SIEMENS

JT File Format Reference

Version 10.0

Rev-B

Acknowledgments

Documents of this type typically require many hands both to author, and to ensure their correctness. However, if one single person can be identified most responsible for bringing this specification document into existence, it is Gary Lance. He authored the JT v8.1 Specification almost single-handedly over the course of four months in 2006, beginning with no knowledge of the DirectModel toolkit from which the JT format springs. This document owes much to the considerable efforts and high standards of quality Gary brought to that first version.

Equally required of documents of this type, come the inevitable erratum or two. Paul Kitchen, of Wilcox Associates, Inc. a Hexagon Metrology Company, is due special thanks for his patient, diligent, and even enthusiastic work with the authors in finding and correcting several bugs in the original JT v8.1 reference document. To our knowledge, Paul is the first outside developer to correctly read all data entities documented in the JT v8.1 specification.

This updated JT Version 10.0 document was written by the JT Format and DirectModel team's developers themselves: Michael Carter, Jianbing Huang, Sashank Ganti, J

Contents

Page

Figures

Tables

1 Siemens JT Data Format Reference Intellectual Property License Terms

The general idea of using an interchange format for electronic documents is in the public domain. Anyone is free to devise a set of unique data structures and operators that define an interchange format for electronic documents. However, Siemens Product Lifecycle Management Software Inc. owns the copyright for the particular data structures and operators, the JT™ Data Format Reference and the written specification constituting the interchange format called the JT Data Format. Thus, these elements of the JT Data Format may not be copied without Siemens‘s permission.

Siemens will enforce its copyright. Siemens‘s intention is to maintain the integrity of the JT Data Format standard, enabling the public to distinguish between the JT Data Format and other interchange formats for electronic documents. However, Siemens desires to promote the use of the JT Data Format for information interchange among diverse products and applications. Accordingly, Siemens gives anyone copyright permission, subject to the conditions stated below, to:

- Prepare and distribute files whose content conforms solely to the JT Data Format.
- Write and distribute software applications that produce discreet output represented in the JT Data Format. Write and distribute software applications that accept input in the form of the JT Data Format and display, print, or otherwise interpret the contents
- Copy Siemens‘s copyrighted list of data structures and operators in the written specification to the extent necessary to use the JT Data Format for the purposes above.
- For avoidance of doubt, the permissions granted in the preceding sentences do not include the reading, writing or distribution of files whose content contains output in the JT Data Format and any other data in any other format and do not include the right to incorporate, integrate, or combine the JT Data Format, structure, or schema into any other data format, structure, or schema.

The conditions of such copyright permission are:

- Anyone who uses the copyrighted list of data structures and operators, as stated above, must include an appropriate copyright notice.

This limited right to use the copyrighted list of data structures and operators does not include the right to copy this document, other copyrighted material from Siemens, or the software in any of Siemens‘s products that use the JT Data Format, in whole or in part, nor does it include the right to use any Siemens patents, except as may be permitted by an official Siemens JT Data Format Reference Patent Clarification Notice.

Nothing in this book is intended to grant you any right or license to use the Marks for any purpose.

2 Scope

This reference defines the syntax and semantics of the JT Version 10.0 file format.

The JT format is an industry focused, high-performance, lightweight, flexible file format for capturing and repurposing 3D Product Definition data that enables collaboration, validation and visualization throughout the extended enterprise. JT format is the de-facto standard 3D Visualization format in the automotive industry, and the single most dominant 3D visualization format in Aerospace, Heavy Equipment and other mechanical CAD domains.

The JT format is both robust, and streamable, and contains best-in-class compression for compact and efficient representation. The JT format was designed to be easily integrated into enterprise translation solutions, producing a single set of 3D digital assets that support a full range of downstream processes from lightweight web-based viewing to full product digital mockups.

