Libraries

COLLABORATORS

	TITLE : Libraries	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 28, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 29 Graphics Library and Text

On the Amiga, rendering text is similar to rendering lines and shapes.
The Amiga graphics library provides text functions based around the
RastPort structure, which makes it easy to intermix graphics and text.

1.2 29 Graphics Library and Text / About Amiga Fonts

In order to render text, the Amiga needs to have a graphical
representation for each symbol or text character. These individual images
are known as glyphs. The Amiga gets each glyph from a font in the system
font list. At present, the fonts in the system list contain a bitmap of a
specific point size for all the characters and symbols of the font.

Fonts are broken up into different font families. For example, the
Amiga's Topaz is a font family. Each font family shares a basic look but
can have a variety of styles and point sizes.

The style of a font refers to a minor alteration in the way the plain
version of the font's characters are rendered. Currently, the Amiga
supports three font styles: bold, italics and underline (the font's style
may also be considered plain when it does not have any of these styles).
Although these styles can be inherent to a font, they are normally added
algorithmically as text is rendered.

On the Amiga, the point size of a font normally refers to the height of

the font in pixels. For example, Topaz-8 is 8 pixels high. Because the size of Amiga pixels varies between display modes, the appearance of a font will also vary between display modes. Future versions of the Amiga OS may measure font size in other units. For example, the standard point in the PostScript page description language normally refers to a point as being a square dot that is 1/72 of an inch on a side. Using a standard measuring unit such as the PostScript point makes it possible to create a WYSIWYG (What You See Is What You Get) display that exactly matches printer or other device output.

When the Amiga first starts up, the only fonts in the system font list are Topaz-8 and Topaz-9, both of which are in ROM. Any other fonts must be loaded from disk or generated somehow. In Amiga operating systems prior to Release 2, additional fonts have to be loaded from disk (usually from the FONTS: directory) using the diskfont.library. For each font size of each font family there is a corresponding bitmap file on disk. If there is no bitmap on disk or in ROM for a specific font size, that font size is not available (if the operating system is 1.3 or earlier).

System Fonts In Release 2

1.3 29 / About Amiga Fonts / System Fonts In Release 2

Under Release 2 and later versions of the OS, the system has additional font sources at its disposal. If an application asks the diskfont.library to load a font of a size that has no corresponding bitmap on disk, the library can generate that size font. If diskfont.library can find a smaller or larger version of the font requested, it can scale that font's bitmap to the size needed. Of course, because the library is scaling a bitmap, the quality of the bitmap can degenerate in the scaling process.

A more significant improvement to the diskfont.library is that contains AGFA's Intellifont® engine. As of Release 2.04 (V37) the diskfont.library can utilize AGFA Compugraphic font outlines. The Compugraphic fonts (CG fonts) are mathematical outlines that describe what the font's characters look like. The advantage of the outline fonts is that they can be scaled to any point size without the loss of resolution inherent in bitmap scaling. From the programmers point of view, no extra information is necessary for using the CG fonts, the diskfont.library takes care of all the scaling. Future releases of the OS may bring finer control over the font scaling engine which will allow an application to rotate and shear glyphs.

1.4 29 Graphics Library and Text / The Text Function

Amiga text rendering is centered around the graphics.library function Text(), which renders text into a rastport:

```
    void Text( struct RastPort *myrp, STRPTR mystring, ULONG count );
```

where myrp is a pointer to the target rastport, mystring is the string to render, and count is the number of characters of mystring to render.

Text() renders at the current rastport position and it takes care of moving the rastport's current X position as it renders each letter. Text() only renders text horizontally, so repositioning the rastport's Y position (for example, for a new line) is the responsibility of the application. This is covered in more detail later in this chapter.

Like the other rastport based graphics primitives, most of the text rendering attributes are specified within the RastPort structure itself. The current position, the color of the text, and even the font itself are all specified in the RastPort structure.

1.5 29 / The Text Function / Choosing the Font

The graphics.library function SetFont() changes the rastport's current font:

```
    void SetFont( struct RastPort *myrp, struct TextFont *mytf );
```

The parameter mytf is a pointer to an open, valid TextFont structure. The system uses the TextFont structure to keep track of fonts (The TextFont structure is discussed in detail later in this chapter). The OpenFont() (from graphics.library) and OpenDiskFont() (from diskfont.library) functions both return a pointer to a valid TextFont structure. The OpenFont() function will only open fonts that have already been loaded and are currently in the system list. Normally applications use the OpenDiskFont() call instead of OpenFont() because OpenDiskFont() can load and open fonts from disk as well as open those that are already in the system list.

Here are prototypes for these functions:

```
    struct TextFont *OpenDiskFont( struct TextAttr *mytextAttr );
    struct TextFont *OpenFont( struct TextAttr *mytextAttr );
```

The mytextAttr argument points to a TextAttr structure that describes the requested font. The TextAttr structure (from <graphics/text.h>) looks like this:

```
    struct TextAttr {
        STRPTR  ta_Name;            /* name of the font */
        UWORD   ta_YSize;           /* height of the font */
        UBYTE   ta_Style;           /* intrinsic font style */
        UBYTE   ta_Flags;           /* font preferences and flags */
    };
```

where ta_Name is a string naming the font to open, ta_YSize is the point size of the font (normally in pixels), ta_Style is a bitfield describing the font style, and ta_Flags is a bitfield that further describes characteristics of the font. Note that the name of the font can either be the font name alone (<font name>.font) or it can be prepended with a full path. Without a path to the font, if the font is not already loaded into the system list, OpenDiskFont() will look in the FONTS: directory for the

font file. If there is a path, OpenDiskFont() will look in that directory for the font files, allowing the user to put fonts in any directory (although this is discouraged). OpenFont() and OpenDiskFont() try to find a font that matches your TextAttr description. An important thing to remember about OpenDiskFont() is that only a process can call it (as opposed to a task). This is primarily because the function has to use dos.library to scan disks for font files.

The font styles for ta_Style (from <graphics/text.h>) are:

FSF_UNDERLINED	The font is underlined
FSF_BOLD	The font is bolded
FSF_ITALIC	The font is italicized
FSF_EXTENDED	The font is extra wide

The flags for ta_Flags (from <graphics/text.h>) are:

FPF_ROMFONT	This font is built into the ROM (currently, only Topaz-8 and Topaz-9 are ROM fonts).
FPF_DISKFONT	This font was loaded from disk (with diskfont.library)
FPF_REVPATH	This font is designed to be printed from from right to left (Hebrew is written from right to left)
FPF_TALLDOT	This font was designed for a Hires screen (640x200 NTSC, non-interlaced)
FPF_WIDEDOT	This font was designed for a Lores Interlaced screen (320x400 NTSC)
FPF_PROPORTIONAL	The character widths of this font are not constant
FPF_DESIGNED	This font size was explicitly designed at this size rather than constructed. If you do not set this bit in your TextAttr, then the system may generate a font for you by scaling an existing ROM or disk font (under V36 and above).

