POSTSCRIPT™
Software From Adobe

Adobe Type Manager Software API: Windows

Adobe Developer Support

Technical Note #5073

26 April 1993

Adobe Systems Incorporated

Corporate Headquarters
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400 Main Number
(415) 961-4111 Developer Support
Fax: (415) 961-3769

Adobe Systems Eastern Region
24 New England
Executive Park
Burlington, MA 01803
(617) 273-2120
Fax: (617) 273-2336

Adobe Systems Europe B.V.
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z-O, Netherlands
+31-20-6511 200
Fax: +31-20-6511 300

Adobe Systems Japan
Swiss Bank House 7F
4-1-8 Toranomon, Minato-ku
Tokyo 105, Japan
+81-3-3437-8950
Fax: +81-3-3437-8968

PN LPS5073

Contents

Adobe Type Manager Software API: Windows

1 Introduction to the Adobe Type Manager Interface

Adobe Type Manager™ software for Windows™ allows well over 10,000 Type 1 PostScript™ language font programs to be used transparently by Windows application programs to image text on displays and printers.

Adobe Type Manager (ATM™) software intercepts the GDI text drawing calls and images all requests that can be satisfied using Type 1 fonts. (The words 'font' and 'fonts' refer to PostScript language font programs.) Windows 3.0 does not have a direct interface that allows imaging of skewed type. The ATM software does have this capability and applications can take advantage of this through an application program interface (API). Additionally, using the ATM software API, an application can access a Type 1 font outline description that can be used for special text effects. Finally, the ATM software API can be used by an application program to create and fill arbitrary paths (including Bézier curves).

The API provides a number of function calls. Using the **ATMProperlyLoaded( )** call, an application can check whether ATM software is loaded and initialize the ATM software API structures. **ATMGetVersion( )** can be called if the application wants to verify whether a specific version of ATM software is running.

The **ATMFontAvailable( )** call is a quick way to check whether a font outline is available to the ATM software. **ATMFontSelected( )** can be used to determine whether the currently selected font can be imaged by ATM software. **ATMXYShowText( )** and **ATMBBoxBaseXYShowText( )** allow an application to pass a *string*, a *handle* to a device context (HDC), and a *matrix* along with other parameters (similar to GDI function **ExtTextOut( )**) to the ATM software. The string is then imaged on the device pointed to by hDC in the font currently selected for the device. The *matrix* is described in the section 1.2, "Transformation Matrix".

ATMGetOutline( ) allows an application to obtain the PostScript language outline description of a Type 1 font character. In addition, **ATMGetOutline2( )** allows a Japanese application to obtain the PostScript language outline description of both Type 1 roman fonts and Japanese fonts. The outline can then be manipulated and re-imaged by using GDI calls or by passing the manipulated data to the ATM software via ATM software's path rendering and filling operators.

Note *The functions defined in this document are followed by a version number. This number indicates the earliest version of ATM software that implements the function. All subsequent versions of ATM software will also support the function. ATM software Japanese version 2.5 is based on ATM software Roman version 2.0.3. Therefore, there are functions that are supported in the Roman version 2.5 but not in the Japanese version 2.5. This will be clarified by each description of such functions.*

Appendix A lists the ATM software API error codes. These error codes are referred to throughout this document.

1.1 Structures and Typedefs

The following data types and MACROS are used throughout this document.

ATMFixed

```
typedef long int ATMFixed;
```

ATMFixedMatrix

```
typedef struct {
    ATMFixed a, b, c, d, tx, ty; /* tx & ty not presently used */
} ATMFixedMatrix, FAR *LPATMFixedMatrix;
```

ATMFixedPoint

```
typedef struct {
    ATMFixed x, y;
} ATMFixedPoint, FAR *LPATMFixedPoint;
```

ATMShortPoint

```
typedef struct { // added in version 1.15
    short int x,y;
} ATMShortPoint, FAR *LPATMShortPoint;
```

ATMBBox

```
typedef struct { // added in version 1.15
    ATMShortPoint ll, ur; // could be called ul & lr
} ATMBBox, FAR *LPATMBBox;
```

Note *Our ATMBBox field names ll (lower left) & ur (upper right) can be confusing. For example, when used with* ***ATMFillGetBBox()****, if the mapping mode is MM_TEXT and the bounding box goes from (0,0) to (100,100), then the* *values in ll will be the point (0,0) and the values in ur will be the point (100,100). Of course, on your display in MM_TEXT mapping mode, (0,0) is actually the upper left not lower left!*

ATMINTTOFIXED #define ATMINTTOFIXED(n) ((ATMFixed) (((long int)(n))<< 16))

ATM software uses the *ATMFixed* data type internally. *ATMFixed* is a 32-bit signed quantity where the fractional part is contained in the low order word and the integer part is contained in the high order word. To convert a *float* to *ATMFixed*, multiply the *float* by 65536.0 then cast to (*ATMFixed*).

1.2 Transformation Matrix

The ATM software does the work of transforming (rotating and skewing) text. The application specifies the transformation by providing a matrix of type *ATMFixedMatrix* to the **ATMXYShowText( )**, **ATMGetOutline( )** and **ATMGetOutline2( )** functions. This matrix maps the coordinate system onto the coordinate system of the device context currently in use.

The matrix is similar (but not identical) to the matrix passed to the PostScript language operator **makefont**. The values used in this transformation matrix are *device* units (pixels). Appendix B discusses the differences between this transformation matrix and that normally passed to the **makefont** operator.

Note that *ATMFixedMatrix* contains variables *tx* and *ty*, which are not currently used by ATM software but are included for completeness. Currently, *tx* and *ty* should be initialized to zero.

The matrix values are as follows:

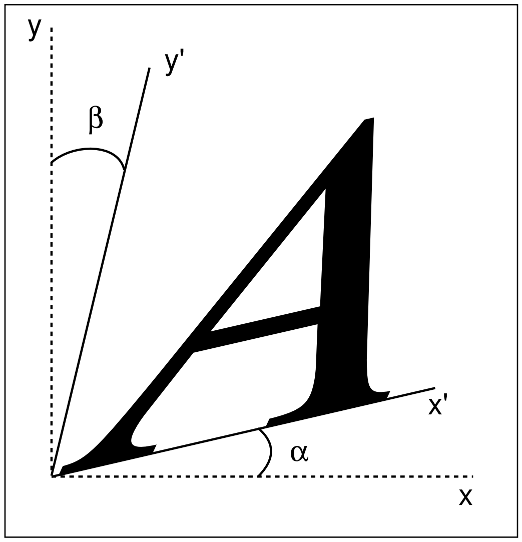
$$a = Mx$$
$$b = -Mx \tan(\alpha)$$
$$c = My \tan(\beta)$$
$$d = -My$$

The elements a and d are the horizontal and vertical scaling factors. Mx is the font size in the x direction. If Mx is 12 then the x dimension of the font will be that of a 12 pixel font. My is the font size in the y direction. The y dimension of the font will be that of an My pixel font. The elements b and c determine the orientation of the axes relative to a normal Cartesian coordinate system. Element b includes the skew of the axis in the x direction.

Figure 1 illustrates the geometry. If α is the upward skew of the transformed x axis relative to the normal x axis, then $b = -Mx \tan(\alpha)$. Similarly if β is the clockwise vertical skew of the transformed y axis relative to the normal y

axis, then $c = My \tan(\beta)$. Note that all these values have been adjusted to map into a coordinate system in which y values increase downward and the x and y units have the same physical dimensions.

Figure 1 *The skew of the coordinate system*



In Figure 1, the skew of the coordinate system is specified by the parameters α and β. The directions of skew indicated in the drawing would result in positive values of both α and β.

A slightly different formula is easier for uniform rotations. If you want a size of M pixels with a counterclockwise rotation of θ degrees, then

$$a = M \cos\theta$$
$$b = -M \sin\theta$$
$$c = -M \sin\theta$$
$$d = -M \cos\theta$$

Examples

The following pages contain some simple code examples showing how the matrix should look when passed into **ATMXYShowText( )**.

