```
//-----------------------------------------------------------------------------------
//  Filename: ADFTypeSystem.h
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Copyright 2004-2016 Mitsubishi Electric Research Laboratories (MERL)
//  Saffron API
//  Version 4.0
//  Ronald Perry, Sarah Frisken, and Eric Chan
//-----------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------
//  OVERVIEW
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Description:
//
//  This file defines the application programming interface (API) for the Adaptively
//  Sampled Distance Field (ADF) type rendering system (a.k.a. Saffron). Saffron is a
//  cross-platform (i.e., OS- and processor-independent) engine that provides
//  efficient, high quality, scalable type rendering with automatic hinting.
//
//  For an introduction to ADFs, see U.S. Patent 6,396,492 "Detail-Directed
//  Hierarchical Distance Fields", Sarah Frisken, Ronald Perry, and Thouis Jones
//  and "Adaptively Sampled Distance Fields: A General Representation of Shape for
//  Computer Graphics", SIGGRAPH 2000 Conference Proceedings, Sarah Frisken, Ronald
//  Perry, Alyn Rockwood, and Thouis Jones. A brief overview follows.
//
//  We define a 2-dimensional signed distance field D representing a closed
//  2-dimensional shape S (such as a glyph) as a mapping D:RxR-->R for all points p
//  contained in RxR such that D(p) = sign(p) * min{||p - q||: for all points q on
//  the zero-valued iso-surface (i.e., edge) of S}, sign(p) = {-1 if p is outside S,
//  +1 if p is inside S}, and ||.|| is the Euclidean norm. Less formally, the
//  distance field of a glyph simply measures the minimum distance from any point p
//  in space to the edge of the glyph, where the sign of this distance is negative if
//  p is outside the glyph and positive if p is inside the glyph. ADFs are generated
//  by performing a detail-directed sampling of the shape’s distance field wherein
//  the sampled values (such as distances and reconstruction data) are stored in a
//  data structure for efficient processing. Distances at arbitrary points in the
//  field can then be reconstructed from the sampled values and used for processing
//  such as rendering and editing. The use of adaptive detail-directed sampling
//  permits higher sampling rates in regions of fine detail and lower sampling rates
//  where the distance field varies smoothly, thereby allowing very high accuracy
//  with minimal memory requirements. ADFs, like TrueType outlines, can be rendered
//  at any size and orientation.
//
//  ADF glyphs are rendered using a new distance-based antialiasing algorithm (see,
//  for example, U.S. Patent 7,034,845 "Method for Antialiasing an Object Represented
//  as a Two-Dimensional Distance Field in Image-Order" by Ronald Perry and Sarah
//  Frisken, U.S. Patent 6,982,724 "Method for Antialiasing an Object Represented as
//  a Two-Dimensional Distance Field in Object-Order" by Sarah Frisken and Ronald
//  Perry, and U.S. Patent 6,917,369 "Method and Apparatus for Rendering Cell-Based
//  Distance Fields Using Texture Mapping" by Ronald Perry and Sarah Frisken). Rather
//  than computing a coverage value at each sample point, we reconstruct the distance
//  at each sample point and then map the distance to a density value. Coverage-based
//  approaches require either a complex and expensive analytic filter or many
//  supersamples to approximate the coverage value at each sample point. ADF
//  rendering requires just a single distance reconstruction per sample and provides
//  quality comparable to exact analytic methods.
//
//  During ADF rendering, strong vertical and horizontal edges of glyphs and
//  characteristic distances (e.g., cap heights) of the typeface can be aligned to
//  the pixel grid to provide better contrast, uniform stroke weights, and consistent
//  characteristic distances. This process, referred to as 'grid fitting' in this
//  API, uses a set of 'alignment zones' that we detect automatically from each ADF.
//  Standard Alignment Zones (SAZ), which are optimized for non-CJK glyphs, are
//  detected once per glyph in a preprocessing step. Multiple Alignment Zones (MAZ),
```

```
//  which are optimized for CJK glyphs, are detected dynamically during rendering.
//  Saffron also supports the rendering of application-hinted glyphs.
//
//  Glyphs are imported into Saffron as ADFPaths. ADFPaths can represent both
//  outline-based glyphs (e.g., from TrueType fonts) comprising a set of closed
//  contours and stroke-based glyphs comprising 1) a set of stroke skeletons that can
//  be either open or closed and 2) a set of stroke attributes, such as stroke width,
//  that determine how the stroke skeletons are rendered.
//
//  This API is structured as a toolkit, and includes the following major functional
//  blocks:
//
//      1) Conversion of ADFPaths to ADFs
//      2) Rendering ADFs as density images
//      3) Determining alignment zones directly from ADFs, thereby enabling grid
//         fitting during ADF rendering
//      4) A dual caching system for ADFs and density images
//
//  Saffron supports two types of ADFs for representing glyphs: explicit ADFs and
//  implicit ADFs. A compile time Boolean flag (ADF_USE_IMPLICIT_ADFS) controls the
//  selection of which ADF type to use throughout the system. The usage note Explicit
//  and Implicit ADFs (see below) describes both ADF types and provides guidance for
//  selecting which ADF type best suits the needs of the application.
//
//  Saffron supports both CRT and LCD (i.e., sub-pixel) rendering. During sub-pixel
//  rendering, each color component is treated as a separate sample; these samples
//  are then combined to determine all the color components for the pixel. Sub-pixel
//  rendering increases the effective resolution of the display, thus enabling higher
//  quality type on displays with addressable components (e.g., LCDs).
//
//  Throughout this document we will refer to traditional pixel-based rendering as
//  CRT rendering (i.e., CRT mode) and sub-pixel rendering as LCD rendering (i.e.,
//  LCD mode). Saffron can apply sub-pixel rendering not only to LCD displays, but
//  to any type of display with addressable sub-pixels such as organic LEDs. The
//  list of supported LCD modes (e.g., ADF_REND_MODE_RGBv) is described in the
//  rendering section of this API. Note that LCD rendering can be used on CRT
//  displays even though the pixel components are not individually addressable:
//  the additional distance field samples result in a cleaner edge which many users
//  prefer.
//
//
//  Table of Contents:
//
//  - Usage Notes
//      - Explicit and Implicit ADFs
//      - Basic Data Flow and Processing Steps
//      - Compositing Glyph Density Images
//      - Combining Glyph Density Images
//      - Coordinate Systems
//      - Continuous Stroke Modulation
//      - Color Reduction
//      - Typesetting
//      - Rendering CJK Glyphs With Multiple Alignment Zones
//      - Rendering Application-Hinted Glyphs
//  - API
//      - Saffron API Version Number
//      - ADF Type Selection (Explicit or Implicit)
//      - Library Component Selection
//      - Math Mode Selection
//      - CPU Byte Order Selection
//      - Memory Allocation
//      - Fundamental Data Types
//      - Initialization and Termination of the System
//      - Glyph Representation (ADFPaths)
//      - ADF Generation from ADFPaths
//      - Alignment Zones
//      - Density Images
```

```
//      - Rendering
//      - Dual Caching System
//  - Revision History
//-----------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------
//  USAGE NOTES
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: EXPLICIT AND IMPLICIT ADFS
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Saffron supports two types of ADFs for representing glyphs: explicit ADFs and
//  implicit ADFs. A compile time Boolean flag (ADF_USE_IMPLICIT_ADFS) controls the
//  selection of which ADF type to use throughout the system.
//
//  Explicit ADF generation uses top-down spatial subdivision to generate a spatial
//  hierarchy of explicit ADF cells, where each explicit ADF cell contains a set of
//  sampled distance values and a reconstruction method; explicit ADF rendering
//  reconstructs the distance field within each explicit ADF cell using its
//  reconstruction method and then maps the reconstructed distances to density
//  values.
//
//  In contrast, implicit ADF cells are not generated during generation (they are
//  generated on-demand during rendering). More specifically, implicit ADF generation
//  preprocesses an ADFPath (which represents a glyph); implicit ADF rendering
//  generates implicit ADF cells from the preprocessed ADFPath and renders each
//  implicit ADF cell by 1) reconstructing the distance field within the implicit ADF
//  cell using its reconstruction method and 2) mapping the reconstructed distances
//  to density values.
//
//  The following should be considered when selecting which ADF type to use:
//
//  1) Implicit ADFs support stroke-based glyphs; explicit ADFs do not.
//
//  2) Implicit ADFs support MAZ alignment zones; explicit ADFs do not.
//
//  3) Implicit ADFs can be processed internally using either floating point
//     arithmetic or fixed point arithmetic (see the Math Mode Selection section
//     below); explicit ADFs are always processed internally using floating point
//     arithmetic.
//
//  4) Implicit ADF generation is nearly instantaneous while explicit ADF generation
//     is in the range of 2000 explicit ADF glyphs per second (assuming level 4
//     explicit ADFs (see below) and a Pentium M 2.0 GHz CPU).
//
//  5) While both ADF types provide fast interactive rendering, implicit ADF
//     rendering is, in general, faster than explicit ADF rendering for large point
//     sizes. The crossover point in performance (i.e., the point size at which the
//     implicit ADF rendering speed and the explicit ADF rendering speed are roughly
//     equal) depends on both the current library implementation and system
//     characteristics such as CPU cache sizes. For optimal performance, applications
//     should be benchmarked and profiled to determine the appropriate ADF type.
//
//  6) Implicit ADFs always provide high quality rendering while the rendering
//     quality of explicit ADFs depends on the generation attributes (see
//     ADFGenAttrs) set by the application. In general, higher quality explicit ADFs
//     are larger and require longer generation times than lower quality explicit
//     ADFs.
//
//  7) In general, implicit ADFs are smaller than explicit ADFs. Implicit ADFs are
//     approximately the same size as their ADFPaths (in the range of 350 bytes to
//     1K bytes) while typical sizes for explicit ADFs of Latin glyphs range from
//     2K bytes (for level 4 explicit ADFs, which provide high quality rendering for
//     20 ppem and below) to 4K bytes (for level 7 explicit ADFs, which provide high
//     quality rendering for all ppems).
```

```
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: BASIC DATA FLOW AND PROCESSING STEPS
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  To illustrate the basic data flow and processing steps when using Saffron, a
//  simple example for rendering with grid fitting is outlined here:
//
//  1) Initialize the system
//
//  2) Preprocess the required non-CJK glyphs to determine their SAZ alignment zones
//
//      - Initialize SAZ alignment zone detection for a specified typeface
//      - Detect the SAZ alignment zones for all required non-CJK glyphs in the
//        specified typeface
//      - Terminate SAZ alignment zone detection for the specified typeface
//
//  3) Create an ADF cache and a density image cache
//
//  4) Typeset and render each glyph
//
//      - Query the density image cache for the rendered glyph
//      - Upon a cache miss
//          - Query the ADF cache for the glyph's ADF
//          - Upon a cache miss
//              - Generate the glyph's ADF from its ADFPath
//              - Insert the glyph's ADF into the ADF cache
//          - If the glyph is a CJK glyph, enable MAZ alignment zone detection, which
//            is performed dynamically during rendering (i.e., the next step)
//          - Render the glyph's ADF into a density image using the glyph's
//            SAZ or MAZ alignment zones for grid fitting
//          - Insert the density image into the density image cache
//      - Composite the glyph's density image into the display buffer
//      - Advance the pen position using typesetting adjustments due to grid fitting
//
//  5) Destroy the caches
//
//  6) Terminate the system
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: COMPOSITING GLYPH DENSITY IMAGES
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  The Saffron type system renders a glyph (represented as an ADF) according to
//  a set of rendering parameters to produce a density image. A density image
//  records a 'density' value for each component of each pixel which can be treated
//  as an alpha value. Density images can be used by the application to blend a
//  foreground color (e.g., a text color) with a background color or a background
//  image to produce a colored image of the rendered glyph. This colored image can
//  then be BLTed to the display. Note that rendered density images can be used
//  directly for blending white text on a background or inverted (i.e., 255 -
//  density) to blend black text on a background.
//
//  The blending, or 'compositing', can be performed in software using the density
//  value for each pixel (in CRT mode) or pixel component (in LCD mode) to blend
//  the corresponding background and foreground pixels or pixel components. Some
//  examples of how the color of a destination pixel can be determined from the
//  corresponding density image pixel, the foreground color, and the background
//  color (or background image pixel) are provided here.
//
//  When rendering in CRT mode, the density image has one component per pixel:
//  A_density. When rendering in LCD mode (i.e., sub-pixel rendering), the density
//  image has four components per pixel: R_density, G_density, B_density, and
//  A_density, where A_density is set to the maximum of R_density, G_density, and
//  B_density. The foreground color is denoted 'fg' and the background color is
//  denoted 'bg'. All color and density values range from 0 to 255.
//
```

```
//
//  ALPHA BLEND:
//
//  Alpha blending uses the density value of each pixel or each pixel component
//  to blend the foreground and background pixel colors to determine the
//  destination pixel color (R, G, B, A). Note that alpha blending can be avoided
//  for each pixel if the pixel's A_density is zero.
//
//  For CRT mode:
//      R <-- R_bg + (R_fg - R_bg) * A_density / 255
//      G <-- G_bg + (G_fg - G_bg) * A_density / 255
//      B <-- B_bg + (B_fg - B_bg) * A_density / 255
//      A <-- A_bg + (A_fg - A_bg) * A_density / 255
//
//  For LCD mode:
//      R <-- R_bg + (R_fg - R_bg) * R_density / 255
//      G <-- G_bg + (G_fg - G_bg) * G_density / 255
//      B <-- B_bg + (B_fg - B_bg) * B_density / 255
//      A <-- A_bg + (A_fg - A_bg) * A_density / 255
//
//
//  ALPHA TEST:
//
//  For white text on a black background or black text on a white background,
//  alpha blending can be replaced by a simple alpha test or a simple alpha
//  test with an inversion as shown here.
//
//  White text on a black background
//      For CRT mode:
//          if (A_density > 0) {
//              R <-- A_density
//              G <-- A_density
//              B <-- A_density
//              A <-- A_density
//          }
//
//      For LCD mode:
//          if (A_density > 0) {
//              R <-- R_density
//              G <-- G_density
//              B <-- B_density
//              A <-- A_density
//          }
//
//  Black text on a white background
//      For CRT mode:
//          if (A_density > 0) {
//              R <-- 255 - A_density
//              G <-- 255 - A_density
//              B <-- 255 - A_density
//              A <-- A_density
//          }
//
//      For LCD mode:
//          if (A_density > 0) {
//              R <-- 255 - R_density
//              G <-- 255 - G_density
//              B <-- 255 - B_density
//              A <-- A_density
//          }
//
//
//  MINMAX BLEND:
//
//  For white text on a black background, minmax blending determines the destination
//  pixel color (R, G, B, A) by computing the maximum of the density value of each
//  pixel or pixel component and the background pixel color. For black text on a
```

```
//  white background, minmax blending determines the destination pixel color (R, G,
//  B, A) by computing the minimum of the inverted density value of each pixel or
//  pixel component and the background pixel color.
