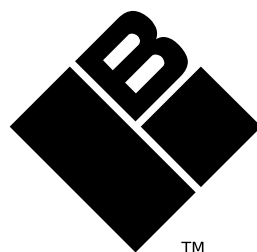

Bitstream® Font Fusion™ 2.1

Reference Guide

May 2001



TM
BITSTREAM

Bitstream® Font Fusion™ Version 2.0

Information in this document is subject to change without notice.

BITSTREAM INC. MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Bitstream Inc. shall not be liable for errors herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

© Copyright 1999-2001 Bitstream Inc., Cambridge, MA. All rights reserved. No part of this document may be photocopied, reproduced, or translated without the prior written consent of Bitstream Inc.

Document printing history:

Bitstream Font Fusion Reference Guide, v. 2.1, May, 2001

Bitstream Font Fusion Reference Guide, v. 2.0, June, 2000

Bitstream Font Fusion Reference Guide, v. 1.1, February, 2000

Bitstream Font Fusion Reference Guide, v. 1.0, October, 1999

Bitstream, the Bitstream logo, and TrueDoc are registered trademarks of Bitstream Inc. and Dutch, Font Fusion, Speedo, Swiss, and Zurich are trademarks of Bitstream Inc. T2K is a registered trademark of Type Solutions, Inc., a Bitstream company.

Bitstream TrueDoc: U.S. Patent Nos. 5,577,177 and 5,583,978

Adobe, ATM, Adobe Type Manager, and PostScript are trademarks of Adobe Systems, Incorporated, and may be registered in some jurisdictions. Apple, Macintosh, and TrueType are registered trademarks of Apple Computer, Inc. Hewlett-Packard, HP, and PCL are registered trademarks of Hewlett-Packard Company. Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation.

All other product or company names are used for identification purposes only, and may be registered trademarks or trademarks of their respective owners.

Bitstream Inc., 215 First Street, Cambridge, MA 02142

Worldwide phone number: 617-497-6222

Phone number in the U.S. and Canada: 800-522-3668

Phone number in Europe: +31 20 5200 300

©1999-2001 Bitstream Inc. All rights reserved.

Printed in the United States of America

Table of Contents

CHAPTER 1: FONT FUSION OVERVIEW

| | |
|--|------|
| General Information | 1-2 |
| Architectural Overview | 1-2 |
| Applications & Operating Systems Supported | 1-2 |
| Font Formats Supported | 1-3 |
| Multilingual Capabilities | 1-4 |
| Devices Supported | 1-5 |
| High-Quality Output | 1-6 |
| Using the Font Fusion Core | 1-9 |
| Using the Font Fusion Font Manager | 1-10 |
| Merging Fonts Dynamically | 1-10 |
| Using the Font Fusion Cache Manager | 1-12 |

CHAPTER 2: GETTING STARTED WITH THE FONT FUSION CORE

| | |
|---|------|
| Getting Started | 2-2 |
| What Files Should I Look at First? | 2-2 |
| What is the Best Way to Get Started? | 2-2 |
| What are the Functions in Font Fusion? | 2-3 |
| Should I Use Public APIs Only? | 2-5 |
| Allocating Memory | 2-6 |
| Using Your Own Memory Allocator and De-allocator with Font Fusion | 2-6 |
| The InputStream Object | 2-6 |
| The tsiMemObject Object | 2-7 |
| Using One tsiMemObject per Font | 2-7 |
| If You Have a lot of Fonts Open and Active at the Same Time | 2-7 |
| Assert Statements | 2-8 |
| Optional: Redefining “Assert” | 2-8 |
| Compile-Time Options | 2-9 |
| Errors | 2-13 |
| What Happens When Font Fusion Returns an Error | 2-13 |

| | |
|---|------|
| What to Do if Font Fusion Returns an Error | 2-13 |
| Font Fusion Objects You Need to Restart if Font Fusion Returns an Error | 2-13 |
| Creating Compact Fonts | 2-14 |

CHAPTER 3: FONT FUSION CORE API

| | |
|--|------|
| tsi Functions: Overview | 3-2 |
| The tsiMemObject Object | 3-2 |
| Using One tsiMemObject per Font | 3-2 |
| Creating and Destroying a Memory Handle | 3-2 |
| tsi Functions | 3-3 |
| tsiMemObject *tsi_NewMemhandler(. | 3-3 |
| void tsi_DeleteMemhandler(. | 3-3 |
| InputStream Functions: Overview | 3-4 |
| The InputStream Object | 3-4 |
| Creating an Input Stream | 3-4 |
| If You Have a lot of Fonts Open and Active at the same Time | 3-5 |
| Using Your Own Memory Allocator and De-allocator with Font Fusion | 3-5 |
| InputStream Functions | 3-6 |
| InputStream *New_InputStream3(. | 3-6 |
| InputStream *New_InputStream(. | 3-6 |
| InputStream *New_NonRamInputStream(. | 3-7 |
| void PF_READ_TO_RAM(. | 3-8 |
| void Delete_InputStream(. | 3-9 |
| sfntClass Functions: Overview | 3-10 |
| The sfntClass Object | 3-10 |
| ALGORITHMIC_STYLES | 3-10 |
| sfntClass Functions | 3-11 |
| sfntClass *FF_New_sfntClass(. | 3-11 |
| void FF_Delete_sfntClass(. | 3-12 |
| PlatformID Functions: Overview | 3-13 |
| The PlatformID | 3-13 |
| Setting the Platform and Platform-Specific ID | 3-13 |
| Mapping Table to Use with TrueType and Native T2K Fonts | 3-13 |
| Getting the Font Name | 3-14 |

| | |
|--|------|
| PlatformID Functions | 3-15 |
| Set_PlatformID(..... | 3-15 |
| Set_PlatformSpecificID(..... | 3-15 |
| T2K Functions: Overview | 3-17 |
| The T2K Scaler Object. | 3-17 |
| Obliquing Text (i.e., Making Algorithmic Italics) | 3-18 |
| T2K_RenderGlyph(): Getting Bitmap and Outline Output | 3-19 |
| T2K_RenderGlyph(): Hinting | 3-24 |
| T2K_RenderGlyph(): Rendering Characters and Strings | 3-25 |
| T2K_RenderGlyph(): Sample Code for Rendering Characters and Strings | 3-29 |
| The Filter Function | 3-33 |
| Getting the Font Name | 3-37 |
| Enabling “sbits”. | 3-38 |
| Measuring the Widths and Other Metrics of Strings (such as X11’s XTextWidth, and XTextExtent) | 3-38 |
| T2K Functions | 3-39 |
| T2K *NewT2K(..... | 3-39 |
| void DeleteT2K(..... | 3-39 |
| void T2K_NewTransformation(..... | 3-40 |
| void T2K_RenderGlyph(..... | 3-41 |
| void T2K_GaspifyTheCmds(..... | 3-43 |
| int T2K_GetBytesConsumed(..... | 3-44 |
| void T2K_ConvertGlyphSplineType(..... | 3-44 |
| void FF_Set_T2K_Core_FilterReference(..... | 3-45 |
| void T2K_PurgeMemory(..... | 3-46 |
| void T2K_SetNameString(..... | 3-46 |
| uint16 T2K_GetGlyphIndex(..... | 3-47 |
| char T2K_FontSbitsExists(..... | 3-47 |
| char T2K_FontSbitsAreEnabled(..... | 3-48 |
| int T2K_GlyphSbitsExists(..... | 3-48 |
| void T2K_TransformXFunits(..... | 3-49 |
| void T2K_TransformYFunits(..... | 3-50 |
| T2K_KernPair *T2K_FindKernPairs(..... | 3-51 |
| uint32 T2K_MeasureTextInX(..... | 3-52 |
| void T2K_GetIdealLineWidth(..... | 3-53 |
| void T2K_LayoutString(..... | 3-55 |
| Functions for Translating Font Data | 3-56 |

| | |
|--|------|
| unsigned char *ExtractPureT1FromPCType1(. | 3-56 |
| char *ExtractPureT1FromMacPOSTResources(. | 3-56 |
| Additional Functions | 3-58 |
| uint8 *FF_GetTTTablePointer(. | 3-58 |
| int FF_GlyphExists(. | 3-59 |
| void FF_ForceCMAPChange(. | 3-59 |
| ff_ColorTableType *FF_NewColorTable(. | 3-60 |
| int FF_PSNameToCharCode(. | 3-62 |
| Sample Code. | 3-63 |
| Macintosh. | 3-63 |
| T2K Scaler. | 3-63 |

