAdvanceWidth16Dot16;
y += scaler->yAdvanceWidth16Dot16;
```

and for vertical text put the left top corner of the bitmap at

```
[((x + 0x8000)>> 16) + (scaler->vert_fLeft26Dot6 >> 6), ((y +
0x8000)>> 16) - (scaler->vert_fTop26Dot6 >>6)]
```

then advance the pen:

```
x += scaler->vert_xAdvanceWidth16Dot16;
y += scaler->vert_yAdvanceWidth16Dot16;
```

If the quality of the rotated text is important then you can use fractional positioning where you maintain fractional accuracy for the pen-position and pass in this information to `RenderGlyph`.

Since both the x and y position have 6 fractional bits for the position (fractional positioning), for a given rotational transform there are 64*64 = 4096 possible bitmap shapes for the same character at that rotation. Here you need to pass in the fractional position in x and y to `RenderGlyph`.

To do this you need to call `RenderGlyph` this way:

```
long iXPen, iYPen;
iXPen = ((x+ 0x8000) >> 16);
iYPen = ((y+ 0x8000) >> 16);
tmp = x - (iXPen <<16); xFracPenDelta = tmp >> 10; /* x is the X-
pen-position */
tmp = y - (iYPen <<16); yFracPenDelta = tmp >> 10 ;/* y is the Y-
pen-position */
T2K_RenderGlyph( scaler, charCode, xFracPenDelta, yFracPenDelta,
grayScaleLevel, cmd, &errCode);
```

If the quality of rotated text is more important then the quality of the text at regular 90 degree angles you should do this to advance the pen and at the same time do not use hinting (turn OFF both `T2K_NAT_GRID_FIT` and `T2K_GRID_FIT`)

```
x += scaler->xLinearAdvanceWidth16Dot16;
y += scaler->yLinearAdvanceWidth16Dot16;
instead of the below with hinting active (T2K_NAT_GRID_FIT):
x += scaler->xAdvanceWidth16Dot16;
y += scaler->yAdvanceWidth16Dot16;
```

Q 31: How do I use my own memory allocator and deallocator with T2K?

A 31: `CONFIG.H` allows you to remap allocation, reallocation and deletion to anything you want.

Q 32: What is the persistence of some of the different T2K data structures needs to be? In particular: Does the input stream need to be allocated for as long as the font is active and allocated? I would like to be able to do this within the initialization stage:

```
memHandler_ = tsi_NewMemHandler( &err );
inStream = New_InputStream3( memHandler_, data, length, &err );
font_ = New_sfntClass( memHandler_, fontType, inStream, NULL, &err
);
...
scaler_ = NewT2K( font_->mem, font_, &err );
...
Delete_InputStream( inStream, &err );
```

That is, I want to delete the input stream and continue to use the font. Is this feasible, or does the font continue to refer to the input stream even after the `sfntClass` and the T2K structures have been built?

A 32: The various object have to be deleted in the reverse order of creation. This means that the `InputStream` object can only be deleted ***`after`*** you have deleted the `T2K` and `sfntClass` object.

Q 33: Does the transformation need to exist as an external (application-managed) structure after calling `T2K_NewTransformation`? If not, I would like to be able to do this within the font initialization stage:

