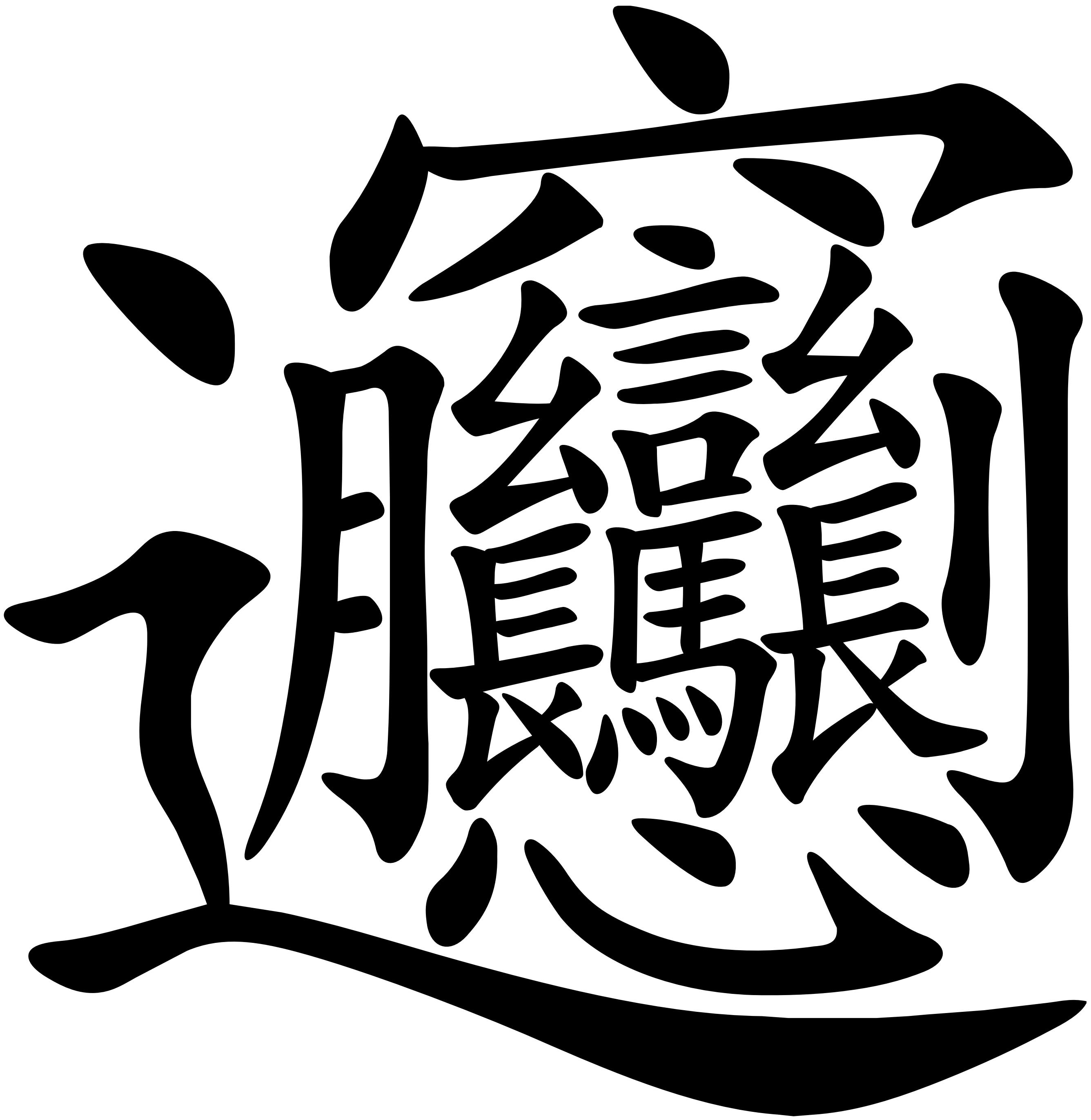
Curve-based glyph UTF-8 extension
Covers all future language characters

Memory use per character: 24068 bytes (including the 10 prefixes, 192544 bits total; 144408 useful bits)
Starting sequence: 11111111
Character Identifier: 10xxxxxx 10xxxxxx 10xxxxxx (this is how you avoid repeating the longer encoding for a symbol that is repeatedly used in the same file, that the second and third and nth use of this symbol in one file only needs these first 4 bytes).
Color swatches: 128 swatches, 8 bytes per swatch.
Length swatches: 128 swatches, 4 bytes per swatch.
Nodes: 2048 nodes, 11 bytes per node.



This is the character I used as the basis for saying that all future characters would need (at most) 2048 nodes, because this SVG file has a total of 1536 coordinates in line 36. This is the SVG of the character known on Wikipedia as “the most complex Chinese character”. Note that some of those coordinates are the control points on each node of this character, so I will just go over the number of necessary nodes as the nice (in base 2) round number as 2048 nodes (2^11).

Here is the design of the memory:
starting sequence: 11111111
character identifier: 10xxxxxx 10xxxxxx 10xxxxxx (if the next character starts with a 0, then use the previously equivalently identified character of this type in the given document instead of trying to calculate anything new).

Color swatches:
128 colors, each with 12 bit encoding for the amount of red, 12 bit encoding for the amount of green, 12 bit encoding for the amount of blue, and 12 bit encoding for the alpha (4095 alpha is the maximum opacity, 0 is completely clear). If 17 four-thousand-ninety-fifths of the blue color is called for, and 311 four-thousand-ninety-fifths of the green color is called for, and 1099 four-thousand-ninety-fifths of the red color is called for with a transparency of around 50%, that swatch is represented in binary within this character as
10010001 10001011 10000100 10110111 10000000 10010001 10011111 10111111 (with the back six bits of the last 2 bytes being the representation of the alpha as 2047, alternatively use 10100000 10000000 to have alpha be 2048, which is also accurate for saying "transparency of around 50%").

