Windows Font Mapping

by Ron Gery

Abstract
This article discusses the Windows font mapper and how it controls the realization of fonts. In the process, the article also looks at what it takes to effectively create a logical font so that the font mapping is predictable and useful. Some of the information is specific to Windows version 3.1, but most of it applies to both versions 3.0 and 3.1.

Introduction

An application requests a font by creating and selecting a font object. In creating the font, the application uses the **CreateFont** or **CreateFontIndirect** function to specify a list of attributes that define the font. A detailed explanation of the specific attributes is found in the SDK The resulting font object is called a *logical* font; it defines an idealized font that may or may not be available on a specific output device.

When the application selects this logical font into a DC using the **SelectObject** function, the font is matched to a *physical font*, one that can be output by the device. This process is called *realization* and is where font mapping takes place. The goal of the realization is to find a font available for the output device that most closely resembles the logical font. Determining this closeness is what font mapping is all about.

Note to the squeamish: in the case that the idealized font does not exist, the Windows font mapper is not perfect, the main controversy being the definition of the closest match. As a result, many disgruntled developers have referred to it with such crafty misnames as "font mangler" and the like. As with most software, the trick to using the font mapper is to understand how it operates and to work with it instead of against it.

An application can get data about the the physical font that is actually realized by using the **GetTextMetrics**, **GetTextFace**, and **GetOutlineTextMetrics** functions (the last one only with TrueType fonts). First select the logical font, then use these functions to get the details on the realized font.

For the duration of the article, the font's attributes are referenced using their field names as defined in the **LOGFONT** structure. While this structure is not a parameter to the **CreateFont** function, that function's parameters map directly to the structure. Inside GDI, all fonts are defined by a **LOGFONT** structure.

The "Right" Way to Create Fonts

The ideal way for an application to create a logical font is to define one whose attributes exactly match those of an existing physical font. This allows the application to control the mapping because it has essentially already picked which physical font it wants to use. Normally an application takes this approach by enumerating all of the fonts in the system using **EnumFonts** or **EnumFontFamilies** (the latter is only available in Windows version 3.1). These functions enumerate the fonts by specifying for each physical font a logical font that exactly describes how the system identifies the physical font.

Font enumeration, like font realization, is based on a specific physical device identified by a DC. Different devices enumerate a different set of physical fonts, so an enumerated logical font from one device may not exactly match a physical font on another device. For example, a given printer may enumerate a hardware-based font named "Courier 10cpi," but that font has no exact match on the display device so the font mapper is forced to choose the closest match among the fonts available to the display. The application can "help out" in this process by choosing an ideal match from the fonts enumerated by the display device and using that font for selection into the display device.

Usually the desired font is chosen by the user, a process that can be greatly simplified by an application's use of the **ChooseFont** common dialog. This dialog presents the user with the list of enumerated fonts and returns to the application a logical font to use. The application can control exactly which types of fonts are presented to the user, thereby limiting the enumeration as desired.

If enumeration and exact picking is not an option, an application still needs to describe the font as unambiguously as possible in order to get a reasonable level of consistency in the mapping process. It is a good idea to always specify an **lfFaceName**, **lfPitchAndFamily**, **lfHeight**, and **lfCharSet**. The use and abuse of default values is discussed in greater detail below.

Point Sizes and Cell Heights

The Windows font interface deals with pixel heights of font cells instead of the more typographically traditional point sizes (1/72nd of an inch). Converting between the two measurements is not too hard, though:

```
CellHeight = (PointSize * VerticalResolution) / 72 + InternalLeading;

and conversely,

PointSize = ((CellHeight - InternalLeading) * 72) / VerticalResolution;
```

Where:

- Ø *CellHeight* is cell height of font. This is requested with the **lfHeight** attribute and returned in the **tmHeight** field of the TEXTMETRIC structure after the font is selected.

- Ø *PointSize* is the point size of the font.

- Ø *VerticalResolution* refers to the resolution of the device being used. Windows provides two possible numbers for this value, neither of which is entirely accurate on display devices. Most applications use the convention of LOGPIXELSY (available via **GetDeviceCaps**), which is an idealized number of pixels per "logical inch" and is used in nonscalable fonts' resolution specification. The alternative is to compute the number of pixels per "physical inch" using VERTSIZE and VERTRES (also available via **GetDeviceCaps**). Thess values do not provide a hardware-specific value either (Windows has no knowledge of the physical size of the screen), but they may be a better approximation. With printer devices, the values for both logical resolution and physical resolution are usually the same.