Example 1: *Upright text*

```
m.a = ATMINTTOFIXED( 126);  /* Upright 126 pixel font */
m.b =  0;
m.c =  0;
m.d = ATMINTTOFIXED( -126);
m.tx =  0;
m.ty =  0;
```

Figure 2 *Upright text*

Example 2: *Reversed text*

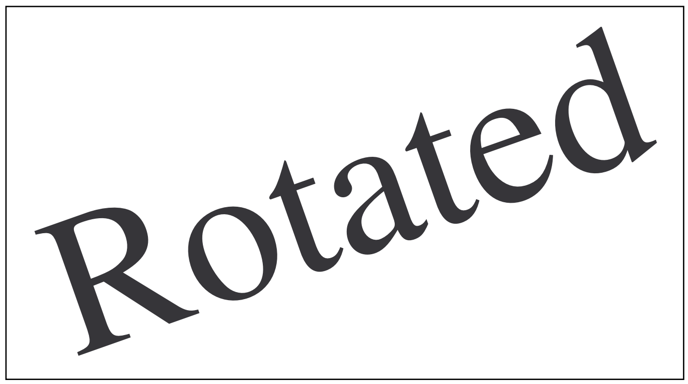
```
m.a = ATMINTTOFIXED( -55);   /* Reversed in x. 55 pixels in x. */
m.b =  0;
m.c =  0;
m.d = ATMINTTOFIXED(85);     /* Reversed in y. 85 pixels       */
m.tx = 0;
m.ty = 0;
```

Figure 3 *Reversed text*

Example 3: *Rotated text*

```
#define    DEGTORAD  (2*3.14159/360) /*   2pi/360              */
#define   FIX(q) (ATMFixed) (q * 65536.0)
   m.a= FIX (44*cos(25*DEGTORAD));     /* 44 pixel font           */
   m.b= FIX (-44*sin(25*DEGTORAD));    /* rotated 25 deg clockwise*/
   m.c= FIX (-44*sin(25*DEGTORAD));
   m.d= FIX (-44*cos(25*DEGTORAD));
   m.tx = 0;
   m.ty = 0;
```

Figure 4 *Rotated text*



Example 4: *Skewed text*

```
#define    DEGTORAD    (2*3.14159/360)  /*   2pi/360 */
#define    FIX(q)  (ATMFixed) (q * 65536.0)
m.a = FIX (44);                          /* 44 pixels in x*/
m.b = FIX ( -44*tan(15*DEGTORAD));       /* X axis 15 degrees CCW*/
m.c = FIX (73*tan(35*DEGTORAD));         /* Y axis 35 deg clockwise*/
m.d = FIX (-73);                         /* 73 pixels in y*/
m.tx = 0;
m.ty = 0;
```

Figure 5 *Skewed text*

2 ATM Software API Functions

These functions are presented in five sections.

- Section 2.1, "Getting Started," includes a discussion of **ATMProperlyLoaded( )**, **ATMGetVersion( )**, **ATMFontAvailable( )**, and **ATMFontSelected( )**.

- Section 2.2, "Text Showing," describes the use of **ATMXYShowText( )** and **ATMBBoxBaseXYShowText( )** and gives some tips on placing skewed text properly.

- Section 2.3, "Obtaining A Character's Outline," describes the use of **ATMGetOutline( )** and **ATMGetOutline2( )**.

- Section 2.4, "Path Rendering and Filling Routines," describe how the API can be used to render and fill arbitrary paths.

- Section 2.5, "Miscellaneous Functions," describes calls that were added in versions 1.1 and 2.0 of the ATM software. These calls alter the way ATM software creates fonts for the application.

2.1 Getting Started

The procedures described in this section are used to initialize the ATM software, determine the version of ATM software that is available, and determine which fonts are available to the application.

ATMProperlyLoaded

```
extern BOOL FAR PASCAL ATMProperlyLoaded ( void );
```

Version **1.0**

ATMProperlyLoaded( ) checks whether the ATM software is running and sets up structures internal to ATM software. **ATMProperlyLoaded( )** *must* be called before any other API calls are made.

ATMProperlyLoaded( ) returns a value that specifies whether or not the ATM software is running and the API structures are properly initialized. It is nonzero if the ATM software is initialized properly. Otherwise, it is zero.

Note *If calls are made to any ATM API functions before a call to* ***ATMProperlyLoaded( )*** *is made, the results are undefined. This might change in future versions of the ATM software.*

ATMGetVersion

```
extern WORD FAR PASCAL ATMGetVersion ( void );
```

Version **1.0**

ATMGetVersion( ) returns the currently running ATM software version number. This is useful if an application requires a specific version of the ATM software.

ATMGetVersion( ) returns the major and the minor version numbers. The major version number is in the low order byte and the minor version number is in the high order byte of the return value. For example, if the version is 1.15, then the major version number is 1 and the minor version number is 15.

ATMFontAvailable

```
extern BOOL FAR PASCAL ATMFontAvailable (
    LPSTR       lpFacename,
    int         nWeight,
    BYTE        cItalic,
    BYTE        cUnderline,
    BYTE        cStrikeOut,
    BOOL FAR*lpFromOutline );
```

Version **1.0**

ATMFontAvailable( ) checks whether a Type 1 font outline is available. Be aware that **ATMFontAvailable( )** does not parse the font to assure that it is a valid Type 1 font definition. Note that all parameters except **lpFromOutline* correspond to parameters passed to the Windows function **CreateFont( )**.

lpFacename is a long pointer to the face name of the font to be checked for. If *lpFacename* does not match one of the face names in the Fonts section of the *ATM.INI* file, **ATMFontAvailable( )** will return false.

Note that in the current version of the ATM software, **ATMFontAvailable( )** does not look at the face names in the Aliases section of the *ATM.INI* file. Therefore, although ATM software can image a font when an aliased name such as Tms Rmn is requested, **ATMFontAvailable( )** will return false if information about the aliased face name is requested. This will change in later versions of ATM software. See the *Adobe Type Manager User's Guide* for more information on the *ATM.INI* file.

nWeight is the weight of the font whose availability is being queried. *nWeight* corresponds to the *nWeight* value passed to the GDI **CreateFont( )** function. Creating a *logical* font with a weight value of 0 to 500 is recognized as a regular weight font by the ATM software. Weights 501 and up are considered bold weights by the ATM software.

Note *There is a bug in the ATM software version 1.0 API function* ***ATMFontAvailable( )****.* ***ATMFontAvailable( )*** *erroneously considers weights 401 and above bold even though the ATM software correctly considers weights of 501 and above bold weights.*

cItalic flag should be set to one if the query is for an italic outline. Otherwise, the flag should be set to zero.

cUnderline flag is not currently in use but is included for completeness and possible future use. To assure future compatibility, *cUnderline* should be set to one if the desired font is to be underlined. Otherwise, *cUnderline* should be set to zero.

cStrikeOut flag is not currently in use but is included for completeness and possible future use. To assure future compatibility, *cStrikeOut* should be set to one if the desired font is to be struck out. Otherwise, *cStrikeOut* should be set to zero.

lpFromOutline flag will be set to one if the font being queried will be rendered directly from an outline and not partially synthesized by ATM software. Otherwise, **lpFromOutline* will be set to zero. This is important if an application program needs to know if the actual requested outline is available or if the ATM software is going to use the "regular" outline and synthesize the bold or italic attributes.

ATMFontAvailable( ) will return 0 if the ATM software cannot find a font outline suitable for imaging the request. Otherwise, 1 is returned.

ATMFontSelected

```
extern BOOL FAR PASCAL ATMFontSelected ( HDC hDC )
```

Version **1.0**

ATMFontSelected( ) can be used to determine whether the currently selected font for the device context selected by *hDC* will be imaged by ATM software.

hDC specifies the device context to be queried.

ATMFontSelected( ) will return 1 if the font will be imaged by the ATM software. Otherwise, 0 is returned.

2.2 Text Showing

The procedures described in this section are used for showing text.

ATMXYShowText

```
extern BOOL FAR PASCAL ATMXYShowText (
    HDC          hDC,
    int          x,
    int          y,
    WORD         wOptions,
    LPRECT       lpRect,
    LPSTR        lpString,
    int          nCount,
    LPATMFixedPoint     lpPoints,
    LPATMFixedMatrix    lpMatrix );
```

Version **1.0**

Note *There is a bug in version 1.15. The lpRect parameter can't be NULL, and any text outside the rectangle will get clipped in "y."*

ATMXYShowText( ) can be used to image type in any orientation, skew, and scale, including non-uniform scaling. **ATMXYShowText( )** is very similar to the GDI function **ExtTextOut( )** with the following exceptions:

- As of ATM version **2.1**, the *wOptions* parameter may include the values *ATM_USELOGCOLOR, ATM_USEPURECOLOR* or *ATM_MAKEVISIBLE* which cause ATM to override the settings in the ATM.INI file.
- Opaquing text is not supported. That is, if *wOptions* includes the ETO_OPAQUE option, it will be ignored.
- **ATMXYShowText( )** has two extra parameters, *lpPoints* and *lpMatrix*.

hDC identifies the device context.

x specifies the *logical* x-coordinate of the origin of the character cell for the first character in the specified string.

y specifies the *logical* y-coordinate of the origin of the character cell for the first character in the specified string.

wOptions can be set to *ETO_CLIPPED, ATM_USELOGCOLOR, ATM_USEPURECOLOR, ATM_MAKEVISIBLE* or *NULL*.

ETO_CLIPPED specifies that Windows will clip text to the rectangle *lpRect*.

ATM_USELOGCOLOR specifies that ATM should use the logical color set for text foreground even when it would normally switch to the nearest pure color based on the settings in the ATM.INI. *ATM_USELOGCOLOR* is mutually exclusive with *ATM_USEPURECOLOR*.

ATM_USEPURECOLOR specifies that ATM should use the nearest pure (device) color instead of the actual logical color set for text foreground. *ATM_USEPURECOLOR* is mutually exclusive with *ATM_USELOGCOLOR*.

ATM_MAKEVISIBLE is used for B/W devices and causes ATM to use either white or black depending on the color of background in order to make the text visible. If *ATM_MAKEVISIBLE* is specified, then *ATM_USELOGCOLOR* and *ATM_USEPURECOLOR* are ignored.