//
//  White text on a black background
//      For CRT mode:
//          R <-- maximum(R_bg, A_density)
//          G <-- maximum(G_bg, A_density)
//          B <-- maximum(B_bg, A_density)
//          A <-- maximum(A_bg, A_density)
//
//      For LCD mode:
//          R <-- maximum(R_bg, R_density)
//          G <-- maximum(G_bg, G_density)
//          B <-- maximum(B_bg, B_density)
//          A <-- maximum(A_bg, A_density)
//
//  Black text on a white background
//      For CRT mode:
//          R <-- minimum(R_bg, 255 - A_density)
//          G <-- minimum(G_bg, 255 - A_density)
//          B <-- minimum(B_bg, 255 - A_density)
//          A <-- minimum(A_bg, A_density)
//
//      For LCD mode:
//          R <-- minimum(R_bg, 255 - R_density)
//          G <-- minimum(G_bg, 255 - G_density)
//          B <-- minimum(B_bg, 255 - B_density)
//          A <-- minimum(A_bg, A_density)
//
//
//  INVERSE MULTIPLY:
//
//  Black text can also be composited by first inverting the density image and
//  then multiplying the background color by the inverted density. The result of
//  the multiply is then normalized to [0, 255]. This compositing method uses the
//  glyph density image to darken the underlying background image. Note that the
//  inverse multiply can be avoided for each pixel if the pixel's A_density is
//  zero.
//
//  For CRT mode:
//      R <-- R_bg * (255 - A_density) / 255
//      G <-- G_bg * (255 - A_density) / 255
//      B <-- B_bg * (255 - A_density) / 255
//      A <-- A_bg * A_density / 255
//
//  For LCD mode:
//      R <-- R_bg * (255 - R_density) / 255
//      G <-- G_bg * (255 - G_density) / 255
//      B <-- B_bg * (255 - B_density) / 255
//      A <-- A_bg * A_density / 255
//
//
//  SPECIAL EFFECTS:
//
//  Density images can be used in a variety of blending methods. For example, density
//  images make effective masks when a direct MULTIPLY, rather than an INVERSE
//  MULTIPLY, is used for blending.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: COMBINING GLYPH DENSITY IMAGES
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  When glyph density images overlap, they can either be composited sequentially
//  onto the background image (as described above), or they can be combined first in
//  a 'combining buffer' and then the combining buffer can be composited onto the
//  background image in a single pass. There are various ways for combining
```

```
//  overlapping density values in the combining buffer. For example, density values
//  can be combined using a CSG union operation, which selects the maximum density
//  value for each pixel (CRT mode) or pixel component (LCD mode). In addition,
//  various other methods can be used as described below.
//
//  Density values in the combining buffer are initialized to zero. Then, for each
//  pixel of a density image being combined, the corresponding pixel of the
//  combining buffer is set according to one of the following methods.
//
//
//  MAXIMUM DENSITY:
//
//  This method selects the maximum density value for each pixel or pixel component
//  and corresponds to a CSG union operation.
//
//  For CRT mode:
//      A_combined <-- maximum(A_combined, A_density)
//
//  For LCD mode:
//      R_combined <-- maximum(R_combined, R_density)
//      G_combined <-- maximum(G_combined, G_density)
//      B_combined <-- maximum(B_combined, B_density)
//      A_combined <-- maximum(R_combined, G_combined, B_combined)
//
//
//  AVERAGE DENSITY:
//
//  This method can provide better antialiasing when rendered glyphs overlap and
//  can reduce color fringing in LCD mode.
//
//  For CRT mode:
//      A_combined <-- average(A_combined, A_density)
//
//  For LCD mode:
//      R_combined <-- average(R_combined, R_density)
//      G_combined <-- average(G_combined, G_density)
//      B_combined <-- average(B_combined, B_density)
//      A_combined <-- maximum(R_combined, G_combined, B_combined)
//
//
//  INVERSE MULTIPLY:
//
//  This method is a generalization of inverse multiply for compositing (as
//  described above); it can be used to perform a darkening blend of density
//  images. Like average density, it can provide better antialiasing when rendered
//  glyphs overlap and can reduce color fringing in LCD mode.
//
//  For CRT mode:
//      A_combined <-- 255 - (255 - A_combined) * (255 - A_density) / 255
//
//  For LCD mode:
//      R_combined <-- 255 - (255 - R_combined) * (255 - R_density) / 255
//      G_combined <-- 255 - (255 - G_combined) * (255 - G_density) / 255
//      B_combined <-- 255 - (255 - B_combined) * (255 - B_density) / 255
//      A_combined <-- maximum(R_combined, G_combined, B_combined)
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: COORDINATE SYSTEMS
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  All coordinate systems within this library are Cartesian: the x-axis increases
//  from left to right, and the y-axis increases from bottom to top. The major
//  coordinate systems are enumerated here.
//
//
//  ADF COORDINATES:
//
```

```
//  ADFs are represented in a 2-dimensional floating point coordinate system defined
//  over [0.0, 1.0] x [0.0, 1.0]. Distance values in an ADF are measured in this
//  coordinate system.
//
//
//  FONT UNITS:
//
//  Glyphs are represented by the ADFPath data structure. Points that define a glyph
//  are located on a grid whose size is defined by the creator of the font. The units
//  of this grid are conventionally referred to as font units.
//
//  When creating the glyph, the font creator makes use of an imaginary square that
//  is derived from the old typographic concept of the em square. This square can be
//  thought of as a tablet on which the characters are drawn, although it is
//  permissible for characters to extend beyond the tablet or em square.
//
//  In order to scale glyphs to a specified point size, this API requires the number
//  of font units per em square (e.g., 2048 in the core Apple and Microsoft TTF
//  fonts). To support both integer-based (e.g., TTF) and floating point-based (e.g.,
//  Type 1) font representations, this library uses floating point font units. It is
//  assumed that the glyph origin (i.e., the reference point for typesetting) lies at
//  (0,0) in font units.
//
//
//  INTEGER PIXEL COORDINATES:
//
//  The image origin of a density image, which is used for positioning the image
//  during compositing, is represented in integer pixel coordinates. The image
//  origin is the bottom-left corner of the image.
//
//
//  FLOATING POINT PIXEL COORDINATES:
//
//  Glyph pen positions for typesetting (i.e., the locations of the glyph origins)
//  are specified in floating point pixel coordinates. When grid fitting, glyph pen
//  position adjustments are measured in floating point pixel coordinates; glyph
//  scale adjustments scale from font units to floating point pixel coordinates.
//  These grid fitting adjustments can be used by the application when typesetting.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: CONTINUOUS STROKE MODULATION
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  The Saffron type system exploits the inherent properties of distance fields to
//  provide 'Continuous Stroke Modulation' (CSM), i.e., continuous modulation of
//  both stroke weight and edge sharpness. CSM uses three rendering parameters to
//  control the mapping of ADF distances to glyph density values. Optimal values
//  for these parameters are highly subjective; they can depend on user preferences,
//  lighting conditions, display properties, typeface, foreground and background
//  colors, and point size. Ideally, an application would provide support for users
//  to tune CSM parameters according to personal preferences. However, under most
//  circumstances, high quality type can be achieved with a small set of default
//  parameters that can be preset by the application.
//
//  The function that maps ADF distances to density values has an outside cutoff
//  value, below which densities are set to zero, and an inside cutoff value, above
//  which densities are set to a maximum density value (e.g., 255). Between these two
//  cutoff values, the mapping function is a gamma curve ranging from zero at the
//  outside cutoff value to the maximum density at the inside cutoff value. The shape
//  of the curve is governed by a gamma exponent; when the exponent is 1.0, the curve
//  is linear.
//
//  Adjusting the outside and inside cutoff values affects stroke weight and edge
//  sharpness. The spacing between these two parameters is comparable to twice the
//  filter radius of classic antialiasing methods; a narrow spacing provides a
//  sharper edge while a wider spacing provides a softer, more filtered edge. When
//  the spacing is zero, the resulting density image is a bi-level bitmap. When the
```

```
//  spacing is very wide, the resulting density image has a watercolor-like edge.
//  Typically, users prefer sharp, high contrast edges at small point sizes and
//  softer edges for larger point sizes and for animated text. Hence, to achieve
//  optimal quality, the default CSM parameters preset by the application should be
//  point size dependent. Note that outside and inside cutoff values are specified in
//  floating point pixel coordinates.
//
//  Because a glyph's edge lies at the zero-valued iso-surface of its corresponding
//  ADF, the outside cutoff value typically has a negative value, the inside cutoff
//  value typically has a positive value, and their midpoint typically lies near
//  zero. Adjusting these parameters to shift the midpoint towards negative infinity
//  will increase the stroke weight; shifting the midpoint towards positive infinity
//  will decrease the stroke weight. Note that the outside cutoff value should always
//  be less than or equal to the inside cutoff value.
//
//  Under most circumstances, a linear mapping from distance to density (i.e., a
//  gamma exponent equal to 1.0) is recommended. The gamma exponent can be used to
//  compensate for the non-linear characteristics of CRT and LCD displays.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: COLOR REDUCTION
//-----------------------------------------------------------------------------------
//  As noted above, Saffron supports sub-pixel rendering for LCD displays. Sub-pixel
//  rendering increases the effective resolution of the display, but it can also lead
//  to undesirable 'color fringing' artifacts. These artifacts appear because the
//  individual sub-pixel components are set to different values. Color fringing tends
//  to be more noticeable at smaller point sizes and when using typefaces with thin
//  strokes, such as Monotype's Courier New (Version 2.90).
//
//  To combat this problem, Saffron provides a color reduction feature that analyzes
//  the amount of color present in the rendered pixels and reduces the color if
//  necessary. This feature is user-controllable via two attributes in the
//  ADFRenderAttrs data structure: useColorReduction and colorReductionAmt.
//
//  The useColorReduction attribute is a Boolean flag that turns the color reduction
//  feature on or off; setting this attribute to true (i.e., a non-zero value)
//  enables color reduction, and setting it to false (i.e., zero) disables color
//  reduction. If set to false, the colorReductionAmt attribute (described below) is
//  ignored.
//
//  The colorReductionAmt attribute is a floating point value in the range [0, 1]
//  that controls how much color reduction to perform. When this attribute is set to
//  0, minimum color reduction is applied. When this attribute is set to 1, maximum
//  color reduction is applied: pixels will be rendered completely desaturated (i.e.,
//  no color at all). Larger values are usually needed at smaller point sizes and for
//  typefaces with thin strokes.
//
//  It is important to note that when the useColorReduction attribute is set to true,
//  Saffron always performs some amount of color reduction, even when the
//  colorReductionAmt attribute is set to its minimum value of 0. In other words,
//  turning on color reduction and setting the colorReductionAmt attribute to 0 is
//  not equivalent to turning off color reduction altogether (i.e., by setting the
//  useColorReduction Boolean attribute to false).
//
//  Also note that setting the useColorReduction attribute to true and the
//  colorReductionAmt attribute to 1 (when LCD rendering) is not equivalent to CRT
//  rendering. The rendered text will appear different because 3 samples are used per
//  pixel in the former case, whereas only 1 sample is used per pixel in the latter
//  case. In general, rendering quality will be higher in the former case, but
//  performance will be slower.
//
//  Finally, the color reduction feature is only relevant to LCD rendering. The two
//  attributes described above are ignored entirely during CRT rendering.
//-----------------------------------------------------------------------------------
//  USAGE NOTE: TYPESETTING
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
```

```
//  Although this library does not typeset glyphs, it provides feedback to the
//  application that can be used during typesetting; we therefore introduce some
//  basic concepts and terminology here. For more details, see, e.g., Donald E.
//  Knuth, "Tex and Metafont: New Directions in Typesetting", Digital Press.
//
//  In typesetting, the 'baseline' of a line of text is an imaginary line that is
//  used to guide the placement of glyphs. The baseline can be horizontal (e.g., when
//  typesetting Latin fonts) or vertical (e.g., when typesetting Japanese fonts). The
//  'pen position' is a virtual point located on the baseline that is used to locate
//  glyphs. Conventionally (and in this library), the pen position determines the
//  placement of the glyph origin.
//
//  Text is typeset and rendered by incrementing the pen position -- either to the
//  left or right for horizontal baselines, or up or down for vertical baselines. The
//  distance between two successive pen positions is glyph-specific and is called the
//  'advance width'.
//
//  In this library, pen positions are specified by the application for each glyph to
//  be rendered. Adjustments to these pen positions (which consist of incremental
//  scales and translations) may be required depending on the application's use of
//  grid fitting, as explained in the following three cases:
//
//  1) When grid fitting is disabled, glyph density images are rendered with glyph
//     origins placed precisely at the specified pen positions. In this case, the
//     grid fitting translation adjustments are set to zero and the grid fitting
//     scale adjustments are effectively set to one.
//
//  2) When grid fitting is enabled using SAZ alignment zones, glyphs may be
//     incrementally scaled and translated to align strong vertical and horizontal
//     glyph edges and characteristic distances (e.g., cap heights) to the pixel grid
//     or to the sub-pixel grid, thereby providing better contrast, uniform stroke
//     weights, and consistent characteristic distances. The incremental scale and
//     translation for a glyph may affect the glyph's advance width (and other font
//     metrics such as the glyph's 'kerning value') which, in turn, may affect how
//     the application sets subsequent pen positions. Therefore, this library returns
//     grid alignment adjustments for each glyph (i.e., adjustments (in x and y) to
//     the glyph's pen position and scale) for use in typesetting.
//
//  3) When grid fitting is enabled using MAZ alignment zones, strong vertical and
//     horizontal glyph edges will be detected and aligned to the pixel grid
//     dynamically during rendering, thereby providing better contrast and uniform
//     stroke weights. Unlike case 2 above, grid fitting using MAZ alignment zones
//     does not require changes to the pen position, advance width, or other font
//     metrics. Therefore, this library returns grid fitting translation adjustments
//     that are set to zero and grid fitting scale adjustments that are effectively
//     set to one for each glyph. Applications that apply these grid fitting
//     adjustments during typesetting will obtain the same results (i.e., the same
//     text layout) as if grid fitting is disabled. Consequently, enabling MAZ grid
//     fitting improves the appearance of glyphs without affecting their placement.
//
//  While grid fitting is advised for small point sizes (e.g., less than 30 point on
//  a 72 dpi device), it is not necessary for larger point sizes, rotated glyphs, or
//  animated type. Hence, there are four common scenarios regarding grid fitting and
//  the use of grid alignment adjustments:
//
//  1) Grid fitting is disabled -- in this scenario, each glyph is placed precisely
//     at the requested pen position and the grid alignment adjustments can be
//     ignored or applied with identical results. As a consequence, a prescribed
//     layout can be maintained exactly but the rendering quality is compromised at
//     small point sizes.
//
//  2) Grid fitting is enabled using SAZ alignment zones and the grid alignment
//     adjustments are used for typesetting -- in this scenario, each glyph's advance
//     width (and other font metrics) are modified according to the glyph alignment
//     adjustments and are used for determining subsequent pen positions. For
//     example, assuming horizontal baselines, if the requested pen position is
//     (penX,penY) and the glyph's advance width is advanceW, the pen position for
```

```
//     the next glyph is computed as nextPenX <-- penX + adjustX + advanceW *
//     adjustScaleX, where adjustX is the x glyph alignment adjustment for
//     translation and adjustScaleX is the x glyph alignment adjustment for scaling.
//     Note that nextPenY <-- penY (i.e., penY is not adjusted) so that a consistent
//     horizontal baseline is maintained. Using the grid alignment adjustments
//     provides optimal typesetting and is strongly recommended in circumstances
//     where a prescribed layout is not mandated.