CHAPTER 4: FONT MANAGER API

| | |
|--|------|
| Getting Started with the Font Manager. | 4-2 |
| Why Use the Font Manager? | 4-3 |
| What Files Should I Look at First? | 4-3 |
| How Do I Use the Font Manager? | 4-4 |
| Why Do Font Fusion, the Font Manager, and the Cache Manager Have a RenderGlyph() Function? | 4-5 |
| 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? | 4-5 |
| How Many Fonts Can I Handle at Once? | 4-5 |
| Are There Any Other Configuration Parameters for the Font Manager? | 4-6 |
| Why Does FF_FM_AddTypefaceStream() Take Two Stream Arguments? | 4-6 |
| Why Does FF_FM_CreateFont() Include a flushCache Parameter? | 4-6 |
| Is There a Coding Example? | 4-7 |
| Functions for Creating, Configuring, and Deleting the Font Manager. | 4-8 |
| FF_FM_Class *FF_FM_New(. | 4-8 |
| void FF_FM_AddTypefaceStream(. | 4-9 |
| void FF_FM_SetPlatformID(. | 4-10 |
| void FF_FM_SetPlatformSpecificID(. | 4-10 |
| void FF_FM_SetLanguageID(. | 4-12 |
| void FF_FM_SetNameID(. | 4-12 |
| void FF_FM_Delete(. | 4-13 |
| Function for Installing Fonts and Getting Font Information | 4-14 |
| enumTypefaceCallback() Function. | 4-14 |

| | |
|---|------|
| int FF_FM_EnumTypefaces(. | 4-14 |
| Functions for Creating and Using Fonts. | 4-16 |
| uint16 FF_FM_CreateFont(. | 4-16 |
| void * FF_FM_SetXYResolution(. | 4-19 |
| T2K * FF_FM_SelectFont(. | 4-19 |
| void FF_FM_DeleteFont(. | 4-20 |
| void FF_FM_RenderGlyph(. | 4-21 |
| Sample Code. | 4-23 |

CHAPTER 5: CACHE MANAGER API

| | |
|---|------|
| Getting Started with the Cache Manager | 5-2 |
| Overview of the Cache Manager. | 5-2 |
| If You Want to Write Your Own Cache Manager | 5-2 |
| Why Do Font Fusion, the Font Manager, and the Cache Manager Have a RenderGlyph() Function? | 5-2 |
| Functions for Creating and Deleting the Cache Manager | 5-4 |
| FF_CM_Class *FF_CM_New(. | 5-4 |
| void FF_CM_Delete(. | 5-4 |
| Functions for Working with the Cache Manager: Overview | 5-6 |
| Filter Function | 5-6 |
| Functions for Working with the Cache Manager | 5-12 |
| void FF_CM_RenderGlyph(. | 5-12 |
| int FF_CM_GlyphInCache(. | 5-13 |
| void FF_CM_Flush(. | 5-14 |
| void FF_CM_SetFilter(. | 5-15 |
| Sample Code. | 5-16 |

APPENDIX A: FONT FUSION API FOR PRINTER DEVELOPERS

| | |
|---------------------------------|-----|
| General Information | A-2 |
| Compile-Time Options | A-3 |
| Font Types. | A-4 |
| Callback Functions. | A-5 |
| int eo_get_char_data(. | A-5 |
| int tt_get_char_data(. | A-6 |

metricsInfo A-7

APPENDIX B: TEXT FLOWS

Overview B-2

Font Fusion Core B-3

Font Manager B-4

Cache Manager B-5

Font Manager and Cache Manager B-6

APPENDIX C: ERROR CODES

Font Fusion Core Error Codes. C-2

Font Manager Error Codes C-3

Cache Manager Error Codes C-4

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 the latest multi-font rasterizing engine available from Bitstream. Font Fusion is designed to support operating systems, software applications, Web applications, low-resolution screen devices, multimedia servers, high-definition television screens (HDTVs), set-top boxes, continuous tone printers, personal digital assistants (PDAs), and other embedded systems and Internet appliances.

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

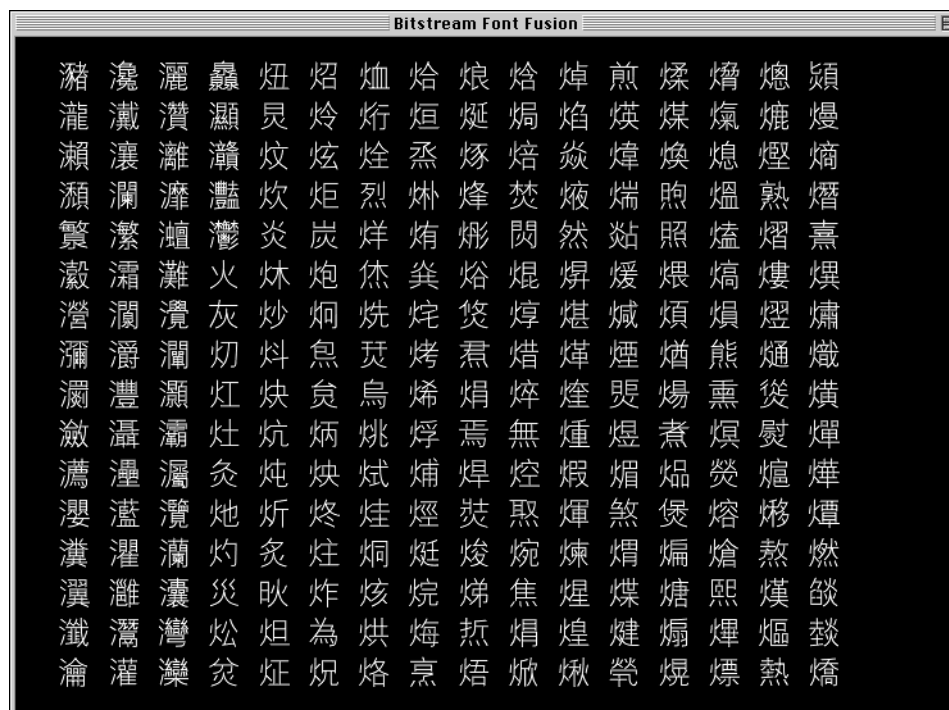
Font Fusion supports a multitude of applications and operating systems, including:

- 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:

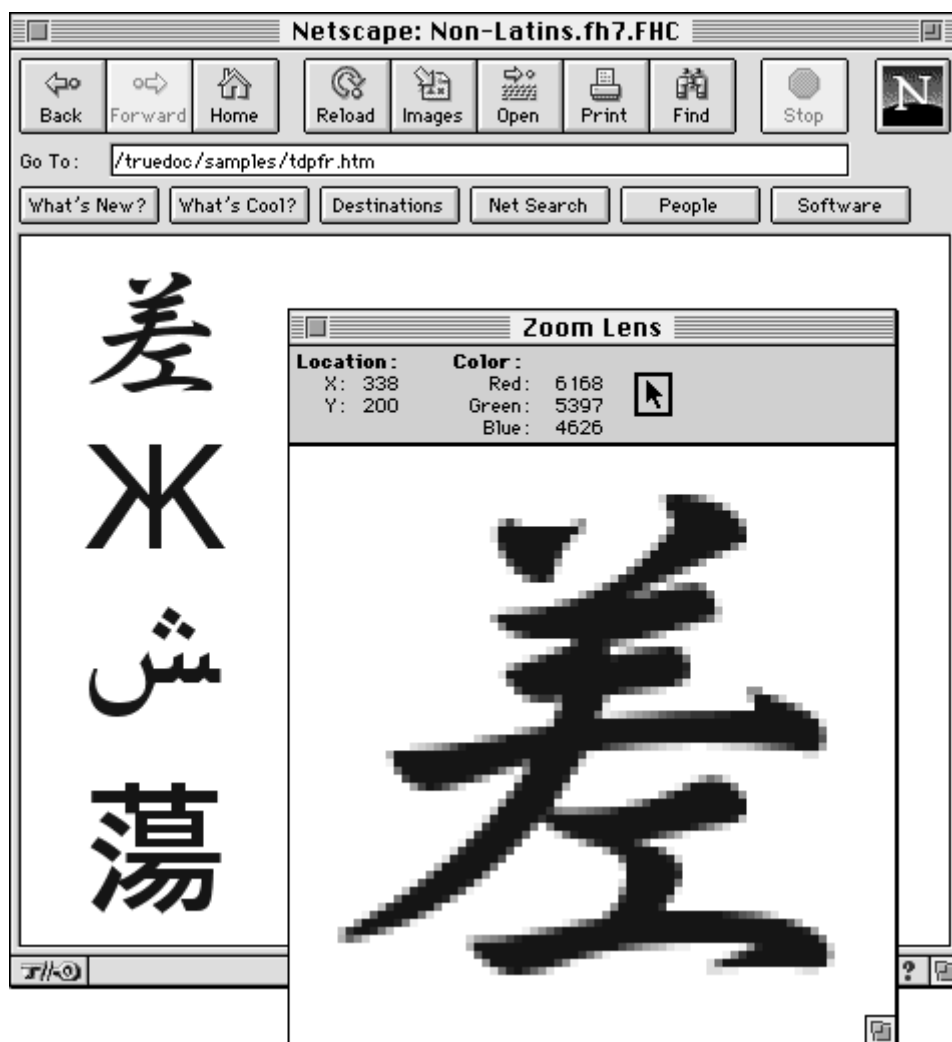
- Type 1
- TrueType®
- TrueType collections
- Compact font format (CFF)/Type 2
- TrueDoc portable font resources (PFRs)/Web fonts
- Bitstream Speedo™
- T2K
- Font Fusion stroke (FFS) format
- Embedded bitmaps (TrueType, TrueDoc, and T2K formats)