- Ø *InternalLeading* is a Windows concept that specifies the difference between a font's actual point size (also known as the *EmHeight*) and its physical cell height. This value is returned in the **tmInternalLeading** field of the TEXTMETRIC structure after the font is selected.

  Gee, so how does an application account for the internal leading value that is not available until after the font is selected? The simplest way to overcome the internal leading concept is to specify the **lfHeight** as a negative value. The font mapper treats the absolute value of that number as the desired point size (i.e. the value defined by CellHeight-InternalLeading) rather than as the cell height. To define a height for a font with a given point size, use:

```
MyFont.lfHeight = -((PointSize * GetDeviceCaps(hDC, LOGPIXELSY)) / 72);
```

Also, the TEXTMETRIC stucture is returned during font enumeration, so it is possible to anticipate a font's internal leading if a specific physical font is being tartgeted. Anticipation is not possible with scalable fonts because the fonts do not enumerate specific sizes and may not scale linearly,

The Mapping Process

Now that the logical font is built, the font is selected into a DC, and GDI calls the font mapper to perform the realization. Well actually because GDI caches physical fonts, if a logical font is being re-selected and the DC has not changed (same device, same mapping mode), there is no need for a full mapping, and the cached realization information is used instead. Unfortunately, this can't always be done.

There is a fundamental difference in the font mapping process between Windows version 3.0 and version 3.1. Both versions share a core mapper that uses weighted penalties to choose the closest matching physical font, but version 3.1 also has a shortcut layer that attempts to bypass the core mapper by looking for exact matches and only resorting to the core mapper when a certain level of exactness cannot be achieved. The basic idea is that if an application requests a logical font in the "right" way (see above), finding a matching physical font can be quick and to the point without excess processing; the shortcut layer is only for speed purposes, the functional behavior is unchanged.

Possible Physical Fonts

There are four types of physical fonts that are available to the font mapper, system-based raster fonts, system-based vector fonts, hardware-specific fonts found on the device, and for Windows version 3.1, TrueType fonts. Fonts generated by most bolt-on font packages like Adobe Type Manager (ATM) appear to the font mapper as device fonts.

The system-based fonts are the non-TrueType fonts that are installed via the Control Panel (TrueType fonts are also installed via the Control Panel). A few are installed automatically when Windows is set up. The raster fonts usually come in several sizes per facename, and because the fonts do not scale, these are the only sizes that are available for mapping. The vector fonts that come with Windows (Modern, Roman, and Script) are scalable fonts defined by polylines and are the only fonts that can be printerd on a plotter device. They are commonly referred to as the "stick fonts" because they are made up of line segments, which results in rather poor typographic quality. Most applications avoid the vector fonts by requesting a font with **lfCharSet** set to something other than OEM_CHARSET (specifically, ANSI_CHARSET or SYMBOL_CHARSET), the charset of the vector fonts, and it is common to see these fonts screened out of the font selection lists shown to the user.

Device fonts are those fonts that are provided by the device driver, either via hardware fonts or by downloading to the device. Usually these only exist for printer drivers, but bolt-on font engines also provide the concept for display drivers. Device fonts are controlled more by the device than by GDI. GDI only has access to descriptive information about the font and not to the actual bit information, and more importantly, the device driver acts as its own font mapper to choose the best match among the available device fonts. During the font mapping process, the Windows font mapper only inspects the one device font that was chosen by the device driver. Windows sends the device driver the requested logical font, and the device chooses the best match from its available hardware and downloaded fonts. If the device has no font choice, there is no device font for the font mapper to inspect.

TrueType font technology is new for Windows version 3.1 and introduces to the system a class of fonts that are scalable and compatible with all raster devices. On printers, these fonts are either built into the hardware, downloaded to the hardware, or drawn to a bitmap which is then blted to the page. TrueType fonts, like raster and vector fonts, are added to the Windows environment via the Control Panel and managed by GDI. Because of their scalability, TrueType fonts are not penalized for height, width, or aspect ratio, but they are treated like the other fonts in all other respects.