At its core the JT format is a scene graph with CAD specific node and attributes support. Facet information (triangles), is stored with sophisticated geometry compression techniques. Visual attributes such as lights, textures, materials and shaders (Cg and OGLSL) are supported. Product and Manufacturing Information (PMI), Precise Part definitions (B-Rep) and Metadata as well as a variety of representation configurations are supported by the format. The JT format is also structured to enable support for various delivery methods including asynchronous streaming of content.

Some of the highlights of the JT format include:

- Built-in support for assemblies, sub-assemblies and part constructs
- Flexible partitioning scheme, supporting single or multiple files
- B-Rep, including integrated support for industry standard Parasolid® (XT) format
- Product Manufacturing Information in support of paperless manufacturing initiatives
- Precise and imprecise wireframe
- Discrete purpose-built Levels of Detail
- Triangle sets, Polygon sets, Point sets, Line sets and Implicit Primitive sets (cylinder, cone, sphere, etc…)
- Full array of visual attributes: Materials, Textures, Lights, Shaders
- Hierarchical Bounding Box and Bounding Spheres
- Advanced data compression that allows producers of JT files to fine tune the tradeoff between compression ratio and fidelity of the data.

Beyond the data contents description of the JT Format, the overall physical structure/organization of the format is also designed to support operations such as:

Offline optimizations of the data contents

- File granularity and flexibility optimized to meet the needs of Enterprise Data Translation Solutions

Asynchronous streaming of content

- Viewing optimizations such as view frustum and occlusion culling and fixed-framerate display modes.

Layers, and Layer Filters.

Along with the pure syntactical definition of the JT Format, there is also series of conventions which although not required to have a reference compliant JT file, have become commonplace within JT format translators. These conventions have been documented in the -Best Practices” section of this JT format reference.

This JT format reference does not specifically address implementation of, nor define, a run-time architecture for viewing and/or processing JT data. This is because although the JT format is closely aligned with a run-time data representation for fast and efficient loading/unloading of data, no interaction behavior is defined within the format itself, either in the form of specific viewer controls, viewport information, animation behavior or other event-based interactivity. This exclusion of interaction behavior from the JT format makes the format more easily reusable for dissimilar application interoperation and also facilitates incremental update, without losing downstream authored data, as the original CAD asset revises.

2.1 What’s New in This Revision

Revision A

Revision B

1. Corrections made in Figure 133 — Wireframe Rep Element data collection, Lag1 is replaced by NULL in two places.
2. Added best practice sections 14.8.5 The SUBNODE property and 14.8.6 Reference Sets and the Reference Set Property

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the following terms apply.

2.1.1
Assembly
A related collection of model parts, represented in a JT format logical scene graph as a logical graph branch.

2.1.2
Attribute
Objects associated with nodes in a logical scene graph and specifying one of several appearances, positioning, or visual characteristics of a shape

2.1.3
Boundary representation
Solid model representation where the solid volume is specified by its surface boundary (both its geometric and topological boundaries)

2.1.4
Code text
Collection of data in encoded form

2.1.5
Coordinate system
A system which uses one or more numbers, or coordinates, to uniquely determine the position of a point or other geometric element

NOTE 1 If not otherwise specified in a data field‘s description, it is assumed that the data is defined in Local Coordinate System.

2.1.6
Directed acyclic graph
Graph that consists of a set of nodes and a set of edges that connect the nodes in a tree like structure

NOTE 1 A directed graph is one in which every edge has a direction such that edge (u,v), connecting node-u with node-v, is different from edge (v,u).

NOTE 2 A directed acyclic graph is a directed graph with no cycles, where a cycle is a path (sequence of edges) from a node to itself.

NOTE 3 With a directed acyclic graph, there is no path that can be followed within the graph such that the first node in the path is the same as the last node in the path.

2.1.7
JT enabled application
Application which supports reading and/or writing reference compliant JT format files

2.1.8
Level of detail
LOD
Alternative graphical representation for some model component such as part

2.1.9
Local coordinate system
LCS
Coordinate system that is used to specify the raw data of the shape geometry with no transforms applied

2.1.10
Logical scene graph
LSG
Scene graph representing the logical organization of a model

NOTE A scene graph contains shapes and attributes representing the model‘s physical components, properties identifying arbitrary metadata (e.g. names, semantic roles) of those components, and a hierarchical structure expressing the component relationships.