Default settings in the ATM.INI file cause ATM to use logical colors for printers and pure colors for screens.

lpRect points to a *RECT* data structure that contains the *logical* coordinates of a rectangle. The lpRect parameter can be NULL. The effect of the rectangle depends on how the *wOptions* parameter is set.

lpString points to the character string to be rendered.

nCount specifies the number of characters in *lpString* to be rendered.

lpPoints points to an array of x and y *device* coordinate values that indicate the distance in x and y between the origins of adjacent characters. The *lpPoints* parameter can be *NULL*.

This array is a series of *ATMFixedPoint* values representing x and y widths for the characters in the string being imaged. This allows for both track kerning and pairwise kerning, as well as special effects. If the *lpPoints* parameter is *NULL*, the default x and y character widths will be used.

Note that the *lpPoints* array differs from the similar array passed to the GDI function **ExtTextOut( )**. This array contains widths for both x and y. Additionally, the units used here are *device* pixels rather than *logical* units as in the similar **ExtTextOut( )** parameter.

Note *In the Japanese version of Windows, the character width for a double-byte encoded Japanese character is considered to be the sum of two integers in the array pointed by the last parameter being passed to* ***ExtTextOut( )****. While each of the integers corresponds to a single-byte character width in* ***ExtTextOut()****,* ***ATMXYShowText( )*** *uses only the first of a LPATMFixedPoint pair. Therefore, if a character width is to be specified for a bouble-byte Japanese character, that width needs to be specified in the first of the LPATMFixedPoint pair.*

lpMatrix points to a transformation matrix of type *ATMFixedMatrix* as described in section 1.2, “Transformation Matrix”. This transformation matrix will be applied to the string being imaged. The *lpMatrix* parameter can be *NULL* in versions of ATM software after 1.1.

Note *In ATM software versions 1.0 and 1.1, lpMatrix cannot be NULL. In subsequent versions, if lpMatrix is NULL then the font will be used unmodified.*

ATMXYShowText( ) will return 1 if the string is drawn. Otherwise, 0 is returned.

In ATM software versions higher than 1.1, if both *lpMatrix* and *lpPoints* are set to *NULL*, then the call is handled as an **ExtTextOut( )** call.

When creating a *logical* font for use with **ATMXYShowText( )**, the *lfHeight* parameter to **CreateFont( )** should be specified as –1. As described below, this allows ATM software to provide optimal placement when displaying skewed or rotated text. This can be avoided by using **ATMBBoxBaseXYShowText( )**, which is available as of version **1.15** of ATM software.

An *lfHeight* of –1 (1 unit character height instead of cell height—see the *Microsoft Windows Software Development Kit Reference* for more information) is needed because ATM software must be able to determine the position of the baseline of the font.

When GDI **TextOut( )** operations are performed, the baseline *y* coordinate is calculated by adding the ascent value for the font to the *y* coordinate of the origin specified in the **TextOut( )** call. When **ATMXYShowText( )** is used, the presence of an arbitrary transformation matrix renders the original ascent value invalid. Therefore, translation from the *y* value specified in the call to the *y* coordinate of the baseline will be meaningless unless the ascent value can be made into a very small predictable number. This is achieved by creating a 1-pixel high font. Effectively, this causes the upper left corner of the font to coincide with its baseline. Thus, **ATMXYShowText( )** receives the coordinates of the baseline.

x and *y* are the coordinates of the upper left of the text box without taking the matrix into consideration. In other words, (x, y) defines the origin of a normal **TextOut( )** or **ExtTextOut( )** call, which would use the font unmodified (that is, with default matrix). This means that the baseline position is determined relative to the unmodified font.

Also note that *x* and *y* are expressed in *logical* units—the same as **TextOut( )** and **ExtTextOut( )**. However, the character widths passed in *lpPoints* are expressed in *device* units and are not subject to the current mapping mode in effect for the *hdc*. This means that for devices capable of rendering graphics at lower than normal resolutions (such as the PCL driver's ability to switch to 150 or 75 dpi), these widths are expressed at the graphics resolution in effect.

ATMBBoxBaseXYShowText

```
extern BOOL FAR PASCAL ATMBBoxBaseXYShowText (
    HDC              hDC,
    int              x,
    int              y,
    WORD             wOptions,
    LPRECT           lpRect,
    LPSTR            lpString,
    int              nCount,
    LPATMFixedPoint  lpPoints,
    LPATMFixedMatrix lpMatrix,
    BOOL             bDoOutput,
    LPATMBBox        lpBBox,
    LPATMFixedPoint  lpDelta);
```

Version **1.5**

Note *There is a bug in version 1.5. The lpRect parameter cannot be NULL and any text outside the rectangle will get clipped in "y."*

ATMBBoxBaseXYShowText( ) can be used to image type in any orientation, skew, and scale, including non-uniform scaling. Except for the *x* and *y* parameters and the additional three parameters *bDoOutput*, *lpBBox*, and *lpDelta*, is similar to the **ATMXYShowText( )** function. Note however, that the *x* and *y* parameters specify the *baseline* location rather than the upper left corner of the character box.

DC identifies the device context.

x specifies the *logical* x-coordinate of the start of the *baseline* of the specified string.

y specifies the *logical* y-coordinate of the start of the *baseline* of the specified string.

wOptions can be set to *ETO_CLIPPED, ATM_USELOGCOLOR, ATM_USEPURECOLOR, ATM_MAKEVISIBLE* or *NULL*.

ETO_CLIPPED specifies that Windows will clip text to the rectangle *lpRect*.

ATM_USELOGCOLOR specifies that ATM should use the logical color set for text foreground even when it would normally switch to the nearest pure color based on the settings in the ATM.INI. *ATM_USELOGCOLOR* is mutually exclusive with *ATM_USEPURECOLOR*.

ATM_USEPURECOLOR specifies that ATM should use the nearest pure (device) color instead of the actual logical color set for text foreground. *ATM_USEPURECOLOR* is mutually exclusive with *ATM_USELOGCOLOR*.

ATM_MAKEVISIBLE is used for B/W devices and causes ATM to use either white or black depending on the color of background in order to make the text visible. If *ATM_MAKEVISIBLE* is specified, then *ATM_USELOGCOLOR* and *ATM_USEPURECOLOR* are ignored.

Default settings in the ATM.INI file cause ATM to use logical colors for printers and pure colors for screens.

lpRect points to a RECT data structure. The *lpRect* parameter can be *NULL*. The effect of the rectangle depends on how the *wOptions* parameter is set. *lpRect* is expressed in *logical* units.

lpString points to the character string to be rendered.

nCount specifies the number of characters in *lpString* to be rendered.

lpPoints points to an array of x and y *device* coordinate values that indicate the distance in x and y between the origins of adjacent characters. The *lpPoints* parameter can be *NULL*.

Note *See description of lpPoints under* ***ATMXYShowText( )*** *about how this parameter is treated by ATM software Japanese version.*

This array is a series of *ATMFixedPoint* values representing x and y widths for the characters in the string being imaged. This allows for both track kerning and pairwise kerning, as well as special effects. If the *lpPoints* parameter is *NULL*, the default x and y character widths will be used.

Note that the *lpPoints* array differs from the similar array passed to the GDI function **ExtTextOut( )**. This array contains widths for both x and y. Additionally, the units used here are *device* pixels rather than *logical* units as in the similar **ExtTextOut( )** parameter.

pMatrix points to a transformation matrix of type *ATMFixedMatrix* as described in section 1.2, “Transformation Matrix.” This transformation matrix will be applied to the string being imaged. The *lpMatrix* parameter can be *NULL* in which case the font will be used unmodified as it was created by the GDI.

bDoOutput specifies whether or not output is produced. If *bDoOutput* is TRUE then output is produced. If *bDoOutput* is FALSE then no output is produced. This is useful if just the *lpBBox* or *lpDelta* information is needed.

lpBBox points to a structure that will receive the bounding box (or extent) of the string in *logical* units. If *NULL* is passed in, then the bounding box information is not returned.

lpDelta will receive the deltas that should be applied to *x* and *y* to calculate the new *x*' and *y*' parameters to the subsequent call to **ATMBBoxBaseXYShowText( )**. Using this *x*', *y*' will assure that the string will continue at the correct point on the baseline. *lpDelta* is returned in *device* units. If *NULL* is passed in, then the delta is not returned.

ATMBBoxBaseXYShowText( ) will return 1 if it executes properly. Otherwise, 0 is returned.