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 (also known as Web font) 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.

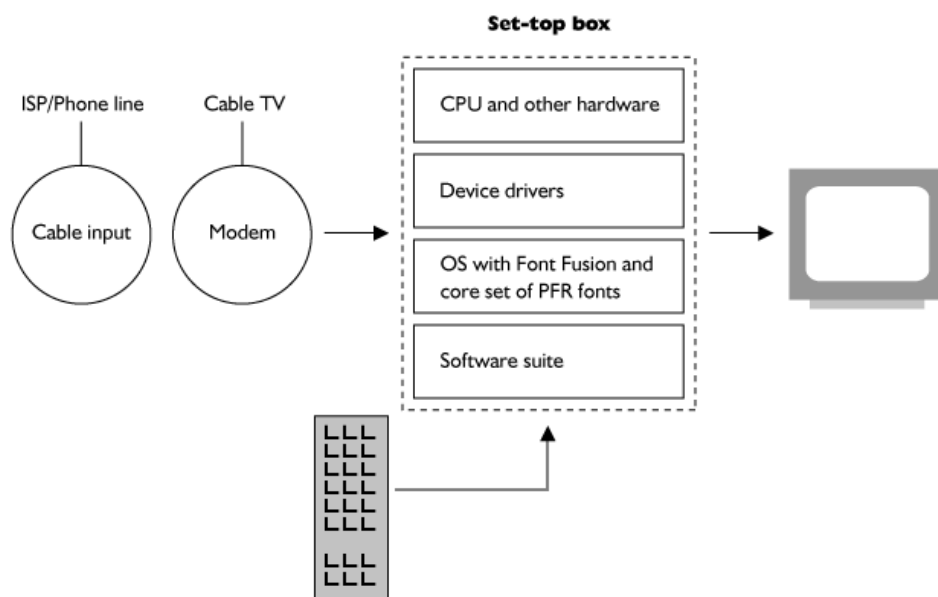


A sample of international characters recorded for a portable font resource (PFR) and regenerated on screen in an Internet browser. The zoomed-in box illustrates anti-aliasing capabilities. (Note the gray pixels at the edges of the character.)

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:

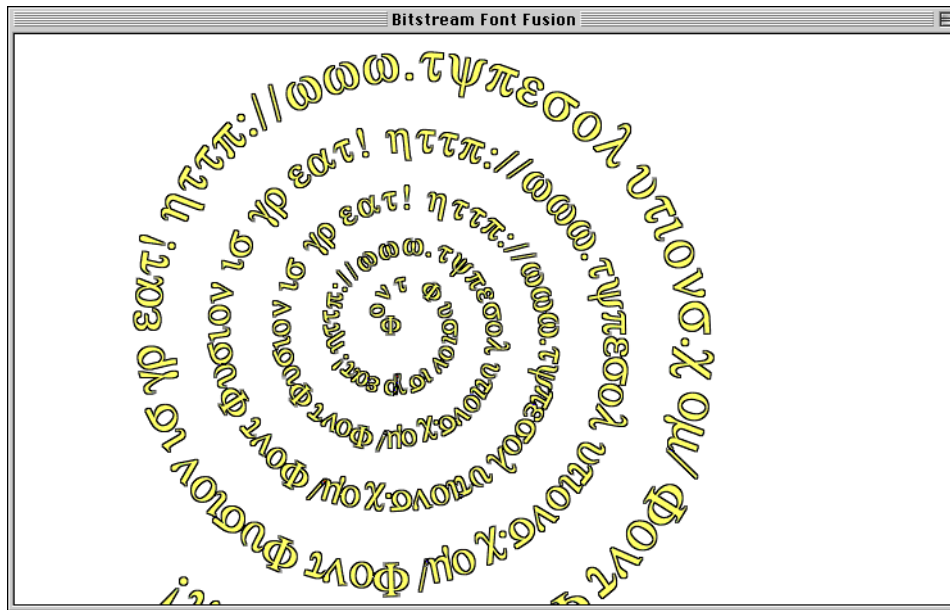
- anti-aliasing
- filtering and other post-processing
- subpixel positioning

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



The anti-aliasing technology developed by Bitstream for Font Fusion provides for layers of pixels of up 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, coloring, or even texture mapping to these bitmaps.

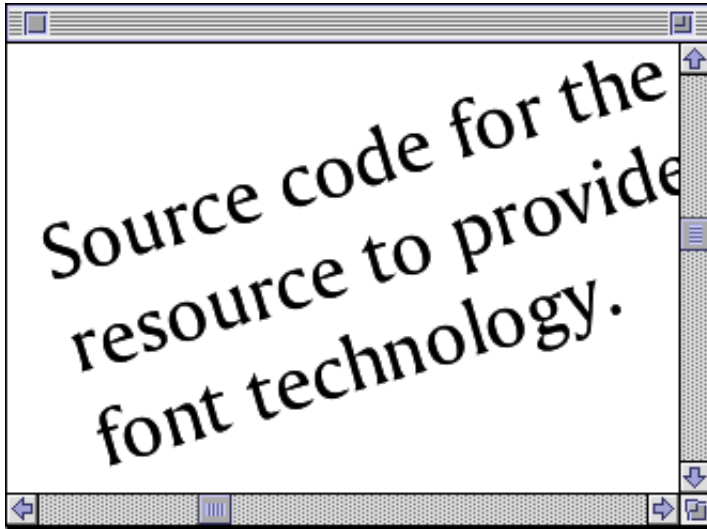


Using Font Fusion, you can take a single-color character and create a multiple-color character, complete with a border. The border is anti-aliased to the background, and the interior color is anti-aliased to the border.

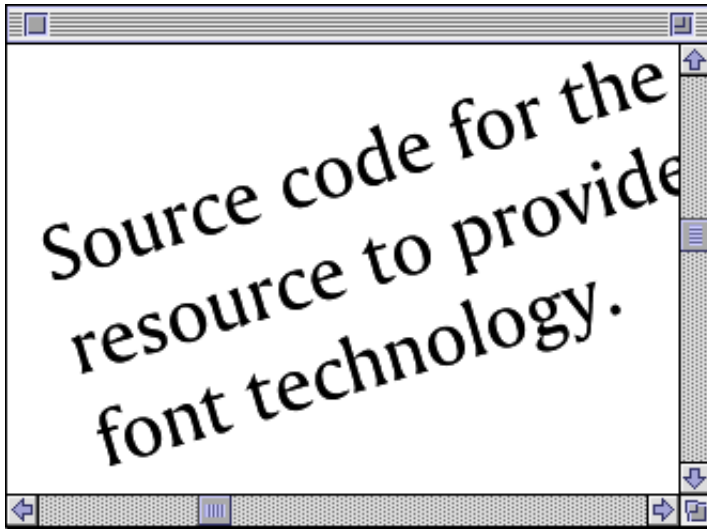
With its **subpixel positioning** technology, Font Fusion can accurately adjust and control the placement of characters. Previously, outlines of character shapes were simply fitted to character grids in an output device (such as a laser printer or a computer screen). If a pixel fell within the outline 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.