For example to open an 11 point bold, italic Topaz font, the code would look something like this:

pseudotext.c

The example above uses the graphics.library's SetFont() function to change the rastport's current font. Notice that this example restores the rastport's original font (myrp->Font) before exiting. This isn't normally necessary unless some other process assumes the rastport's font (or other drawing attributes) will not change. Intuition does not rely on the window's RPort.Font field for rendering or closing the default window font, so applications can change that font without having to restore it.

Prior to Release 2, some applications assumed that any window they opened would always use Topaz-8 without bothering to explicitly set it. Since Topaz-8 was the normal default font before Release 2, this was usually not a problem. However, under Release 2 and later versions of the OS, the user can easily change the default system fonts with the Font Preferences editor. Hence, applications that make assumptions about the size of the

default font look terrible under Release 2 (and in some cases are unusable). Program designers should not make assumptions about the system font, and wherever possible, honor the user font preferences. See the "Preferences" chapter of this manual for more information on how to find user preferences.

1.6 29 / The Text Function / Setting the Text Drawing Attributes

In addition to SetFont(), there are three rastport control functions that set attributes for text rendering:

```
void SetAPen( struct RastPort *rp, ULONG pen );
void SetBPen( struct RastPort *rp, ULONG pen );
void SetDrMd( struct RastPort *rp, ULONG drawMode );
```

The color of the text depends upon the rastport's current drawing mode and pen colors. You set the draw mode with the SetDrMd() function passing it a pointer to a rastport and a drawing mode: JAM1, JAM2, COMPLEMENT or INVERSEID.

If the drawing mode is JAM1, the text will be rendered in the RastPort.FgPen color. Wherever there is a set bit in the character's bitmap image, Text() will set the corresponding bit in the rastport to the FgPen color. This is known as overstrike mode. You set the FgPen color with the SetAPen() function by passing it a pointer to the rastport and a color number.

If the drawing mode is set to JAM2, Text() will place the FgPen color as in the JAM1 mode, but it will also set the bits in the rastport to the RastPort.BgPen color wherever there is a corresponding cleared bit in the character's bitmap image. Basically, this prints the character themselves in the FgPen color and fills in the surrounding parts of the character image with the BgPen color. You set the BgPen color with the SetBPen() function by passing it a pointer to the rastport and a color number.

If the drawing mode is COMPLEMENT, for every bit set in the character's bitmap image, the corresponding bits in the rastport (in all of the rastport's bitplanes) will have their state reversed. cleared bits in the character's bitmap image have no effect on the destination rastport. As with the other drawing modes, the write mask can be used to protect certain bitplanes from being modified (see the "graphics primitives" chapter for more details).

The JAM1, JAM2, and COMPLEMENT drawing modes are mutually exclusive of each other but each can be modified by the INVERSVID drawing mode. If you combine any of the drawing modes with INVERSVID, the Amiga will reverse the state of all the bits in the source drawing area before writing anything into the rastport.

The idea of using a RastPort structure to hold all the rendering attributes is convenient if the rastport's drawing attributes aren't going to change much. This is not the case where several processes need to render into a rastport using very different drawing attributes. An easy way around this problem is to clone the RastPort. By making an exact duplicate of a RastPort, you can change the various rendering parameters

of your RastPort without effecting other programs that render into the RastPort you cloned. Because a RastPort only contains a pointer to the rendering area (the bitmap), the original RastPort and the cloned RastPort both render into the bitmap, but they can use different drawing parameters (font, drawing mode, colors, etc.).

1.7 29 / The Text Function / Rendering the Text

When the Text() routine renders text, it renders at the current rastport position along the text's baseline. The baseline is an imaginary line on top of which the text is rendered. Each font has a baseline that is a constant number of pixels from the top of the font's bitmap. For most fonts, parts of some characters are rendered both above and below the baseline (for example, y, g, and j usually have parts above and below the baseline). The part below the baseline is called the descender.

Figure 29-1: Descenders and Baseline of Amiga Fonts

Because Text() only increments the rastport's current X position as it renders text horizontally, programs that need to print several lines of text have to take care of moving the current pointer to the beginning of the next line, usually with the graphics.library's Move() function:

```
void Move( struct RastPort *rp, LONG x, long y );
```

When moving the current position to the beginning of the next line, an application must make sure it leaves enough space above and below the baseline to prevent characters on different lines from overlapping each other. There are a few fields in the TextFont structure returned by OpenFont() and OpenDiskFont() that are useful for spacing and rendering text:

```
struct TextFont {
    struct Message tf_Message; /* reply message for font removal */
                               /* font name in LN       | used in this */
    UWORD   tf_YSize;          /* font height           | order to best */
    UBYTE   tf_Style;          /* font style            | match a font */
    UBYTE   tf_Flags;          /* preferences and flags / request. */
    UWORD   tf_XSize;      /* nominal font width */
    UWORD   tf_Baseline;   /* distance from the top of char to baseline */
    UWORD   tf_BoldSmear;  /* smear to affect a bold enhancement */

    UWORD   tf_Accessors;  /* access count */

    UBYTE   tf_LoChar;     /* the first character described here */
    UBYTE   tf_HiChar;     /* the last character described here */
    APTR    tf_CharData;   /* the bit character data */

    UWORD   tf_Modulo;     /* the row modulo for the strike font data */
    APTR    tf_CharLoc;    /* ptr to location data for the strike font */
                           /*   2 words: bit offset then size */
    APTR    tf_CharSpace;  /* ptr to words of proportional spacing data */
    APTR    tf_CharKern;   /* ptr to words of kerning data */
};
```

The fields of interest to applications are as follows.

tf_YSize
    The "height", in pixels, of this font. None of the characters in
    this font will be taller than this value.

tf_XSize
    This is the character width for monospaced (non-proportional) fonts.
    The width includes the extra space needed to the left and right of
    the character to keep the characters from running together.

tf_Baseline
    The distance in pixels from the top line of the font to the baseline.

tf_LoChar
    This is the first character glyph (the graphical symbol associated
    with this font) defined in this font. All characters that have ASCII
    values below this value do not have an associated glyph.

tf_HiChar
    This is the last character glyph defined in this font. All
    characters that have ASCII values above this value do not have an
    associated glyph. An application can use these values to avoid
    rendering characters which have no associated glyphs. Any characters
    without an associated glyph will have the default glyph associated to
    them. Normally, the default glyph is a rectangle.