2.1.11
Mipmap
Reduced resolution version of a texture map

NOTE Mipmaps are used to texture a geometric primitive whose screen resolution differs from the resolution of the source texture map originally applied to the primitive.

2.1.12
Model
Representation, in JT format, of a physical or virtual product, part, assembly; or collections of such objects

2.1.13
Model coordinate system
MCS
Local coordinates transformed by any transforms specified as attributes at or above the node

2.1.14
Product and manufacturing information
PMI
Collection of information created on a 3D/2D CAD model to completely document the product with respect to design, manufacturing, inspection, etc.

NOTE This can include data such as:

— dimensions (tolerances for each dimension);

— geometric tolerances of feature (datums, feature control frames);

— manufacturing information (surface finish, welding notations);

— inspection information (key locations points);

— assembly instructions;

— product information (materials, suppliers, part numbers).

2.1.15
Property
Object associated with a logical scene graph node and identifying arbitrary application or enterprise specific information (meta-data) related to that node

2.1.16
Quantize
Constrain something to a discrete set of values, such as an integer or integral multiplier of a common factor, rather than a continuous set of values, such as a real number

2.1.17
Scene graph
Directed acyclic graph that arranges the logical and often (but not necessarily) spatial representation of a graphical scene

2.1.18
Shader

User-definable program, expressed directly in a target assembly language or in high-level form to be compiled, that calculates colour values at a pixel based upon data such as lighting, surface colour and texture

NOTE 1 A shader program replaces a portion of the otherwise fixed-functionality graphics pipeline with some user-defined function.

NOTE 2 At present, hardware manufacturers have made it possible to run a shader for each vertex that is processed or each pixel that is rendered.

2.1.19
Streaming

Loading from disk based medium only the portions of data that are required by the user to perform the tasks at hand

NOTE 1 The motivation for streaming is to more efficiently manage system memory.

NOTE 2 Transfer of data in a stream of packets, over the internet on an on-demand basis, where the data is interpreted in real-time by the application as the data packets arrive.

NOTE 3 The motivation for streaming is that the user can begin using or interacting with the data almost immediately - no waiting for the entire data file(s) to be transferred before beginning.

NOTE 4 The desired end result of streaming is to deliver only the JT data that the user needs, where the user needs it, when the user needs it.

2.1.20
Shape

Logical scene graph leaf node containing or referencing the geometric shape definition data (such as vertices, polygons and normals) of a model component

2.1.21
Texture channel

Texture unit plus the texture environment.

NOTE The JT format meaning for texture channel is the same as in OpenGL [1].

2.1.22
Texture object

Named cache that stores texture data, such as the image array, associated mipmaps, and associated texture parameter values: width, height, border width, internal format, resolution of components, minification and magnification filters, wrapping modes, border colour, and texture priority

NOTE The JT format meaning for texture object is the same as in OpenGL [1].

2.1.23
Texture unit

A hardware unit used to sample and filter a texture image.

NOTE The JT format meaning for texture unit is the same as in OpenGL [1].

2.1.24
View coordinate system

World coordinates transformed by a view matrix

2.1.25
World coordinate system
WCS

Node coordinates transformed by transforms inherited from a node‘s parent (i.e. the coordinate system at the root of the graph)

3.2 Abbreviated terms

For the purposes of this document, the following abbreviated terms apply.