//
//  3) Grid fitting is enabled using SAZ alignment zones and the grid alignment
//     adjustments are ignored -- in this scenario, each glyph is placed precisely at
//     the requested pen position which allows the application to maintain a
//     prescribed layout. Note, however, when grid fitting to the pixel grid,
//     sub-optimal inter-character spacing occurs due to glyph translations of at
//     most half a pixel which are performed during grid fitting. The inter-character
//     spacing can be improved by grid fitting to the sub-pixel grid, in which case
//     the glyph translations due to grid fitting are at most one-sixth of a pixel.
//     However, grid fitting to the sub-pixel grid results in softer edges and more
//     color fringing (in LCD mode). In addition, grid fitting to the sub-pixel grid
//     requires more memory when caching glyph density images since each glyph can be
//     aligned to three different sub-pixels.
//
//  4) Grid fitting is enabled using MAZ alignment zones -- in this scenario, each
//     glyph is placed precisely at the requested pen position and the grid alignment
//     adjustments can be ignored or applied with identical results. Consequently,
//     enabling MAZ grid fitting improves the appearance of glyphs without affecting
//     their placement.
//
//  When typesetting text at small point sizes, it is important to note that
//  antialiasing increases the perceived size of each glyph, thereby decreasing
//  the space between adjacent characters. Adding extra space (e.g., equal to the
//  filter radius) between glyphs during typesetting can result in more readable
//  text.
//
//  Finally, optimal metric-preserving non-CJK typesetting when grid fitting requires
//  more than just placing glyphs at pen positions mandated by a prescribed layout
//  (see, e.g., Microsoft's approach for formatting text in GDI+ which adjusts space
//  bands during glyph placement).
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: RENDERING CJK GLYPHS WITH MULTIPLE ALIGNMENT ZONES
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  If requested, MAZ alignment zones are detected and applied dynamically to glyphs
//  during rendering. This automatic grid fitting process significantly improves the
//  appearance of both outline-based CJK glyphs and stroke-based CJK glyphs. Although
//  the MAZ alignment zone detection and grid fitting system can be applied to any
//  glyph (including non-CJK glyphs), the system is designed and optimized for CJK
//  glyphs. Applications are strongly discouraged from using MAZ alignment zones to
//  grid fit non-CJK glyphs.
//
//  Grid fitting using MAZ alignment zones can be enabled by setting the gridFitType
//  element of the ADFRenderAttrs data structure to ADF_GRID_FIT_MAZ_PIXEL (see the
//  Rendering section below for details).
//
//  There are two issues that applications need to be aware of when using MAZ
//  alignment zones:
//
//  1) In general, this library returns grid fitting adjustments which may be used by
//     the application to perform typesetting (i.e., to compute the pen position of
//     each glyph). When grid fitting is enabled using MAZ alignment zones, this
//     library returns grid fitting translation adjustments that are set to zero and
//     grid fitting scale adjustments that are effectively set to one. Applications
//     that apply these grid fitting adjustments during typesetting will obtain the
//     same results (i.e., the same text layout) as if grid fitting is disabled.
//     Consequently, MAZ alignment zones improve the appearance of glyphs without
//     affecting their placement.
//
```

```
//  2) The 'alignment behavior' of the MAZ alignment zone detection and grid fitting
//     system (i.e., the manner in which strong vertical and horizontal glyph edges
//     are aligned to the pixel grid) is independent of the requested pen position.
//     For example, the same glyph placed at two different pen positions (e.g.,
//     (20.7,25.8) and (30.2,29.3)) will appear identical when rendered.
//     Consequently, when caching density images, only one version of the glyph needs
//     to be stored into the cache.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  USAGE NOTE: RENDERING APPLICATION-HINTED GLYPHS
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Applications can perform their own grid fitting (e.g., by using TrueType hints)
//  instead of using the automatic hinting (based on SAZ or MAZ alignment zones)
//  provided by this library. This is useful for Unicode character sets not supported
//  by the current implementations of the SAZ and MAZ alignment zone detection and
//  grid fitting systems. Applications that perform their own grid fitting must
//  follow certain steps to ensure that this library renders application-hinted
//  glyphs correctly. These steps are described below in the Rendering section.
//-----------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------
//  To avoid multiple inclusion of header files
//-----------------------------------------------------------------------------------
#ifndef _ADF_TYPE_SYSTEM_
#define _ADF_TYPE_SYSTEM_


//-----------------------------------------------------------------------------------
//  To make functions accessible from C++ code
//-----------------------------------------------------------------------------------
#ifdef __cplusplus
extern "C" {
#endif


//-----------------------------------------------------------------------------------
//  SAFFRON API VERSION NUMBER
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  The 16 most significant bits of ADF_API_VER_NUMBER identify the major revision
//  number of the Saffron API; the 16 least significant bits of ADF_API_VER_NUMBER
//  identify the minor revision number of the Saffron API.
//
//  This is version 4.0 of the API.
//-----------------------------------------------------------------------------------
#define ADF_API_VER_NUMBER 0x00040000


//-----------------------------------------------------------------------------------
//  ADF TYPE SELECTION (EXPLICIT OR IMPLICIT)
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  The compile time Boolean flag used to control which type of ADF to use throughout
//  the system (if true: use implicit ADFs, if false: use explicit ADFs)
//-----------------------------------------------------------------------------------
#define ADF_USE_IMPLICIT_ADFS 0


//-----------------------------------------------------------------------------------
//  LIBRARY COMPONENT SELECTION
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  This API contains a set of compile-time switches that can be used to enable or
//  disable the implementations of specific library features. These features
//  include:
```

```
//
//    - The dual caching system
//    - The library validation system
//    - The stylized stroke font (SSF) rendering system
//    - The SAZ alignment zone detection system
//    - The MAZ alignment zone detection and grid fitting system for outline-based
//      glyphs
//    - The MAZ alignment zone detection and grid fitting system for uniform-width
//      stroke-based glyphs
//    - The MAZ alignment zone detection and grid fitting system for stylized
//      stroke-based glyphs
//
//  The purpose of these compile-time switches is to allow applications to optimize
//  this library's object code size.
//
//  Some applications may not require all features of this library. For example,
//  applications that implement their own caches may not need this library's dual
//  caching system. Setting ADF_ENABLE_DUAL_CACHING to false (see below) will exclude
//  the implementation of the dual caching system from the compiled object code,
//  thereby reducing object code size.
//-----------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------
//  The compile time Boolean flag used to control whether the implementation of the
//  dual caching system is enabled (if true: included, if false: excluded). The
//  Boolean flag must be set to true to enable the implementations of the following
//  API functions:
//
//    - ADFCreateCache()
//    - ADFDestroyCache()
//    - ADFGetCacheElement()
//    - ADFInsertCacheElement()
//    - ADFFlushCacheElement()
//    - ADFFlushCacheAll()
//    - ADFGetCacheState()
//    - ADFGetCacheKeys()
//
//  For applications that intend to use this library's dual caching system, this
//  Boolean flag must be set to true. For applications that intend to use their own
//  caching systems or do not require caching, this Boolean flag should be set to
//  false.
//-----------------------------------------------------------------------------------
#define ADF_ENABLE_DUAL_CACHING 1


//-----------------------------------------------------------------------------------
//  The compile time Boolean flag used to control whether the implementation of the
//  library validation system is enabled (if true: included, if false: excluded). The
//  Boolean flag must be set to true to enable the implementations of the following
//  functions in ADFExplicit.c:
//
//    - ADFDrawADFAndAlgnZonesExplicit()
//    - DrawADFGrid()
//    - RecursiveDrawADFGrid()
//
//  and the following functions in ADFImplicitFloat.c:
//
//    - ADFDrawADFAndAlgnZonesImplicit()
//
//  For most applications, this Boolean flag should be set to false.
//-----------------------------------------------------------------------------------
#define ADF_ENABLE_VALIDATION 1


//-----------------------------------------------------------------------------------
//  The compile time Boolean flag used to control whether the implementation of the
```

```
//  stylized stroke font renderer is enabled (if true: included, if false: excluded).
//  For applications that require rendering stylized stroke-based glyphs, this
//  Boolean flag must be set to true. For applications that do not require rendering
//  stylized stroke-based glyphs, this Boolean flag should be set to false.
//-----------------------------------------------------------------------------------
#define ADF_ENABLE_SSF_RENDERER 1


//-----------------------------------------------------------------------------------
//  The compile time Boolean flag used to control whether the implementation of the
//  SAZ alignment zone detection system is enabled (if true: included, if false:
//  excluded). The Boolean flag must be set to true to enable the implementations of
//  the following API functions:
//
//    - ADFInitAlgnZoneDetection()
//    - ADFDetectAlgnZones()
//    - ADFTermAlgnZoneDetection()
//
//  For applications that require the detection of SAZ alignment zones, this Boolean
//  flag must be set to true. For applications that do not require the detection of
//  SAZ alignment zones, this Boolean flag should be set to false.
//
//  Note that this Boolean flag affects only the detection of SAZ alignment zones,
//  not the application of SAZ alignment zones during grid fitting, which occurs when
//  rendering. Existing (i.e., preprocessed) SAZ alignment zones can be used to
//  perform grid fitting even when this Boolean flag is set to false.
//-----------------------------------------------------------------------------------
#define ADF_ENABLE_SAZ_DETECTION 1


//-----------------------------------------------------------------------------------
//  The compile time Boolean flag used to control whether the implementation of the
//  MAZ alignment zone detection and grid fitting system for outline-based glyphs is
//  enabled (if true: included, if false: excluded). For applications that require
//  automatic grid fitting for outline-based glyphs using MAZ alignment zones (e.g.,
//  for CJK typefaces), this Boolean flag must be set to true. For applications that
//  do not require automatic grid fitting for outline-based glyphs using MAZ
//  alignment zones, this Boolean flag should be set to false.
//-----------------------------------------------------------------------------------
#define ADF_ENABLE_MAZ_OUTLINES 1


//-----------------------------------------------------------------------------------
//  The compile time Boolean flag used to control whether the implementation of the
//  MAZ alignment zone detection and grid fitting system for uniform-width
//  stroke-based glyphs is enabled (if true: included, if false: excluded). For
//  applications that require automatic grid fitting for uniform-width stroke-based
//  glyphs using MAZ alignment zones (e.g., for CJK typefaces), this Boolean flag
//  must be set to true. For applications that do not require automatic grid fitting
//  for uniform-width stroke-based glyphs using MAZ alignment zones, this Boolean
//  flag should be set to false.
//-----------------------------------------------------------------------------------
#define ADF_ENABLE_MAZ_UNIFORM_WIDTH_STROKES 1


//-----------------------------------------------------------------------------------
//  The compile time Boolean flag used to control whether the implementation of the
//  MAZ alignment zone detection and grid fitting system for stylized stroke-based
//  glyphs is enabled (if true: included, if false: excluded). For applications that
//  require automatic grid fitting for stylized stroke-based glyphs using MAZ
//  alignment zones (e.g., for CJK typefaces), this Boolean flag must be set to true.
//  For applications that do not require automatic grid fitting for stylized
//  stroke-based glyphs using MAZ alignment zones, this Boolean flag should be set to
//  false.
//-----------------------------------------------------------------------------------
#define ADF_ENABLE_MAZ_STYLIZED_STROKES 1
```

```
//-----------------------------------------------------------------------------------
//  MATH MODE SELECTION
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Explicit ADFs are always processed internally using floating point arithmetic.
//  Implicit ADFs, however, can be processed internally using either floating point
//  arithmetic or fixed point arithmetic. The latter option provides better
//  performance on systems that lack floating point hardware support.
//
//  When fixed point arithmetic is enabled, applications must ensure that certain
//  rendering parameters lie within specific ranges. These range requirements are
//  described below.
//
//  The ADF_MATH_MODE macro controls the type of arithmetic used internally by the
//  following API functions:
//
//    - ADFGenerateADF()
//    - ADFRenderSetup()
//    - ADFRenderSetupFromPath()
//    - ADFRenderGlyph()
//    - ADFPositionCachedImage()
//    - ADFInitAlgnZoneDetection()
//    - ADFDetectAlgnZones()
//
//  ADF_MATH_MODE must be set to one of the following constants:
//
//    - ADF_MATH_FLOAT
//    - ADF_MATH_FIXED_C_32
//    - ADF_MATH_FIXED_C_64
//    - ADF_MATH_FIXED_ASM_X86
//
//  If ADF_MATH_MODE is set to ADF_MATH_FLOAT, this library uses a floating point
//  implementation for the above API functions.
//
//  If ADF_MATH_MODE is set to ADF_MATH_FIXED_C_32, this library uses a fixed point
//  implementation for the above API functions. This fixed point implementation is
//  written in ANSI C and is portable across systems that support 32-bit integers and
//  two's complement arithmetic.
//
//  If ADF_MATH_MODE is set to ADF_MATH_FIXED_C_64, this library uses a fixed point
//  implementation for the above API functions. This fixed point implementation is
//  portable across systems that support 32-bit integers, 64-bit integers, and two's
//  complement arithmetic. Applications must define ADF_I64 appropriately (see
//  below). This implementation requires that the system performs sign extension when
//  the left operand of a right shift is a signed integer. In general, this
//  implementation is significantly faster than ADF_MATH_FIXED_C_32.
//
//  If ADF_MATH_MODE is set to ADF_MATH_FIXED_ASM_X86, this library uses a fixed
//  point implementation for the above API functions. This fixed point implementation
//  is written in optimized x86 assembly. Applications must define ADF_I64
//  appropriately (see below). This implementation compiles only to x86-based systems
//  (e.g., Pentium M systems) and is usually significantly faster than
//  ADF_MATH_FIXED_C_64.
//
//  All of the fixed point implementations provide bit-identical results with each
//  other.
//
//  Note that the other API functions (i.e., all API functions described in this file
//  except the ones listed above) use only integer arithmetic and therefore ignore
//  the setting of ADF_MATH_MODE.
//
//  If ADF_USE_IMPLICIT_ADFS is false, then ADF_MATH_MODE must be set to
//  ADF_MATH_FLOAT.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  When fixed point arithmetic is enabled, applications must ensure that certain
```

```
//  rendering parameters lie within specific ranges. These range requirements are
//  as follows:
//
//    1. All floating point elements in the ADFRenderAttrs and ADFPath data
//       structures (see below) must be in the range [-32767, 32767].
//
//    2. The scaleX and scaleY elements in the ADFRenderAttrs data structure must be
//       non-negative.
//
//    3. The outsideCutoff and insideCutoff elements in the ADFRenderAttrs data
//       structure must be in the range [-20, 20]. Furthermore, outsideCutoff must be
//       less than or equal to insideCutoff.
//
//    4. The gamma element in the ADFRenderAttrs data structure must be positive.
//
//    5. The pointSize, dpi, scaleX, and scaleY elements in the ADFRenderAttrs data
//       structure must satisfy the following two inequalities:
//
//           0 <= (pointSize * dpi * scaleX / 72) <= 2048
//           0 <= (pointSize * dpi * scaleY / 72) <= 2048
//
//       For example, if dpi is 72 and scaleX and scaleY are both 1, then pointSize
//       must lie in the range [0, 2048].