The Core Mapper

The mapper itself, the big kahuna, compares each one of the possible physical fonts to the requested font. Each of the physical fonts is identified using a set of attributes that roughly parallel those found in the LOGFONT structure, and each attribute is compared to the requested attribute. There is a penalty assessed for each mismatch, and the penalties are accumulated. The mapper tracks the physical font with the lowest penalty, and once all of the physical fonts have been inspected, one font remains as the closest match. In case of a tie, the first closest match encountered is the one selected.

The interesting part, of course, is the relative size of the penalties. At the end of this article is a table that lists the specific penalties and their values. A quick inspection of this table reveals that the penalties are heavily weighted to discourage physical fonts whose major typographical features are not compatible with the logical font. The big ticket items are charset, output precision, variable instead of fixed pitch, facename, family type, and height. The lesser penalties only come into play in cases where the logical font is too vague to hone in on a desired physical font.

The height penalty is often a cause for confusion. The goal for the mapper is to pick a font that is as large as possible without being taller than the requested height. For example, when using raster fonts (height problems only apply to fonts that do not scale arbitrarily), if the logical font has a height of 11 and the available physical fonts have heights of 10 and 12, the shorter font is chosen. If the logical font has a height of 8 and there was no font shorter than 10, then the font of height 10 is chosen. This "under-sizing" scheme can get undesired results if a super-small font such as SMALLE.FON is available on the system because the font maps to a 6-pixel hight font that is not exactly legible. Then again, an actual small "filler" font may be exactly what the application wants.

In the case where two different fonts have exactly the same penalty, the first font that was inspected is the one that is selected. An application does not have control on this ordering. Non-device fonts are assessed an extra penalty to encourage the selection of a device font over an identical non-device font. The selection priority is 1) device, 2) raster, and 3) TrueType.

Shortcut Mapping for Verstion 3.1

New for Windows version 3.1 is an attempt to circumvent the above penalty scheme by searching for exact matches instead. For this algorithm, a physical font is an exact match if it shares the following attributes with the logical font:

- charset
- face name
- height
- italicness
- weight

Because an exact match is the goal, GDI's italic and emboldening simulations are not considered.

The shortcut mechanism is only invoked if the device is a raster device that can accept raster fonts or a device that can output TrueType fonts (presumably by downloading). Also, the logical font must specify either a face name or use the OUT_TT_ONLY_PRECIS or CLIP_EMBED_PRECIS flags (in its **lfOutputPrecision** and **lfClipPrecision** attributes, respectively) in order to be considered a candidate for an exact match. Obviously if no face name is specified, then an exact match cannot be found, but the two flags indicate that only TrueType fonts should be considered for a match so some of the standard mapping process can be avoided. If the shortcut method cannot be used, the standard penalty-based mapper is used to pick the font.

Similar to the full font mapper, the shortcut inspects all the possible physical fonts for a match, but instead of tracking penalties, it throws out of consideration any candidate font that does not exactly match the above attributes. So if the logical font is based directly on an existing physical font, the matching is very simple, which is the entire goal of the shortcut.

If no exact match is found, the font mapping defaults to the penalty-based system. Ties (that is, multiple physical fonts providing an exact match) are resolved based on the filter setting of the **lfOutputPrecision** attribute. If no special filter is specified, the default behavior is that device fonts beat out everything else and raster fonts beat out TrueType fonts. The default behavior can be altered to have the TrueType font win a tie by using the TTIfCollisions entry in the [TrueType] section of win.ini. The possible filters are discussed in more detail below.

Extra Commentary

Below is a variety of issues and suggestions that affect the application's control of the font mapping process.

TrueType

The existence of the TrueType font technology in Windows version 3.1 changes the font mapping game to some degree. The basic mapping process remains intact, but the availability of more fonts and the ability of these fonts to be scaled to any desired size greatly increases the chance for a very close font match.

This is especially true for loosely defined fonts, those fonts that use a lot of default attribute settings. For example, an application that assumes a logical font created by specifying a height of 12 and setting all other attributes to 0 will map to the system font is making a very bad assumption. While it might work under Windows version 3.0, under version 3.1 at least 6 fonts can match that request exactly even if only the base TrueType fonts are loaded. The one that is chosen by the font mapper is suddenly not so obvious and depends on such untangibles as the ordering of the fonts during initialization.

It is up to the application to be specific enough in its definition of the logical font in order to avoid mapping confusion. The first thing to consider is that TrueType fonts are fully scalable, so an apparently off-size height can be matched—there is no penalty for height (or for width or aspect ratio).