Abs	Absolute Value
Bbox	Bounding Box
B-Rep	Boundary Representation
CAD	Computer Aided Design
CODEC	Coder-Decoder
GD&T	Geometric Dimensioning and Tolerancing
GUID	Globally Unique Identifier
HSV	Hue, Saturation, Value
LsbFirst	Least Significant Byte First
Max	Maximum
Min	Minimum
MsbFirst	Most Significant Byte First
N/A	Not Applicable
PCS	Parameter Coordinate Space
PLM	Product Lifecycle Management
RGB	Red, Green, Blue
RGBA	Red, Green, Blue, Alpha
TOC	Table of Contents
VPCS	Viewpoint Coordinate System
URL	Uniform Resource Locator

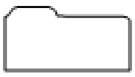
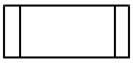
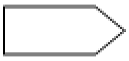
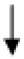
4 Notational conventions

Diagrams and field descriptions

Symbolic diagrams are used to describe the structure of the JT file. The symbols used in these diagrams have the following meaning:

Table 1 — Symbols

Symbol	Description
▭	Rectangles represent a data field of one of the standard data types.

	Folders represent a logical collection of one or more of the standard data types. This information is grouped for clarity and the basic data types that compose the group are detailed in following sections of the document.
	Rectangles with extra lines at left and the right sides corners clipped off represent information logical stepsthat has been compressed.
	Rectangles with the right side corners clipped off represent information that has been compressed.
	Arrows convey the ordering of the information.

The format used to title the diagram symbols is dependent upon the symbol type as follows:

Diagram —rectangle box” (i.e. standard data types) symbols are titled using a format of —Data_Type : Field_Name.” The Data_Type is an abbreviated data type symbol as defined in 3.2 Data Types. In the example below the Data_Type is —I32” (a signed 32 bit integer) and Field_Name is —Count.”

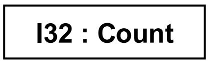


Figure 1 — rectangle box diagram

Diagram —folder” (i.e. logical data collections) symbols are simply titled with a collection name. In the example below the collection name is —Graph Elements.”

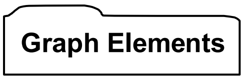


Figure 2 — folder diagram

Diagram —rectangle box with lines at left and right sides” are simply titled with a logic step name. In the example below the logic step name is —Recover First Shell Indices”.

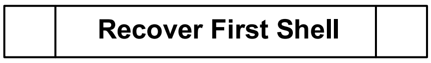


Figure 3 — rectangle box with lines at left and right sides diagram

Diagram —rectangle box with clipped right side corners” (i.e. compressed/encoded data fields) are titled using one of the following three formats:

Data Type; followed by open brace —{—number of bits used to store value, closed brace “}”, and a colon —:” followed by the Field Name. This format for titling the diagram symbol indicates that the data is compressed but not encoded. The compression is achieved by using only a portion of the total bit range of the data type to store the value (e.g. if a count value can never be larger than the value —63” then only 6 bits are needed to store all possible count values). In the example below the Data Type is —U32”, —6”bits are used to store the value, and Field Name is —Count”

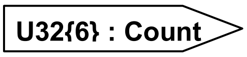


Figure 4 — rectangle box with clipped right side corners

Data Type followed by open brace —{-compressed data packet type, –,", Predictor Type, closed brace "}", and a colon —:" followed by the field name. This format for titling the diagram indicates that a vector of —Data Type" data (i.e. *primal* values) is ran through —Predictor Type" algorithm and the resulting output array of *residual* values is then compressed and encoded into a series of symbols using one of the two supported compressed data packet types.

The two supported compressed data packet types are:

Int32CDP – The Int32CDP (i.e. Int32 Compressed Data Packet) represents a third-generation format used to encode/compress a collection of data into a series of Int32 based symbols. This version of the Int32CDP supersedes the two similarly-named ones from the Version 9 JT Specification, and should not be confused with either of its predecessors. A complete description for Int32 Compressed Data Packet can be found in 13.1.1 Int32 Compressed Data Packet.

The Int32 Compressed Data Packet type is used for compressing/encoding both –integer" and –float" (through quantization) data.

In the example below the Data Type is —VecU32", Int32 Compressed Data Packet type is used, Lag1 Predictor Type is used, and Field Name is —First Shell Index."