To erase text, the graphics.library provides two functions that were specifically designed to clear parts of a rastport based on the dimensions of the current font:

```
void ClearEOL( struct RastPort *rp );
void ClearScreen( struct RastPort *rp );
```

Using the current font, ClearEOL() will clear the rest of the current text line from the rastport's current position to the edge of the rastport. ClearEOL() was introduced in the Release 2 graphics.library. ClearScreen() will clear the rest of the line as ClearEOL() does, but it will also clear the rastport below the current line of text.

1.8 29 / The Text Function / Setting the Font Style

The OpenFont() and OpenDiskFont() functions both search through the fonts available to them, looking for the font that most closely matches the TextAttr structure. If these functions can't find a font that matches exactly, they will open the one with the same name that most closely matches the TextAttr structure's ta_YSize, ta_Style, and ta_Flags fields (in that order of preference).

If the font doesn't match your style choice exactly, it is possible to ask the system to alter how it renders the font so it matches the style you need. The rastport contains some flags that tell the system's text rendering functions to algorithmically add styles to characters as they

are rendered. Currently, the system can add up to three styles to a font: italics, bold, and underline. The system cannot alter the style of a font if the style is already intrinsic to the font. For example, it is not possible to add (or remove) the bold styling to a font if the font was designed to be bolded. There are two graphics.library functions that deal with software font style setting:

```
ULONG AskSoftStyle( struct RastPort *rp );
ULONG SetSoftStyle( struct RastPort *rp, ULONG newstyle,
                    ULONG enable );
```

The AskSoftStyle() function returns a bitmask of the style bits available to the rastport's current font. The style bits are the same ones used by the TextAttr's ta_Style field (from <graphics/text.h>). SetSoftStyle() changes the rastport's current software style setting according to the style bits set in the newstyle field (from the function prototype above).

SetSoftStyle() pays attention only to the bits of newstyle that have the corresponding bit in the enable field set as well. This function returns the style, which is the combined result of previous soft style selection, the effect of this function, and the style inherent in the set font. The following code fragment turns on the algorithmic font attributes for the rastport (myrastport) based on those style attributes that were requested in the OpenDiskFont() call (mytextattr.ta_Style) and not inherent in the font.

```
/* Set the font and add software styling to the text if I asked for
   a style in OpenFont() and didn't get it. Because most Amiga fonts
   do not have styling built into them (with the exception of the CG
   outline fonts), if the user selected some kind of styling for the
   text, it will have to be added algorithmically by calling
   SetSoftStyle().
*/
if (myfont = OpenDiskFont(mytextattr))
{
        SetFont(myrastport, myfont);
        SetSoftStyle(myrastport,
                     mytextattr.ta_Style ^ myfont->tf_Style,
                     (FSF_BOLD | FSF_UNDERLINED | FSF_ITALIC));
...
...
        CloseFont(myfont);
}
```

1.9 29 Graphics Library and Text / Does the Text Fit?

The Text() function renders its text on a single horizontal line without considering whether or not the text it renders will actually fit in the visible portion of the display area. Although for some applications this behavior is acceptable, other applications, for example a word processor, need to render all of their text where the user can see it. These applications need to measure the display area to determine how much text can fit along a given baseline. The graphics.library contains several functions that perform some of the necessary measurements:

```
    WORD TextLength( struct RastPort *my_rp, STRPTR mystring,
                     ULONG mycount );

    void TextExtent( struct RastPort *my_rp, STRPTR mystring, LONG mycount,
                     struct TextExtent *textExtent );

    void FontExtent( struct TextFont *font,
                     struct TextExtent *fontExtent );

    ULONG TextFit  ( struct RastPort *rp, STRPTR mystring, ULONG strLen,
                     struct TextExtent *textExtent,
                     struct TextExtent *constrainingExtent,
                     LONG strDirection, ULONG constrainingBitWidth,
                     ULONG constrainingBitHeight );
```

The TextLength() function is intended to mimic the Text() function without rendering the text. Using the exact same parameters as the Text() function, TextLength() returns the change in my_rp's current X position (my_rp.cp_x) that would result if the text had been rendered using the Text() function. As in Text(), the mycount parameter tells how many characters of mystring to measure.

Some fonts have characters that intrinsically render outside of the normal rectangular bounds. This can result for example, from the Amiga's version of kerning (which is discussed later in this chapter) or from algorithmic italicizing. In such cases, TextLength() is insufficient for determining whether a text string can fit within a given rectangular bounds.

The TextExtent() function offers a more complete measurement of a string than the TextLength() function. TextExtent(), which was introduced in Release 2, fills in the TextExtent structure passed to it based on the current rendering settings in my_rp. T The TextExtent structure <graphics/text.h>) supplies the dimensions of mystring's bounding box:

```
    struct TextExtent {
        UWORD   te_Width;           /* same as TextLength */
        UWORD   te_Height;          /* same as tf_YSize   */
        struct Rectangle te_Extent; /* relative to CP     */
    };
```

The Rectangle structure (from <graphics/gfx.h>):

```
    struct Rectangle
    {
        WORD   MinX,MinY;
        WORD   MaxX,MaxY;
    };
```

TextExtent() fills in the TextExtent as follows:

te_Width — the same value returned by TextLength().

te_Height — the font's Y size.

te_Extent.MinX — the pixel offset from the rastport's current X position to the left side of the bounding box defined by the rectangle structure. Normally, this is zero.

```
    te_Extent.MinY  the distance in pixels from the baseline to the top of
                    the bounding box.

    te_Extent.MaxX  the pixel offset from the rastport's current X position
                    to the right side of the bounding box.  Normally, this
                    is te_Width - 1.

    te_Extent.MaxY  the distance from the baseline to the bottom of the
                    bounding box.
```

The FontExtent() function is similar to the TextExtent() function. It fills in a TextExtent structure that describes the bounding box of the largest possible single character in a particular open font, including the effects of kerning. Because the FontExtent() function looks at an open TextFont structure rather than a rastport to figure out values of the TextExtent structure, it cannot consider the effects of algorithmic styling. Like TextExtent(), FontExtent() was introduced in Release 2, so it is not available under the 1.3 or earlier OS releases.