Example 5:

```
/* Produces skewed text */

•••
#define    DEGTORAD (2*3.14159/360)          /*   2pi/360 */
#define    FIX(q)   (ATMFixed) (q * 65536.0)
HDC     hDC;
HFONT   hFont;
BOOL    showResult;
ATMFixedMatrix matrix;
static char someText [] = "Skewed";
int x = 100, y = 100;       /* text will start at 100, 100*/
hWnd = CreateWindow(•••);   /* get a window handle*/
hDC = GetDC(hWnd));         /* get the device context to write to*/
hFont = CreateFont(
   -1,               /* font height- use -1 with ATMXYShowText   */
    0,               /* width      */
    0,               /* escapement */
   •••               /* fill in the rest of the params       */
   "Times");         /* Font name - a Type 1 font outline    */

matrix.a = FIX (44);                    /* 44 pixels in x     */
matrix.b = FIX ( -44*tan(15*DEGTORAD)); /* X axis 15 degrees CCW */
matrix.c = FIX (73*tan(35*DEGTORAD));   /* Y axis 35 deg CW   */
matrix.d = FIX (-73);                   /* 73 pixels in y     */
matrix.tx = (ATMFixed)0;
matrix.ty = (ATMFixed)0;

showResult = ATMXYShowText(hDC,x,y,NULL,(LPRECT) NULL,
      someText, sizeof(someText)-1,
      (LPATMFixedPoint) NULL,
      &matrix);
```

Figure 6

2.3 Obtaining a Character's Outline

An application has the capability to get, using **ATMGetOutline( )** and call-back routines, the PostScript language Type 1 font definition of a character outline. The outline is returned as a series of points in *device* space. These points can then be manipulated and used to reconstruct the altered character using **moveto**, **lineto**, **curveto**, and **closepath** operations. **ATMGetOutline( )** works similarly to the PostScript language operator **pathforall**.

ATMGetOutline

```
extern int FAR PASCAL ATMGetOutline (
    HDC                 hDC,
    char                c,
    LPATMFixedMatrix    lpMatrix,
    FARPROC             lpProcMoveTo,
    FARPROC             lpProcLineTo,
    FARPROC             lpProcCurveTo,
    FARPROC             lpProcClosePath,
    LPSTR               lpData );
```

Version **1.0**

ATMGetOutline( ) allows an application to retrieve a character's font outline.

A PostScript language Type 1 font character can be represented by a series of the four basic elements of a path, which are the PostScript language operators **moveto**, **lineto**, **curveto**, and **closepath**. An application program supplies **ATMGetOutline( )** with a call-back routine for each of the four elements. Each time **ATMGetOutline( )** processes one of these elements, it will execute the corresponding application-supplied call-back routine, handing the routine *lpData* and the data points that correspond to the element being processed. The application-supplied call-back routine can then manipulate or store the data.

hDC is the device context handle that identifies the font to be used.

c is the character whose outline is to be retrieved.

lpMatrix is the transformation matrix that the application wants ATM software to apply to the character before the data is returned. This matrix is of the same form as the matrix used with **ATMXYShowText( )** and is described in section 1.2, "Transformation Matrix."

lpProcMoveTo, *lpProcLineTo*, *lpProcCurveTo*, and *lpProcClosePath* are long pointers to the application- supplied call-back routines that will be called by **ATMGetOutline( )**. See below for more information about the call-back functions.

lpData is a long pointer to a data string that the application program can use as a message. This pointer is passed by **ATMGetOutline( )** to the application supplied call-back routines when they are executed. *lpData* can be *NULL*.

ATMGetOutline( ) returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned. If the selected font is a Japanese font in which outline data is protected,
ATM_INVALIDFONT is returned. See the error code listing in Appendix A.

The call-back routines correspond to the PostScript language operators **moveto**, **lineto**, **curveto**, and **closepath**. The call-back routines should return 1 if they are to continue being called by **ATMGetOutline( )** for the remainder of the current character. They should return 0 if they want to terminate **ATMGetOutline( )**.

As shown in the following call-back prototype example, all the call-back routines take arguments that include *lpData* and the appropriate number of pointer(s) to *ATMFixedPoint* points. The **moveto** and **lineto** call-backs each get one *ATMFixedPoint* parameter, the **curveto** call-back gets three *ATMFixedPoint* arguments, and **closepath** requires no *ATMFixedPoint* arguments. The call-back routines passed to **ATMGetOutline( )** correspond to the procedure arguments passed to the PostScript language operator **pathforall**.

Note that **ATMGetOutline( )** returns paths in the same order in which the font designer created the character. This means that an application can not depend on information such as the outer path being returned before the inner path. Additionally, be aware that the path returned by **ATMGetOutline( )** does not incorporate any hints that might be in the font outline description.

A typical use of the data returned by **ATMGetOutline( )** might be to have the application alter the data before sending the new data to the ATM software fill routines or to use the data to create some type of wire frame (that is, a 3-D font) and image the frame using GDI functions. (See section 2.4, "Path Rendering and Filling Routines.")

The call-back procedure prototypes should look something like this:

```
BOOL FAR PASCAL ClosePath ( LPSTR lpData )
BOOL FAR PASCAL MoveTo ( LPATMFixedPoint lpFixPt, LPSTR lpData )
BOOL FAR PASCAL LineTo ( LPATMFixedPoint lpFixPt, LPSTR lpData )
BOOL FAR PASCAL CurveTo( LPATMFixedPoint lpFixPt1,
   LPATMFixedPoint lpFixPt2,
   LPATMFixedPoint lpFixPt3,
   LPSTR lpData );
```

where the *LPATMFixedPoint* parameters are the points passed by **ATMGetOutline( )** to the call-back functions and *lpData* is the message that the application program has **ATMGetOutline( )** pass to the call-back functions.

Note *As with all call-back functions in Windows, the call-back parameters to **ATMGetOutline( )** must be listed in the exports section of the module definition (.DEF) file.*

Example 6:

```
•••
Handle  ghInst;
BOOL    result;
ATMFixedMatrix gMatrix = {ATMINTTOFIXED(100),0,0,
                          ATMINTOFIXED(–100),0,0};
HDC     hDC;
HFONT   hFont;
BOOL    showResult;
ATMFixedMatrix matrix;
char    myData[] = "Hello";
hWnd = CreateWindow(•••);           /* get a window handle*/
hDC  = GetDC(hWnd)); /* get the device context to write to*/

hFont = CreateFont(
   –1,   /* font height, use –1   */
   0, 0, /* width, escapement       */
   •••
   "Times");   /* Font name – a Type 1 font outline */

BOOL FAR PASCAL MyLineTo (LPATMFixedPoint lpFixPnt, LPSTR lpData)
{
   /* manipulate and/or save the points lpFixPnt for use with GDI
   /* functions or for use with the ATMFILL fns.(See the     */
   /* section "Path Rendering and Filling Routines" ) */
}
```

```
/* make 3 more fns as above for lineto, curveto & closepath */
•••

/* get proc instance address */
lpMoveTo = MakeProcInstance( MyMoveTo, ghInst);
lpLineTo = MakeProcInstance( MyLineTo, ghInst);
lpCurveTo = MakeProcInstance( MyCurveTo, ghInst);
lpClosePath = MakeProcInstance( MyClosePath, ghInst);
•••
result = ATMGetOutline ( HdC, 'A', &gMatrix, lpMoveTo,
       lpLineTo,
       lpCurveTo,
       lpClosePath,
       (LPSTR) myData );
•••

/* use the saved data */
```

ATMGetOutline2

```
extern int FAR PASCAL ATMGetOutline2 (
   HDC                  hDC,
   short                c,
   LPATMFixedMatrix     lpMatrix,
   FARPROC              lpProcMoveTo,
   FARPROC              lpProcLineTo,
   FARPROC              lpProcCurveTo,
   FARPROC              lpProcClosePath,
   LPSTR                lpData );
```

Version **2.5** Japanese version only

ATMGetOutline2( ) allows an application to retrieve a character's font outline, for a character with a double-byte character code. **ATMGetOutline2( )** is a variant of **ATMGetOutline( )** with the same functionality and error codes described above. **ATMGetOutline2( )** obtains outline data not only from Japanese fonts, specified using *SHIFTJIS_CHARSET*, but also from non-Japanese single-byte encoded fonts. In such case, **ATMGetOutline2( )** works exactly like **ATMGetOutline( )**.

All the parameters are the same, except the second. With **ATMGetOutline2( )**, *c* is a double-byte character code that represents a Japanese character whose outline is to be retrieved.

The encoding scheme used is *Shift-JIS* encoding. The character set used depends upon the PC platform on which application software is developed. ATM software Japanese version supports three OEM-dependent character sets: 78JIS NEC Extended, 83JIS Fujitsu Additional, and 83JIS. These character sets are used on NEC PCs or NEC PC-compatibles, Fujitsu PCs, and IBM PC and IBM PC-compatibles, respectively. The character code is interpreted the same way **TextOut( )** would interpret characters in the character array passed to it, as the properties of the currently selected font are taken into account.

ATMGetOutline2( ) returns *ATM_NOERR* if the call was successful, after it has enumerated the character path by calling the callbacks as described for **ATMGetOutline( )**. Otherwise, one of the ATM software error codes is returned. When **ATMGetOutline2( )** is invoked to obtain outline data for a character in Japanese font whose outline data is protected, the error code *ATM_INVALIDFONT* is returned. See the error code listing in Appendix A.

2.4 Path Rendering and Filling Routines

Several routines allow an application to use the ATM software to create and fill arbitrary paths. The ATM software fill routines **ATMFillStart( )** and **ATMFillEnd( )** must surround all calls to the path construction routines **ATMFillMoveTo( )**, **ATMFillLineTo( )**, **ATMFillCurveto( )**, **ATMFillClosePath( )**, and **ATMFillGetBBox( )**. The ATM software fill routines return *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned (see Appendix A). Note that these fill routines closely parallel the PostScript language operators **moveto**, **lineto**, **curveto**, **closepath**, and **eofill**.