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.'



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 Director 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 a C++-like API to your application.

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

There are three types of fonts that your application deals with 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 scaleable 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. As the user of the Cache Manager, either finding the image in the cache, or creating it, storing it in the cache, and then exposing it to your application, 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

2

Topics

- Getting started
- Allocating memory
- Assert statements
- Compile-time options
- Errors
- Creating compact fonts

Getting Started

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.

This section discusses the following topics.

- What files should I look at first?
- What is the best way to get started?
- What are the functions in Font Fusion?
- What is the T2K scaler object?
- What is the `sfntClass` object?
- Should I use public APIs only?

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, and lets you build debuggable or non-debuggable versions of the code. There are many features included in this file. Turn off features you do not need in order to minimize the size of the Font Fusion Core.

What is the Best Way to Get Started?

First, configure `config.h`. Refer to the variables listed in “Compile-Time Options” on page 9 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 *Appendix B* 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 **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 **PlatformID** defines the computer platform which will use the font.
- The **T2K scaler** object represents an 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.

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 forces the changing of the TrueType character map
- **FF_PSNameToCharCode()**, which supports color combinations other than black and white.

Font Manager Functions:

- Functions for creating, configuring, and deleting the font manager.
- Function for installing fonts and getting font information.
- Functions for creating and using fonts.

Cache Manager Functions:

- Functions for creating and deleting the cache manager.
- Functions for working with the cache manager.

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 )ReallocateTaggedMemo-
ryNilAllowed(ptr, size, "t2k") */

```

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 optional features described in this section increase ROM and RAM needs, so only enable them if you are using them. These features can be enabled or disabled by modifying the `config.h` file.

Note: Do not use the option `ENABLE_T2KE`, reserved for future development.

| Options: Allocating Memory and Assert Statements | Description |
|--|---|
| <code>CLIENT_MALLOC</code> | See “Allocating Memory” on page 2-6. |
| <code>CLIENT_FREE</code> | See “Allocating Memory” on page 2-6. |
| <code>CLIENT_REALLOC</code> | See “Allocating Memory” on page 2-6. |
| <code>CLIENT_ASSERT</code> | See “Assert Statements” on page 2-8. |
| Options: Layout & Kerning | Description |
| <code>ENABLE_LINE_LAYOUT</code> | Enable if you plan to use <code>T2K_FindKernPairs()</code> or <code>T2K_MeasureTextInX()</code> . |
| <code>ENABLE_KERNING</code> | Enable if you plan to use <code>T2K_FindKernPairs()</code> or <code>T2K_MeasureTextInX()</code> . |
| <code>LAYOUT_CACHE_SIZE somesize</code> | This option speeds up <code>T2K_MeasureTextInX()</code> . Enable only if you also enable both <code>ENABLE_LINE_LAYOUT</code> and <code>ENABLE_KERNING</code> , and you plan to use <code>T2K_MeasureTextInX()</code> . Note that Font Fusion consumes eight times <code>somesize</code> for the cache. |
| Options: Algorithmic Styles | Description |
| <code>ALGORITHMIC_STYLES</code> | Use with <code>FF_New_sfntClass()</code> to enable algorithmic styles, such as emboldening and obliquing. |
| Options: Font Support | Description |
| <code>ENABLE_T1</code> | Enable if you need Type 1 font support. |
| <code>ROM_BASED_T1</code> | Disable this option to save memory if Type 1 fonts are not in ROM (to disable, comment the option out or remove it). This option is enabled by default. |
| <code>ENABLE_MAC_T1</code> | Enable if you have also defined <code>ENABLE_T1</code> and you need Macintosh Type 1 font support for the Macintosh platform. |
| <code>ENABLE_CFF</code> | Enable if you need Compact Font Format (CFF)/Type 2 font support. |

| | |
|-----------------------------------|--|
| ENABLE_ORION | Enable if you need to support entropy-encoded Font Fusion fonts (i.e., compact Kanji fonts, not stroke-based fonts). |
| ENABLE_T2KS | Enable if you need to support stroke-based fonts. |
| ENABLE_SPD | Enable if you need Bitstream Speedo™ font support. |
| ENABLE_PFR | Enable if you need Bitstream WebFont® or TrueDoc® PFR (portable font resource) font support. |
| ENABLE_SBIT | Enable if you need embedded bitmap font support. Currently, Font Fusion supports embedded bitmaps in TrueType, Native T2K, and TrueDoc PFR formats. |
| Options: “Seat-Belts” Mode | Description |
| 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. |
| Options: Anti-Aliasing | Description |
| ENABLE_GASP_TABLE_SUPPORT | Enable if you plan to use T2K_GaspifyTheCmds() . |
| Options: Hinting | Description |
| ENABLE_NATIVE_TT_HINTS | Enable if you need native TrueType hint support. |
| ENABLE_NATIVE_T1_HINTS | Enable if you need native Type 1 hint support. |
| 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. |
| ENABLE_AUTO_GRIDDING_CORE | <p>Enable only if you plan to use T2K_LCD_MODE_2, T2K_LCD_MODE_3, T2K_LCD_MODE_4, or T2K_TV_MODE_2.</p> <p>Note that if you use T2K_LCD_MODE_3 or T2K_LCD_MODE_4, you must also enable ENABLE_NATIVE_TT_HINTS (if using TrueType fonts) or ENABLE_NATIVE_T1_HINTS (if using Type 1 fonts), as applicable.</p> <p>Bitstream recommends using T2K_TV_MODE_2 to get the best-quality output on a TV or an LCD device.</p> <p>Bitstream recommends using T2K_LCD_MODE_4 to get the best-quality output on an LCD device.</p> |
| Options: Curve Conversion | Description |
| ENABLE_FF_CURVE_CONVERSION | Enable if you want to use the function T2K_ConvertGlyphSplineType() , documented in “void T2K_ConvertGlyphSplineType(” on page 44. |

| | |
|-------------------------------------|---|
| ENABLE_STRKCONV | Set the default to 36 lines. This improves the performance of the stroke font processor below 36 lines. At or near 36 lines, anomalies in quality appear because of the high-speed method that Font Fusion uses. |
| Options: For Scan Converter | Description |
| ENABLE_NON_ZERO_WINDING_RULE | <p>Enable if you want to use a non-zero winding rule in the scan converter instead of an even-odd fill. For example, the even-odd fill rule turns an area where two strokes overlap—which is rare—into white, but the non-zero winding rule keeps such areas black.</p> <p>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. Therefore, we recommend keeping this setting on.</p> |
| ENABLE_MORE_TT_COMPATIBILITY | <p>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.</p> <p>Only define this option if rendering TrueType fonts on black-and-white, low-resolution devices.</p> |
| MAKE_SC_ROWBYTES_A_4BYTE_MULTIPLE | Enable this option to get the scan converter to pad all bitmap rows to four-byte multiples. |
| REVERSE_SC_Y_ORDER | Enable this option to force the y-axis to go up, not down, in bitmaps. |
| Options: Non-RAM, -ROM Fonts | Description |
| 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. |
| Options: Output Modes | Description |
| ENABLE_LCD_OPTION | Enable if you need to use any of the LCD modes. |

| Options: Printer Fonts | Description |
|------------------------|--|
| ENABLE_PCL | <p>Enable this option to process scalable Intellifont® fonts that have been downloaded to a Hewlett-Packard® printer or printer emulation as encapsulated outlines.</p> <p>Bitstream's font reader for this format is included in the source modules pclread.c and pclread.h.</p> <p>When using ENABLE_PCL, you need to write a callback function, eo_get_char_data(). See <i>Appendix A</i> for instructions.</p> |
| ENABLE_PCLETTO | <p>Enable this option to process scalable TrueType fonts that have been downloaded to a Hewlett-Packard printer or printer emulation as encapsulated outlines.</p> <p>No additional font reader module is required.</p> <p>When using ENABLE_PCLETTO, you need to write a callback function, tt_get_char_data(). See <i>Appendix A</i> for instructions.</p> |

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

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.

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 *Appendix C* for a list of error codes and descriptions.

Creating Compact Fonts

Contact Bitstream if you want to use the Font Fusion format when creating your own fonts. This will make your fonts more compact.

Font Fusion Core API

3

- tsi Functions
- InputStream Functions
- sfntClass Functions
- PlatformID Functions
- T2K Functions
- Functions for Translating Font Data
- Additional Functions
- Sample Code

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)
```

Arguments

errCode is a pointer to the returned error code.