//
//       If the ADFPath represents a stroke-based glyph, then the pointSize, dpi,
//       scaleX, and scaleY elements of the ADFRenderAttrs data structure and the
//       pathWidth and fontUnitsPerEM elements of the ADFPath data structure must
//       also satisfy the following two inequalities:
//
//           0 <= (pointSize * dpi * scaleX * pathWidth / fontUnitsPerEM) <= 7200
//           0 <= (pointSize * dpi * scaleY * pathWidth / fontUnitsPerEM) <= 7200
//
//    6. The fontUnitsPerEM element in the ADFPath data structure must lie in the
//       range (0,2048].
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_MATH_FLOAT             0
#define ADF_MATH_FIXED_C_32        1
#define ADF_MATH_FIXED_C_64        2
#define ADF_MATH_FIXED_ASM_X86     3
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_MATH_MODE              ADF_MATH_FLOAT
//-----------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------
//  Check at compile time that fixed point arithmetic is not being used to process
//  explicit ADFs
//-----------------------------------------------------------------------------------
#if ((ADF_USE_IMPLICIT_ADFS == 0) && (ADF_MATH_MODE != ADF_MATH_FLOAT))
#error "Fixed point arithmetic is not supported for explicit ADFs."
#endif


//-----------------------------------------------------------------------------------
//  Applications should set ADF_INLINE to the keyword used by their compiler to
//  identify inline functions. The default setting is __inline, the keyword used by
//  the Microsoft Visual Studio 6.0 compiler to identify inline functions.
//-----------------------------------------------------------------------------------
#define ADF_INLINE __inline


//-----------------------------------------------------------------------------------
//  Applications should set ADF_I64 to the keyword used by their compiler to
//  represent a 64-bit signed integer. The default setting is __int64, the keyword
//  used by the Microsoft Visual Studio 6.0 compiler to represent a 64-bit signed
```

```
//  integer.
//-----------------------------------------------------------------------------------
typedef __int64 ADF_I64;


//-----------------------------------------------------------------------------------
//  CPU BYTE ORDER SELECTION
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Applications must specify the CPU byte order by setting the ADF_CPU_BYTE_ORDER
//  define accordingly
//-----------------------------------------------------------------------------------
#define ADF_LITTLE_ENDIAN   0
#define ADF_BIG_ENDIAN      1
#define ADF_CPU_BYTE_ORDER  ADF_LITTLE_ENDIAN


//-----------------------------------------------------------------------------------
//  MEMORY ALLOCATION
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  The macros described below are used by this system to perform all memory
//  allocation tasks. The first argument to each macro is an opaque pointer appInst
//  to application specific instance data which is specified by the application when
//  the ADF font rendering system is initialized (i.e., when ADFInitSystem() is
//  invoked). appInst can be used by the application if it chooses to replace the
//  default memory allocation macro definitions with a custom memory management
//  system tailored specifically to its needs. If the default memory allocation macro
//  definitions are used, appInst is ignored.
//
//  ADF_ALLOC() returns a pointer to space for an object of size numBytes, or NULL
//  if the request cannot be satisfied. The space is uninitialized.
//
//  ADF_CALLOC() returns a pointer to space for an array of numElems elements, each
//  of length bytesPerElem bytes, or NULL if the request cannot be satisfied. Each
//  element is initialized to 0.
//
//  ADF_FREE() deallocates the space pointed to by object; it does nothing if object
//  is NULL. object must be a pointer to space previously allocated by ADF_ALLOC(),
//  ADF_CALLOC(), or ADF_REALLOC().
//
//  ADF_REALLOC() changes the size of object to numBytes. The contents of object will
//  be unchanged up to the minimum of object's old and new sizes. If the new size of
//  object is larger, the additional space is uninitialized. ADF_REALLOC() returns a
//  pointer to the reallocated space for object, or NULL if the request cannot be
//  satisfied, in which case object is unchanged. Note that the returned pointer may
//  be different than the input (object) pointer.
//-----------------------------------------------------------------------------------
#define ADF_ALLOC(appInst,numBytes)                 malloc(numBytes)
#define ADF_CALLOC(appInst,numElems,bytesPerElem)   calloc(numElems,bytesPerElem)
#define ADF_FREE(appInst,object)                    free(object)
#define ADF_REALLOC(appInst,object,numBytes)        realloc(object,numBytes)


//-----------------------------------------------------------------------------------
//  FUNDAMENTAL DATA TYPES
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Fundamental data types for characters, Booleans, integers, and floating point
//  numbers
//-----------------------------------------------------------------------------------
typedef     char            ADF_I8;
typedef     short           ADF_I16;
typedef     int             ADF_I32;
typedef     unsigned char   ADF_U8;
typedef     unsigned short  ADF_U16;
typedef     unsigned int    ADF_U32;
```

```
typedef     float           ADF_F32;
typedef     double          ADF_F64;
typedef     int             ADF_Bool;


//-----------------------------------------------------------------------------------
//  INITIALIZATION AND TERMINATION OF THE SYSTEM
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Initialize the ADF font rendering system. This call must precede all other
//  ADFxxx() invocations. This function returns an opaque pointer to an 'instance'
//  data structure which records all the global and static state for the rendering
//  system. A NULL is returned if the request cannot be satisfied.
//
//  All memory allocation tasks required by this system are performed via a set of
//  memory allocation macros; each macro requires appInst as an input argument (for
//  a discussion of appInst, see the Memory Allocation section above). If the default
//  memory allocation macro definitions are used, appInst is ignored.
//-----------------------------------------------------------------------------------
void *ADFInitSystem (void *appInst);


//-----------------------------------------------------------------------------------
//  Terminate the ADF font rendering system. This function must be invoked when no
//  further ADFxxx() invocations are required. The instance pointer returned by the
//  corresponding ADFInitSystem() invocation is expected as input.
//-----------------------------------------------------------------------------------
void ADFTermSystem (void *libInst);


//-----------------------------------------------------------------------------------
//  GLYPH REPRESENTATION (ADFPATHS)
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Glyphs are represented by the ADFPath data structure. Similar to Postscript, an
//  ADFPath is composed of a series of pen commands (e.g., moveto, lineto, and
//  curveto) that define a closed shape for an outline-based glyph or a set of stroke
//  skeletons for a stroke-based glyph. Note that a skeleton of a stroke-based glyph
//  can be open (e.g., for a 'u') or closed (e.g., for an 'o'). The pen commands
//  specify the movement and drawing of a virtual pen and allow both line segments
//  and quadratic Bezier curve segments to be drawn. The endpoints and the control
//  vertices of each segment are specified in floating point font units. This library
//  follows the TrueType convention for rendering glyphs, where rasterization is
//  performed using samples located at pixel centers. For outline-based glyphs, we
//  assume a non-zero winding rule for distinguishing between filled and unfilled
//  areas of the shape. The ADFPath consists of the following elements:
//
//  pathType: ADF_OUTLINE_PATH, ADF_UNIFORM_STROKE_PATH, or ADF_STYLIZED_STROKE_PATH.
//  This element indicates whether the ADFPath represents an outline-based glyph
//  (e.g., from a TrueType font) comprising a set of closed contours
//  (ADF_OUTLINE_PATH), a uniform-width stroke-based glyph comprising a stroke width
//  and a set of stroke skeletons representing the centerlines of the glyph
//  (ADF_UNIFORM_STROKE_PATH), or a stylized stroke-based glyph (referred to as a
//  stylized stroke font (SSF) glyph in this library) comprising a set of stroke
//  skeletons representing the approximate centerlines of the glyph and a set of
//  stroke attributes (see ADFSSFAttrs definition below) that determine how the
//  stroke skeletons are rendered (ADF_STYLIZED_STROKE_PATH). Note that uniform-width
//  stroke-based glyphs are always rendered with round endcaps and round corners.
//
//  pathWidth: The stroke width of a stroke-based glyph in floating point font units.
//  pathWidth is used only when pathType is ADF_UNIFORM_STROKE_PATH or
//  ADF_STYLIZED_STROKE_PATH.
//
//  ssfAttrs: A set of stroke attributes (see ADFSSFAttrs definition below) that
//  determine how the stroke skeletons of a stylized stroke-based glyph are rendered.
//  This element is used only when pathType is ADF_STYLIZED_STROKE_PATH.
//
```

```
//  glyphMinX, glyphMinY, glyphMaxX, glyphMaxY: The exact bounding box of the glyph
//  in floating point font units. If the pathType is ADF_UNIFORM_STROKE_PATH or
//  ADF_STYLIZED_STROKE_PATH, these values define the exact bounding box of the pen
//  commands that comprise the stroke skeletons (i.e., the bounding box does not
//  account for the stroke width (i.e., pathWidth)).
//
//  fontUnitsPerEM: The em box size in floating point font units (i.e., font units
//  per em).
//
//  charCode: The Unicode character code of the glyph. A zero character code
//  indicates that the character code is not available.
//
//  numContours: For an outline-based glyph, the number of closed contours of the
//  glyph, where each contour is described by a series of pen commands and begins
//  with a moveto command. This element is not used when pathType is
//  ADF_UNIFORM_STROKE_PATH or ADF_STYLIZED_STROKE_PATH.
//
//  numPenCmds: The total number of pen commands in the ADFPath.
//
//  penCmds: A pointer to a contiguous block of fixed-sized pen commands defining the
//  glyph shape. The first command of each contour or stroke skeleton must be a
//  moveto command. For an outline-based glyph, the x and y coordinates of the last
//  command of each contour must equal the x and y coordinates of the contour's
//  initial moveto command.
//
//  algnZones: algnZones is an encoding of the glyph's SAZ alignment zones, which are
//  used to build the appropriate transformation needed for grid fitting the ADF
//  glyph to the pixel grid or to the sub-pixel grid.
//
//  algnZonesMask: algnZones contains a valid encoding of the glyph's SAZ alignment
//  zones if algnZonesMask is non-zero. If algnZonesMask is zero, no SAZ alignment
//  zones are available for the glyph.
//
//  appData: A pointer to application specific data.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_OUTLINE_PATH            0   // An outline-based glyph
#define ADF_UNIFORM_STROKE_PATH     1   // A uniform-width stroke-based glyph
#define ADF_STYLIZED_STROKE_PATH    2   // A stylized stroke-based glyph
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  A summary of the pen commands follows:
//
//  ADF_PEN_MOVETO_CMD x y -- Starts a new subpath (i.e., contour or stroke skeleton)
//  of the ADFPath. This command sets the current point to (x,y).
//
//  ADF_PEN_LINETO_CMD x y -- Appends a straight line segment to the current subpath
//  (i.e., contour or stroke skeleton) of the ADFPath. The line segment extends from
//  the current point to (x,y). After constructing the line segment, (x,y) becomes
//  the new current point.
//
//  ADF_PEN_CURVTO_CMD cx cy x y -- Appends a quadratic Bezier curve segment to the
//  current subpath (i.e., contour or stroke skeleton) of the ADFPath between the
//  current point and (x,y) using (cx,cy) as the Bezier control point. After
//  constructing the curve segment, (x,y) becomes the new current point.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_PEN_MOVETO_CMD  0   // Set the current point of the subpath
#define ADF_PEN_LINETO_CMD  1   // Append a line segment to the subpath
#define ADF_PEN_CURVTO_CMD  2   // Append a quadratic Bezier segment to the subpath
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_U32     opCode; // ADF_PEN_MOVETO_CMD, ADF_PEN_LINETO_CMD, etc.
    ADF_F32     x;      // The x coordinate of the pen command
    ADF_F32     y;      // The y coordinate of the pen command
    ADF_F32     cx;     // The x coordinate of the control point of curvto commands
```

```
    ADF_F32     cy;     // The y coordinate of the control point of curvto commands
}   ADFPenCmd;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  SSF definitions
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_SSF_ENDCAP_NONE         0   // Do not render an endcap
#define ADF_SSF_ENDCAP_ROUND        1   // A round endcap
#define ADF_SSF_ENDCAP_SQUARE       2   // A square endcap
#define ADF_SSF_ENDCAP_OBLONG       3   // An oblong endcap
#define ADF_SSF_ENDCAP_TRIANGLE     4   // A triangle endcap
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_SSF_CORNER_NONE         0   // Do not render a corner join
#define ADF_SSF_CORNER_ROUND        1   // A round corner join
#define ADF_SSF_CORNER_BEVEL        2   // A beveled corner join
#define ADF_SSF_CORNER_MITER        3   // A mitered corner join
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_SSF_PROFILE_UNIFORM     0   // A uniform-width profile
#define ADF_SSF_PROFILE_LINEAR      1   // A linearly tapered profile
#define ADF_SSF_PROFILE_QUADRATIC   2   // A quadratic profile
#define ADF_SSF_PROFILE_CUBIC       3   // A cubic profile
#define ADF_SSF_PROFILE_CUSTOM      4   // A custom profile
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_SSF_MAX_PROFILE_COEFS   4   // Maximum # of coefs. to define a profile
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  SSF profile data structures and callback functions: When rendering a stylized
//  stroke-based glyph, applications can specify custom profile callback functions
//  (see the ADFSSFDistFromProfileCB() function prototype below) for determining the
//  perpendicular distance from each stroke skeleton to the left and right edges of
//  its stroke body. Upon entry to a profile callback function, t is the normalized
//  length along the current stroke skeleton being processed; t ranges from [0,1].
//  The output of each profile evaluation (i.e., the return value of the profile
//  callback function) must reside in the range [0,1], where a value of V results in
//  a stroke radius of V * 1/2 * pathWidth. attrs (a pointer to a ADFSSFCBAttrs data
//  structure) can be used by applications to achieve various effects. For example,
//  the unit normal vector N at t can be computed (via interpolation) from the input
//  unit normal vector (nx0,ny0) at t0 and the input unit normal vector (nx1,ny1) at
//  t1; N can then be used to determine the profile distance for simulating the nib
//  of a calligraphic pen. Note that t always resides in the interval [t0,t1].