The TextFit() function looks at a string and returns the number of characters of the string that will fit into a given rectangular bounds. TextFit() takes the current rastport rendering settings into consideration when measuring the text. Its parameters (from the prototype above) are:

```
    my_rp                   tells which rastport to get the rendering
                            attributes from

    mystring                the string to "fit"

    strLen                  number of characters of mystring to "fit"

    constrainingExtent      a TextExtent describing the bounding
                            box in which to "fit" mystring

    strDirection            the offset to add to the string pointer to get
                            to the next character in mystring (can be
                            negative)

    constrainingBitWidth    an alternative way to specify the width of the
                            bounding box in which to "fit" mystring

    constrainingBitHeight   an alternative way to specify the height of the
                            bounding box in which to "fit" mystring
```

TextFit() will only pay attention to the constrainingBitWidth and constrainingBitHeight fields if constrainingExtent is NULL.

Text Measuring Example

1.10 29 Graphics Library and Text / Font Scaling and Aspect Ratio

The Release 2 OS offers a significant improvement over the Amiga's previous font resources: it now has the ability to scale fonts to new sizes and dimensions. This means, if the diskfont.library can't find the

font size an application requests, it can create a new bitmap font by scaling the bitmap of a different size font in the same font family. The 2.04 (V37) release of the OS improved upon the diskfont.library's font scaling ability so the Amiga now can utilize AGFA Compugraphic outline fonts, yielding scalable fonts that don't have the exaggerated jagged edges inherent in bitmap scaling.

The best thing about the Amiga's font scaling is that its addition to the system is completely invisible to an application program. Because the diskfont.library takes care of all the font scaling, any program that uses OpenDiskFont() to open a font can have scalable fonts available to it. For simple scaling, the programming interface is the same using Release 2 as it was under 1.3.

However, there is one feature of the Release 2 diskfont.library that the 1.3 programming interface cannot handle. When scaling a font (either from an outline or from another bitmap), the Release 2 diskfont.library can adjust the width of a font's glyphs according to an aspect ratio passed to OpenDiskFont(). A font glyph is the graphical representations associated with each symbol or character of a font.

The aspect ratio refers to the shape of the pixels that make up the bitmap that diskfont.library creates when it scales a font. This ratio is the width of a pixel to the height of the pixel (XWidth/ YWidth). The diskfont.library uses this ratio to figure out how wide to make the font glyphs so that the look of a font's glyphs will be the same on display modes with very different aspect ratios.

To add this new feature, several changes to the OS were necessary:

1) The OS needed to be able to store an aspect ratio for any font loaded into the system list.

2) The structures that diskfont.library uses to store bitmap fonts on disk had to be updated so they can store the aspect ratio a font was designed for.

3) The way in which an application requests fonts from diskfont.library had to be altered so that an application could ask for a specific aspect ratio.

For the diskfont.library to be able to scale a font to a new aspect ratio, it needs to know what the font's current aspect ratio is. The Amiga gets the aspect ratio of a font currently in the system list from an extension to the TextFont structure called (oddly enough) TextFontExtension. Under Release 2, when the system opens a new font (and there is sufficient memory), it creates this extension.

A font's TextFont structure contains a pointer to its associated TextFontExtension. While the font is opened, the TextFont's tf_Message.mn_ReplyPort field points to a font's TextFontExtension. The <graphics/text.h> include file #defines tf_Message.mn_ReplyPort as tf_Extension.

The TextFontExtension structure contains only one field of interest: a pointer to a tag list associated with this font:

```
    struct TagItem *tfe_Tags;               /* Text Tags for the font */
```

If a font has an aspect ratio associated with it, the OS stores the aspect ratio as a tag/value pair in the tfe_Tags tag list.

The TA_DeviceDPI tag holds a font's aspect ratio. The data portion of the TA_DeviceDPI tag contains an X DPI (dots per inch) value in its upper word and a Y DPI value in its lower word. These values are unsigned words (UWORD). At present, these values do not necessarily reflect the font's true X and Y DPI, so their specific values are not relevant. At present, only the ratio of the X aspect to the Y aspect is important (more on this later in the article).

Notice that the X and Y DPI values are not aspect values. The X and Y aspect values are the reciprocals of the X and Y DPI values, respectively:

```
    XDPI = 1/XAspect
    YDPI = 1/YAspect
```

so, the aspect ratio is YDPI/XDPI, not XDPI/YDPI.

Before accessing the tag list, an application should make sure that this font has a corresponding TextFontExtension. The ExtendFont() function will return a value of TRUE if this font already has an extension or ExtendFont() was able to create an extension for this font.

The Amiga has a place to store a font's X and Y DPI values once the font is loaded into memory, but where do these X and Y values come from? A font's X and Y DPI values can come from several sources. The X and Y DPI can come from a font's disk-based representation, or it can be set by the programmer.

For the traditional Amiga bitmap fonts, in order to store the X and Y DPI values in a bitmap font's ".font" file, the structures that make up the ".font" file had to be expanded slightly. See the discussion of the FontContentsHeader structure in the "Composition of a Bitmap Font on Disk" section later in this chapter for more information. Currently, out of all the system standard bitmap fonts (those loaded from bitmaps on disk or ROM, not scaled from a bitmap or outline), only one has a built in aspect ratio: Topaz-9.

For the Compugraphic outline fonts, the X and Y DPI values are built into the font outline. Because the format of the Compugraphic outline fonts is proprietary, information about their layout is available only from AGFA Compugraphic. For most people, the format of the outline fonts is not important, as the diskfont.library handles converting the fonts to an Amiga-usable form.

The other place where a font can get an aspect ratio is an application. When an application opens a font with OpenDiskFont(), it can supply the TA_DeviceDPI tag that the diskfont.library uses to scale (if necessary) the font according to the aspect ratio. To do so, an application has to pass OpenDiskFont() an extended version of the TextAttr structure called the TTextAttr:

```
    struct TTextAttr {
        STRPTR  tta_Name;           /* name of the font           */
```

```
        UWORD   tta_YSize;          /* height of the font         */
        UBYTE   tta_Style;          /* intrinsic font style       */
        UBYTE   tta_Flags;          /* font preferences and flags */
        struct TagItem *tta_Tags;   /* extended attributes        */
    };
```

The TextAttr and the TTextAttr are identical except that the tta_Tags field points to a tag list where you place your TA_DeviceDPI tag. To tell OpenDiskFont() that it has a TTextAttr structure rather than a TextAttr structure, set the FSF_TAGGED bit in the tta_Style field.