Because most TrueType fonts are designed with a normal, italic, bold, and bold italic version, a request for one of these variations maps directly to a non-simulated variant of the standard font. Many fonts are also available in non-traditional weights such as light, demibold, and black, so the exact **lfWeight** that is specified could be meaningful. Lastly, an application needs to be careful in choosing the **lfFaceName** to deal with the potential explosion of face names. The **EnumFontFamilies** function is designed to help an application sort through the face names and their family relationships. In the eyes of the font mapper, the **lfFaceName** specified by the application can be either the full face name (for example, "Arial Italic") or only the family name (in this example, "Arial"). Assuming that all of the attributes match (in this example, **lfItalic** needs to be set), the two names are considered equivalent.

TrueType doesn't really introduce anything new into the font mapping picture, but it does make the choice of physical fonts large enough to necessitate an application to carefully define its logical fonts.

Substituted Face Names

Font face names are usually copyrighted strings, which adds a level of confusion in trying to identify various fonts that are produced by different vendors but look remarkably similar. Because much of the font mapping process relies on the exact use of face names, the font mapper has the concept of *face name substitution* to allow one face name to be substituted for another in the mapping process. This is a feature new to Windows version 3.1. For example, Windows version 3.0 had a raster font named "Helv" that is renamed to "MS Sans Serif" in Windows version 3.1. With version 3.1's built-in substitutions, a logical font with **lfFaceName** set to "Helv" is matched by face name to the physical font with the name "MS Sans Serif".

Face name substitutions are used only for raster and TrueType fonts and not for device fonts. Also, when a face name is matched via substitution, it is not considered an exact match— a penalty is assessed for substitute face names, and an exact match always takes precedence.

The substitutions come from two places: one list is predefined in GDI and a second is defined by the user in win.ini under the [FontSubstitutes] section. The user-defined list can override GDI's predefined list to customize the Windows environment to the user's font needs. GDI's list consists of approximately 20 substitutions that map "Tms Rmn" to "MS Serif", "Helv" to "MS Sans Serif", and various PostScript names to the corresponding TrueType fonts that ship in Windows and in the Windows Font Pack.

Filters

An application can, to some extent, filter which physical fonts are examined by the font mapper. Aspect ratio filtering, which is available in both Windows version 3.0 and version 3.1 allows an application to specify that only fonts designed for the device's aspect ratio should be considered by the font mapper. An application enables and disables this filter by using the **SetMapperFlags** function. Because nonscaling raster fonts are designed with a certain aspect ratio in mind, it is sometimes desirable to ensure that only fonts actually designed for the device are used. When this filter is enabled, the font mapper does not consider any physical fonts whose design aspect ratio does not match that of the device. Aspect ratio filtering does not affect TrueType fonts because they can scale to match any aspect ratio. This filter affects all font selections to the DC until the filter is turned off. Aspect ratio filtering is a holdover from earlier times and is not a recommended approach in today's font world.

The second type of filtering is new for Windows version 3.1 and allows an application to choose what type of physical font should be chosen in the case of a facename conflict. These filters only affect the shortcut mapping described above; if no shortcut match is found, the font mapper continues with its standard matching scheme and ignores these filters. The application defines the filter in the **lfOutputPrecision** attribute of the logical font. Unlike the aspect ratio filter, this filter is specific to a given logical font. The three possible filter settings are OUT_TT_PRECIS for TrueType fonts, OUT_DEVICE_PRECIS for device fonts, and OUT_RASTER_PRECIS for raster fonts. For example, if the shortcut mechanism finds two fonts, one TrueType and one raster, that provide an exact match for the logical font and the logical font is defined with OUT_TT_PRECIS, the TrueType font is chosen.

Another shortcut filter, OUT_TT_ONLY_PRECIS, specifies that only TrueType fonts should be considered for the exact matching. None of the other types of physical fonts is considered. This filter is useful for an application that wants to guarantee that the realized font is a TrueType font.

Default Values

Many of the attributes that define a font have a defined default setting. For most attributes, the font mapper ignores default attributes in its penalty scheme so that no penalty is given for that attribute for any candidate font. This is a simple way to indicate that an attribute is not important in the font mapping.