Description

tsi_NewMemhandler() creates an object to handle all memory allocation.

```
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 3-8, 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 void (*PF_READ_TO_RAM) ( void *id, uint8 *dest_ram,
unsigned long offset, long numBytes );
#endif

#ifdef JUST_AN_EXAMPLE_OF_PF_READ_TO_RAM
void 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 );
}
#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_AlgorithmStyleDescriptor *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`.

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), also known as Web fonts. |
| FONT_TYPE_SPD | Use for Bitstream Speedo fonts |

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->preferredPlatformID
= (ID))
#define Set_PlatformSpecificID( t, ID ) ((t)->font->preferred-
PlatformSpecificID = (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 3-13 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 3 (Microsoft). 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 3-13 for an example.

Description

Set_PlatformSpecificID() 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.

T2K Functions: Overview

- `NewT2K()`
- `DeleteT2K()`
- `T2K_NewTransformation()`
- `T2K_RenderGlyph()`
- `T2K_GaspifyTheCmds()`
- `T2K_GetBytesConsumed()`
- `T2K_ConvertGlyphSplineType()`
- `FF_T2K_Core_FilterReference()`
- `T2K_PurgeMemory()`
- `T2K_SetNameString()`
- `T2K_GetGlyphIndex()`
- `T2K_FontSbitsExists()`
- `T2K_FontSbitsAreEnabled()`
- `T2K_GlyphSbitsExists()`
- `T2K_TransformXFunits()`
- `T2K_TransformYFunits()`
- `T2K_FindKernPairs()`
- `T2K_MeasureTextInX()`
- `T2K_GetIdealLineWidth()`
- `T2K_LayoutString()`

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.

Obliquing Text (i.e., Making Algorithmic Italics)

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.

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`.

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

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.

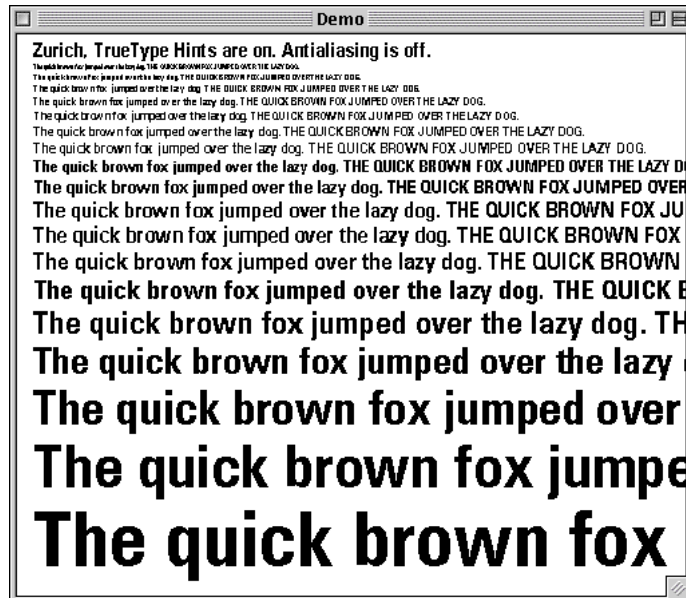
The Difference Between `T2K_TV_MODE_2` and `T2K_TV_MODE`

The `_2` modifier 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.”

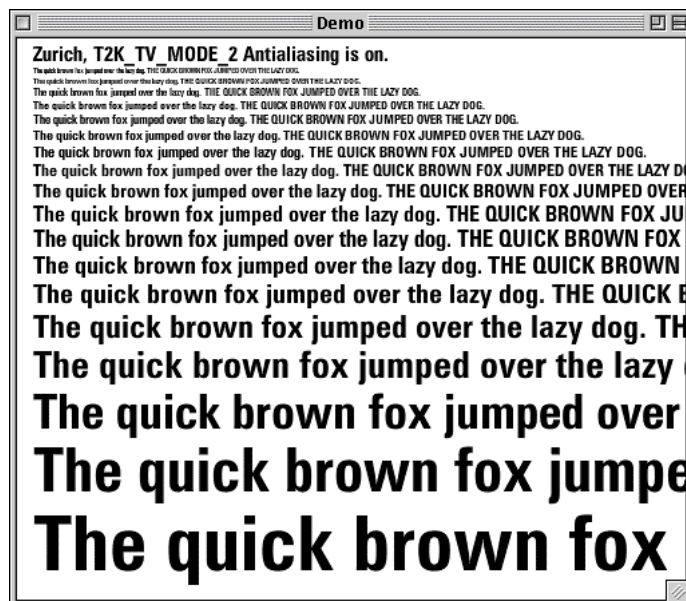
We recommend using `T2K_TV_MODE_2` to get the best-quality output on TV devices. For instance, compare the figures below, which show no anti-aliasing and `T2K_TV_MODE_2`.

We recommend using `T2K_LCD_MODE_4` to get the best-quality output on LCD devices.

For optimal quality, the regular TV modes already make characters left-to-right symmetrical.



No anti-aliasing.



T2K TV MODE 2, anti-aliasing is on.

LCD Modes

As with TV modes, the `_2` modifier 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.”

The difference between `T2K_LCD_MODE_4` and `T2K_LCD_MODE_3` is that `T2K_LCD_MODE_4` uses native hints, while `T2K_LCD_MODE_3` relies on hints automatically generated at run-time. This makes `T2K_LCD_MODE_4` faster than `T2K_LCD_MODE_3`.

We recommend using `T2K_LCD_MODE_4` to get the best-quality output on LCD devices.

For optimal quality, the regular LCD modes already make characters left-to-right symmetrical.

Driving Color LCD Displays

If you are driving color LCD displays, you need to use Font Fusion with the Gibson LCD option, `T2K_WriteToGrayPixels()` in `t2kextra.c`. See the procedure below for details.

► To set up a color LCD display:

- 1 When you call `T2K_NewTransformation()`, set the `xRes` to three times the `yRes`. This means that, if you look closely at the screen, there are three times as many colored pixels in the x direction as there are in the y direction, i.e., a non-square aspect ratio, where x resolution is three times more than the y.
- 2 Set `T2K_LCD_MODE_2` in the `cmd` parameter to `T2K_RenderGlyph()`.
- 3 Right after invoking `T2K_RenderGlyph()`, invoke the Gibson LCD option by calling `T2K_WriteToGrayPixels()`.
- 4 You now have a bitmap with three times as many pixels for x as you have in y. When you draw the bitmap, you need to take into account that the bitmap contains values for the colored pixels, so you need to address them directly, at least conceptually. Note that the bitmap may align with the colors in three different ways depending on the pen position.

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 actually reserved 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 `T2K_CreateBorderedCharacter()`, in `t2kextra.c`, right after you invoke `T2K_RenderGlyph()`.

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
- Glyph-specific metrics

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.

Glyph-Specific Metrics

Here is a lowercase letter “g” showing glyph-specific metrics.

```
t2k->vert_fTop26Dot6 (26.6) This is the vertical line below, from base
line to the top of the bitmap.
<----> t2k->vert_fLeft26Dot6 (26.6)
      ^          *****          ^          t2k->height (integer number of
      |          *                *          |          scanlines) (32.0)
      |          *                *          |
      |          *                *          |
      |          *                *          |
      |          *                *          |
-----0 v-----*****-----|-----0-----
      ^          *                ^          |
pen pos- |          *                | next pen position.
ition    |          *                |
          *                *          |
          *****          v
          <----->          t2k->width (integer number of
                               pixels) (32.0)
          <----->
          t2k->[xAdvanceWidth16Dot16,yAdvanceWidth16Dot16] (16.16)
```

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;
```

The Filter Function

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. The Core creates bitmaps in either 1-bit or 8-bit depth (alpha values range from 0 to 126). These fundamental images can experience post-processing in the form of Gaussian fuzz-filtering, smearing, colorizing, or even texture mapping in a filter function.

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.

If You’re Using the Font Fusion Core without the Cache Manager

If using the Font Fusion Core without the Cache Manager, you plug in the filter function through the `setter` macro defined in `t2k.h`.

If You’re Using the Cache Manager

For users of the Font Fusion Cache Manager, you plug in the filter function through the Cache Manager interface, **FF_CM_SetFilter()**.

See “void FF_CM_SetFilter(” on page 5-15 for more information about this function.