For example, to ask for Topaz-9 scaled to an aspect ratio of 75 to 50 the code might look something like this:

```
    #define MYXDPI (75L << 16)
    #define MYYDPI (50L)

    struct TTextAttr mytta = {
        "topaz.font", 9,
        FSF_TAGGED,   0, NULL
    };

    struct TagItem tagitem[2];
    struct TextFont *myfont;
    ULONG dpivalue;

     . . .

    tagitem[0].ti_tag = TA_DeviceDPI;
    tagitem[0].ti_Data = MYXDPI | MYYDPI;
    tagitem[1].ti_tag = TAG_END;
    mytta.tta_tags = tagitem;

     . . .

    if (myfont = OpenDiskFont(&mytta))
    {
        dpi = GetTagData(TA_DeviceDPI,
                         OL,
                         ((struct TextFontExtension *)
                          (myfont->tf_Extension))->tfe_Tags);
        if (dpi) printf("XDPI = %d    YDPI = %d\n",
                        ((dpi & 0xFFFF0000)>>16),
                        (dpi & 0x0000FFFF));
              . . .                              /* Blah Blah print blah */

        CloseFont(myfont);
    }
```

1.11 29 Graphics Library and Text / Some Things to Look Out For

One misleading thing about the TA_DeviceDPI tag is that its name implies that the diskfont.library is going to scale the font glyphs according to an actual DPI (dots per inch) value. As far as scaling is concerned, this tag serves only as a way to specify the aspect ratio, so the actual values

of the X and Y elements are not important, just the ratio of one to the other. A font glyph will look the same if the ratio is 2:1 or 200:100 as these two ratios are equal.

To makes things a little more complicated, when diskfont.library scales a bitmap font using an aspect ratio, the X and Y DPI values that the OS stores in the font's TextFontExtension are identical to the X and Y DPI values passed in the TA_DeviceDPI tag. This means the system can associate an X and Y DPI value to an open font size that is very different from the font size's actual X and Y DPI. For this reason, applications should not use these values as real DPI values. Instead, only use them to calculate a ratio.

For the Compugraphic outline fonts, things are a little different. The X and Y DPI values are built into the font outline and reflect a true X and Y DPI. When the diskfont.library creates a font from an outline, scaling it according to an application-supplied aspect ratio, diskfont.library does not change the Y DPI setting. Instead, it calculates a new X DPI based on the font's Y DPI value and the aspect ratio passed in the TA_DeviceDPI tag. It does this because the Amiga thinks of a font size as being a height in pixels. If an application was able to change the true Y DPI of a font, the diskfont.library would end up creating font sizes that were much larger or smaller than the YSize the application asked for. If an application needs to scale a font according to height as well as width, the application can adjust the value of the YSize it asks for in the TTextAttr.

As mentioned earlier, almost all of the system standard bitmap fonts do not have a built in aspect ratio. This means that if an application loads one of these bitmap fonts without supplying an aspect ratio, the system will not put a TA_DeviceDPI tag in the font's TextFontExtension: the font will not have an aspect ratio. If a font size that is already in the system font list does not have an associated X and Y DPI, the diskfont.library cannot create a new font of the same size with a different aspect ratio.

The reason for this is the diskfont.library cannot tell the difference between two instances of the same font size where one has an aspect ratio and the other does not. Because diskfont.library cannot see this difference, when an application asks, for example, for Topaz-8 with an aspect ratio of 2:1, OpenDiskFont() first looks through the system list to see if that font is loaded. OpenDiskFont() happens to find the ROM font Topaz-8 in the system font list, which has no X and Y DPI. Because it cannot see the difference, diskfont.library thinks it has found what it was looking for, so it does not create a new Topaz-8 with an aspect ratio of 2:1, and instead opens the Topaz-8 with no aspect ratio.

This also causes problems for programs that do not ask for a specific aspect ratio. When an application asks for a font size without specifying an aspect ratio, OpenDiskFont() will not consider the aspect ratios of fonts in the system font list when it is looking for a matching font. If a font of the same font and style is already in the system font list, even though it may have a wildly distorted aspect ratio, OpenDiskFont() will return the font already in the system rather than creating a new one.

Font Aspect Ratio Example }

1.12 29 Graphics Library and Text / What Fonts Are Available?

The diskfont.library function AvailFonts() fills in a memory area designated by you with a list of all of the fonts available to the system. AvailFonts() searches the AmigaDOS directory path currently assigned to FONTS: and locates all available fonts. If you haven't issued a DOS Assign command to change the FONTS: directory path, it defaults to the sys:fonts directory.

```
LONG AvailFonts( struct AvailFontsHeader *mybuffer, LONG bufBytes,
                 LONG flags );
```

AvailFonts() fills in a memory area, mybuffer, which is bufBytes bytes long, with an AvailFontsHeader structure:

```
struct AvailFontsHeader {
    UWORD   afh_NumEntries;      /* number of AvailFonts elements */
    /* struct AvailFonts afh_AF[], or struct TAvailFonts afh_TAF[]; */
};
```

This structure is followed by an array of AvailFonts structures with the number of entries in the array equal to afh_NumEntries:

```
struct AvailFonts {
    UWORD   af_Type;            /* MEMORY, DISK, or SCALED */
    struct TextAttr af_Attr;    /* text attributes for font */
};
```

Each AvailFonts structure describes a font available to the OS. The flags field lets AvailFonts() know which fonts you want to hear about. At present, there are four possible flags:

AFF_MEMORY Create AvailFonts structures for all TextFont's currently in the system list.

AFF_DISK Create AvailFonts structures for all TextFont's that are currently available from disk.

AFF_SCALED Create AvailFonts structures for TextFont's that do not have their FPF_DESIGNED flag set. If the AFF_SCALED flag is not present, AvailFonts() will not create AvailFonts structures for fonts that have been scaled, which do not have the FPF_DESIGNED flag set.

AFF_TAGGED These AvailFonts structures are really TAvailFonts structures. These structures were created for Release 2 to allow AvailFonts() to list tag values:

```
struct TAvailFonts {
    UWORD   taf_Type;           /* MEMORY, DISK, or SCALED */
    struct TTextAttr taf_Attr;  /* text attributes for font */
};
```

Notice that AFF_MEMORY and AFF_DISK are not mutually exclusive; a font that is currently in memory may also be available for loading from disk. In this case, the font will appear twice in the array of AvailFonts (or

TAvailFonts) structures.