The path is rendered and filled by the call **ATMFillEnd(** *hDC*, *x*, *y*, *rop* **)** using the GDI function **BitBlt( )**. Therefore, like **BitBlt( )** the result will be determined by a combination of the pixels turned on by the fill operators, the Brush selected for the destination device context specified by *hDC*, the pixels previously turned on in the device context, the mapping mode selected for the device context, and the raster operation specified by the parameter *rop*. See the *Windows Programmer's Reference* included with the Windows SDK for information on the available raster operations.

The following are concepts used in this section:

- *point* – all references to a *point* in this discussion refer to a point in the *device* coordinate system of the device context selected by *hDC*.

- *current path* – the path being built with the ATM software fill operators since the last call to **ATMFillStart( )**.

- *subpath* – a sequence of connected segments. The current path consists of one or more disconnected subpaths. The operator **ATMFillClosePath( )** explicitly connects the end of a subpath back to the subpath's starting point. Such a subpath is said to be closed.

- *current point* – the trailing endpoint of the most recently constructed path segment. The path construction operators **ATMFillLineTo( )**, **ATMFillCurveTo( )**, and **ATMFillClosePath( )** add a segment to the current path starting at the current point. If no current point exists, a call to one of these operators will cause an error.

The following *ATMfill* functions are available in all versions since 1.0.

Note *A bug in the ATM* software *API versions 1.0 and 1.1 prevented the coordinates passed into these routines from being interpreted correctly in any mode other than MM_TEXT. Versions 1.15 and higher work correctly in all mapping modes although we recommend using MM_TEXT to minimize round off errors.*

ATMFillStart

```
extern int FAR PASCAL ATMFillStart ( void );
```

ATMFillStart( ) places the ATM software in a state ready to accept the rendering commands **ATMFillMoveTo( )**, **ATMFillLineTo( )**, **ATMFillCurveto( )**, and **ATMFillClosePath( )**.

ATMFillStart( ) returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned.

ATMFillMoveTo

```
extern int FAR PASCAL ATMFillMoveTo (LPATMFixedPoint lpPnt );
```

ATMFillMoveTo( ) establishes a new current point without adding a segment to the current path.

lpPnt represents a *device* x,y coordinate to move to. **ATMFillMoveTo( )** starts a new subpath of the current path and sets the current point to x,y.

ATMFillMoveTo( ) returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned.

Note ***ATMFillMoveTo( )*** *must be called at least once to establish a starting point before any calls to* ***ATMFillLineTo( )*** *or* ***ATMFillCurveTo( )*** *are executed.*

ATMFillLineTo

```
extern int FAR PASCAL ATMFillLineTo ( LPATMFixedPoint lpPnt  );
```

ATMFillLineTo( ) takes a pointer to an *ATMFixedPoint*, which represents a *device* x,y coordinate and appends a straight line segment from the current point to the point (x,y). The point (x,y) becomes the new current point.

lpPnt specifies the *device* x,y coordinates of the line endpoint.

ATMFillLineTo( ) returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned.

ATMFillCurveTo

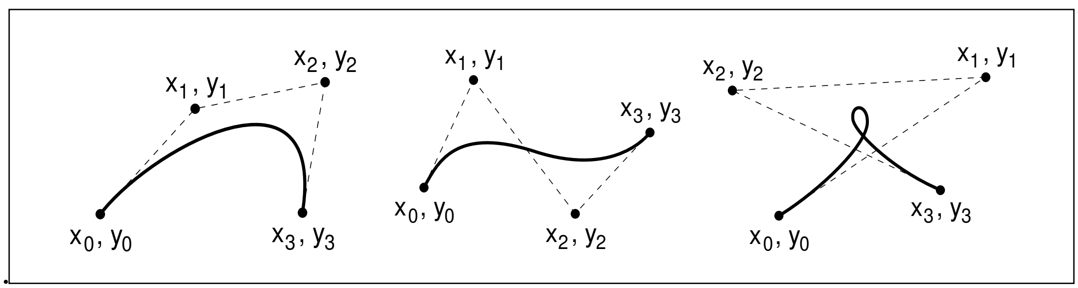
```
extern int FAR PASCAL ATMFillCurveTo (
    LPATMFixedPoint   lpPoint1,
    LPATMFixedPoint   lpPoint2,
    LPATMFixedPoint   lpPoint3 );
```

ATMFillCurveTo( ) adds a section of a Bézier cubic curve to the current path. It takes three pointers to *ATMFixedPoint* points, which represent the locations (x1,y1), (x2,y2), and (x3,y3) and adds a Bézier cubic section to the current path between the current point, referred to below as (x0, y0) and the point (x3, y3), using (x1, y1) and (x2, y2) as the Bézier cubic control points. The point (x3,y3) becomes the new current point. **ATMFillCurveTo( )** is similar to the PostScript language **curveto** operator.

lpPoint1, *lpPoint2*, and *lpPoint3* specify the *device* x,y coordinates that represent Bézier control points (*lpPoint1* and *lpPoint2*) and an endpoint (*lpPoint3*).

ATMFillCurveTo( ) returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned

Figure 7 *Curves showing Bézier control points and end points*



The mathematical formulation of a Bézier cubic curve is derived from a pair of parametric cubic equations:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + x_0$$
$$y(t) = a_y t^3 + b_y t_2 + c_{yt} + y_0$$

The points returned by **ATMFillCurveTo( )** represent the path traced by x(t) and y(t) as t ranges from 0 to 1. The Bézier control points corresponding to this curve are

$$x_1 = x_0 + c_x/3 \qquad y_1 = y_0 + c_y/3$$
$$x_2 = x_1 + (c_x + b_x)/3 \qquad y_2 = y_1 + (c_y + b_y)/3$$
$$x_3 = x_0 + c_x + b_x + a_x \qquad y_3 = y_0 + c_y + b_y + a_y$$

ATMFillClosePath extern int FAR PASCAL ATMFillClosePath (void)

ATMFillClosePath( ) closes the current subpath by appending a straight line segment connecting the *current point* to the subpath's starting point (generally the point most recently specified by **ATMFillMoveTo( )**. **ATMFillClosePath( )** returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned.

ATMFillEnd

```
extern int FAR PASCAL ATMFillEnd (
    HDC      hDC,
    int      x,
    int      y,
    DWORD    rop);
```

Version **1.0**

ATMFillEnd( ) paints the area enclosed by the current path (the path drawn since the last **ATMFillStart( )** according to the *Brush* selected in the device context specified by *hDC* and the GDI raster operation specified by *rop*. The inside of the current path is determined by the PostScript language even-odd rule. This is similar to the GDI *ALTERNATE* fill mode. See the *PostScript Language Reference Manual, Second Edition* for details of the even-odd rule.

hDC identifies the device context.

x specifies the *logical* x coordinate of the origin of the image to be drawn. Note that the origin is the upper-left corner of the image bounding rectangle.

y specifies the *logical* y coordinate of the origin of the image to be drawn. Note that the origin is the upper-left corner of the image bounding rectangle.

rop specifies the raster operation to be used. Any of the raster operations that can be used with the GDI function **BitBlt( )** can be used with **ATMEndFill( )**.

ATMEndFill( ) returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned.

ATMBaseFillEnd

```
extern int FAR PASCAL ATMBaseFillEnd (
    HDC      hDC,
    int      x,
    int      y,
    DWORD    rop);
```

Version **1.5**

ATMBaseFillEnd( ) paints the area enclosed by the current path (the path drawn since the last **ATMFillStart( )** according to the *Brush* selected in the device context specified by *hDC* and the GDI raster operation specified by *rop*. The inside of the current path is determined by the PostScript language even-odd rule. This is similar to the GDI *ALTERNATE* fill mode.

Note that this routine is different from **ATMFillEnd( )** only in the way that *x* and *y* are interpreted. In the case of this routine, *x* and *y* are the coordinates of the start of the *baseline* (point 0,0) for the object being drawn and filled.

hDC identifies the device context.

x specifies the *logical* x coordinate of the start of the baseline of the image to be drawn.

y specifies the *logical* y coordinate of the start of the baseline of the image to be drawn.

rop specifies the raster operation to be used. Any of the raster operations that can be used with the GDI function **BitBlt( )** can be used with **ATMBaseFillEnd( )**.

ATMBaseFillEnd( ) returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned.

ATMFillGetBBox

```
extern int FAR PASCAL ATMFillGetBBox (
    HDC          hDC,
    LPATMBBox    lpBBox );
```

Version **2.0**

ATMFillGetBBox( ) can be called before one of the *FillEnd* routines to obtain the bounding box and width information for the image being drawn.

hdc is the device context handle which identifies the device which will be imaged on.

See the *ATMBBox* structure in the section “Structures and Typedefs”.

lpBBox is a pointer to a structure which is a bounding box of the image about to be filled with one of the *FillEnd* routines. *lpBBox* is returned in *logical* units.