How to Write a Filter Function

► There are several basic rules for writing a filter function plug in:

- 1 Allocate destination memory from the right place.
- 2 Find the source image in the T2K class.
- 3 Leave the destination image in the right place in the T2K class.
- 4 Update glyph metrics that the filter affects.
- 5 Dispose of the source image memory properly.
- 6 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 |
|--------------------------------|---|
| <code>internal_baseAddr</code> | true |
| <code>baseAddr</code> | set and valid, allocated by the Core and not by the Cache Manager |
| <code>internal_baseARGB</code> | false |
| <code>baseARGB</code> | 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

When you are finished with a filter, you simply set the filter function pointer to NULL. If you are using the cache, turn it off if using this kind of call:

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

We suggest using a filter tag code (second parameter) of 0, implying no filter, but that's really up to you.

If you are not using the Cache Manager, turn it off if using this kind of call:

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

Both these examples also set the parameter's pointer to NULL for the sake of neatness. You can switch to another filter simply by setting a new function pointer/parameter's-block pointer combination.

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.

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 or native T2K font contains any “sbits.”

T2K_GlyphSbitsExists() is a query method to check if a particular glyph exists in sb1t 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.

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.

T2K Functions

```
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 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 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_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.

BIT FLAGS for setting the greyScaleLevel argument

```
#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
grayscale */
#define GREY_SCALE_BITMAP_HIGHER_QUALITY 4
#define GREY_SCALE_BITMAP_EXTREME_QUALITY 5 /* Slowest */
/* When doing grayscale, the scan converter returns values in
the range T2K_WHITE_VALUE to 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, where
   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 */

```

BIT FLAGS for setting the cmd argument

```

/* Use the macro names below, not the actual hex values. */
#define T2K_GRID_FIT0x0001
#define T2K_SCAN_CONVERT0x0002
#define T2K_RETURN_OUTLINES 0x0004
#define T2K_CODE_IS_GINDEX0x0008 /* Otherwise, it is the character
code */
#define T2K_USE_FRAC_PEN0x0010
#define T2K_SKIP_SCAN_BM0x0020 /* Everything works as normal;
but
   we do _not_ generate the actual bitmap */
#define T2K_TV_MODE0x0040 /* Ideal for TV if you use integer
metrics and grayscale (Turn off T2K_GRID_FIT) */
#define T2K_NAT_GRID_FIT0x0080 /* Enables native TrueType hint
and
   gridding support */
#define T2K_LCD_MODE0x0100 /* Ideal for LCD screens */
#define T2K_Y_ALIGN0x0200 /* Ideal with T2K_LCD_MODE and
T2K_TV_MODE */

```

```

/* The first TV mode is T2K_TV_MODE. The first LCD mode is
T2K_LCD_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 )

```

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.

```
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 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.

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_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()`**.

`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);`)

`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 or native T2K font contains any “sbits” (embedded bitmaps).

```
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).

```
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 or native T2K 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_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 $0.07 \times 1000 = 70$ 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 $0.07 \times 1000 = 70$ 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.

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

```

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_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.

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.

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.

Additional Functions

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

`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 those setter 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 now 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 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.
- 4 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()**.
- 5 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.

**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.

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
- Functions for Creating, Configuring, and Deleting the Font Manager
- Function for Installing Fonts and Getting Font Information
- Functions for Creating and Using Fonts
- Sample Code

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 3-4 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 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_AlgorithmDescriptor *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
```

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 number 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 number in 16.16 format.

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 number in 16.16 format. 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: error set by `setjmp()`.

See also

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)
{
#ifdef 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 with it, empty it, and set filter parameters if you want to apply special effects to characters (drop shadows, cross-hatch fills, etc.).

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)
```

Arguments

`long` is the size of the cache, including `FF_CM_Class`.

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

Description

`FF_CM_New()` is a pointer to a function that creates a new instance of the Font Fusion Cache Manager.

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

```
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: Overview

- `FF_CM_RenderGlyph ()`
- `FF_CM_GlyphInCache ()`
- `FF_CM_Flush ()`
- `FF_CM_SetFilter ()`

Filter Function

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. The Core creates bitmaps in either 1-bit or 8-bit depth (alpha values range from 0 to 126). These fundamental images can experience post-processing in the form of Gaussian fuzz-filtering, smearing, colorizing, or even texture mapping in a filter function.

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 probably 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.

If You’re Using the Font Fusion Core without the Cache Manager

If using the Font Fusion Core without the Cache Manager, you plug in the filter function through the `setter` macro defined in `t2k.h`.

See **FF_SET_T2K_Core_FilterReference ()** in Chapter 3 for more information about this function.

If You're Using the Cache Manager

For users of the Font Fusion Cache Manager, you plug in the filter function through the Cache Manager interface, **FF_CM_SetFilter()**.

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.

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.

► How to Write a Filter Function

There are several basic rules for writing a filter function plug in:

- 1 Allocate destination memory from the right place.
- 2 Find the source image in the T2K class.
- 3 Leave the destination image in the right place in the T2K class.
- 4 Update glyph metrics that the filter affects.
- 5 Dispose of the source image memory properly.
- 6 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.

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 a 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 describes the state of the T2K class after this filter function completes:

| T2K class | State |
|--------------------------------|---|
| <code>internal_baseAddr</code> | false |
| <code>baseAddr</code> | NULL, and disposed of by this filter function |
| <code>internal_baseARGB</code> | true if from <code>tsi_AllocMem()</code> , false if from cache memory |
| <code>baseARGB</code> | set and valid |

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

Removing or Changing Filters

When you are finished with a filter, you simply set the filter function pointer to NULL. If you are using the cache, turn it off if using this kind of call:

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

We suggest using a filter tag code (second parameter) of 0, implying no filter, but that's really up to you.

If you are not using the Cache Manager, turn it off if using this kind of call:

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

Both these examples also set the parameter's pointer to `NULL` for the sake of neatness. You can switch to another filter simply by setting a new function pointer/parameter's-block pointer combination.

Functions for Working with the Cache Manager

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 if you are also using fractional (subpixel) character positioning.

`greyScaleLevel` describes the level of anti-aliasing you want to apply. See “BIT FLAGS for setting the `greyScaleLevel` argument” on page 3-41 for more information.

`cmd` describes to Font Fusion what to do with various bitflags. See “BIT FLAGS for setting the `cmd` argument” on page 3-42 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.

This function returns a non-zero value if everything is successful.

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 “BIT FLAGS for setting the `greyScaleLevel` argument” on page 3-41 for more information.

`cmd` describes to Font Fusion what to do with various `bitflags`. See “BIT FLAGS for setting the `cmd` argument” on page 3-42 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.

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.