If AvailFonts() fails without any major system problems, it will be because the buffer for the AvailFontsHeader structure was not big enough to contain all of the AvailFonts or TAvailFonts structures. In this case, AvailFonts() returns the number of additional bytes that mybuffer needed to contain all of the TAvailFonts or AvailFonts structures. You can then use that return value to figure out how big the buffer needs to be, allocate that memory, and try AvailFonts() again:

```
int afShortage, afSize;
struct AvailFontsHeader *afh;
 . . .

    afSize = AvailFonts(afh, 0L, AFF_MEMORY|AFF_DISK|AFF_SCALED|
                                                    AFF_TAGGED);
    do
    {
        afh = (struct AvailFontsHeader *) AllocMem(afSize, 0);
        if (afh)
        {
            afShortage = AvailFonts(afh, afSize, AFF_MEMORY|AFF_DISK|
                                                 AFF_SCALED|AFF_TAGGED);
            if (afShortage)
            {
                FreeMem(afh, afSize);
                afSize += afShortage;
            }
        }
        else
        {
            fail("AllocMem of AvailFonts buffer afh failed\n");
            break;
        }
    } while (afShortage); /* if (afh) non-zero here, then:             */
                          /* 1. it points to a valid AvailFontsHeader, */
                          /* 2. it must have FreeMem(afh, afSize)      */
                          /* called for it after use. */
```

The following code, AvailFonts.c, uses AvailFonts() to find out what fonts are available to the system. It uses this information to open every available font (one at a time), print some information about the font (including the TA_DeviceDPI tag values if they are present), and renders a sample of the font into a clipping region.

AvailFonts.c

1.13 29 Graphics Library and Text / How an Amiga Font Structured in Memory?

So far, this chapter has concentrated on using library functions to render text, letting the system worry about the layout of the underlying font data. As far as the OS representation of a loaded font is concerned, outline fonts and normal bitmap fonts are structured identically. Color fonts have some extras information associated with them and are discussed a little later. Every loaded font, including color fonts, has a TextFont

structure associated with them:

```
    struct TextFont {
        struct Message tf_Message;  /* reply message for font removal */
        UWORD   tf_YSize;
        UBYTE   tf_Style;
        UBYTE   tf_Flags;
        UWORD   tf_XSize;
        UWORD   tf_Baseline;
        UWORD   tf_BoldSmear; /* smear to affect a bold enhancement */

        UWORD   tf_Accessors;
        UBYTE   tf_LoChar;
        UBYTE   tf_HiChar;
        APTR    tf_CharData;  /* the bit character data */

        UWORD   tf_Modulo;  /* the row modulo for the strike font data  */
        APTR    tf_CharLoc; /* ptr to location data for the strike font */
                            /*   2 words: bit offset then size          */
        APTR    tf_CharSpace;
                           /* ptr to words of proportional spacing data */
        APTR    tf_CharKern; /* ptr to words of kerning data             */
    };
```

The first field in this structure is a Message structure. The node in this Message structure is what the OS uses to link together the fonts in the system list. From this node, an application can extract a font's name. The other fields in the TextFont structure are as follows:

tf_YSize
: The maximum height of this font in pixels.

tf_Style
: The style bits for this particular font, which are defined in <graphics/text.h>. These include the same style bits that were mentioned in the discussion of the TextAttr structure in the "Choosing the Font" section of this chapter. In addition to those bits, there is also the FSF_COLORFONT bit, which identifies this as a special kind of TextFont structure called a ColorTextFont structure. This is discussed later in this chapter.

tf_Flags
: The flags for this font, which were mentioned along with the style bits in the section, "Choosing the Font".

tf_XSize
: If this font is monospaced (non-proportional), tf_XSize is the width of each character. The width includes the extra space needed to the left and right of the character to keep the characters from running together.

tf_Baseline
: The distance in pixels from the top line of the font to the baseline.

tf_BoldSmear
: When algorithmically bolding, the Amiga currently "smears" a glyph by rendering it, moving over tf_BoldSmear number of pixels, and

rerendering the glyph.

tf_Accessors
: The number of currently open instances of this font (like the open count for libraries).

tf_LoChar
: This is the first character glyph (the graphical symbol associated with this font) defined in this font. All characters that have ASCII values below this value do not have an associated glyph.

tf_HiChar
: This is the last character glyph defined in this font. All characters that have ASCII values above this value do not have an associated glyph. An application can use these values to avoid rendering characters which have no associated glyphs. Any characters without an associated glyph will have the default glyph associated to them. Normally, the default glyph is a rectangle.

tf_CharData
: This is the address of the bitmap from which the OS extracts the font's glyphs. The individual glyphs are bit-packed together. The individual bitmaps of the glyphs are placed in ASCII order side by side, left to right. The last glyph is the default glyph. The following is what the bitmap of the suits-8 font example looks like (suits8 is the complete, disk-based bitmap font example used later in this chapter):

```
  .@@@...@@@......@........@.......@@@....@@@@@@@@@@@@.............
  @@@@@.@@@@@...@@@@@.....@@@.....@@@@@...@@........@@.............
  .@@@@@@@@@..@@@@@@@@@..@@@@@..@@..@..@@.@@........@@.............
  ..@@@@@@@..@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@.......@@.............
  ...@@@@@....@@@.@.@@@..@@@@@..@@..@..@@.@@........@@.............
  ....@@@.........@.......@@@.......@.....@@........@@.............
  .....@........@@@@@......@......@@@@@...@@@@@@@@@@@@.............
  ................................................................
```

This font is rather sparse, as it only has five glyphs. Most fonts at least have glyphs for each letter of the alphabet. In this example, each glyph represents a symbol for a suit in a standard deck of cards (from left to right: hearts, spades, diamonds, and clubs). Notice that there is no space between the individual glyphs. The spacing information is kept in separate tables to reduce the amount of memory occupied by the font.

tf_Modulo
: This is number of bytes the pointer must move to go from one line of a glyph to the next. This is the pixel width of the entire font bitmap divided by eight. Notice that the bitmap above does not stop after it gets to the end of the last glyph. It is padded with zero bits to the nearest WORD boundary.

tf_CharLoc
: This is a pointer to the CharLoc, the character location table. This table tells the OS how far into the bitmap to look for a character and how many pixels to fetch from each row. The CharLoc table for the suits-8 font looks like this:

```
        $0000000B,$000B000B,$00160007,$001D000B,$0028000C
```