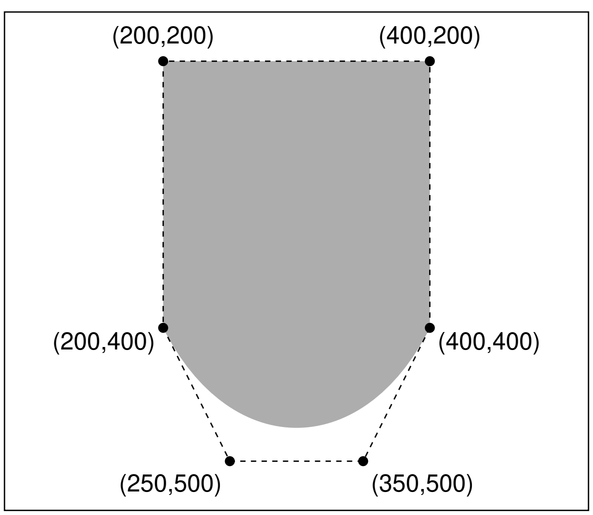
Note that **ATMFillGetBBox( )** should not be called for separate segments of an object being imaged. It should only be called once, immediately before **ATMFillEnd( )** or **ATMBaseFillEnd( )**.

ATMFillGetBBox( ) returns *ATM_NOERR* if the call was successful, otherwise, the appropriate ATM software error code is returned.

Example 7:

```
/* Produces output as in figure 7 */

•••
hDC = GetDC(hWnd);/* hWnd is the window to be drawn in*/
int x = 200,y = 200;    /* Draw with origin at 200,200 */
ATMFixedPoint p0, p1, p2;
ATMFillStart();

p0.x = (ATMFixed) (0);
p0.y = (ATMFixed) (0);
ATMFillMoveTo((LPATMFixedPoint) &p0);

p0.x = (ATMFixed) (0);
p0.y = ATMINTTOFIXED(200);
ATMFillLineTo((LPATMFixedPoint) &p0);

p0.x = ATMINTTOFIXED(50);
p0.y = ATMINTTOFIXED(300);
p1.x = ATMINTTOFIXED(150);
p1.y = ATMINTTOFIXED(300);
p2.x = ATMINTTOFIXED(200);
p2.y = ATMINTTOFIXED(200);
ATMFillCurveTo((LPATMFixedPoint) &p0,(LPATMFixedPoint) &p1,
               (LPATMFixedPoint) &p2);

p0.x = ATMINTTOFIXED(200);
p0.y = (ATMFixed) (0);
ATMFillLineTo((LPATMFixedPoint) &p0);

ATMFILLClosePath()
ATMFillEnd(hDC,x,y,SRCCOPY);
•••
```

Figure 8



In Figure 8, the gray-filled area was drawn by the code in Example 7. The points represent the line end points and the Bézier control points. The upper-left of the bounding rectangle is placed at (x,y) as specified in the ATMFillEnd( ) function.

2.5 Miscellaneous Functions

This section contains information on changing ATM's default actions when creating fonts and adding fonts on–the–fly.

Changing ATM's Default Actions

ATMSelectObject

```
extern int FAR PASCAL ATMSelectObject (
                    HD          hDC,
                    HANDLE      hNewObject,
                    WORD        options,
                    LPHANDLE    lphOldObject);
```

Version **2.01** and above, return values changed in version **2.5**.

Note ***ATMSelectObject()*** *was introduced to replace the collection of* ***ATMForceXXX( )*** *and the corresponding* ***ATMClearAllForce( )*** *routines. The obsolete* ***ATMForceXXX( )*** *functions are documented in "Appendix D".*

ATMSelectObject( ) allows applications to override some of ATM's default actions including font deferral as specified by the user in the ATM control panel and width data adjustment for the screen.

hDC is the DC into which the new object is being selected.

hNewObject is the handle to the object being selected into *hDC*.

options flags select the function that **ATMSelectObject( )** performs. Its values may be *ATM_DEFER, ATM_SELECT, ATM_USEEXACTWIDTH,* or *(ATM_SELECT | ATM_USEEXACTWIDTH)* which are defined below.

lphOldObject returns a handle to the previously selected object. If this parameter is *NULL*, the old object handle is not returned.

The *options* parameter flags are:

ATM_USEEXACTWIDTH

ATM_USEEXACTWIDTH creates a font without adjusting the width data for the screen. This has an effect only when a font is being created for a *display* device context.

In order to correct for the text justification behavior of some Windows applications, ATM renders screen text compressed by about 3% along the x axis. This may be undesirable for some of the more advanced text applications. An application may avoid this compression by using the *ATM_USEEXACTWIDTH* flag.

This flag is to be used in conjunction with the *GDI* text outputting functions such as **TextOut( )** and **ExtTextOut( )**. Regardless of how a font is created, ATM will not apply any compression to text rendered using the ATM API. That is, the matrix passed to **ATMXYShowText( )** or **ATMBBoxBaseXYShowText( )** is honored exactly as specified.

ATM_SELECT

ATM_SELECT causes the font to actually be created from an outline font even when normally deferring to a bitmap font would be possible.

When the "Use Pre-built or Resident Bitmap Fonts" checkbox in the ATM Control Panel is checked, ATM will defer to a device bitmap font if a font creation request can be satisfied with one. If an application desires a Type 1 font regardless of the status of this box, it should use the *ATM_SELECT* flag.

ATM_SELECT is the complement of *ATM_DEFER* and therefore they are mutually exclusive.

When using **ATMXYShowText( )**, it is recommended that an application create a 1 pixel high font (see the section "ATMShowText") in order to make the baseline and the upper left corner of character cells coincide.

Creating a 1 pixel high font would cause a problem in cases where ATM would normally defer to a screen bitmap for a GDI synonym font. For example, ATM would defer to GDI for "Helv" requests below 9 points (See *Adobe Type Manager – User Guide*). In such cases an application should use the ATM_SELECT flag.

Note *This specific issue can be avoided by using* ***ATMBBoxBaseXYShowText( )*** *instead of* ***ATMXYShowText( )****.*

ATM_DEFER

If the *ATM_DEFER* flag is used, regardless of the setting of the "Use Pre-built or Resident Bitmap Fonts" checkbox in the ATM Control Panel, ATM will defer to the device font. If the device font is not available, the application will get whatever the device substitutes for the requested font.

ATM_DEFER is the complement of *ATM_SELECT* and therefore they are mutually exclusive.

ATMSelectObject( ) returns the following:

Successful return value:

ATM_SELECTED	*hNewFont* successfully selected.

Note *ATM_SELECTED is the same return value as ATM_NEWFONTSELECTED in ATM versions **2.01** and **2.02**.*

Failure return values are:

ATM_NOT_ON	ATM is not running.
ATM_BADPARM	Conflicting *options* flags.
ATM_CANTHAPPEN	An internal error is encountered.
ATM_SELECTERROR	*hDC* or *hNewObject* is bad/invalid.

In versions **2.01** and **2.02**, **ATMSelectObject( )** returned the additional values:

ATM_NONFONTSELECTED, ATM_OLDFONTSELECTED, and *ATM_FOREIGNFONTSELECTED* which are now obsolete in ATM software Roman version 2.5. However, these return codes are still used in ATM software Japanese version 2.5.

Adding and Removing Fonts On–The–Fly

The following API functions allow font utility applications to make newly installed fonts available to ATM versions **2.5** and higher on–the–fly (no reboot of Windows). These calls update ATM's structures but don't make the corresponding changes to the WIN.INI or ATM.INI files. Neither do these functions install the .pfm and .pfb files on the users disk.

The .pfb and .pfm files must be properly installed on the user's disk before the **ATMAddFont( )** function can be used.

The ATM.INI file must be updated before the user exits the application. See the section "ATM.INI" below for information on the [Fonts] section of the ATM.INI file.

WIN.INI contains information on .pfm and .pfb files for some PostScript language printer drivers. The softfont entries in the WIN.INI file should be appropriately updated when installing fonts. If this isn't done, the user may be able to use the font with ATM but not with PostScript language Printers! Documentation on the WIN.INI settings can be found in the Microsoft® Windows™ Device Development Kit, Printers and Fonts Kit.

Note *Any call or series of calls to the* ***ATMAddFont( )*** *or* ***ATMRemoveFont( )*** *functions must be encapsulated between a set of* ***ATMBeginFontChange( )*** *and* ***ATMEndFontChange( )*** *functions. These functions are not supported in the Japanese version of ATM software.*

ATMAddFont

```
extern int  FAR PASCAL ATMAddFont (
        LPSTR          lpMenuName,
        WORD           style,
        LPSTR          lpMetricsFile,
        LPSTR          lpFontFile);
```

Version **2.5**. Not supported in Japanese version 2.5.

ATMAddFont( ) makes a newly installed font available to ATM.

lpMenuName contains the Windows name of the font.
style contains none, one or both of the flags: *ATM_ITALIC, ATM_BOLD*
lpMetricsFile contains the full pathname to the .pfm file for the font.
lpFontFile contains the full pathname to the .pfb file for the font.