Relative polar coordinate length swatches:
128 possible lengths from a previous node (or the top left corner of the glyph area for the position of the first node) to the next node; each is a 24 bit unsigned integer of the form 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx. A length of 16777215 is the length from the top of the area the glyph is allowed to use to the bottom of the area the glyph is allowed to use, with all other lengths being scaled from that length (10111111 10111111 10111111 10111111).

The nodes: 66 bits of useful data (11 bytes) per node.
**7 bits** representing which length swatch is chosen for how far away this node is from the previous node in the list (if this is the first node, then the length swatch is taken from the top left corner of where the gylph is supponed to be written from).
**11 bits** representing the angle (an unsigned integer that converts to an angle linearly such that 10*00000 1000000 is just straight horizontal and to the right of the previous node; and 10*11111 10111111 is technically just straight horizontal and to the right of the previous node, but it is also technically a full rotation; and 10*10000 10000000 is straight horizontal and to the left of the previous node; and 10*01000 10000000 is 90.0 degrees straight up from the previous node; and 10*11000 10000000 is 270.0 degrees straight down from the previous node (* represents the last bit of the previous sequence (since the length swatch could be represented as 10xxxxxx 10x***** (where * would represent the bits of the next sequence (the first 5 bits of the angle)))).
**11 bits** (10xxxxxx 10xxxxx* (where * represents the first bit of the next sequence)) representing the index of which node is "connected node 1" for the purpose of drawing a curve between this node and that node.
**11 bits** (10*****x 10xxxxxx 10xxxx** (where the first 5 stars represent the previous sequence, and the next 2 * represent the first two bits of the next sequence)) representing the index of which node is "connected node 2" for the purpose of drawing a curve between this node and that node.
**8 bits** (10****xx 10xxxxx) representing the angle (in a specific number of degrees, to a maximum of 255 degrees) (where 0 is horizontal to the node position and to the right) to the spot 3 pixels away from the node where the swatch color starts being imposed (gradually filling the area without crossing any curves or lines, and stopping at the places where another region of color has filled (the spots within the space that are topologically closer to the point that another swatch starts at (with no topological path over a line or curve))).
**9 bits** (10xxxxxx 10xxx***) representing the angle (a number of 1/512ths of a full rotation, where 10000000 10000*** is straight horizontal to the right, and 10010000 10000*** is straight up, and 10100000 10000*** is straight horizontal to the left, and 10110000 10000*** is straight down) of "arm 1" where the control point of arm 1 is the same length away from the node as the "connected node 1" is away from the node. After calculating the x and y of arm 1, and similarly the global coordinates of

connected node 1 and this current node, use the following algorithm (written in Python3) for finding all the points on the curve that is meant to be drawn.

```
def points_on_curve(x_base_node: int, y_base_node: int, x_connected_node_1: int,
y_connected_node_1: int, x_arm_1: int, y_arm_1: int, delta: float)->list:
    x=x_connected_node_1
    y=y_connected_node_1
    result: list=[(x, y)]
    dist: float=sqrt((x-x_base_node)**2 + (y - y_base_node)**2)
    num: int=int(dist/delta)
    for i in range(num):
        fraction: float=(i+0.0)/num
        xi: float=(x*(1-fraction) + x_base_node*fraction)
        yi: float=(y*(1-fraction) + y_base_node*fractiom)
        xp: float=(x_arm_1*(1-fraction) + x_base_node*fraction)
        yp: float=(y_arm_1*(1-fraction) + y_base_node*fraction)
        xf: float=(xi*(1-fraction) + xp*fraction)
        yf: float=(yi*(1-fraction) + yp*fraction)
        result.append((xf, yf))
    result.append((x_base_node, y_base_node))
    return result
```

**9 bits** (10***xxx 10xxxxxx) representing the angle (a number of 1/512ths of a full rotation, where 10000000 10000*** is straight horizontal to the right, and 10010000 10000*** is straight up, and 10100000 10000*** is straight horizontal to the left, and 10110000 10000*** is straight down) of “arm 2” where the control point of arm 2 is the same length away from the node as the “connected node 2” is away from the node. After calculating the x and y of arm 2, and similarly the global coordinates of connected node 2 and this current node, use the following algorithm (written in Python3) for finding all the points on the curve that is meant to be drawn.

```
def points_on_curve(x_base_node: int, y_base_node: int, x_connected_node_2: int,
y_connected_node_2: int, x_arm_2: int, y_arm_2: int, delta: float)->list:
    x=x_connected_node_2
    y=y_connected_node_2
    result: list=[(x, y)]
    dist: float=sqrt((x-x_base_node)**2 + (y - y_base_node)**2)
    num: int=int(dist/delta)
    for i in range(num):
        fraction: float=(i+0.0)/num
        xi: float=(x*(1-fraction) + x_base_node*fraction)
        yi: float=(y*(1-fraction) + y_base_node*fractiom)
        xp: float=(x_arm_2*(1-fraction) + x_base_node*fraction)
        yp: float=(y_arm_2*(1-fraction) + y_base_node*fraction)
        xf: float=(xi*(1-fraction) + xp*fraction)
        yf: float=(yi*(1-fraction) + yp*fraction)
        result.append((xf, yf))
    result.append((x_base_node, y_base_node))
    return result
```