There are times when this gets dangerous. Using DEFAULT_CHARSET for specifying the font's **lfCharSet** can lead to the font mapper choosing a non-ANSI font that may not contain needed characters and may not even be composed of alphanumeric glyphs. Internally, the DEFAULT_CHARSET is an actual charset definition, so all candidate fonts get an equally large penalty for not matching the charset. It is always a good idea to use a non-default charset when defining a logical font.

The font mapper treats a logical font with an **lfHeight** of 0 as a font that is 12 points high. The actual pixel height depends on the resolution of the device being used. With the advent of TrueType technology in Windows version 3.1, this default height becomes even less meaningful (more on that below). The moral: always specify a height.

By specifying no face name (**lfFaceName**) for the logical font, an application indicates to the font mapper that the exact face name of the physical font is not critical. An application that does not specify a face name is just asking for trouble. Because the face name is the most unique identifier of a font, ignoring it leaves the application open to more ambiguity in realizing a physical font. The more fonts are available on the system, the less control the application has over the mapping process. An application can overcome some of this ambiguity by defining the generalized look of the font using the **lfPitchAndFamily** field, but as more and more fonts become available in the Windows system, a font's unique face name becomes critical in its identification. The moral: always specify a face name.

The font mapper interprets an **lfWeight** of FW_DONTCARE as though it was really FW_NORMAL, and any corresponding penalty is tabulated.

It is generally considered a good thing to specify a default **lfWidth** by setting this attribute to 0. While for most fonts the character widths cannot be changed and are not actually affected, scalable fonts (TrueType and many device fonts) are altered in shape by a non-default width. This is usually not desired. Because height is more important in the matching process, a non-default width usually does not affect the matching, but it could. The font mapper compares this value to the average width value of the candidate font (accessible via the **tmAvgWidth** field of the TEXTMETRIC structure), a value which for nonscalable, variable pitch fonts is an approximation at best. For scalable fonts, the **lfWidth** field is used to define an *x*-scaling factor for the font (*x*-scale = **lfWidth**/**tmAvgWidth**) where an **lfWidth** of 0 means a scaling factor of 1.0, the default.

Font Rotation

An application specifies the desired rotation of a logical font using the **lfEscapement** and **lfOrientation** attributes. The main font mapper does not use these attributes in its font selection process—no penalties are assessed for candidate fonts that are not rotated or rotatable—but the shortcut method does not select a raster font if either of the attributes is nonzero. The key issue here is that not all fonts, raster fonts especially, can be rotated effectively. Because font rotation is not a factor in the mapping, it is possible that the chosen physical font is not able to rotate as desired by the application. Fonts that do rotate are TrueType fonts, the vector fonts, and some device fonts. It is wise for an application that desires rotated fonts to specifically ask for a font that can actually be rotated.

Other Warnings

Most applications do not want a vector font selected as the physical font because vector fonts are ugly. Before TrueType, the vector fonts were the only fonts that could be arbitrarily scaled and rotated, and it could be argued that they maintained more dignity than a GDI-scaled raster fonts at very large size. Mostly, though, they are to be avoided. The simplest way to do this is to explicitly use ANSI_CHARSER or SYMBOL_CHARSET when defining a logical font. The penalty for the charset is large enough to keep the vector fonts from font matching contention. Vector fonts can also be culled out of an enumeration by ignoring any enumerated font that is not a raster, TrueType, or device font. With the **ChooseFont** common dialog, vector fonts can be removed from the user's view through use of the CF_NOVECTORFONTS flag.

The height and width of a logical font are defined in logical units. When the font is selected into a DC, these values are converted to device units before the font mapping takes place. As a result, the realized font may be larger or smaller, depending on the DC's current mapping mode. Also, if an application changes the mapping mode of a DC, the currently selected font is re-realized based on the new coodinate system. This is entirely consistent with GDI's handling of logical and physical coordinates and is something to keep in mind when specifying the height of a font. The one exception to this process is new to Windows version 3.1: if the logical font is one of the stock objects (available through **GetStockObject** function-- copies don't count), the height and width are always treated as MM_TEXT values, regardless of the DC's current mapping mode.

GDI simulations

GDI provides simulations for some font attributes. It emboldens a normal weight font by overstriking, italicizes a nonitalic font by shearing the output (TrueType fonts are sheared during rasterization), underlines or strikes-out fonts, and scales raster fonts independently in *x* and *y* by integral amounts. In cases where GDI's simulations help a font to become a better match, the font mapper reduces but does not eliminate the penalty for a mismatch of the simluated attribute. This makes the font a more palatable choice. Unfortunately, those simulations that alter the shape of the font (emboldening, italicizing, and scaling) are not always typographically pleasing. With the exception of scaling, all of these simulations can be performed on any of the available fonts.