//  invRange is set to the value of the expression (t0 == t1 ? 0 : 1 / (t1 - t0)) and
//  can be used to interpolate t within the interval [t0,t1]. If endCap is 1, then
//  the profile callback function is being invoked to determine the perpendicular
//  distance for constructing the chosen begEndCap (e.g., to determine the radius of
//  a round endcap); in this case, t will be 0 and both (nx0,ny0) and (nx1,ny1) will
//  be set to the unit normal vector of the stroke skeleton at t = 0. If endCap is 2,
//  then the profile callback function is being invoked to determine the
//  perpendicular distance for constructing the chosen endEndCap; in this case, t
//  will be 1 and both (nx0,ny0) and (nx1,ny1) will be set to the unit normal vector
//  of the stroke skeleton at t = 1. If endCap is 0, then the profile callback
//  function is not being invoked for endcap processing. If corner is 1, then the
//  profile callback function is being invoked to determine the perpendicular
//  distance for constructing the chosen corner; in this case, t will reside in the
//  range [0,1], both t0 and t1 will be set to t, and either the left or the right
//  profile callback function will be invoked twice to determine the perpendicular
//  distance for the incoming stroke skeleton at the corner point (where both
//  (nx0,ny0) and (nx1,ny1) will be set to the unit normal vector of the incoming
//  stroke skeleton at t) and for the outgoing stroke skeleton at the corner point
//  (where both (nx0,ny0) and (nx1,ny1) will be set to the unit normal vector of the
//  outgoing stroke skeleton at t). If the outside of the corner is on the left side
//  of the stroke skeleton then the left profile callback function is used to
//  determine the corner data; similarly, if the outside of the corner is on the
//  right side of the stroke skeleton then the right profile callback function is
```

```
//  used to determine the corner data. If corner is 0, then the profile callback
//  function is not being invoked for corner processing. coefs is set to the left
//  profile coefficients ssfAttrs.leftProfileCoefs (see ADFSSFAttrs definition below)
//  when invoking a left profile callback function and to the right profile
//  coefficients ssfAttrs.rightProfileCoefs when invoking a right profile callback
//  function. appData is a pointer to application specific data which can be used by
//  applications to pass arbitrary data to a profile callback function; appData is
//  set to ssfAttrs.appData upon entry.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  START: FLOATING POINT MATH ONLY
//-----------------------------------------------------------------------------------
#if (ADF_MATH_MODE == ADF_MATH_FLOAT)
typedef struct {
    ADF_F32 nx0;        // x-coordinate of the unit normal vector at t0
    ADF_F32 ny0;        // y-coordinate of the unit normal vector at t0
    ADF_F32 t0;         // Normalized length along the stroke skeleton at (nx0,ny0)
    ADF_F32 nx1;        // x-coordinate of the unit normal vector at t1
    ADF_F32 ny1;        // y-coordinate of the unit normal vector at t1
    ADF_F32 t1;         // Normalized length along the stroke skeleton at (nx1,ny1)
    ADF_F32 invRange;   // Set to the value of (t0 == t1 ? 0 : 1 / (t1 - t0))
    ADF_U32 endCap;     // 0: not an endcap, 1: beginning endcap, 2: ending endcap
    ADF_U32 corner;     // 0: not a corner, 1: processing a corner
}   ADFSSFCBAttrs;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef ADF_F32 (*ADFSSFDistFromProfileCB) (ADF_F32 t, ADFSSFCBAttrs *attrs,
ADF_F32 coefs[ADF_SSF_MAX_PROFILE_COEFS], void *appData);
//-----------------------------------------------------------------------------------
//  END:   FLOATING POINT MATH ONLY
//  START: FIXED POINT MATH ONLY
//-----------------------------------------------------------------------------------
#else
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef ADF_I32 ADF_I1616;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_I1616 nx0;      // x-coordinate of the unit normal vector at t0
    ADF_I1616 ny0;      // y-coordinate of the unit normal vector at t0
    ADF_I1616 t0;       // Normalized length along the stroke skeleton at (nx0,ny0)
    ADF_I1616 nx1;      // x-coordinate of the unit normal vector at t1
    ADF_I1616 ny1;      // y-coordinate of the unit normal vector at t1
    ADF_I1616 t1;       // Normalized length along the stroke skeleton at (nx1,ny1)
    ADF_I1616 invRange; // Set to the value of (t0 == t1 ? 0 : 1 / (t1 - t0))
    ADF_U32 endCap;     // 0: not an endcap, 1: beginning endcap, 2: ending endcap
    ADF_U32 corner;     // 0: not a corner, 1: processing a corner
}   ADFSSFCBAttrs;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef ADF_I1616 (*ADFSSFDistFromProfileCB) (ADF_I1616 t, ADFSSFCBAttrs *attrs,
ADF_I1616 coefs[ADF_SSF_MAX_PROFILE_COEFS], void *appData);
//-----------------------------------------------------------------------------------
//  END: FIXED POINT MATH ONLY
//-----------------------------------------------------------------------------------
#endif
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  SSF attributes:
//
//  begEndCap: ADF_SSF_ENDCAP_NONE, ADF_SSF_ENDCAP_ROUND, ADF_SSF_ENDCAP_SQUARE,
//  ADF_SSF_ENDCAP_OBLONG, or ADF_SSF_ENDCAP_TRIANGLE. Indicates the type of endcap
//  to apply to the beginning of each stroke skeleton of a stylized stroke-based
//  glyph.
//
//  endEndCap: ADF_SSF_ENDCAP_NONE, ADF_SSF_ENDCAP_ROUND, ADF_SSF_ENDCAP_SQUARE,
```

```
//  ADF_SSF_ENDCAP_OBLONG, or ADF_SSF_ENDCAP_TRIANGLE. Indicates the type of endcap
//  to apply to the end of each stroke skeleton of a stylized stroke-based glyph.
//
//  corner: ADF_SSF_CORNER_NONE, ADF_SSF_CORNER_ROUND, ADF_SSF_CORNER_BEVEL, or
//  ADF_SSF_CORNER_MITER. Indicates the type of corner join to apply to every corner
//  of each stroke skeleton of a stylized stroke-based glyph. Corners are determined
//  procedurally and are placed at every path point of each stroke skeleton that
//  exhibits a significant C1 discontinuity (e.g., when the angle between two
//  consecutive line segments L1 and L2 of a stroke skeleton is less than some
//  threshold a corner join is applied to the point connecting L1 and L2).
//
//  asoEnabled: If this Boolean flag is set to true, then the pen commands of a
//  stylized stroke-based glyph are reordered (during ADF generation) to adhere to
//  traditional Chinese handwriting conventions (e.g., a stroke skeleton consisting
//  of a single horizontal line segment directed from left-to-right is correctly
//  ordered according to traditional conventions; in contrast, a stroke skeleton
//  consisting of a single horizontal line segment directed from right-to-left is
//  incorrectly ordered and therefore would be reversed during ADF generation).
//
//  leftProfile: ADF_SSF_PROFILE_UNIFORM, ADF_SSF_PROFILE_LINEAR,
//  ADF_SSF_PROFILE_QUADRATIC, ADF_SSF_PROFILE_CUBIC, or ADF_SSF_PROFILE_CUSTOM.
//  Indicates the type of profile to use to determine the left edge of each stroke
//  skeleton of a stylized stroke-based glyph. The left profile defines the
//  perpendicular distance from each stroke skeleton to the left edge of its stroke
//  body. The general parametric equation defining the left profile is L(t) = L0 +
//  (t * L1) + (t * t * L2) + (t * t * t * L3), where t is the normalized length
//  along the current stroke skeleton being processed (i.e., t ranges from [0,1]) and
//  L0, L1, L2, and L3 correspond to leftProfileCoefs[0], leftProfileCoefs[1],
//  leftProfileCoefs[2], and leftProfileCoefs[3], respectively. If leftProfile is
//  ADF_SSF_PROFILE_UNIFORM, then only L0 is used. If leftProfile is
//  ADF_SSF_PROFILE_CUSTOM, then the callback function leftProfileCB() is used to
//  determine the perpendicular distance. The output of each leftProfile evaluation
//  must reside in the range [0,1], where a value of V results in a left stroke
//  radius of V * 1/2 * pathWidth.
//
//  rightProfile: ADF_SSF_PROFILE_UNIFORM, ADF_SSF_PROFILE_LINEAR,
//  ADF_SSF_PROFILE_QUADRATIC, ADF_SSF_PROFILE_CUBIC, or ADF_SSF_PROFILE_CUSTOM.
//  Indicates the type of profile to use to determine the right edge of each stroke
//  skeleton of a stylized stroke-based glyph. The right profile defines the
//  perpendicular distance from each stroke skeleton to the right edge of its stroke
//  body. The general parametric equation defining the right profile is R(t) = R0 +
//  (t * R1) + (t * t * R2) + (t * t * t * R3), where t is the normalized length
//  along the current stroke skeleton being processed (i.e., t ranges from [0,1]) and
//  R0, R1, R2, and R3 correspond to rightProfileCoefs[0], rightProfileCoefs[1],
//  rightProfileCoefs[2], and rightProfileCoefs[3], respectively. If rightProfile is
//  ADF_SSF_PROFILE_UNIFORM, then only R0 is used. If rightProfile is
//  ADF_SSF_PROFILE_CUSTOM, then the callback function rightProfileCB() is used to
//  determine the perpendicular distance. The output of each rightProfile evaluation
//  must reside in the range [0,1], where a value of V results in a right stroke
//  radius of V * 1/2 * pathWidth.
//
//  leftProfileCoefs: The left profile coefficients L0, L1, L2, and L3 defined above
//  (see leftProfile).
//
//  rightProfileCoefs: The right profile coefficients R0, R1, R2, and R3 defined
//  above (see rightProfile).
//
//  leftProfileCB: A custom left profile callback function which can be used by
//  applications to define their own pen styles (see the SSF profile data structures
//  and callback functions section above).
//
//  rightProfileCB: A custom right profile callback function which can be used by
//  applications to define their own pen styles (see the SSF profile data structures
//  and callback functions section above).
//
//  appData: A pointer to application specific data. appData is an input argument to
//  the custom profile callback functions leftProfileCB and rightProfileCB.
```

```
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  Note: Due to the inexact nature of the representation of numbers during the
//  rasterization of SSF glyphs, the normalized length t along a stroke skeleton will
//  range from 0-epsilon to 1+epsilon, where epsilon is a very small fractional value
//  (e.g., 0.0001).
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
typedef struct {
    ADF_U32 begEndCap;      // ADF_SSF_ENDCAP_NONE, ADF_SSF_ENDCAP_ROUND, etc.
    ADF_U32 endEndCap;      // ADF_SSF_ENDCAP_NONE, ADF_SSF_ENDCAP_ROUND, etc.
    ADF_U32 corner;         // ADF_SSF_CORNER_NONE, ADF_SSF_CORNER_ROUND, etc.
    ADF_U32 asoEnabled;     // Asian Stroke Orientation is 1: enabled, 0: disabled
    ADF_U32 leftProfile;    // ADF_SSF_PROFILE_UNIFORM, ADF_SSF_PROFILE_LINEAR, etc.
    ADF_U32 rightProfile;   // ADF_SSF_PROFILE_UNIFORM, ADF_SSF_PROFILE_LINEAR, etc.
    ADF_F32 leftProfileCoefs[ADF_SSF_MAX_PROFILE_COEFS];    // Left profile coefs
    ADF_F32 rightProfileCoefs[ADF_SSF_MAX_PROFILE_COEFS];   // Right profile coefs
    ADFSSFDistFromProfileCB leftProfileCB;  // Custom left profile callback function
    ADFSSFDistFromProfileCB rightProfileCB; // Custom right profile callback function
    void *appData;          // Pointer to application specific data
}   ADFSSFAttrs;
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  ADFPath attributes
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
typedef struct {
    ADF_U32     pathType;          // ADF_OUTLINE_PATH, ADF_UNIFORM_STROKE_PATH, etc.
    ADF_F32     pathWidth;         // The stroke width of a stroke-based glyph
    ADFSSFAttrs ssfAttrs;          // SSF attributes
    ADF_F32     glyphMinX;         // The minimum x-coordinate of the glyph's bBox
    ADF_F32     glyphMinY;         // The minimum y-coordinate of the glyph's bBox
    ADF_F32     glyphMaxX;         // The maximum x-coordinate of the glyph's bBox
    ADF_F32     glyphMaxY;         // The maximum y-coordinate of the glyph's bBox
    ADF_F32     fontUnitsPerEM;    // Em box size in floating point font units
    ADF_U32     charCode;          // Unicode character code of the glyph
    ADF_U32     numContours;       // # of glyph contours of an outline-based glyph
    ADF_U32     numPenCmds;        // Total number of pen commands in the ADFPath
    ADFPenCmd   *penCmds;          // Pointer to the pen commands defining the glyph
    ADF_U32     algnZones[2];      // An encoding of the glyph's SAZ alignment zones
    ADF_U32     algnZonesMask;     // If non-zero, then algnZones is valid
    void        *appData;          // Pointer to application specific data
}   ADFPath;
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
//  ADF GENERATION FROM ADFPATHS
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  Attributes for generating an ADF from an ADFPath. These attributes are ignored
//  when generating implicit ADFs, but are required when generating explicit ADFs.
//
//  Explicit ADFs are represented as a spatial hierarchy of explicit ADF cells, where
//  each explicit ADF cell contains a set of sampled distance values and a
//  reconstruction method which is used to reconstruct the distance field within the
//  explicit ADF cell. Explicit ADFs are generated using top-down spatial
//  subdivision; maxLevel limits the depth of the spatial hierarchy. During explicit
//  ADF generation, each explicit ADF cell is subdivided until maxLevel is reached or
//  until the error between the reconstructed distance field and the true distance
//  field of the ADFPath is less than maxError within the explicit ADF cell. In
//  applications that only require an accurate representation of the edge of the
//  ADFPath, such as in glyph rendering, it is not necessary for explicit ADF cells
//  that do not contain the edge to meet the maxError constraint. In this case,
//  distEps should be set to 0. However, some applications (e.g., applications that
//  perform collision detection between glyphs) require an accurate representation of
//  the distance field away from the edge. A positive distEps will force explicit ADF
```

```
//  cells within distEps of the edge to meet the maxError constraint. distEps is
//  expressed in ADF units.
//
//  After explicit or implicit ADF generation, all relevant ADFPath data required for
//  rendering (e.g., the SAZ alignment zones) is represented internally in the ADF
//  data structure, thereby allowing the application to destroy the ADFPath.
//-----------------------------------------------------------------------------------
typedef struct {        // Attributes required for generating explicit ADFs
    ADF_U32 maxLevel;   // Never exceed this level when generating
    ADF_F32 maxError;   // Maximum error between ADF and ADFPath's distance field
    ADF_F32 distEps;    // Use to force non-edge cells to pass maximum error test
}   ADFGenAttrs;


//-----------------------------------------------------------------------------------
//  Generate an ADF from the specified ADFPath using the specified generation
//  attributes. This function returns an opaque pointer to the generated ADF; a
//  NULL is returned if the request cannot be satisfied.
//-----------------------------------------------------------------------------------
void *ADFGenerateADF (void *libInst, ADFPath *path, ADFGenAttrs *genAttrs);


//-----------------------------------------------------------------------------------
//  Destroy the given ADF
//-----------------------------------------------------------------------------------
void ADFDestroyADF (void *libInst, void *ADF);


//-----------------------------------------------------------------------------------
//  The representation of an ADF is opaque (i.e., a void * pointer) and not
//  accessible to the application. Applications can, however, query the ADF for its
//  size, thus enabling the ADF to be copied, cached, transmitted, etc., for various
//  application specific uses. ADFs are represented as a contiguous block of memory
//  beginning at the opaque pointer. All internal ADF references (e.g., to ADF cells)
//  are stored as offsets, rather than direct pointers, thereby allowing applications
//  to freely move ADFs throughout their system.
//-----------------------------------------------------------------------------------
ADF_U32 ADFGetADFSize (void *libInst, void *ADF);


//-----------------------------------------------------------------------------------
//  ALIGNMENT ZONES
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Alignment zones identify strong vertical and horizontal edges of glyphs and
//  characteristic distances (i.e., zones) of a typeface (e.g., baseline to x-height
//  and baseline to cap-height distances). Alignment zones are determined directly
//  from the ADF of each glyph and are used to build the appropriate transformation
//  needed for grid fitting each ADF to the pixel grid or to the sub-pixel grid.
//
//  This library supports two alignment zone systems: Standard Alignment Zones (SAZ)
//  and Multiple Alignment Zones (MAZ). There are three important differences between
//  these two alignment zone systems:
//
//      1) SAZ alignment zones are designed and optimized for non-CJK glyphs (e.g.,
//         Latin or Hebrew glyphs), whereas MAZ alignment zones are designed and
//         optimized for CJK glyphs.