```
trans_ = new T2K_TRANS_MATRIX;
trans_->t00 = ONE16Dot16 * pointSize;
trans_->t01 = 0;
trans_->t10 = 0;
trans_->t11 = ONE16Dot16 * pointSize;
T2K_NewTransformation( scaler_, true, 72, 72, trans_, true, &err );
delete trans_; ?
```

A 33: The `T2K_TRANS_MATRIX` structure can be deleted right after `T2K_NewTransformation()`.

Q 34: Our application will typically have between three and a dozen fonts open and active at the same time. Am I correct to assume this is not going to cause any problems within T2K?

A 34: This is OK. You should have one `tsiMemObject` per font.

You can choose to keep multiple fonts open simultaneously and you can also decide to have one or multiple T2K scalers simultaneously, or you could decide to only allow one font open at a time.

You can also decide to either use memory based fonts when you create the `InputStream` class or you can choose to use disk based fonts. These choices are a trade-off between memory use and speed.

Q 35: I turn ON the `T2K_RETURN_OUTLINES` bit flag to `T2K_RenderGlyph` and I am using the `t2k->glyph` structure. Unfortunately my code can only handle 3rd degree Beziers. What can I do when `t2k->glyph->curveType == 2`?

A 35: First we encourage you to see if your current code can handle the 2nd degree curves directly since they can be rendered quicker than 3rd degree curves.

If not, then this is how you go between them:

A First we need to find all straight lines and parabolas: The function `Make2ndDegreeEdgeList()` function from `T2KSC.c` shows how that is done:

B Now each Parabola (== 2nd degree Bezieer) is described by the points A,B,C. Each 3rd degree Bezier is described by points P1,P2,P3,P4.

To map points [A,B,C] to [P1,P2,P3,P4] do this:

P1 = A;

P2 = (2B + A)/3;

P3 = (2B + C)/3;

P4 = C;

Q 36: What should I do if T2K returns an error?

A 36: Once you get an error from T2K it frees up ALL memory. This means that you need to set all references to T2K to NULL and not call any T2K delete routines etc. Basically you have to start from the beginning again as if T2K never existed when you get an error.

Q 37: Now, if an error is encountered, is it just that particular T2K that must be "restarted", or all the T2K objects?

A 37: Not all T2K objects just all objects that shared the same tsiMemObject have to be restarted. You should have one tsiMemObject per font.

Q 38: How do I make a colored bordered character?

A 38: Just invoke invoked T2K_CreateBorderedCharacter (in T2KEXTRA.c) right after T2K_RenderGlyph().

After this you can find the 32 bit colored bordered character in t2kscaler->baseARGB. The format is ARGB, 8 bits each.

Q 39: How do I drive LCD displays?

A 39: There are two categories of LCD modes that you can use. Bitstream recommends you use the new one to get the best result.

Old LCD mode:

Here are step by step instructions:

A Enable the ENABLE_LCD_OPTION in config.h.

B Use FF_NewColorTable to get the RGB colors for LCD display. These colors will be indexed by any bitmap produced by T2K. If your platform is using a Color Lookup Table, you will need to set these colors in that table.

This is how you extract the actual RGB colors from the T2K color table:

```
      ff_ColorTableType *pColorTable;

      /* for black text on white Set Rb = Gb = Bb = 0xff, and Rf =
Gf = Bf = 0. */
      pColorTable = FF_NewColorTable( mem, Rb, Gb, Bb, Rf, Gf, Bf
);

      /* For all the indeces in the bitmap you get the color by
doing this. */
      /* pColorTable->N will contain # elements in the array */
      /* pColorTable->ARGB[0] contains the first ARGB value */
      ARGB = pColorTable->ARGB[ byte index from the bitmap ];
      B = (ARGB & 0xff); ARGB >>= 8;
      G = (ARGB & 0xff); ARGB >>= 8;
      R = (ARGB & 0xff);

      /* When done free up the color-table, but please do not call
this per-character for speed reasons. */
      FF_DeleteColorTable( mem, pColorTable);
```

C Do not invoke either FF_SetBitRange255() or FF_SetRemapTable(). If you need to shift the range, we recommend using a filter function.

D When you invoke `T2K_NewTransformation()` set the `xRes` to 3 times the `yRes`, since if you look close at the screen there are 3 times as many colored pixels in the x direction as there are in the y direction. This tells T2K we have a non-square aspect ratio where the x resolution is 3 times higher than the y resolution.

E Then set `T2K_LCD_MODE_4` in the cmd parameter and `GREY_SCALE_BITMAP_HIGH_QUALITY` in the `greyScalelevel` parameter to the function `T2K_RenderGlyph()`.

F You now have an indexed color bitmap. When you draw the bitmap you need to take into account that the bitmap contains indices for the colored pixels.

New LCD mode:

Here are step by step instructions:

A Enable the `ENABLE_EXTENDED_LCD_OPTION` in `config.h`.

B Use `FF_NewColorTable` to get the RGB colors for LCD display. These colors will be indexed by any bitmap produced by T2K. If your platform is using a Color Lookup Table, you will need to set these colors in that table.

This is how you extract the actual RGB colors from the T2K color table:

```
ff_ColorTableType *pColorTable;

      /* for black text on white Set Rb = Gb = Bb = 0xff, and Rf =
Gf = Bf = 0. */
      pColorTable = FF_NewColorTable( mem, Rb, Gb, Bb, Rf, Gf, Bf
);

      /* For all the indeces in the bitmap you get the color by
doing this. */
      /* pColorTable->N will contain # elements in the array */
      /* pColorTable->ARGB[0] contains the first ARGB value */
      ARGB = pColorTable->ARGB[ byte index from the bitmap ];
      B = (ARGB & 0xff); ARGB >>= 8;
      G = (ARGB & 0xff); ARGB >>= 8;
      R = (ARGB & 0xff);