    Each of the five long words in this character location table corresponds to a glyph in Suits-8. Each long word is broken up into two word values. The first word is the offset in pixels from the left edge of the bitmap to the first column containing the corresponding glyph. The second word is the width in pixels of the corresponding glyph image in the bitmap (note, this is not the width of the actual glyph as the actual glyph will have some space on either side of it). For example, the diamond character (the third character) starts at offset $16 (22) and it is 7 pixels wide.

tf_CharSpace
    This is a pointer to an array of WORDs containing the width of each glyph in the font. Each entry tells the OS how much to increment the current horizontal position (usually RastPort.cp_X). For reverse path fonts, these values can be negative.

tf_CharKern
    This is a pointer to an array of "kerning" values. As it is used here, the term "kerning" is unorthodox. On the Amiga, kerning refers to the number pixels to leave blank before rendering a glyph. The normal typographical definition of the word refers to the number of pixels to back up before rendering the current glyph and is usually associated with a specific pair of glyphs rather than one particular glyph.

For each glyph the system renders, it has to do several things:

    1) Get the value from the kerning table that corresponds to this glyph and begin the rendering that number of pixels to the right.

    2) Find this glyph's bitmap using the CharLoc table and blit the glyph to the rastport.

    3) If this is a proportional font, look in the spacing table and figure how many pixels to advance the rastport's horizontal position. For a monospaced font, the horizontal position advance comes from the TextFont's tf_XSize field.

Under Release 2, when the system opens a new font, it creates an extension to the TextFont structure called the TextFontExtension. This extension is important because it contains a pointer to the font's tag list, which is where the system keeps the font's TA_DeviceDPI values. The TextFont's tf_Message.mn_ReplyPort field contains a pointer to the TextFontExtension structure (the <graphics/text.h> include file #defines tf_Message.mn_ReplyPort as tf_Extension). The only field of interest in the TextFontExtension structure is:

```
    struct TagItem *tfe_Tags;           /* Text Tags for the font */
```

which points to the font's tag list. Before accessing the tag list, an application should make sure that this font has a corresponding TextFontExtension. The ExtendFont() function will return a value of TRUE if this font already has an extension or ExtendFont() was able to create

an extension for this font.

1.14 29 Graphics Library and Text / But What About Color Fonts?

When the Amiga loads a color font, it has to account for more information than will fit into the TextFont structures. For color fonts, the Amiga uses a superset of the TextFont structure called the ColorTextFont structure (defined in <graphics/text.h>):

```
    struct ColorTextFont {
        struct TextFont ctf_TF;
        UWORD ctf_Flags;      /* extended flags */
        UBYTE ctf_Depth;      /* number of bit planes */
        UBYTE ctf_FgColor;    /* color that is remapped to FgPen */
        UBYTE ctf_Low;        /* lowest color represented here */
        UBYTE ctf_High;       /* highest color represented here */
        UBYTE ctf_PlanePick;      /* PlanePick ala Images */
        UBYTE ctf_PlaneOnOff;     /* PlaneOnOff ala Images */
        struct ColorFontColors *ctf_ColorFontColors; /* colors for font */
        APTR ctf_CharData[8]; /* pointers to bit planes ala tf_CharData */
    };
```

The ctf_TF field is the TextFont structure described in the previous section. There are two minor differences between the data stored in a color font's TextFont structure and an ordinary TextFont structure. The first is that the color font's TextFont.tf_Style field has the FSF_COLORFONT bit set. The other difference is that the bitmap that TextFont.tf_CharData points to can be a multi-plane bitmap.

The ctf_Flags field is a bitfield that supports the following flags:

CT_COLORFONT The color map for this font contains colors specified by the designer.

CT_GREYFONT The colors for this font describe evenly stepped gray shades from low to high.

The ctf_Depth field contains the bitplane depth of this font's bitmap.

The ctf_FgColor contains the color that will be dynamically remapped during output by changing ctf_FgColor to RastPort.FgPen. This field allows a ColorTextFont to contain color outlines, shadows, etc. while also containing a predominant color that can be changed by the user. If the font does not have a predominant color, ctf_FgColor is 0xFF. For example, given a color font that has a blue and red outline and a white center, the person designing the font can set ctf_FgColor equal to white. Then when the font is used in a paint package that supports color fonts, the white will change to the current foreground color.

The fields ctf_Low and ctf_High contain the lowest and highest color values in the ColorTextFont. For example, a four bitplane color font can have sixteen colors, but the font may use only nine of those colors, thus ctf_Low=0 and ctf_High=8. The most important use of these colors is for defining the boundaries of a gray scale font. If the font uses less than the total number of colors around but needs white as the lowest and black

as the highest level of gray, the boundaries would have to be defined in order for the font to be rendered correctly. Defaults for these values should be the lowest and the highest values for the given number of bitplanes.

The ctf_PlanePick and ctf_PlaneOnOff contain information for saving space in memory for some types of ColorTextFont structures. The ctf_PlanePick field contains information about where each plane of data will be rendered in a given bitmap. The ctf_PlaneOnOff field contains information about planes that are not used to render a plane of font data. If ctf_PlaneOnOff contains a zero bit for a given plane, that bitplane is cleared. If ctf_PlaneOnOff contains a set bit for a given plane, that bitplane is filled. For more information on how the ctf_PlaneOnOff and ctf_PlanePick fields work see the "Specifying the Colors of a Bob" section of the "Graphics Sprites, Bobs and Animation" chapter of this book.

The ctf_ColorFontColors field contains a pointer to a ColorFontColors structure:

```
    struct ColorFontColors {
        UWORD   cfc_Reserved;   /* *must* be zero */
        UWORD   cfc_Count;      /* number of entries in cfc_ColorTable */
        UWORD  *cfc_ColorTable;
                            /* 4 bit per component color map packed xRGB */
    };
```

Which specifies the colors used by this font. The ColorFontColors cfc_Count field contains the number of colors defined in this structure. Each color is defined as a single, UWORD entry in the cfc_ColorTable. For each entry in cfc_ColorTable, the lowest four bits make up the blue element, the next four bits the green element, the next four bits the red element, and the upper four bits should be masked out.

The ctf_CharData[] fields is an array of pointers to each of the bitplanes of the color font data.

1.15 29 Graphics Library and Text / Composition of a Bitmap Font on Disk

For each Amiga bitmap font stored on disk (normally in the FONTS: assign directory), there is a corresponding ".font" file, a directory, and within that directory, a series of files bearing numeric names. For example, for the font Sapphire, within FONTS:, there is a file called sapphire.font, a directory called Sapphire, and within the directory Sapphire are the files 14 and 19.