//
//      2) SAZ alignment zones are determined in a three step preprocess (described
//         immediately below), whereas MAZ alignment zones are determined dynamically
//         during rendering.
//
//      3) SAZ alignment zones can be used to align glyphs either to the pixel grid
//         or to the sub-pixel grid, whereas MAZ alignment zones can only be used to
//         align glyphs to the pixel grid (i.e., alignment to the sub-pixel grid is
//         not supported by MAZ alignment zones).
//
```

```
//  SAZ alignment zones are determined in a three step preprocess:
//
//      1) Initialize SAZ alignment zone detection for a specified typeface
//      2) Detect SAZ alignment zones for each required glyph in the specified
//         typeface
//      3) Terminate SAZ alignment zone detection for the specified typeface
//
//  Because initialization involves some overhead, it is recommended that the SAZ
//  alignment zones for all of the required glyphs of a particular typeface be
//  determined before termination.
//
//  Note that initialization requires the ADFPaths of a small set of glyphs in the
//  typeface from which characteristic distances (i.e., zones) are determined. Hence,
//  the application must supply a callback function for fetching the ADFPath of an
//  arbitrary glyph from the typeface; when the ADFPath has been processed and is no
//  longer needed, a corresponding callback function for 'releasing' the ADFPath is
//  invoked. The release callback function can be used to free the ADFPath if the
//  corresponding fetch callback function allocated the ADFPath on demand.
//
//  After a glyph's SAZ alignment zones are determined, they are encoded in the
//  corresponding ADFPath. During ADF generation, the SAZ alignment zones of the
//  ADFPath are copied into the ADF data structure and used for grid fitting during
//  rendering.
//
//  The current implementation supports SAZ alignment zone detection for the
//  following Unicode character code sets:
//
//      - Basic Latin
//      - Latin 1
//      - Latin Extended A
//      - Latin Extended B
//      - Devanagari
//      - Arabic
//      - Arabic Supplement
//      - Arabic Presentation Forms A
//      - Arabic Presentation Forms B
//      - Hebrew
//      - Hebrew Presentation Forms
//      - Thai
//
//  The current implementation supports MAZ alignment zone detection for all Unicode
//  character code sets. However, MAZ alignment zones are designed and optimized for
//  CJK glyphs. Applications are strongly discouraged from using MAZ alignment zones
//  to grid fit non-CJK glyphs.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  A callback function provided by the application and invoked by
//  ADFInitAlgnZoneDetection() to fetch the ADFPath of a specified glyph of the
//  typeface being processed. A NULL is returned if the ADFPath is not available.
//  appInst is the application instance data provided by the application when
//  ADFInitSystem() was invoked (it can be used for memory allocation). fontID is an
//  opaque pointer provided by the application to ADFInitAlgnZoneDetection() that
//  identifies the typeface. charCode is the Unicode value for the glyph whose
//  ADFPath is required.
//-----------------------------------------------------------------------------------
typedef ADFPath *ADFGetGlyphCB (void *appInst, void *fontID, ADF_U32 charCode);


//-----------------------------------------------------------------------------------
//  A callback function provided by the application and invoked by
//  ADFInitAlgnZoneDetection() to release the ADFPath of a specified glyph of the
//  typeface being processed. appInst is the application instance data provided by
//  the application when ADFInitSystem() was invoked. fontID is an opaque pointer
//  provided by the application to ADFInitAlgnZoneDetection() that identifies the
//  typeface. charCode is the Unicode value for the glyph represented by path. path
//  is the ADFPath to be released.
//-----------------------------------------------------------------------------------
```

```
typedef void ADFReleaseGlyphCB (void *appInst, void *fontID, ADF_U32 charCode,
ADFPath *path);


//-----------------------------------------------------------------------------
//  Initialize SAZ alignment zone detection for the specified typeface fontID.
//  fontName is a null-terminated string identifying the name of the typeface; an
//  empty string is specified if no name is available. As described above, getGlyphCB
//  and releaseGlyphCB are application callback functions for fetching and releasing
//  the ADFPath of a specified glyph of the typeface fontID. A NULL releaseGlyphCB is
//  valid, thus allowing the application to avoid specifying a release function when
//  it is not necessary.
//
//  This function returns an opaque pointer to SAZ 'alignment zone state' that is
//  used internally for detecting the SAZ alignment zones of individual glyphs via
//  ADFDetectAlgnZones(). This state is destroyed by ADFTermAlgnZoneDetection() after
//  the SAZ alignment zones for all of the required glyphs of the fontID typeface are
//  determined. A NULL opaque pointer is returned if the request cannot be satisfied.
//-----------------------------------------------------------------------------
void *ADFInitAlgnZoneDetection (void *libInst, void *fontID, ADF_I8 *fontName,
ADFGetGlyphCB *getGlyphCB, ADFReleaseGlyphCB *releaseGlyphCB);


//-----------------------------------------------------------------------------
//  Detect the SAZ alignment zones of a glyph described by the ADFPath path. Upon a
//  successful return, path->algnZonesMask will be set to a non-zero value and
//  path->algnZones will contain the SAZ alignment zones for the glyph; if the
//  detection fails, path->algnZonesMask is set to zero. algnZoneState is the SAZ
//  alignment zone state returned by ADFInitAlgnZoneDetection(). This function
//  requires that the Unicode character code in the path (path->charCode) be valid.
//-----------------------------------------------------------------------------
void ADFDetectAlgnZones (void *libInst, void *algnZoneState, ADFPath *path);


//-----------------------------------------------------------------------------
//  Terminate SAZ alignment zone detection for the typeface corresponding to the
//  algnZoneState returned by ADFInitAlgnZoneDetection().
//-----------------------------------------------------------------------------
void ADFTermAlgnZoneDetection (void *libInst, void *algnZoneState);


//-----------------------------------------------------------------------------
//  DENSITY IMAGES
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  Density images are used in this library to record a rendered ADF. When
//  performing CRT rendering, density images must have one channel (i.e., type is
//  ADF_IMAGE_TYPE_GRAY); for LCD rendering, density images must have four channels
//  (i.e., type is ADF_IMAGE_TYPE_RGBA).
//
//  Application-specific data can be stored along with a density image, thus
//  providing a convenient and efficient mechanism for associating arbitrary data
//  with density images. The application-specific data is comprised of a
//  variable-length array of ADF_U32s; the length of this array is specified when
//  creating a density image. One possible use of this array is to store data
//  required for typesetting and displaying a glyph whose density image is stored in
//  a density image cache.
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
#define ADF_IMAGE_TYPE_GRAY     0   // 1 channel density image
#define ADF_IMAGE_TYPE_RGBA     1   // 4 channel density image
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
typedef struct {
    ADF_U16  type;        // ADF_IMAGE_TYPE_GRAY or ADF_IMAGE_TYPE_RGBA
    ADF_U16  w;           // Width in pixels
    ADF_U16  h;           // Height in pixels
```

```
    ADF_U16  appDataLen; // Length of appData[] array
    ADF_U32  *appData;   // Application-specific data
    ADF_U8   *base;      // Ptr to pixels (GG... or RGBARGBA); 8 bits per channel
}   ADFImage;
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  This library uses the following macro to pack density values R, G, B, and A into
//  a 32 bit word. Applications can #ifdef this macro to accommodate their specific
//  requirements (e.g., the byte order (ADF_CPU_BYTE_ORDER) of particular CPUs and
//  the preferred packing order of the RGBA values in a 32 bit word).
//-----------------------------------------------------------------------------
#define ADF_PACK_RGBA(R,G,B,A)  (((A) << 24) | ((B) << 16) | ((G) << 8) | (R))


//-----------------------------------------------------------------------------
//  Create a density image of the specified type and size. A variable-length array of
//  ADF_U32s can be allocated along with the density image to store
//  application-specific data associated with the density image; appDataLen specifies
//  the length of this array. A NULL is returned if the request cannot be satisfied.
//-----------------------------------------------------------------------------
ADFImage *ADFCreateImage (void *libInst, ADF_U16 type, ADF_U16 w, ADF_U16 h,
ADF_U16 appDataLen);


//-----------------------------------------------------------------------------
//  Destroy the given density image
//-----------------------------------------------------------------------------
void ADFDestroyImage (void *libInst, ADFImage *image);


//-----------------------------------------------------------------------------
//  RENDERING
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  ADF rendering proceeds in four steps:
//
//  1) Application rendering attributes, such as point size and display mode, are
//     set by the application for a particular ADF glyph. These attributes are
//     defined in the ADFRenderAttrs data structure.
//
//  2) The application rendering attributes are then processed by ADFRenderSetup()
//     to determine:
//
//     a) ADFRenderGlyphData, a structure containing data, such as the rendering
//        transformation, required by ADFRenderGlyph() to render the ADF glyph
//        according to the application rendering attributes.
//
//     b) ADFRenderImageAttrs, a structure containing attributes, such as the image
//        size and image origin, required by the application to prepare a density
//        image for rendering the ADF glyph. The image size can be used by the
//        application to allocate a density image via ADFCreateImage(); the image
//        origin specifies the location of the bottom-left corner of the density
//        image in integer pixel coordinates. This location can be used by the
//        application to position the density image in the display buffer when
//        compositing.
//
//     c) ADFTypesetAttrs, a structure containing attributes required by the
//        application for adjusting the font metrics due to grid fitting.
//
//     d) ADFCacheGlyphData, a structure containing data required by the application
//        for positioning and typesetting a cached density image of the ADF glyph
//        via ADFPositionCachedImage(). Refer to ADFPositionCachedImage() for an
//        overview on typesetting cached density images.
//
//  3) A density image of the required size is prepared by the application (e.g.,
//     allocated via ADFCreateImage()).
//
```

```
//  4) ADFRenderGlyph() is invoked by the application to render the ADF glyph into
//     the prepared density image using the corresponding ADFRenderGlyphData. If
//     requested, ADFRenderGlyph() performs MAZ alignment zone detection and grid
//     fitting on the ADF glyph during the rendering process.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  ADFRenderAttrs (set by the application):
//
//  penX, penY: The x and y coordinates of the pen position, which determines the
//  placement of the ADF glyph origin. penX and penY are specified in floating
//  point pixel coordinates.
//
//  pointSize: The requested point size for the ADF glyph.
//
//  dpi: The dots-per-inch of the display device.
//
//  scaleX, scaleY: Additional x and y scale factors for scaling the ADF glyph
//  beyond the requested point size. If the ADF glyph is an explicit ADF, non-uniform
//  scaling (i.e., where scaleX does not equal scaleY) distorts the distance field of
//  the ADF glyph, which can result in poor quality antialiasing. If the ADF glyph is
//  an implicit ADF, non-uniform scaling has no negative impact on the quality of the
//  antialiasing.
//
//  rotationPtX, rotationPtY: The x and y coordinates of the center of rotation
//  for the ADF glyph. rotationPtX and rotationPtY are specified in floating point
//  pixel coordinates.
//
//  rotationAngle: The rotation angle applied to the ADF glyph. rotationAngle is
//  specified in radians. Note that grid fitting is automatically disabled when a
//  non-zero rotationAngle is specified.
//
//  displayMode: ADF_REND_MODE_CRT, ADF_REND_MODE_RGBv, ADF_REND_MODE_BGRv,
//  ADF_REND_MODE_RGBh, or ADF_REND_MODE_BGRh. This element determines whether
//  the ADF glyph is rendered in CRT mode (ADF_REND_MODE_CRT) or LCD mode (all
//  others). If an LCD mode is chosen, displayMode also specifies the physical
//  layout of the display's sub-pixels, i.e., whether the display is horizontally
//  or vertically striped, and whether the striping is in RGB or BGR order. This
//  library assumes that the striped sub-pixels have identical dimensions.
//
//  gridFitType: ADF_GRID_FIT_NONE, ADF_GRID_FIT_PIXEL, ADF_GRID_FIT_SUB_PIXEL, or
//  ADF_GRID_FIT_MAZ_PIXEL. ADF_GRID_FIT_NONE disables grid fitting.
//  ADF_GRID_FIT_PIXEL aligns a glyph to the pixel grid using SAZ alignment zones.
//  ADF_GRID_FIT_SUB_PIXEL aligns a glyph to 1/3 of a pixel using SAZ alignment zones
//  (see Usage Note: Typesetting for a discussion on the advantages and disadvantages
//  of sub-pixel grid fitting). ADF_GRID_FIT_MAZ_PIXEL aligns a glyph to the pixel
//  grid using MAZ alignment zones.
//
//  outsideCutoff, insideCutoff: The filter cutoff values for the mapping function
//  which maps ADF distances to density values as described above in Usage Note:
//  Continuous Stroke Modulation. outsideCutoff and insideCutoff are specified in
//  floating point pixel coordinates.
//
//  gamma: The exponent of the gamma curve mapping ADF distances to density values
//  as described above in Usage Note: Continuous Stroke Modulation.
//
//  useColorReduction: The Boolean flag that enables color reduction during LCD
//  rendering. Setting this variable to true enables color reduction. Setting this
//  variable to false disables color reduction. This variable is ignored during CRT
//  rendering (i.e., when displayMode is set to ADF_REND_MODE_CRT).
//
//  colorReductionAmt: If useColorReduction is set to true, this variable controls
//  the amount of color reduction applied during LCD rendering. The value must lie in
//  the range [0, 1]. A value of 0 specifies minimum color reduction. A value of 1
//  specifies maximum color reduction: pixels will be completely desaturated (i.e.,
//  no color at all). Note that a value of 0 will still cause some amount of color
//  reduction to be performed; it will NOT give the same results as turning off color
//  reduction entirely (i.e., by setting useColorReduction to false). This variable
```

```
//  is ignored if useColorReduction is set to false. This variable is also ignored if
//  displayMode is set to ADF_REND_MODE_CRT.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Applications can perform their own grid fitting instead of using the automatic
//  hinting (based on SAZ or MAZ alignment zones) provided by this library. In order
//  to render application-hinted glyphs correctly, applications must perform certain
//  steps to ensure that this library makes no adjustments to the points (i.e., the x
//  and y coordinates) that define the hinted glyphs.
//
//  The steps required to set up an ADFPath for an application-hinted glyph are:
//
//  1) The application obtains the points (in font units) that define the glyph.
//
//  2) The application scales the coordinates of each point to pixel coordinates. For
//     example, if the desired ppem is 10.0 and the em box size (in font units) is
//     2048.0, the application scales the coordinates of each point by 10.0 / 2048.0.
//
//  3) The application applies its own grid fitting to the point data in pixel
//     coordinates (e.g., by using TrueType hints). If the glyph is a stroke-based
//     glyph, the application also determines the hinted stroke width (typically an
//     integral value).
//
//  4) The application sets the ADFPath data structure. In particular:
//       - The application sets fontUnitsPerEM to the ppem value (e.g., 10.0).
//       - The application specifies the coordinates of each pen command in floating
//         point pixel coordinates (instead of in floating point font units).
//       - The application specifies the glyph's bounding box (i.e., glyphMinX,
//         glyphMinY, glyphMaxX, glyphMaxY) in floating point pixel coordinates
//         (instead of in floating point font units).