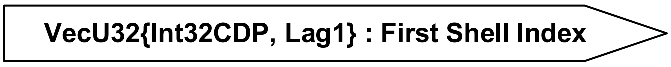


Figure 5 — compressed data packet diagram

As mentioned above (with Predictor Type algorithm), the *primal* input data values are NOT always what is encoded/compressed. This is because the *primal* input data is first run through a Predictor Type algorithm, which produces an output array of residual values (i.e. difference from the predicted value), and this resulting output array of *residual* values is the data which is actually encoded/compressed. The JT format supports several Predictor Type algorithms and each use of Int32CDP specifies, using the above described notation format, what Predictor Type algorithm is being used on the data. The JT format supported Predictor Type algorithms are as follows (note that a sample implementation of decoding the predictor *residual* values back into the primal values can be found in Annex C.

Table 2 — Predictor Type

Predictor Type	Description
Lag1	Predicts as last value
Xor1	Predict as last, but use XOR instead of subtract to compute residual
NULL	No prediction applied

Each predictor type can be combined with additional processing steps, and in such case the predictor type is prefixed with —Combined:". For example, —Combined:Lag1" means that predictor type —Lag1" is combined with additional preprocessing steps. Additional description about the processing steps is provided whenever such combined predictor is used.

—Data Type : Field Name" . This format for titling the diagram symbol indicates that the data is both compressed and encoded. The Data_Type is an abbreviated data type symbol as defined in Data Types and usually represent a vector/array of data. How the data is compressed and encoded into the Data Type is indicated by a CODEC type and other information stored before the particular data in the file. In the example below the Data_Type is —VecU32" and Field_Name is –CodeText."

VecU32 : CodeText

Figure 6 — data type : field name diagram

Note that for some JT file Segment Types there is LZMA compression also applied to all bytes of element data stored in the segment. This LZMA compression applied to all the segment‘s data is not indicated in the diagrams through the use of ―retangle box with clipped right side corners”. Instead, one shall examine information stored with the first Element in the file segment to determine if LZMA compression is applied to all data in the segment. A complete description of the JT format data compression and encoding can be found in 5.1.3 Data Segment and 13 Data Compression and Encoding.

Following each data collection diagram is detailed descriptions for each entry in the data diagram.

For rectangles this detail includes the abbreviated data type symbol, field name, verbal data description, and compression technique/algorithm where appropriate. If the data field is documented as a collection of flags, then the field is to be treated as a bit mask where the bit mask is formed by combining the flags using the binary OR operator. Each bits usage is documented, and bit ON indicates flag value is TRUE and bit OFF indicates flag value is FALSE. All bits fields that are not defined as in use should be set to ―0‖

For folders (i.e. data collections), if the collection is not detailed under a sub-section of the particular document section referencing the data collection, then a comment is included following the diagram indicating where in the document the particular data collection is detailed.

If an arrow appears with a branch in its shaft, then there are two or more options for data to be stored in the file. Which data is stored will depend on information previously read from the file. The following example shows data field A followed by (depending on value of A) either data field B, C, or D.

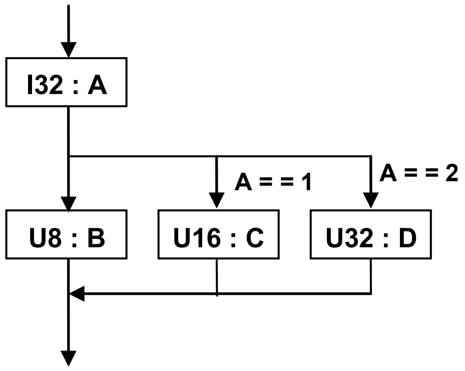


Figure 7 — data filed dependency example

In cases where the same data type repeats, a loop construct is used where the number of iterations appears next to the loop line. There are two forms of this loop construct. The first form is used when the number of iterations is not controlled by some previous read count value. Instead the number of iterations is either a hard coded count (e.g. always 80 characters) or is indicated by some end-of-list marker in the data itself (thus the count is always minimum of 1). This first form of the loop construct looks as follows:

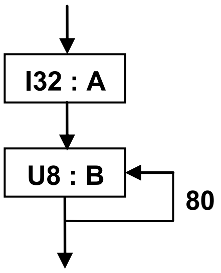


Figure 8 — loop construct example

The second form of this loop construct is used when the number of iterations is based on data (e.g. count) previously read from the file. In this case it is valid for there to be zero data iterations (zero count). This second from of the loop construct looks as follows (data field D is repeated C value times).

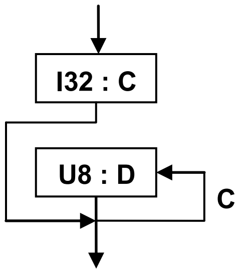


Figure 9 — loop construct with iterations example

Data Types

The data types that can occur in the JT binary files are listed in the following two tables.

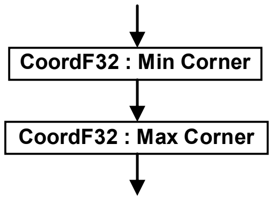
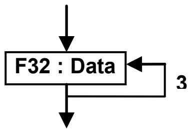
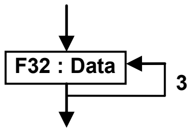
Table 3 — Basic Data Types lists the basic/standard data types which can occur in JT file.

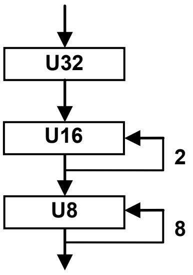
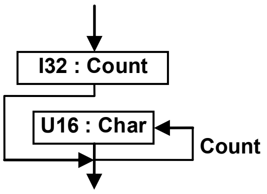
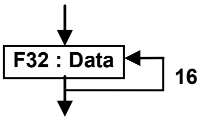
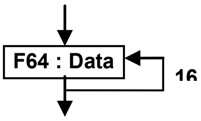
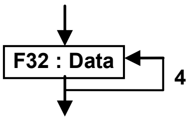
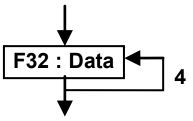
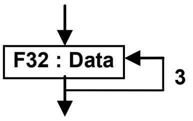
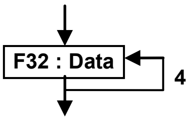
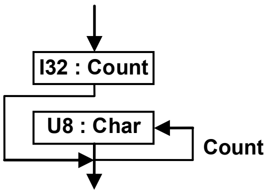
Table 3 — Basic Data Types

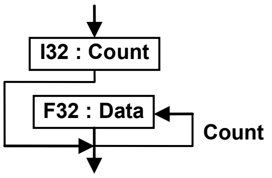
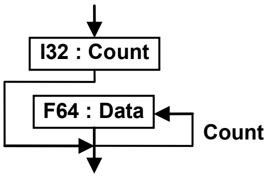
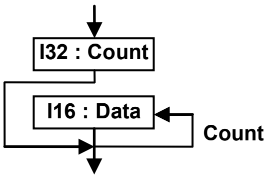
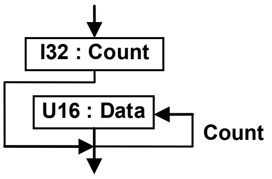
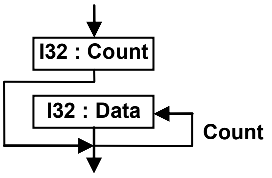
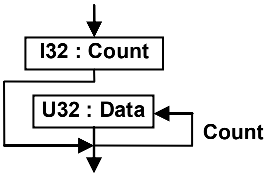
Type	Description
UChar	An unsigned 8-bit byte.
U8	An unsigned 8-bit integer value.
U16	An unsigned 16-bit integer value.
U32	An unsigned 32-bit integer value.
U64	An unsigned 64-bit integer value.
I16	A signed two‘s complement 16-bit integer value.
I32	A signed two‘s complement 32-bit integer value.
I64	A signed two's complement 64-bit integer value.
F32	An IEEE 32-bit floating point number.
F64	An IEEE 64-bit double precision floating point number

Table 4 — Composite Data lists some composite data types which are used to represent some frequently occurring groupings of the basic data types (e.g. Vector, RGBA colour). The composite data types are defined in this reference simply for convenience/brevity in describing the JT file contents.