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 = "AABCCabcde fghijklmnopqrsabcde fghijklmnopqrsabcde f-
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");
printf("*****      **      **\n");
printf("    **      **      **      **\n");
printf("    **          **      **\n");
printf("    **          **      **\n");
printf("    **          **      **\n");
printf("    **          **      **\n");
printf("    **          **      **\n");
printf("    **          **      **\n");
printf("    **          **      **\n");
printf("    **          **      **\n");
printf("    **          **      **\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, 0L, SEEK_END ); assert( errCode == 0 );
length= (unsigned long)ftell( fpID ); assert( ferror(fpID) == 0 );
errCode= fseek( fpID, 0L, 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;
}
}

```

Font Fusion API for Printer Developers



Topics

- General Information
- Compile-Time Options
- Font Types
- Callback Functions

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 | <p>Enable this option to process scalable Intellifont fonts that have been downloaded to a Hewlett-Packard printer or printer emulation as encapsulated outlines.</p> <p>We have supplied a font reader for this format with two new source modules, pclread.c and pclread.h.</p> <p>When using ENABLE_PCL, you need to write a callback function, eo_get_char_data(), documented in this appendix.</p> |
| ENABLE_PCLETTO | <p>Enable this option to process TrueType fonts that have been downloaded to a Hewlett-Packard printer or printer emulation as encapsulated outlines.</p> <p>No additional font reader module is required.</p> <p>When using ENABLE_PCLETTO, you need to write a callback function, tt_get_char_data(), documented in this appendix.</p> |

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_PCLETT0`.

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.

Text flows

B

- Overview
- Font Fusion Core
- Font Manager
- Cache Manager
- Font Manager and Cache Manager

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

C

- 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 |
| 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 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 |

Cache Manager Error Codes

The Cache Manager has no specific error codes.

Index

Numerics

2K_AlgorithmDescription 3-11

A

AdvanceWidth16Dot16 3-54
algorithmic italics 3-18
ALGORITHMIC_STYLES 2-9, 3-10, 4-17
allocation 2-6
anti-aliasing 1-6, 3-21, 3-41, 3-43, 5-12, 5-14
applications supported 1-2, A-3
architecture 1-2, A-3
ARGB 3-28
Assert 2-8
assert statements 2-8
auto-hinting 2-10, 3-20
auxiliary metrics files 4-6, 4-9
aw A-7
awSet A-7

B

baseAddr 3-25, 3-26, 3-35, 5-8
baseARGB 3-25, 3-26, 3-28, 3-35, 5-8
baseLength 3-51
baseSet 3-51
Bb 3-60
Beziér curves 3-28, 3-31
Bf 3-60
BIT_FLAGS 3-41, 3-42
bitflags 3-41, 3-59, 5-12, 5-14
bitmap data 3-25
bitmap output 3-19
BitmapFilter 5-15
bitmaps 3-48
BLACK_AND_WHITE_BITMAP 3-20, 3-41
bordered character 3-28
b-spline curve 3-28
bufSize 3-58

C

Cache Manager 1-12
cArr array 3-54
cCode A-5, A-6
CFF 2-9
char T2K_FontSbitsAreEnabled() 3-48
char T2K_FontSbitsExists() 3-47
char_code 5-12
character map 3-15
characters, rendering 3-29
charCode 3-47, 3-51, 3-54, A-5
CJK fonts 4-3
CLIENT_ASSERT 2-8
CLIENT_FREE 2-6

CLIENT_MALLOC 2-6
CLIENT_REALLOC 2-6
cmap 3-15, 4-10
cmd 3-23, 3-24, 3-41, 4-21, 5-12, A-5, A-6
cmd argument bit flags 3-42
code 3-41, 4-21
color LCD display 3-23
colored, bordered character 3-28
colorizing 1-7, 3-33, 5-6
compile-time options 2-9
computer monitor 3-24
config.h 2-2, 4-3
ep 3-27
sp 3-27
contourCount 3-27
Core 1-9
Corner 3-54
curveTypeOut 3-44

D

data 3-6
dataSize A-5, A-6
Delete_InputStream 3-4
Delete_InputStream() 3-9
DeleteT2K() 3-39
dest_ram 3-8
devices supported 1-5
Dimension 3-54
doSetUpNow 3-40
dynamic fonts 1-10, 1-11

E

embedded bitmaps 3-48
ENABLE_AUTO_GRIDDDING 2-10
ENABLE_AUTO_GRIDDDING_CORE 2-10
ENABLE_CFF 2-9
ENABLE_FF_CURVE_CONVERSION 2-10
ENABLE_GASP_TABLE_SUPPORT 2-10
ENABLE_KERNING 2-9, 3-51
ENABLE_LCD_OPTION 2-11
ENABLE_LINE_LAYOUT 2-9, 3-52
ENABLE_MAC_T1 2-9
ENABLE_MORE_TT_COMPATIBILITY 2-11
ENABLE_NATIVE_T1_HINTS 2-10
ENABLE_NATIVE_TT_HINTS 2-10, 3-24
ENABLE_NON_RAM_STREAM 2-11, 3-8
ENABLE_NON_ZERO_WINDING_RULE 2-11
ENABLE_ORION 2-10
ENABLE_PCL 2-12, A-3
ENABLE_PCLETTO 2-12, A-3
ENABLE_PFR 2-10
ENABLE_SBIT 2-10
ENABLE_SPEEDO 2-10

ENABLE_STRKCONV 2-11
 ENABLE_T1 2-9
 ENABLE_T2KE 2-9
 ENABLE_T2KS 2-10
 enableSbits 3-40
 enumTypefaceCallBack() 4-12, 4-14
 enumTypefaceCallback() 4-14
 eo_get_char_data() 2-12, A-3, A-5
 errCode 3-6, 3-7, 3-9, 3-11, 3-12, 3-39, 3-40, 3-41, 3-44, 3-46, 3-48, 3-56, 4-8, 4-9, 4-13, 4-16, 4-19, 4-20, 4-21, 5-4, 5-12, 5-14, B-2
 error B-2
 Errors 2-13
 even-odd fill 2-11, 3-26
 ExtractPureT1FromMacPOSTResources() 3-56
 ExtractPureT1FromPCType1() 3-56

F

F26Dot6 3-27
 faceName16 4-14
 faceName8 4-14
 FF_CM_Delete() 5-4
 FF_CM_Flush() 5-14
 FF_CM_New() 3-45, 5-4
 FF_CM_RenderGlyph() 1-11, 1-12, 4-5, 5-3, 5-12, 5-13
 FF_CM_SetFilter() 3-33, 3-37, 5-7, 5-10, 5-15
 FF_Delete_sfntClass() 3-11, 3-12
 ff_fm.h 4-3
 FF_FM_AddTypefaceStream() 4-6, 4-9, 4-12
 FF_FM_CreateFont() 4-6, 4-16
 FF_FM_Delete() 4-13
 FF_FM_DeleteFont() 4-19, 4-20
 FF_FM_EnumTypefaces() 4-4, 4-12, 4-14
 FF_FM_MAX_DYNAMIC_FONTS 4-6
 FF_FM_New() 4-8
 FF_FM_RenderGlyph() 4-21
 FF_FM_SelectFont() 4-10, 4-19
 FF_FM_SetLanguageID() 4-12
 FF_FM_SetNameID() 4-12
 FF_FM_SetPlatformID() 4-10
 FF_FM_SetPlatformSpecificID() 4-10
 FF_FM_SetXYResolution() 4-19
 FF_FontTypeFromStream() 3-11
 FF_ForceCMAPChange() 3-59
 FF_GetTTTablePointer() 3-58
 FF_GlyphExists() 3-59
 FF_New_sfntClass() 3-11
 FF_NewColorTable() 3-60
 FF_PSNameToCharCode() 3-62
 FF_SET_T2K_Core_FilterReference() 5-6
 FF_Set_T2K_Core_FilterReference() 3-37, 3-41, 3-44, 3-45, 5-10
 FF_T2K_FilterFuncPtr 3-34, 3-45, 5-7, 5-15
 Filter Function 5-6
 filter function 1-7, 3-33, 5-6
 filterParamsPtr 5-15
 FilterTag 5-7, 5-15
 flushCache 4-6, 4-16
 font 3-39