//       - If the glyph is a stroke-based glyph, the application sets pathWidth to
//         the hinted stroke width in floating point pixel coordinates (instead of in
//         floating point font units).
//
//  After the application generates an ADF from the ADFPath, ADF rendering of an
//  application-hinted glyph proceeds as described above, except that the application
//  must set certain fields of the ADFRenderAttrs data structure as follows:
//       - The application sets dpi to 72.
//       - The application sets both scaleX and scaleY to 1.
//       - The application sets gridFitType to ADF_GRID_FIT_NONE.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_REND_MODE_CRT   0   // CRT rendering
#define ADF_REND_MODE_RGBv  1   // LCD rendering; RGB components are vertical RGB
#define ADF_REND_MODE_BGRv  2   // LCD rendering; RGB components are vertical BGR
#define ADF_REND_MODE_RGBh  3   // LCD rendering; RGB components are horizontal RGB
#define ADF_REND_MODE_BGRh  4   // LCD rendering; RGB components are horizontal BGR
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_GRID_FIT_NONE       0   // No grid fitting
#define ADF_GRID_FIT_PIXEL      1   // Grid fit to the pixel grid using SAZ
#define ADF_GRID_FIT_SUB_PIXEL  2   // Grid fit to the sub-pixel grid using SAZ
#define ADF_GRID_FIT_MAZ_PIXEL  3   // Grid fit to the pixel grid using MAZ
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_F32     penX;               // x-coordinate of ADF glyph origin
    ADF_F32     penY;               // y-coordinate of ADF glyph origin
    ADF_F32     pointSize;          // Requested point size
    ADF_U32     dpi;                // Dots-per-inch of the display device
    ADF_F32     scaleX;             // Additional ADF glyph scaling in x
    ADF_F32     scaleY;             // Additional ADF glyph scaling in y
    ADF_F32     rotationPtX;        // x-coord of center of rotation for ADF glyph
    ADF_F32     rotationPtY;        // y-coord of center of rotation for ADF glyph
    ADF_F32     rotationAngle;      // Rotation angle in radians
    ADF_U32     displayMode;        // ADF_REND_MODE_CRT, ADF_REND_MODE_RGBv, etc.
    ADF_U32     gridFitType;        // ADF_GRID_FIT_NONE, ADF_GRID_FIT_PIXEL, etc.
```

```
    ADF_F32     outsideCutoff;      // Outside filter cutoff value for CSM
    ADF_F32     insideCutoff;       // Inside filter cutoff value for CSM
    ADF_F32     gamma;              // Gamma curve exponent for CSM
    ADF_U32     useColorReduction;  // Boolean that turns on/off color reduction
    ADF_F32     colorReductionAmt;  // Controls the amount of color reduction
}   ADFRenderAttrs;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  ADFRenderGlyphData (determined from ADFRenderAttrs by ADFRenderSetup() and used
//  by ADFRenderGlyph() to render the ADF glyph into a density image):
//
//  xform[3][3]: The rendering transformation matrix used to render the ADF glyph.
//
//  displayMode: The original displayMode attribute specified in ADFRenderAttrs.
//
//  gridFitType: The original gridFitType attribute specified in ADFRenderAttrs or
//  ADF_GRID_FIT_NONE if a non-zero rotation angle was specified in ADFRenderAttrs.
//
//  outsideCutoff, insideCutoff: If using implicit ADFs, the original filter cutoff
//  values specified in ADFRenderAttrs. If using explicit ADFs, the filter cutoff
//  values specified in ADFRenderAttrs transformed into ADF coordinates.
//
//  gamma: The original gamma attribute specified in ADFRenderAttrs.
//
//  filterCutoffScale: If using implicit ADFs, filterCutoffScale is not used and
//  therefore is set to one. If using explicit ADFs, filterCutoffScale specifies the
//  scale factor for scaling the filter cutoff values outsideCutoff and insideCutoff
//  from pixel coordinates (as specified in ADFRenderAttrs) to ADF coordinates (as
//  required in ADFRenderGlyphData); by manually scaling the filter cutoff values,
//  applications can perform CSM without repeated calls to ADFRenderSetup(), thereby
//  avoiding repeated computation of ADFRenderGlyphData, ADFRenderImageAttrs, and
//  ADFTypesetAttrs.
//
//  useColorReduction: The original useColorReduction attribute specified in
//  ADFRenderAttrs.
//
//  colorReductionAmt: The original colorReductionAmt attribute specified in
//  ADFRenderAttrs.
//
//  ppem: The pixels per em used during MAZ grid fitting. This element is ignored if
//  gridFitType is not set to ADF_GRID_FIT_MAZ_PIXEL.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_F32     xform[3][3];        // Rendering transformation matrix
    ADF_U32     displayMode;        // ADF_REND_MODE_CRT, ADF_REND_MODE_RGBv, etc.
    ADF_U32     gridFitType;        // ADF_GRID_FIT_NONE, ADF_GRID_FIT_PIXEL, etc.
    ADF_F32     outsideCutoff;      // Filter cutoff for CSM in pixel or ADF coords
    ADF_F32     insideCutoff;       // Filter cutoff for CSM in pixel or ADF coords
    ADF_F32     gamma;              // Gamma curve exponent for CSM
    ADF_F32     filterCutoffScale;  // Used to convert cutoff values to ADF coords
    ADF_U32     useColorReduction;  // Boolean that turns on/off color reduction
    ADF_F32     colorReductionAmt;  // Controls the amount of color reduction
    ADF_F32     ppem;               // PPEM used for MAZ grid fitting
}   ADFRenderGlyphData;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  ADFRenderImageAttrs (determined from ADFRenderAttrs by ADFRenderSetup() and
//  used by the application):
//
//  imageOriginX, imageOriginY: The location of the bottom-left corner of the
//  density image in integer pixel coordinates. This location can be used by the
//  application to position the density image in the display buffer when
//  compositing.
//
//  imageW, imageH: The width and height in pixels of the density image required for
//  rendering the ADF glyph via ADFRenderGlyph(). These elements can be used by
```

```
//  the application to allocate a density image via ADFCreateImage().
//
//  imageSubPixelType: This element is valid if gridFitType is set to
//  ADF_GRID_FIT_SUB_PIXEL in ADFRenderAttrs, i.e., if the application has chosen to
//  align the ADF glyph to the sub-pixel grid using SAZ alignment zones. In this
//  case, when imageSubPixelType is ADF_SUB_PIXEL_1 the ADF glyph will be aligned to
//  the first sub-pixel (e.g., the R component when displayMode is
//  ADF_REND_MODE_RGBv); when imageSubPixelType is ADF_SUB_PIXEL_2 the ADF glyph will
//  be aligned to the second sub-pixel (e.g., the G component when displayMode is
//  ADF_REND_MODE_RGBv); etc. Distinguishing between rendered density images aligned
//  to different sub-pixels is necessary when caching the images.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_SUB_PIXEL_1     0   // Sub-pixel grid fitting to the first sub-pixel
#define ADF_SUB_PIXEL_2     1   // Sub-pixel grid fitting to the second sub-pixel
#define ADF_SUB_PIXEL_3     2   // Sub-pixel grid fitting to the third sub-pixel
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_I32     imageOriginX;       // x-coordinate of bottom-left corner of image
    ADF_I32     imageOriginY;       // y-coordinate of bottom-left corner of image
    ADF_U16     imageW;             // Width of glyph density image in pixels
    ADF_U16     imageH;             // Height of glyph density image in pixels
    ADF_U32     imageSubPixelType;  // ADF_SUB_PIXEL_1, ADF_SUB_PIXEL_2, etc.
}   ADFRenderImageAttrs;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  ADFTypesetAttrs (determined from ADFRenderAttrs by ADFRenderSetup() and used
//  by the application):
//
//  FUToPixelScaleX, FUToPixelScaleY: These elements are scale factors from font
//  units to floating point pixel units for the requested point size, dpi, and
//  additional scale factors, scaleX and scaleY. These elements are computed as
//  follows:
//
//      pixelsPerEM = pointSize * (dpi / 72)
//      FUToPixelScaleX = scaleX * pixelsPerEM / fontUnitsPerEM
//      FUToPixelScaleY = scaleY * pixelsPerEM / fontUnitsPerEM
//
//  Because scaleX and scaleY have been factored into this computation,
//  FUToPixelScaleX and FUToPixelScaleY will not necessarily match the corresponding
//  scale factors that various font standards (e.g., TTF and OpenType) would specify.
//  Note that FUToPixelScaleX and FUToPixelScaleY do not take into account the
//  incremental scaling of the ADF glyph due to grid fitting.
//
//  adjFUToPixelScaleX, adjFUToPixelScaleY: If a non-zero rotation angle is specified
//  in the ADFRenderAttrs data structure, then adjFUToPixelScaleX and
//  adjFUToPixelScaleY are set to FUToPixelScaleX and FUToPixelScaleY, respectively.
//  Otherwise, these elements are set as follows. If the gridFitType element of the
//  ADFRenderAttrs data structure is set to either ADF_GRID_FIT_PIXEL or
//  ADF_GRID_FIT_SUB_PIXEL, these elements are scale factors from font units to
//  floating point pixel units that have been adjusted for the incremental scaling of
//  the ADF glyph due to grid fitting. These values can be used to modify the font
//  metrics for typesetting as described above in Usage Note: Typesetting. If the
//  gridFitType element of the ADFRenderAttrs data structure is set to either
//  ADF_GRID_FIT_NONE or ADF_GRID_FIT_MAZ_PIXEL, these elements are set to
//  FUToPixelScaleX and FUToPixelScaleY, respectively.
//
//  gridAlgnAdjX, gridAlgnAdjY: If a non-zero rotation angle is specified in the
//  ADFRenderAttrs data structure, then gridAlgnAdjX and gridAlgnAdjY are both set to
//  zero. Otherwise, these elements are set as follows. If the gridFitType element of
//  the ADFRenderAttrs data structure is set to either ADF_GRID_FIT_PIXEL or
//  ADF_GRID_FIT_SUB_PIXEL, these elements provide the incremental translation of the
//  pen position from the requested pen position due to grid fitting. These values
//  can be used to modify the font metrics for typesetting as described above in
//  Usage Note: Typesetting. If the gridFitType element of the ADFRenderAttrs data
//  structure is set to either ADF_GRID_FIT_NONE or ADF_GRID_FIT_MAZ_PIXEL,
```

```
//  gridAlgnAdjX and gridAlgnAdjY are both set to zero.
//
//  Note that all of the attributes in ADFTypesetAttrs are determined in unrotated
//  space; if the application is typesetting rotated ADF glyphs, the attributes in
//  ADFTypesetAttrs must be adjusted accordingly. For example, assuming a rotated
//  horizontal baseline and left-to-right typesetting, the advancement of the pen
//  position (penX, penY) for the next glyph can be computed as follows:
//
//      dx = advWidthCurrentGlyph * FUToPixelScaleX;
//      penX += dx * cos(rotationAngle);
//      penY += dx * sin(rotationAngle);
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_F32     FUToPixelScaleX;    // Unadjusted x-scale from font units to pixels
    ADF_F32     FUToPixelScaleY;    // Unadjusted y-scale from font units to pixels
    ADF_F32     adjFUToPixelScaleX; // Adjusted x-scale from font units to pixels
    ADF_F32     adjFUToPixelScaleY; // Adjusted y-scale from font units to pixels
    ADF_F32     gridAlgnAdjX;       // Pen adjustment in x due to grid fitting
    ADF_F32     gridAlgnAdjY;       // Pen adjustment in y due to grid fitting
}   ADFTypesetAttrs;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  ADFCacheGlyphData is private data determined from ADFRenderAttrs by
//  ADFRenderSetup() to enable the application to position and typeset a cached
//  density image of the ADF glyph via ADFPositionCachedImage(). Refer to
//  ADFPositionCachedImage() for an overview on typesetting cached density images.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
#define ADF_CACHE_GLYPH_DATA_LEN    12  // Length of the private data block
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_U32 data[ADF_CACHE_GLYPH_DATA_LEN];
}   ADFCacheGlyphData;
//-----------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------
//  Setup for ADF glyph rendering. Application rendering attributes are processed
//  by ADFRenderSetup() to determine:
//
//  a) ADFRenderGlyphData, such as the rendering transformation, required by
//     ADFRenderGlyph() to render the ADF glyph according to the application
//     rendering attributes.
//
//  b) ADFRenderImageAttrs, such as the image size and image origin, required by
//     the application to prepare a density image for rendering the ADF glyph.
//
//  c) ADFTypesetAttrs, required by the application for adjusting the font metrics
//     due to grid fitting.
//
//  d) ADFCacheGlyphData, required by the application for positioning and typesetting
//     a cached density image of the ADF glyph via ADFPositionCachedImage(). If this
//     data is not required (e.g., when density images are not being cached), set the
//     pointer to this structure to NULL.
//-----------------------------------------------------------------------------------
void ADFRenderSetup (void *libInst, void *ADF, ADFRenderAttrs *renderAttrs,
ADFRenderGlyphData *renderGlyphData, ADFRenderImageAttrs *renderImageAttrs,
ADFTypesetAttrs *typesetAttrs, ADFCacheGlyphData *cacheGlyphData);


//-----------------------------------------------------------------------------------
//  ADFRenderSetupFromPath() determines the same output attributes as
//  ADFRenderSetup() (i.e., ADFRenderGlyphData, ADFRenderImageAttrs, ADFTypesetAttrs,
//  and ADFCacheGlyphData), but it determines these output attributes from an ADFPath
//  rather than from an ADF. Only a subset of the elements of the ADFPath is
```

```
//  required to perform this operation: pathType, pathWidth, glyphMinX, glyphMinY,
//  glyphMaxX, glyphMaxY, fontUnitsPerEM, algnZones[2], and algnZonesMask. The
//  determined output attributes are identical to those that would be computed by
//  first generating an ADF from the ADFPath via ADFGenerateADF() and then calling
//  ADFRenderSetup() as long as the required elements of the ADFPath used in both
//  approaches are identical.
//
//  ADFRenderSetupFromPath() can be used by applications in various ways. For
//  example, it can be used to determine the attributes for typesetting an ADF glyph
//  directly from an ADFPath, thus allowing applications to typeset glyphs without
//  generating the ADFs of the glyphs (which can be useful when using explicit ADFs
//  where the generation overhead may degrade typesetting performance).
//  ADFRenderSetupFromPath() can also be used as an alternative to
//  ADFPositionCachedImage() for positioning and typesetting cached density images
//  (see below). Applications may find ADFRenderSetupFromPath() simpler to use than
//  ADFPositionCachedImage() and more convenient depending on their particular data
//  structures. In general, however, ADFRenderSetupFromPath() is slower than
//  ADFPositionCachedImage(), particularly when using explicit ADFs which require an
//  inverse transformation to be computed during ADFRenderSetupFromPath().