ATMAddFont( ) will return *ATM_NOERR* if successful. Otherwise, one of the following will be returned:

ATM_INVALIDFONT	Invalid font format.
ATM_PATHTOOLONG	At least one path is too long.
ATM_BADFONTTYPE	The font is un-compressed.
ATM_BADSTYLE	The *style* parameter not specified as documented.
ATM_NOROOM	Can't add to internal tables.
ATM_NOFONT	At least one path doesn't point to the correct file.
ATM_BADMENUNAME	Missing or invalid menu name given.
ATM_FONTINUSE	The paths and menu name specify an existing font which is in use by ATM.

See Also: **ATMBeginFontChange( )** and **ATMEndFontChange( )**.

ATMRemoveFont

```
extern int FAR PASCAL ATMRemoveFont (
        LPSTR      lpMenuName,
        WORD       style);
```

Version **2.5**. Not supported in Japanese version 2.5.

This call makes a font inaccessible to ATM. Any subsequent requests to ATM for this font will fail.

lpMenuName contains the Windows name of the font.
style contains none, one or both of the flags:*ATM_ITALIC* and *ATM_BOLD.*

ATMRemoveFont( ) returns ATM_NOERR if successful. Otherwise, one of the following values are returned:

ATM_BADSTYLE	The style parameter has bad bits.
ATM_NOFONT	The specified font is not available.
ATM_BADMENUNAME	Missing or invalid menu name.

See Also: **ATMBeginFontChange( )** and **ATMEndFontChange( )**.

ATMBeginFontChange

```
extern int FAR PASCAL ATMBeginFontChange (void);
```

Version **2.5**. Not supported in Japanese version 2.5.

ATMBeginFontChange( ) starts a series of font change operations consisting of calls to the routines **ATMAddFont( )** and **ATMRemoveFont( )**.

ATMBeginFontChange( ) must be called before starting a call or series of calls to **ATMAddFont( )** and/or **ATMRemoveFont( )**.

ATMEndFontChange

```
extern int FAR PASCAL ATMEndFontChange (void);
```

Version **2.5**. Not supported in Japanese version 2.5.

ATMEndFontChange( ) finishes a series of font change operations consisting of calls to the routines **ATMAddFont( )** and **ATMRemoveFont( )**.

ATMEndFontChange( ) must be called after a call or series of calls to **ATMAddFont( )** and/or **ATMRemoveFont( )**.

ATMFontStatus

```
extern int FAR PASCAL ATMFontStatus (
        LPSTR    lpMenuName,
        WORD     style,
        LPSTR    lpMetricsFile,
        LPSTR    lpFontFile);
```

Version **2.5**. Not supported in Japanese version 2.5.

ATMFontStatus( ) determines whether or not a given font is currently installed and/or in use by ATM. It should be called prior to installing a new font.

If the return value is *ATM_FONTINUSE*, then no attempt should be made to install the font. That is, don’t copy over .pfm and .pfb files that are in use by ATM.

lpMenuName	contains the Windows name of the font.
style	contains none, one or both of the flags ATM_ITALIC and ATM_BOLD.

lpMetricsFile	Contains the full pathname to the .pfm file for the font.
lpFontFile	Contains the full pathname to the .pfb file for the font.

Successful return values are:

ATM_FONTINUSE	The paths and menu name specify an existing font which is in use by ATM.
ATM_FONTPRESENT	The paths and menu name specify an existing font which is NOT in use.
ATM_FONTDIFFERENT	The font, as specified by its menu name and styles, exists but its data files are different from those specified by *lpMetricsFile* & *lpFontFile* parameters.
ATM_FONTABSENT	There is no such font installed.

Failure return values are:

ATM_PATHTOOLONG	At least one path is too long.
ATM_BADFONTTYPE	The font is un-compressed.
ATM_BADSTYLE	The style parameter has bad bits!
ATM_NOFONT	At least one path doesn't point to the correct file.
ATM_BADMENUNAME	Missing or invalid menu name given.

The ATM.INI file

The *[fonts]* section of the atm.ini file must be updated appropriately when installing fonts. An entry must be made for each of the fonts added. The entry is as follows:

<Windows Menu Name>*[,STYLE]* = < full .pfm path name > , < full .pfb path name >

Where *STYLE* corresponds to the style flags used in the **ATMAddFont( )** function.

The following is an example of five different font entries:

```
[Fonts]

Courier=c:\psfonts\pfm\com_____.pfm,c:\psfonts\com_____.pfb
Courier,BOLD=c:\psfonts\pfm\cob_____.pfm,c:\psfonts\cob_____.pfb
Courier,BOLDITALIC=c:\psfonts\pfm\co____.pfm,c:\psfonts\co_____.pfb
Courier,ITALIC=c:\psfonts\pfm\coo_____.pfm,c:\psfonts\coo_____.pfb
ZapfDingbats=c:\psfonts\pfm\zd______.pfm,c:\psfonts\zd______.pfb
```

An entry must be removed when the font is removed.

Note *Please use **ATMFontStatus( )** to determines whether or not a given font is in use by ATM. It should be checked prior to installing new .pfb and .pfm files.*

Appendix A: ATM software Error Codes

A.1 ATM Software Error Codes

The following are ATM software error codes as defined in *ATM.h*. These codes are returned by the non-Boolean routines, for example, **ATMGetOutline( )** and the ATM software fill routines.

```
#define  ATM_NOERR  (0)                /* Normal return */
#define  ATM_INVALIDFONT(-1)           /* Font not consistent or outline data
                                          is protected */
#define  ATM_CANTHAPPEN (-2)           /* ATM internal error */
#define  ATM_BADMATRIX  (-3)           /* Invalid Matrix */
#define  ATM_MEMORY (-4)               /* Out of Memory */
```

A.1.1 ATM Software Error Codes Added in 1.15

The following error codes were added in version 1.15

```
#define  ATM_NOTSUPPORTED (-5)         /* Wrong version of ATM*/
#define  ATM_NOTRUNNING (-6)           /* ATM not running */
#define  ATM_FILLORDER  (-7)           /* Inconsistent fill calls */
#define  ATM_CANCELLED(-8)             /* Client halted operation */
#define  ATM_NOCHAR (-9)               /* No outline for char code */
#define  ATM_BADPROC(-100)             /* Bad callback proc addr. */
#define  ATM_CANTDRAW (-101)           /* Error in imaging, a bad ROP? */
```

A.1.2 ATM Software Error Codes Added in Version 2.5

The following error codes are used also in ATM software Japanese version 2.5.

```
#define  ATM_CANTDRAW (-101)           /* Error in imaging */
#define  ATM_BADPARAM (-102)           /* Bad input parameter */
#define  ATM_SELECTED (-200)           /* See ATMSelectObject() */
#define  ATM_SELECTERROR(-204)         /* See ATMSelectObject() */
```

Appendix B: More On Transformation Matrices

A clarification of the discussion about transformation matrices is in order for those familiar with the PostScript language or transformation matrices in general. In the PostScript language, the user space coordinate system is a standard Cartesian coordinate system with each unit exactly 1/72 inch. This is the same for all PostScript language output devices. The Current Transformation Matrix (CTM) performs the mapping of user space coordinates into device space coordinates. The device space coordinate system is that of the physical output device where 1 unit is equal to one device pixel and is therefore resolution dependent. In addition, the origin of coordinates and the orientation of the coordinate system frequently differs from the user space coordinate system.

A PostScript language program requests that a transformation be applied to the user space coordinate system by using the **rotate**, **scale**, or **translate** operators. The appropriate transformation matrix is then concatenated to the CTM to create a new CTM´ which corresponds to the mapping of the new coordinate system into device space. This new CTM´ is calculated as follows:

CTM´ = T x CTM

where *T* represents the transformation of the current user space into the new user space and the multiplication is a matrix multiplication.

Consider the default CTM necessary to map one unit of user space into a coordinate system which has a screen pixel of 1/72" In this case, there is no scaling necessary to map user space into device space. Additionally, assume that user space has a coordinate system in which *y* increases going down (as in the Windows video coordinate system), while the PostScript language coordinate system has *y* increasing up the screen. In order to account for this scaling, the *y* scale factor in the CTM must be -1. If we assume that the coordinate origins coincide, then the CTM should look like:

[1 0 0 –1 0 0]

To transform user space into a new user space, we multiply by our transformation *T*. In the PostScript language, the matrix *T* looks like:

$T = [\, Mx \quad (Mx \tan(a)) \quad (My \tan(b)) \quad My \quad 0 \quad 0\,]$

where *Mx* is the scale factor in *x*, *My* is the scale factor in *y* and α and β are the skew of the transformed *x* and *y* axes respectively. Figure 1 indicates this geometry.

When we concatenate *T* and the CTM we get

$CTM' = [\, Mx \quad (-Mx \tan(a)) \quad (My \tan(b)) \quad -My \quad 0 \quad 0\,]$

This is identical to the matrix we specified in equation 1 above (in the section "Transformation Matrix"). This is the matrix we pass to the **ATMXYShow-Text( )**, **ATMGetOutline( )** and **ATMGetOutline2( )** calls.