For a bitmap font (including color fonts), the ".font" file is a FontContentsHeader structure:

```
    struct FontContentsHeader {
        UWORD   fch_FileID;     /* FCH_ID */
        UWORD   fch_NumEntries; /* the number of FontContents elements */
        struct FontContents fch_FC[];
                                /* or struct TFontContents fch_TFC[];  */
    };
```

```
    #define MAXFONTPATH 256
```

Where the fch_FileID field can be be either:

```
    FCH_ID    0x0f00    This FontContentsHeader uses FontContents
                        structures to describe the available sizes of
                        this font.


    TFCH_ID   0x0f02    This FontContentsHeader uses TFontContents
                        structures to describe the available sizes of
                        this font.
```

The fch_FileID can also be equal to 0x0f03, but that is only for scalable outline fonts.

The FontContents structure:

```
    struct FontContents {
        char    fc_FileName[MAXFONTPATH];
        UWORD   fc_YSize;
        UBYTE   fc_Style;
        UBYTE   fc_Flags;
    };
```

describes one of the sizes of this font that is available to the system as a designed font size. For each FontContents structure, there should be a corresponding font descriptor file in this font's directory that contains data for this size font. The FontContents fields correspond to the similarly named field in the TextFont structure.

The TFontContents structure is almost the same as the FontContents structure except that it allows the OS to store tag value pairs in the extra space not used by the file name. Currently, this allows the OS to preserve the X and Y DPI (TA_DeviceDPI) values for a font.

```
    struct TFontContents {
        char    tfc_FileName[MAXFONTPATH-2];
        UWORD   tfc_TagCount;       /* including the TAG_DONE tag */
        /*
         *  if tfc_TagCount is non-zero, tfc_FileName is overlaid with
         *  Text Tags starting at:  (struct TagItem *)
         *      &tfc_FileName[MAXFONTPATH-(tfc_TagCount*sizeof
         *                                 (struct TagItem))]
         */
        UWORD   tfc_YSize;
        UBYTE   tfc_Style;
        UBYTE   tfc_Flags;
    };
```

The fch_NumEntries contains the number of font sizes (and the number of FontContents or TFontContents structures) that this ".font" file describes. The fch_FC[] is the array of FontContents or TFontContents structures that describe this font.

For each font size described in a FontContents (or TFontContents) structure, there is a corresponding file in that font's directory whose

name is its size. For example, for the font size Sapphire-19, there is a file in the Sapphire directory called 19. That file is basically a DiskFontHeader disguised as a loadable DOS hunk and is known as a font descriptor file. This allows the diskfont.library to use the dos.library to load the module just like it was a hunk of relocatable 680x0 instructions. It even contains two instructions before the real DiskFontHeader structure that will cause the 680x0 to stop running the DiskFontHeader if it does inadvertently get executed.

```
    #define  MAXFONTNAME    32      /* font name including ".font" */

    struct DiskFontHeader {
        /* the following 8 bytes are not actually considered a part of */
        /* the DiskFontHeader, but immediately precede it. The         */
        /* NextSegment is supplied by the linker/loader, and the       */
        /* ReturnCode is the code at the beginning of the font in case */
        /* someone runs it...                                          */
        /*   ULONG dfh_NextSegment;               actually a BPTR      */
        /*   ULONG dfh_ReturnCode;                MOVEQ #0,D0 : RTS    */
        /* here then is the official start of the DiskFontHeader...    */
        struct Node dfh_DF;           /* node to link disk fonts */
        UWORD dfh_FileID;            /* DFH_ID */
        UWORD dfh_Revision;          /* the font revision */
        LONG  dfh_Segment;           /* the segment address when loaded */
        char  dfh_Name[MAXFONTNAME]; /* the font name (null terminated) */
        struct TextFont dfh_TF;        /* loaded TextFont structure */
    };

    /* unfortunately, this needs to be explicitly typed */
    /* used only if dfh_TF.tf_Style FSB_TAGGED bit is set */
    #define dfh_TagList     dfh_Segment     /* destroyed during loading */
```

The dfh_DF field is an Exec Node structure that the diskfont library uses to link together the fonts it has loaded. The dfh_FileID field contains the file type, which currently is DFH_ID (defined in <libraries/diskfont.h>). The dfh_Revision field contains a revision number for this font. The dfh_Segment field will contain the segment address when the font is loaded. The dfh_FontName field will contain the font's name after the font descriptor is LoadSeg()'ed. The last field, dfh_TextFont is a TextFont structure (or ColorTextFont structure) as described in the previous section. The following is a complete example of a proportional, bitmap font.

suites8.asm

1.16 29 Graphics Library and Text / Function Reference

The following are brief descriptions of the Graphics and Diskfont library functions that deal with text. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details on each function call.

Table 29-1: Graphics Library Text Functions

Function	Description
Text()	Render a text string to a RastPort.
SetFont()	Set a RastPort's font.
AskFont()	Get the TextAttr for a RastPort's font.
OpenFont()	Open a font currently in the system font list.
CloseFont()	Close a font.
AddFont()	Add a font to the system list.
RemFont()	Remove a font from the system list.
StripFont()	Remove the tf_Extension from a font (V36).
WeighTAMatch()	Get a measure of how well two fonts match (V36).
ClearScreen()	Clear RastPort from the current position to the end of the RastPort.
ClearEOL()	Clear RastPort from the current position to the end of the line.
AskSoftStyle()	Get the soft style bits of a RastPort's font.
SetSoftStyle()	Set the soft style bits of a RastPort's font.
TextLength()	Determine the horizontal raster length of a text string using the current RastPort settings.
TextExtent()	Determine the raster extent (along the X and Y axes) of a text string using the current RastPort settings (V36).
FontExtent()	Fill in a TextExtent structure with the bounding box for the characters in the specified font (V36).
TextFit()	Count the number of characters in a given string that will fit into a given bounds, using the current RastPort settings (V36).

Table 29-2: Diskfont Library Text Functions

Function	Description
AvailFonts()	Inquire which fonts are available from disk and/or memory.
NewFontContents()	Create a FontContents image for a font.
DisposeFontContents()	Free the result from NewFontContents().
NewScaledDiskFont()	Create a DiskFont scaled from another font (V36).
OpenDiskFont()	Open a font, loading it from disk if necessary.