//-----------------------------------------------------------------------------------
void ADFRenderSetupFromPath (void *libInst, ADFPath *path, ADFRenderAttrs
*renderAttrs, ADFRenderGlyphData *renderGlyphData, ADFRenderImageAttrs
*renderImageAttrs, ADFTypesetAttrs *typesetAttrs, ADFCacheGlyphData *cacheGlyphData);


//-----------------------------------------------------------------------------------
//  Render the ADF glyph into the specified density image using the rendering data
//  ADFRenderGlyphData determined by ADFRenderSetup()
//-----------------------------------------------------------------------------------
void ADFRenderGlyph (void *libInst, void *ADF, ADFRenderGlyphData *renderGlyphData,
ADFImage *image);


//-----------------------------------------------------------------------------------
//  POSITIONING AND TYPESETTING CACHED DENSITY IMAGES:
//
//  Given a new pen position for a cached density image, ADFPositionCachedImage()
//  determines the new image origin, the new image sub-pixel type (if sub-pixel grid
//  fitting was specified when the cached density image was rendered), and the new
//  typesetting attributes for the cached density image whose cached glyph data is
//  given by cacheGlyphData (which is determined via ADFRenderSetup()). This function
//  assumes that the cached density image was grid fit to either the pixel grid or
//  the sub-pixel grid; determining the image placement and typesetting attributes
//  for cached density images that were not grid fit is application-dependent (it
//  typically requires application-specific quantizing of rendering attributes such
//  as the pen position) and is not supported by this function.
//
//  Upon entry, renderAttrs must contain the new pen position (penX, penY) for the
//  ADF glyph represented by the cached density image. Upon exit:
//
//  1) renderImageAttrs contains the new image origin for the cached density image
//     and the new image sub-pixel type (if sub-pixel grid fitting was specified when
//     the cached density image was rendered) based on the new pen position. Note
//     that the image width and height in renderImageAttrs are not set by this
//     function (the width and height of the cached density image are invariant to
//     pen position changes and therefore are not recomputed).
//
//  2) typesetAttrs contains the new typesetting attributes for the cached density
//     image based on the new pen position.
//
//
//  WHEN GRID FITTING TO THE PIXEL GRID:
//
//  When caching density images and grid fitting to the pixel grid using either SAZ
//  or MAZ alignment zones, a typical application can augment the ADF rendering
//  process as follows to support the typesetting of cached density images:
//
```

```
//  1) When rendering the ADF glyph for the first time
//     a) Invoke ADFRenderSetup() to determine the cacheGlyphData for the ADF glyph
//        and the given renderAttrs
//     b) Create a density image with application-specific data for storing the
//        cacheGlyphData
//     c) Invoke ADFRenderGlyph() to render the ADF glyph into the density image
//     d) Store the cacheGlyphData as appData in the density image
//     e) Insert the density image into the cache
//     f) Typeset the density image according to the typesetAttrs and
//        renderImageAttrs determined by ADFRenderSetup()
//
//  2) When typesetting a cached density image for a new pen position
//     a) Determine the new typesetAttrs and renderImageAttrs via
//        ADFPositionCachedImage() using the cacheGlyphData stored as
//        application-specific data in the cached density image
//     b) Typeset the density image according to the new typesetAttrs and
//        renderImageAttrs
//
//
//  WHEN GRID FITTING TO THE SUB-PIXEL GRID:
//
//  When caching density images and grid fitting to the sub-pixel grid using SAZ
//  alignment zones, up to three density images must be cached for renderAttrs which
//  differ only by the pen position, where each image corresponds to the alignment of
//  the ADF glyph with a different sub-pixel. For a new pen position, the appropriate
//  density image and the new typesetAttrs and renderImageAttrs can be determined
//  from the new pen position and the cacheGlyphData of any of the three density
//  images.
//
//  Determining the placement and typesetting of a cached density image when grid
//  fitting to the sub-pixel grid is similar to the approach outlined for grid
//  fitting to the pixel grid but requires a two-step cache-retrieval process:
//
//  1) Determine the new image origin, new image sub-pixel type, and new typesetAttrs
//     via ADFPositionCachedImage() using the cacheGlyphData of any of the three
//     cached density images.
//
//  2) Retrieve the appropriate density image from the cache (i.e., the density image
//     which corresponds to the new image sub-pixel type) and typeset the density
//     image according to the new typesetAttrs and renderImageAttrs determined in
//     step 1.
//
//  Three plausible approaches for storing and retrieving cacheGlyphData are outlined
//  here:
//
//  1) Store cacheGlyphData as application-specific data for every cached density
//     image. In this approach, the application can query the cache for any of the
//     three density images to retrieve the cacheGlyphData required by
//     ADFPositionCachedImage().
//
//  2) Store cacheGlyphData as application-specific data for a 'data image' of zero
//     width and height (using a cache key similar to the key used for the cached
//     density image) and query the cache for the data image to retrieve the
//     cacheGlyphData required by ADFPositionCachedImage().
//
//  3) Store cacheGlyphData in a glyph- and renderAttrs-dependent array managed
//     by the application and fetch the cacheGlyphData required by
//     ADFPositionCachedImage() accordingly.
//-----------------------------------------------------------------------------
void ADFPositionCachedImage (void *libInst, ADFRenderAttrs *renderAttrs,
ADFCacheGlyphData *cacheGlyphData, ADFRenderImageAttrs *renderImageAttrs,
ADFTypesetAttrs *typesetAttrs);


//-----------------------------------------------------------------------------
//  DUAL CACHING SYSTEM
//-----------------------------------------------------------------------------
```

```
//-----------------------------------------------------------------------------------
//  Cache Overview:
//
//  The Saffron type system provides a least recently used (LRU) caching system for
//  caching ADFs and density images; cached ADFs and density images are referred to
//  as 'elements' in this API. This dual caching system allows the application to
//  increase the overall effective rendering performance. For most applications using
//  explicit ADFs, the use of an ADF cache is prudent because explicit ADF generation
//  from an ADFPath is the slowest process in the rendering pipeline. When using
//  implicit ADFs, the use of an ADF cache is less critical because implicit ADF
//  generation is much faster than implicit ADF rendering. Use of a density image
//  cache can significantly increase rendering performance when viewing documents
//  with repeated use of identically rendered glyphs (e.g., when viewing most PDF
//  documents). Because ADFs can be rendered at any size and orientation, cached ADFs
//  can be reused in animations that scale and rotate type. In contrast, because each
//  density image represents a single size and orientation, the effectiveness of a
//  density image cache can be reduced when animating type.
//
//  The dual caching system can be used during rendering as follows:
//
//  - Create an ADF cache and a density image cache using ADFCreateCache()
//  - For each glyph to be rendered
//      - Build a density image key for the glyph based on its fontID, charCode,
//        and rendering parameters (e.g., pointSize)
//      - Use ADFGetCacheElement() to retrieve the density image from the density
//        image cache with the built key
//      - If ADFGetCacheElement() returns a cache miss:
//          - Build an ADF key for the glyph based on its fontID and charCode
//          - Use ADFGetCacheElement() to retrieve the ADF from the ADF cache
//            with the built key
//          - If ADFGetCacheElement() returns a cache miss:
//              - Generate an ADF from the glyph's ADFPath
//              - Insert the ADF into the ADF cache via ADFInsertCacheElement()
//          - Render the ADF to produce a density image
//          - Insert the density image into the density image cache via
//            ADFInsertCacheElement()
//      - Composite the density image into the display buffer
//  - Destroy both caches before terminating the type rendering system
//
//  Note that the ADFs and density images stored in the dual caching system must
//  have been created with ADFGenerateADF() and ADFCreateImage(), respectively,
//  thereby allowing them to be properly destroyed by the system.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Cache creation attributes (an argument of ADFCreateCache()):
//
//  maxCacheSizeBytes: The total size (in bytes) of all the elements in the cache
//  will never exceed this limit. This size does not include the size of the hash
//  table used by the cache (see hashTableSizeExp).
//
//  maxCacheNumElms: The total number of elements in the cache will never exceed
//  this limit.
//
//  cacheType: A cache can store either ADFs (ADF_CACHE_TYPE_ADF) or density images
//  (ADF_CACHE_TYPE_IMAGE).
//
//  hashTableSizeExp: The exponent that governs the number of entries in the hash
//  table used by the cache. The number of entries is 2^hashTableSizeExp and the
//  size of the table (in bytes) is 2 * sizeof(void *) * 2^hashTableSizeExp. Larger
//  hash tables result in fewer collisions during hashing but require more memory.
//
//  keySize: The cache uses a hashing algorithm that hashes a variable length key
//  of 32 bit quantities into a 32 bit value. keySize indicates the integral number
//  of 32 bit quantities comprising the key. The bigger the key size, the slower
//  the hash; consequently, keySize should be kept to an absolute minimum.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
```

```
#define ADF_CACHE_TYPE_ADF      0   // A cache of ADFs
#define ADF_CACHE_TYPE_IMAGE    1   // A cache of density images
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_U32 maxCacheSizeBytes;  // Total size of elements cannot exceed this limit
    ADF_U32 maxCacheNumElms;    // Total number of elements cannot exceed this limit
    ADF_U32 cacheType;          // ADF_CACHE_TYPE_ADF or ADF_CACHE_TYPE_IMAGE
    ADF_U32 hashTableSizeExp;   // Exponent governing number of entries in hash table
    ADF_U32 keySize;            // Number of 32 bit quantities comprising the key
}   ADFCacheAttrs;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//  Cache state (returned for a specified cache using ADFGetCacheState()):
//
//  maxCacheSizeBytes: The original maxCacheSizeBytes creation attribute specified
//  to ADFCreateCache() when the cache was created.
//
//  curCacheSizeBytes: The current size (in bytes) of all the elements in the
//  cache. This size does not include the size of the hash table used by the cache
//  (see hashTableSizeExp).
//
//  maxCacheNumElms: The original maxCacheNumElms creation attribute specified to
//  ADFCreateCache() when the cache was created.
//
//  curCacheNumElms: The current number of elements in the cache.
//
//  numCacheHits: The total number of cache hits during the lifetime of the cache.
//
//  numCacheMisses: The total number of cache misses during the lifetime of the
//  cache.
//
//  cacheType: The original cacheType creation attribute specified to
//  ADFCreateCache() when the cache was created.
//
//  hashTableSizeExp: The original hashTableSizeExp creation attribute specified
//  to ADFCreateCache() when the cache was created.
//
//  keySize: The original keySize creation attribute specified to ADFCreateCache()
//  when the cache was created.
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
typedef struct {
    ADF_U32 maxCacheSizeBytes;  // Total size of elements cannot exceed this limit
    ADF_U32 curCacheSizeBytes;  // Current total size of all elements in the cache
    ADF_U32 maxCacheNumElms;    // Total number of elements cannot exceed this limit
    ADF_U32 curCacheNumElms;    // Current number of elements in the cache
    ADF_U32 numCacheHits;       // Total number of cache hits during cache lifetime
    ADF_U32 numCacheMisses;     // Total number of cache misses during cache lifetime
    ADF_U32 cacheType;          // ADF_CACHE_TYPE_ADF or ADF_CACHE_TYPE_IMAGE
    ADF_U32 hashTableSizeExp;   // Exponent governing number of entries in hash table
    ADF_U32 keySize;            // Number of 32 bit quantities comprising the key
}   ADFCacheState;
//-----------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------
//  Create a cache with the specified creation attributes. This function returns
//  an opaque pointer to the created cache; a NULL is returned if the request cannot
//  be satisfied.
//-----------------------------------------------------------------------------------
void *ADFCreateCache (void *libInst, ADFCacheAttrs *cacheAttrs);


//-----------------------------------------------------------------------------------
//  Destroy the specified cache and all of its elements
//-----------------------------------------------------------------------------------
```

```
void ADFDestroyCache (void *libInst, void *cache);


//-----------------------------------------------------------------------------
//  Search the specified cache for the element with the given key. Upon return:
//
//      if (result == ADF_CACHE_HIT) the return value is a pointer to the
//      cached element (i.e., the ADF or the density image). This element
//      is owned and managed by the cache and therefore should be treated
//      as read-only memory.
//
//      if (result == ADF_CACHE_MISS) the return value is an opaque pointer
//      to an insertion node required for inserting the element (if mandated
//      by the application) into the cache using ADFInsertCacheElement(). The
//      insertion node is returned to avoid rehashing the element key and
//      repeating the search for the appropriate insertion point in the cache.
//
//  If updateLRU is true (the typical case), this function updates the least
//  recently used list accordingly (i.e., the element with the given key is marked
//  as the most recently used element when this function results in a cache hit).
//  If updateLRU is false, the LRU list is untouched.
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
#define ADF_CACHE_MISS  0   // The requested element was not found in the cache
#define ADF_CACHE_HIT   1   // The requested element was found in the cache
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void *ADFGetCacheElement (void *libInst, void *cache, ADF_U32 *key, ADF_U32
updateLRU, ADF_U32 *result);


//-----------------------------------------------------------------------------
//  Insert into the specified cache the element (i.e., the ADF or the density image)
//  with the given key at the specified location insertNode. insertNode is an opaque
//  pointer to an insertion node returned by ADFGetCacheElement() when a cache miss
//  occurs. A return value of zero indicates success; a non-zero return value
//  indicates failure. Upon success, the element is owned and managed by the cache,
//  and cannot be freed or in any way altered by the application. Upon failure, the
//  element is owned by the application.
//
//  A successful insertion updates the LRU list by marking the inserted element as
//  the most recently used element.
//-----------------------------------------------------------------------------
ADF_U32 ADFInsertCacheElement (void *libInst, void *cache, void *insertNode,
ADF_U32 *key, void *element);


//-----------------------------------------------------------------------------
//  Flush the element (i.e., the ADF or the density image) with the given key from
//  the specified cache
//-----------------------------------------------------------------------------
void ADFFlushCacheElement (void *libInst, void *cache, ADF_U32 *key);


//-----------------------------------------------------------------------------
//  Flush all of the elements (i.e., the ADFs or the density images) from the
//  specified cache
//-----------------------------------------------------------------------------
void ADFFlushCacheAll (void *libInst, void *cache);


//-----------------------------------------------------------------------------
//  Get the cache state of the specified cache
//-----------------------------------------------------------------------------
void ADFGetCacheState (void *libInst, void *cache, ADFCacheState *cacheState);
```

```
//-----------------------------------------------------------------------------------
//  Return a pointer to a contiguous block of memory containing the keys of all the
//  elements (i.e., the ADFs or the density images) of the specified cache. A NULL is
//  returned if the request cannot be satisfied. Upon a successful return, cacheState
//  will be populated with the cache state, thereby providing the application with
//  the number of keys (i.e., cacheState->curCacheNumElms), the key size (i.e.,
//  cacheState->keySize), the current size (in bytes) of all the elements in the
//  cache (i.e., cacheState->curCacheSizeBytes), and other potentially useful data
//  pertaining to the specified cache. The contiguous block of keys (which can be
//  treated as a 1D array of 32-bit quantities, with cacheState->keySize 32-bit
//  quantities per key), is ordered from the most-recently-used element to the
//  least-recently-used element. The contiguous block of memory should be freed by
//  the application (via ADF_FREE()) when it is no longer required.
//-----------------------------------------------------------------------------------
ADF_U32 *ADFGetCacheKeys (void *libInst, void *cache, ADFCacheState *cacheState);


//-----------------------------------------------------------------------------------
//  End of C++ wrapper
//-----------------------------------------------------------------------------------
#ifdef __cplusplus
}
#endif


//-----------------------------------------------------------------------------------
//  End of _ADF_TYPE_SYSTEM_
//-----------------------------------------------------------------------------------
#endif
```