When GDI scales a raster font, it does so by simply stretching the font's original bitmap definition horizontally and vertically, as appropriate. Because the scaling is always integral, none of the really nasty side-effects of bitmap stretching occur, but a stretched font still looks very clunky, especially at larger scaler factors.

GDI can trivially simulate underlines and strikethroughs using horizontal lines, so these attributes are not a real concern. While TrueType fonts have special metrics for the placement of underlines and strikethroughs, GDI approximates these values for other fonts with the underline on the baseline and the strikethrough 1/3 of the ascent above the baseline.

Stock Object Special Case

Under Windows version 3.0, a font's logical height and width are always translated to physical units before font mapping, so a stock font could potentially map to an unexpected font if the DC's coordinate system results in a different-sized physical font. This can be a real "gotcha" for an application that assumes it can use the SYSTEM_FONT stock object (for example) without regard to the scaling being done. The physical font scales with the coordinate system.

Under Windows version 3.1, if an application selects a stock font into a DC, the requested font's logical height and width are not translated into logical units. The effect is that a stock font remains the same size regardless of the DC's mapping mode. A copy of a logical object does not qualify for this special feature, only the actual stock object does. Because objects recorded in metafiles are simply copies of objects, a stock font recorded in a metafile is an ordinary font when the metafile is played back.

Actual Weights

The following table specifies the penalty weights used by the font mapper in Windows versions 3.0 and 3.1. Penalties that are new for Windows version 3.1 are identified as such. Note very carefully that this information is *version specific* and is subject to change without warning in future versions of the Windows product. In the table, *requested* refers to the logical font, and *candidate* refers to the physical font to which it is being compared. The candidate with the smallest penalty is the one that is elected.

penalty description	**penalty weight**	**comments**
CharSet	65000	charset does not match
Output Precision	19000	request OUT_STROKE_PRECIS, but device can't do it or candidate is not vector font OR don't request

		OUT_STROKE_PRECIS, candidate is a vector font, and device doesn't support it
FixedPitch	15000	request fixed pitch, candidate variable pitch
FaceName	10000	request face name, candidate does not match
FaceNameSubst	500	new for 3.1 candidate is a substitute face name
Family	9000	request a family, candidate's family is different
FamilyUnknown	8000	request a family, but candidate has no family
FamilyUnlikely	50	new for 3.1 request roman/modern/swiss, candidate decorative/script or vice versa

HeightBigger	600	candidate non-vector font and bigger than requested
PitchVariable	350	request variable pitch, candidate not variable pitch
HeightSmaller	150	raster candidate and smaller than requested penalty * height difference
HeightBigger	150	raster candidate and bigger than requested penalty * height difference
Width	50	request a width, candidate doesn't match penalty * width difference
SizeSynth	50	raster candidate needs scaling by GDI
UnevenSizeSynth	4	raster font scaled unequally in width and

		height penalty * (100 * bigger multiplier / smaller multiplier)
IntSizeSynth	20	raster font needs scaling penalty * (height multiplier + width multiplier)
Aspect	30	candidate aspect ratio different from device penalty * ((100 * devY/devX) - (100 * candidateY / candidateX))
Italic	4	request and candidate do not agree on italic status, and the desired result cannot be simulated
ItalicSim	1	new for win 3.1 request italic, candidate not italic but can be simulated to be italic
Weight	3	candidate's weight does not match penalty * (weight difference/10)

Underline	3	request no underline, candidate is underlined
StrikeOut	3	request no strike-out, candidate is struck
DefaultPitchFixed	1	request DEFAULT_PITCH and candidate is fixed pitch
SmallPenalty	1	new for win 3.1 request rotated font, candidate needs bold or italic simulation and is a raster or vector font
VectorHeightSmaller	2	candidate vector font is smaller penalty * height difference
VectorHeightBigger	1	candidate vector font is bigger penalty * height difference

DeviceFavor	2	extra penalty for all non-device fonts new for win 3.1, request OUT_TT_PRECIS and candidate is not TrueType. Penalty is twice this value.

End.