Note that the size of the font is built into *Mx* and *My* and is specified in pixels. If a font is to be displayed on screen in logical inches, and the user interface is specified in points (1/72"), then an application program can set *Mx* and *My* by scaling the user input for the *x* point size by *LOGPIXELSX* / 72.0 and by scaling input for the *y* point size by *LOGPIXELSY* / 72.0.

To calculate the matrix for rotated text, note that for *uniform* rotation, the angle $\beta = -\alpha$. If we write that $Mx=M\cos(a)$ and $My=M\cos(b)$ (*M* is the point size of the font we wish to rotate), then the CTM can be written as:

$CTM' = [\, M\cos(a) \quad -M\sin(a) \quad -M\sin(a) \quad -M\cos(a) \quad 0 \quad 0\,]$

just as indicated in the equations in section 1 of this document.

For those more interested in these transformations of coordinate spaces, see the *PostScript Language Reference Manual* ("the Red Book"), written by Adobe Systems Incorporated and published by Addison-Wesley. For a complete mathematical explanation of matrix operations and their uses in describing geometrical transformations, see *Principles of Interactive Computer Graphics* by W.M. Newman and R.F. Sproull, published by McGraw-Hill.

Appendix C: Changes Since Earlier Versions

Changes since April 14, 1993

- Added more descriptions relating to ATM software for Windows Japanese version: Section 2.2 under **ATMXYShowText( )**.

Changes since March 1, 1993

- Added descriptions relating to ATM software for Windows Japanese version.

- Added a new function **ATMGetOutline( )**.

Changes since February 14, 1992

- Update to the note on the *ATMBBox* structure.

- Changes to **ATMXYShowText( )** and **ATMBBoxBaseXYShowText( )**.

- Added new functions: **ATMSelectObject( )**, **ATMAddFont()**, **ATMRemoveFont( )**, **ATMBeginFontChange( )**, **ATMEndFontChange( )**, and **ATMFontStatus( )**.

- Appendix D, "Obsolete API Functions" was added.

Changes since August 9, 1991

- The document name changed from "Advanced Type Capabilities Using Adobe Type Manager Software For Windows" to "Adobe Type Manager Software API: Windows".

- The document now reflects various changes to the API as of ATM version 2.0. In addition to some bug fixes, two API functions were added. These include: **ATMForceDefer( )**, and **ATMClearAllForce( )**.

- The format of the document was updated.

- Various minor typographical errors were corrected.

Appendix: D: Obsolete API Functions

Note ***ATMSelectObject( )*** *was introduced to replace the collection of* ***ATMForceXXX( )*** *and the corresponding* ***ATMClearAllForce( )*** *routines which are documented here. Instead of calling one of the* ***ATMForceXXX( )*** *routines followed by a call to* ***SelectObject( )****, an application should call* ***ATMSelectObject( )*** *with the appropriate flag set in the options parameter. The Flags, ATM_DEFER, ATM_SELECT, and ATM_USEEXACTWIDTH specify the equivalent actions to* ***ATMForceDefer( )****,* ***ATMForceCreate( )****, and* ***ATMForceExactWidth( )****, respectively. It is not necessary to call* ***ATMClearAllForce( )*** *following a call to* ***ATMSelectObject( )****.*

The **ATMForceExactWidth( )**, **ATMForceCreate( )**, and **ATMForceDefer( )** functions alter the way logical fonts are created. **ATMClearAllForce( )** can be used to undo pending calls to the *ATMForce* functions.

A font is created when it is selected into a target device context (DC) and *not* when **CreateFont( )** or **CreateFontIndirect( )** is called. Therefore, an application must call the *ATMForce* calls immediately before calling **SelectObject( )** in order to achieve the desired effects. If a given font is selected into more than one DC, then the *ATMForce* function must be called before the first call to **SelectObject( )** for each DC.

The *first* time a particular `hFont` is selected into a specific DC, an application should call the appropriate *ATMForce* routines. On subsequent calls to **SelectObject( )** on the same `hFont` into the same DC, the application should not re-call the *ATMForce* routines. If an application is not tracking whether or not it is the first time the particular `hFont` is selected into the DC, then the application should use **ATMClearAllForce( )** in combination with the *ATMForce*, **SelectObject( )** sequence.

Since these calls affect the next font created, it is important that an application making these calls not give up its time slice between calling these functions and actually creating (selecting) the font. Doing so may have adverse effects on both the calling application and any other application that may create a font during the released time slice.

ATMForceExactWidth extern int BOOL PASCAL ATMForceExactWidth (*void*);

Version **1.1**

ATMForceExactWidth( ) causes the *next* Windows font creating call to create a font without adjusting the width data for the screen. This has an effect only when a font is being created for a *display* device context.

It is important that an application making this call not give up its time slice between calling this routine and actually creating (selecting) the font. Doing so may render the call to this routine void as another application may attempt to create a font which will be affected by the first application's call to **ATMForceExactWidth( )**.

In order to correct for the text justification behavior of some Windows applications, ATM renders screen text compressed by about 3% along the x axis. This may be undesirable for some of the more advanced text applications. An application may avoid this compression by first calling **ATMForceExactWidth( )** and then creating the desired font.

This function is to be used in conjunction with the GDI **TextOut( )** and **ExtTextOut( )** functions. Regardless of how a font is created, ATM will not apply any compression to text rendered using the ATM API. That is, the matrix passed to **ATMXYShowText( )** is honored exactly as specified.

If the application is not tracking the **SelectObject** calls as described in the introduction to this section, **ATMClearAllForce( )** must be called immediately after the call to **SelectObject** that attempts to fulfill the **ATMForceExactWidth( )** request.

ATMForceExactWidth( ) returns 1 if successful. Otherwise, 0 is returned.

ATMForceCreate extern int BOOL PASCAL ATMForceCreate (*void*);

Version **1.1**

#define **ATMForceSelect** ATMForceCreate

Version **1.15**

Note *The name **ATMForceSelect( )** is a more appropriate than **ATMForceCreate( )**, therefore, this MACRO was added in version 1.15. It is recommended that **ATMForceSelect( )** be used.*

ATMForceCreate( ) is the complement of **ATMForceDefer( )** and therefore mutually exclusive with **ATMForceDefer( )**. **ATMForceCreate( )** causes the *next* GDI font creating call to actually create a font from an outline font even when normally deferring to a bitmap font would be possible.

It is important that an application making this call not give up its time slice between calling this routine and actually creating (selecting) the font. Doing so may render the call to this routine void as another application may attempt to create a font which will be affected by the first application's call to **ATMForceCreate( )**.

When the "Use Pre-built or Resident Bitmap Fonts" checkbox in the ATM Control Panel is checked, ATM will defer to a device bitmap font if a font creation request can be satisfied with one. If an application desires an ATM font regardless of the status of this box, it may call **ATMForceCreate( )** (**ATMForceSelect( )**) before creating the font.

When using **ATMXYShowText( )**, it is recommended that an application create a 1 pixel high font (see the section "ATMShowText") in order to make the baseline and the upper left corner of character cells coincide.

Creating a 1 pixel high font would cause a problem in cases where ATM would normally defer to a screen bitmap for a GDI synonym font. For example, ATM would defer to GDI for "Helv" requests below 9 points (See *Adobe Type Manager – User Guide*). In such cases an application should first call **ATMForceCreate( )** (**ATMForceSelect( )**) and then create (select) its desired font.

If the application is not tracking the **SelectObject( )** calls as described in the introduction to this section, **ATMClearAllForce( )** must be called immediately after the call to **SelectObject( )** that attempts to fulfill the **ATMForceCreate( )** request.

ATMForceCreate( ) returns 1 if successful. Otherwise, 0 is returned.

ATMForceDefer

```
extern int BOOL PASCAL ATMForceDefer ( void );
```

Version **2.0**

ATMForceDefer( ) is the complement of **ATMForceCreate( )** and therefore mutually exclusive with **ATMForceCreate( )**. If **ATMForceDefer( )** is called, regardless of the setting of the "Use Pre-built or Resident Bitmap Fonts" checkbox in the ATM Control Panel, ATM will defer to the device font. If the device font is not available, the application will get whatever the device substitutes for the requested font.

It is important that an application making this call not give up its time slice between calling this routine and actually creating (selecting) the font. Doing so may render the call to this routine void as another application may attempt to create a font which will be affected by the first application's call to **ATMForceDefer( )**.

If the application is not tracking the **SelectObject( )** calls as described in the introduction to this section, **ATMClearAllForce( )** must be called immediately after the call to **SelectObject( )** that attempts to fulfill the **ATMForceDefer( )** request.

ATMForceDefer( ) returns 1 if successful. Otherwise, 0 is returned.

ATMClearAllForce

extern int BOOL PASCAL ATMClearAllForce (*void*);

Version **2.0**

ATMClearAllForce( ) undoes preceding calls to any and all of **ATMForceCreate( )**, **ATMForceDefer( )**, or **ATMForceExactWidth( )** whose effects are pending.

If the application is not tracking the **SelectObject** calls as described in the introduction to this section, **ATMClearAllForce( )** must be called immediately after the call to **SelectObject** that attempts to fulfill any *ATMForce* request.

ATMClearAllForce( ) returns 1 if successful. Otherwise, 0 is returned.

Index

B

C

E

L

T