 font formats supported 1-3, A-3

font fragments 1-10, 4-3
 Font Fusion Core 1-9
 Font Manager 1-10
 font metrics 4-9
 font units 3-49, 3-50
 font code 5-12
 FONT_TYPE_1 3-11
 FONT_TYPE_2 3-11
 FONT_TYPE_PCL A-4
 FONT_TYPE_PCLETTO A-4
 FONT_TYPE_PFR 3-11
 FONT_TYPE_TT_OR_T2K 3-11
 fontCode 4-19, 4-20, 4-21
 fontNum 3-11
 fontType 3-11
 fractional pixel positioning 3-21
 funcptr 3-45
 FUnits 3-49, 3-50

G

Gaussian fuzz-filtering 1-7, 3-33, 5-6
 Gb 3-60
 Gf 3-60
 Gibson LCD option 3-23
 gIndex A-5, A-6
 glyph 3-26
 glyph metrics 3-36
 glyph without pixels 3-26
 GlyphClass 3-27
 glyphIndex 3-48, 3-54
 glyph-specific metrics 3-32
 grayscale 3-20
 grayscale output 3-20
 GREY_SCALE_BITMAP_EXTREME_QUALITY 3-42
 GREY_SCALE_BITMAP_HIGH_QUALITY 3-20, 3-42
 GREY_SCALE_BITMAP_HIGHER_QUALITY 3-42
 GREY_SCALE_BITMAP_LOW_QUALITY 3-42
 GREY_SCALE_BITMAP_MEDIUM_QUALITY 3-42
 greyScaleLevel 3-20, 3-41, 4-21, 5-12
 greyScaleLevel argument bit flags 3-41
 gridding 2-10, 3-20

H

high-quality display device 3-24
 hinting 3-24
 hints 2-10
 HPXL_MetricsInfo_t A-6
 HPXL_metricsInfo_t A-7

I

ID 3-15
 id 3-8
 in1 3-11
 in2 3-11
 index 4-16
 input streams 1-10, 4-6
 InputStream 2-6, 3-4, 3-8
 int FF_CM_GlyphInCache() 5-13
 integer metrics 3-21

Intellifon 2-12
 Intellifont A-3
 interlaced TV device 3-21
 internal_baseARGB 3-35, 5-8

K

Kanji fonts 2-10
 kerning 2-9, 3-63, 4-6, 4-9
 kerning pairs 3-51
 kerning value 3-49, 3-50

L

languageID 3-46, 4-12
 Latin fonts 3-21
 LAYOUT_CACHE_SIZE 2-9
 LCD devices 3-21, 3-23
 LCD display 3-23
 LCD modes 2-11
 length 3-6, 3-7, 3-56
 level 3-46
 line layout 2-9
 linear text layout 3-63
 LinearAdvanceWidth16Dot16 3-54
 logical fonts 1-10, 4-6
 low-quality display device 3-24
 lsb A-7
 lsbSet A-7

M

MAKE_SC_ROWBYTES_A_4BYTE_MULTIPLE 2-11
 Make2ndDegreeEdgeList() 3-28, 3-31
 Make3rdDegreeEdgeList() 3-31
 MAX_PURGE_LEVEL 2 3-46
 mem 3-6, 3-7, 3-11, 3-39, 3-56
 Memhandler 2-3, 3-2, 3-4
 memory handle 2-3, 3-2
 merging fonts dynamically 1-10
 metrics information 4-6, 4-9
 metricsInfo A-6, A-7
 monochrome output 3-20, 3-24
 multilingual capabilities 1-4
 MyDrawCharExample 3-29
 MyDrawCharExample() 3-31

N

nameID 3-46, 4-12
 native TrueType hinting 3-24
 New_InputStream() 3-6
 New_InputStream3 3-6
 New_NonRamInputStream() 3-7
 NewT2K() 3-39
 nonRamID 3-7
 non-zero winding rule 2-11, 3-26
 numBytes 3-8
 numChars 3-52

O

obliquing 3-18
 offset 3-8
 onCurve 3-27
 operating systems supported 1-2
 outline output 3-19
 outline spline data 3-26
 outline winding direction 3-25

P

pairCountPtr 3-51
 parabolas 3-31
 params 3-45
 pCharData A-5, A-6
 PCLeTTo fonts A-7
 pclread.c A-3
 pclread.h A-3
 PF_READ_TO_RAM() 3-8
 PFB 3-56
 pFM 4-9, 4-10, 4-12, 4-13, 4-14, 4-16, 4-19, 4-20, 4-21
 PFR 2-10
 physical fonts 1-10, 4-6
 PlatformID 3-13
 platformID 4-10
 platformSpecificID 4-10
 ppTbl 3-58
 pScaler 4-21
 PSName 3-62

R

Rb 3-60
 readFunc 3-7, 3-8
 refNum 3-56
 RenderGlyph() 1-10, 1-11, 1-12, 3-33, 4-5, 5-2, 5-3, 5-6
 rendering characters and strings 3-29
 REVERSE_SC_Y_ORDER 2-11
 Rf 3-60
 ROM_BASED_T1 2-9
 rotating text 1-7
 run-time hinting 3-24, 3-25

S

sample code 4-23
 SBIT 2-10
 Sbits 3-38
 sbits 3-48
 scan converter 3-26
 scan converters 2-11
 second-degree b-spline curve 3-28
 Set_PlatformID() 3-15, 3-47
 Set_PlatformSpecificID() 3-15, 3-47
 sfntClass 3-10, 3-11, 4-4
 sizeofCache 5-4
 smearing 1-7, 3-33, 5-6
 SPEEDO 2-10
 spline curve 3-28
 spline data 3-26
 src 3-56

StreamA 4-9
 StreamB 4-9
 streams 4-6, 4-9
 strikes 4-6
 strings, rendering 3-29
 styling 3-10, 3-11, 4-16
 subpixel positioning 1-7

T

t 3-3, 3-9, 3-12, 3-39, 3-40, 3-41, 3-44, 3-46, 3-47, 3-49,
 3-50, 3-51, 3-52, 3-53
 T2K 1-2
 t2k 3-45
 T2K class 3-34, 4-4
 T2K scaler 4-10
 T2K scaler object 3-17
 t2k.h 2-2, 3-38, 4-3, 5-6
 T2K_AlgStyleDescriptor 3-10, 4-16
 T2K_BLACK_VALUE 3-20, 3-42
 T2K_BorderfilterParams 3-34, 5-8
 T2K_CODE_IS_GINDEX 3-41, A-5, A-6
 T2K_ConvertGlyphSplineType() 3-44
 T2K_CreateBorderedCharacter() 3-28, 3-34, 3-35, 5-7,
 5-9
 T2K_FindKernPairs() 2-9, 2-10, 3-51
 T2K_GetGlyphIndex() 3-47, 3-48
 T2K_GetIdealLineWidth() 3-38, 3-53, 3-55
 T2K_GetNumGlyphsInFont() 3-47
 T2K_GlyphSbitsExists() 3-48
 T2K_GRID_FIT 3-20, 3-24
 T2K_KernPair() 3-51
 T2K_LayoutString() 3-38, 3-53, 3-55
 T2K_LCD_MODE_2 2-10, 3-23
 T2K_LCD_MODE_3 2-10
 T2K_MeasureTextInX() 2-9, 3-52
 T2K_NAT_GRID_FIT 3-20, 3-24
 T2K_NewTransformation() 3-17, 3-23, 3-40
 T2K_PurgeMemory() 3-18, 3-37, 3-40, 3-42, 3-46
 T2K_RenderGlyph() 3-18, 3-20, 3-23, 3-25, 3-29, 3-40,
 3-41, 3-46, 4-5, 5-3
 T2K_RETURN_OUTLINES 3-26, 3-28
 T2K_SetNameString() 3-14, 3-46
 T2K_TRANS_MATRIX 3-40, 4-16
 T2K_TransformXUnits() 3-49
 T2K_TransformYUnits() 3-50
 T2K_TV_MODE 3-20, 3-21
 T2K_TV_MODE_2 2-10, 3-21, 3-23
 T2K_WHITE_VALUE 3-20, 3-42
 T2K_WHITE_VALUE to T2K_BLACK_VALUE 3-42
 T2K_WriteToGrayPixels() 3-23
 T2KCharInfo 3-53, 3-54, 3-55
 t2kextra.c 3-23, 3-28, 3-34
 T2KLayout 3-53, 3-55
 T2KS 2-10
 t2ksc.c 3-28, 3-31

t2kScaler 3-15
 tag 3-58
 text 3-52
 texture mapping 1-7, 3-33, 5-6
 theCache 5-4, 5-12, 5-14, 5-15
 theScaler 5-12
 third-degree Beziér curves 3-28, 3-31
 trans 3-40, 4-16
 transformation matrix 3-40, 4-16
 TrueDoc 1-2, 2-10
 TrueType fonts 2-12, A-3
 TrueType hinting 3-24
 TrueType hints 2-10
 tsb A-7
 tsbSet A-7
 tsi_AllocMem 3-35, 5-8
 tsi_DeAllocMem() 3-51
 tsi_DeleteMemhandler 2-3, 3-2, 3-4
 tsi_DeleteMemhandler() 3-3
 tsi_FastAllocMem() 3-36, 5-10
 tsi_FastDeAllocN() 3-36, 5-10
 tsi_NewMemhandler 2-3, 3-2, 3-4
 tsi_NewMemhandler() 3-3
 tsiMemObject 2-3, 2-7, 3-2, 3-4
 tt_get_char_data() 2-12, A-3, A-6
 TV devices 3-21, 3-23
 TV monitor 3-24
 TV_MODE 2-10
 Type 1 2-9
 Type 1 hints 2-10
 Type 2 2-9

U

USE_NON_ZERO_WINDING_RULE 3-26
 USE_SEAT_BELTS 2-10

W

WebFont 2-10
 white-space characters 3-26
 winding direction 3-25
 winding rule 2-11

X

xFracPenDelta 3-41, 4-21, 5-12
 xKernValuesInFUnits 3-52
 xRes 3-23, 3-40, 4-19
 XTextWidth() 3-38
 xValueInFUnits 3-49

Y

yFracPenDelta 3-41, 4-21, 5-12
 yRes 3-23, 3-40, 4-19
 yValueInFUnits 3-50