      /* When done free up the color-table, but please do not call
this per-character for speed reasons. */
      FF_DeleteColorTable( mem, pColorTable);
```

C Do not invoke either FF_SetBitRange255() or FF_SetRemapTable(). If you need to shift the range, we recommend using a filter function.

D There are four LCD modes in this new LCD category. They are horizontal left-to-right RGB, horizontal left-to-right BGR ,vertical top-to-down RGB and vertical Top-to-down BGR. So turn on the appropriate `bitflag` in cmd. For example, if you need horizontal left-to-right RGB mode, set `T2K_EXT_LCD_H_RGB` in the cmd parameter. Set `GREY_SCALE_BITMAP_HIGH_QUALITY` in the `greyScalelevel` parameter to the function `T2K_RenderGlyph()`.

E You now have an indexed color bitmap. When you draw the bitmap you need to take into account that the bitmap contains indices for the colored pixels.

F Unlike the old LCD mode, you do NOT have to set the `xRes` to 3 times in the `T2K_NewTransformation()`.

Q 40: What is the difference between T2K_TV_MODE_2 and T2K_TV_MODE, and also between T2K_LCD_MODE and T2K_LCD_MODE_2?

A 40: The _2 modifier activates a special light-weight y hint strategy which improves the quality. It for instance makes the antialiased bitmap pattern on top and bottom of an 'o' symmetrical. It is recommended for both TV and LCD output for the best quality.

The regular LCD and TV modes already also make such characters left-right symmetrical for optimal quality.

For best quality, `T2K_LCD_MODE_4` is recommended because it employs native hints.

Q 41: What is a tsiMemObject object, what does it do?

A 41: It is an object that handles all memory allocation, deallocation and reallocation.

The reason we have it done by one object instead of direct calls to the OS is that the `tsiMemObject` object does a lot of error checking. This creates a more stable product.

For instance it puts special markers both before and after allocated memory so that we can detect any attempts to write outside the allocated memory. It also detects any memory leaks and attempts to free no-pointers or or already freed memory and other memory errors. Basically the `tsiMemObject` provides a solid foundations for the product.

Q 42: Why do we need one tsiMemObject per font?

A 42: We could have shared this. But it seems we get a slight performance advantage using only one per font.

The current implementation of `tsiMemObject` also has a maximum limit on the number of pointers it can allocate.

Q 43: What is a InputStream object, what does it do?

A 43: The `InputStream` object provides a level of abstraction for the core. Basically the `InputStream` object exposes certain methods that `T2K` uses to access the data. This means that `T2K` does not need to know if the data is in memory, on the disk, or a across a network, etc.

This provides a cleaner design. The `InputStream` object also checks for out-of bounds read attempts. This error checking together with the nice abstraction `InputStream` provides produces a more solid and robust design and therefore a better product.

Q 44: What does a sfntClass do?

A 44: The `sfntClass` is an internal class that represents a font. It is shared by all supported font formats.

Q 45: What is a T2K class, what does it do?

A 45: The `T2K` object represents an instance of the font scaler. Main task for the fonts scaler is to produce good looking bitmap images for characters at different sizes and transformations like rotation.

Q 46: T2K_NewTransformation transform which item to what?

A 46: `T2K_NewTransformation` is a method that informs the `T2K` object about the current transformation and size is.

Q 47: What does `T2K_RenderGlyph()` do?

A 47: It produces the bitmap image that we want.

Q 48: I am using one of the Font Fusion stroke fonts and the output is too light! Can you make a heavier/bolder font?

A 48: Font Fusion stroke font has variable weight. You are probably just using the default weight which is pretty light. The default is at 0.5 in 16.16 space. The weight is variable between 0.0 and 1.0 in 16.16 space. For instance try this:

```
if ( T2K_GetNumAxes( scaler ) == 1 ) {
  T2K_SetCoordinate( scaler, 0, 0x10000*7/10 ); /* instead of
default 0x10000/2 */
}
```

You can do this setting right after `T2K_NewTransformation();`

You can tune the number to whatever looks best in your circumstance.

Q 49: Please explain more about anti-aliasing and how to blend letters with a random background image.

A 49: Instead of setting `greyScaleLevel` to the function `T2K_RenderGlyph` to `BLACK_AND_WHITE_BITMAP` set it to `GREY_SCALE_BITMAP_MEDIUM_QUALITY` and Font Fusion will return a gray-scale bitmap. This is the same as an alpha mask.

Think of an alpha value and a bitmap filled with alpha values (an alpha mask) conceptually as a partially transparent glass window. Also think of the alpha value as conceptually going between 0.0 (fully transparent) to 1.0 (fully opaque).

When the alpha value is 0.0 you see everything behind the window (the background), but not the foreground. When the alpha value is 1.0 you only see the foreground color (or image).

When the alpha value is 0.5 you see a blend consisting of 50% the background and 50% the foreground. To perfectly blend the foreground and background you need to pixel-by-pixel blend the background pixel values and foreground pixel values according to this linear formula

```
newPixelColor = alpha*foregoundPixelColor + (1.0-alpha) *
backgroundPixelColor
```

When Font Fusion operates in its normal greyscale mode then a value of `T2K_BLACK_VALUE (126)` in the alphamask returned by Font Fusion corresponds to alpha == 1.0, and 0 corresponds to alpha == 0.

Q 50: What are the benefits of using Cache Manager in Font Fusion?

A 50: Cache manager speeds up the performance of your application. If a character exists in the cache, the Cache Manager will deliver the bitmap to the calling application instead of Font Fusion having to create the character each time it is needed. The trade off is memory; the more memory allocated for cache purposes, the more characters can be stored in the cache.

Q 51: What files should I look at for the Cache Manager?

A 51: There are two files that involve the cache: "`cachemgr.c`" and "`cachemgr.h`". "`cachemgr.h`" contains documentation, a coding example and the actual Cache Manager API, while the source code is contained in the file "`cachemgr.c`".

Q 52: How much memory should I allocate for the cache?

A 52: The Cache Manager uses the amount of memory allocated to it at the time of creation. No memory is allocated by the Cache Manager on its own. The amount of memory that is set at creation time will hold all the characters in the cache as well as the cache framework itself. Therefore, the final amount of usable memory for the cache is the total declared at creation time minus the size of the cache management structures.

The amount of memory desired depends on the proposed uses of the application. For example, if large 500 line bitmaps will be created, then allocate plenty of memory for the cache manager. If large gray scale images are needed, allocate over 100K for the cache to use. The more memory the cache is given, the more characters it can hold. As the Cache Manager runs out of space the cache will get rid of the oldest characters that it was holding in order to make room for the newly created characters.

Q 53: How do I use the Cache Manager?

A 53: There is not much for the application to do with the cache. It can create the cache, make render glyph calls through it, flush and delete. Filter functions are attached to Font Fusion through the cache. The function calls are:

`FF_CM_New ( )`, `FF_CM_Delete ( )`, `FF_CM_RenderGlyph ( )`, `FF_CM_Flush ( )`, and `FF_CM_SetFilter ( )`.

To use the Cache Manager:

A Call the Cache Manager constructor `FF_CM_New` specifying the amount of memory to use

B Set a filter tag with `FF_CM_SetFilter`

C Create characters with `FF_CM_RenderGlyph`

Q 54: I noticed that T2K, the Font Manager AND the Cache Manager each have RenderGlyph functions. What is the story?

A 54: They were designed that way so they could be independent of each other and work together. The real `RenderGlyph` work is always done in T2K Core. If you are using the Cache Manager, the `FF_CM_RenderGlyph()` will first check the cache for the glyph or call another module to render the glyph into the cache. It will use either the Font Manager `RenderGlyph` function or call the T2K core. The Font Manager `RenderGlyph` function will look for the requested glyph from among the font fragments of the font, and then call the `T2K_RenderGlyph()` function.

Q 55: How does the Cache Manager know if the Font Manager should render a glyph? What is the configuration requirement for me to make these work together?

A 55: There are no configuration requirements. You just build the Font or Cache Managers and use them at run time. If you "register" a font with the Manager, when you create and select a strike, the Font Manager stamps or marks itself in the T2K Class to let the Cache Manager know it is present. If you are using the Cache Manager, it will respect this little stamp, which consists of enough information for the Cache Manager to use the Font Managers API.

Q 56: Tell me more about the setting up of a post-processing filter and why does this involve the cache manager?

A 56: Font Fusion allows filters to be registered with the core. Each filter has a separate ID. By setting that filter ID with the cache the ID will be stored with the created bitmap in the cache. This is used as further search criteria when retrieving characters. Filter functions are registered through the cache manager to the core. If a filter function has been set up the core will call this function.

Index