Table 4 — Composite Data Types

Type	Description	Symbolic Diagram
BBoxF32	The BBoxF32 type defines a bounding box using two CoordF32 types to store the XYZ coordinates for the bounding box minimum and maximum corner points.	
CoordF32	The CoordF32 type defines X, Y, Z coordinate values. So a CoordF32 is made up of three F32 base types.	
DirF32	The DirF32 type defines X, Y, Z components of a direction vector. So a DirF32 is made up of three F32 base types.	

GUID	The GUID type is a 16 byte (128-bit) number. GUID is stored/written to the JT file using a four-byte word (U32), 2 two-byte words (U16), and 8 one-byte words (U8) such as: {3F2504E0-4F89-11D3-9A-0C-03-05-E8-2C-33-01} In the JT format GUIDs are used as unique identifiers (e.g. Data Segment ID, Object Type ID, etc.)	
MbString	The MbString type starts with an I32 that defines the number of characters (NumChar) the string contains. The number of bytes of character data is —2* NumChar” (i.e. the strings are written out as multi-byte characters where each character is U16 size).	
Mx4F32	Defines a 4-by-4 matrix of F32 values for a total of 16 F32 values. The values are stored in row major order (right most subscript, column varies fastest), that is, the first 4 elements form the first row of the matrix.	
Mx4F64	Defines a 4-by-4 matrix of F64 values for a total of 16 F64 values. The values are stored in row major order (right most subscript, column varies fastest), that is, the first 4 elements form the first row of the matrix.	
PlaneF32	The PlaneF32 type defines a geometric Plane using the General Form of the plane equation (Ax + By + Cz + D = 0). The PlaneF32 type is made up of four F32 base types where the first three F32 define the plane unit normal vector (A, B, C) and the last F32 defines the negated perpendicular distance (D), along normal vector, from the origin to the plane.	
Quaternion	The Quaternion type defines a 3-dimensional orientation (no translation) in quaternion linear combination form (a + b*i* + c*j* + d*k*) where the four scalar values (a, b, c, d) are associated with the 4 dimensions of a quaternion (1 real dimension, and 3 imaginary dimensions). So the Quaternion type is made up of four F32 base types.	
RGB	The RGB type defines a colour composed of Red, Green, Blue components, each of which is a F32. So a RGB type is made up of three F32 base types. The Red, Green, Blue colour values typically range from 0.0 to 1.0.	
RGBA	The RGBA type defines a colour composed of Red, Green, Blue, Alpha components, each of which is a F32. So a RGBA type is made up of four F32 base types. The Red, Green, Blue colour values typically range from 0.0 to 1.0. The Alpha value ranges from 0.0 to 1.0 where 1.0 indicates completely opaque.	
String	The String type starts with an I32 that defines the number of characters (NumChar) the string contains. The number of bytes of character data is —NumChar” (i.e. the strings are written out as single-byte characters where each character is U8 size).	

VecF32	The VecF32 type defines a vector/array of F32 base type. The type starts with an I32 that defines the count of following F32 base type data. So a VecF32 is made up of one I32 followed by that number of F32. Note that it is valid for the I32 count number to be equal to —0 indicating no following F32.	
VecF64	The VecF64 type defines a vector/array of F64 base type. The type starts with an I32 that defines the count of following F64 base type data. So a VecF64 is made up of one I32 followed by that number of F64. Note that it is valid for the I32 count number to be equal to —0 indicating no following F64.	
VecI16	The VecI16 type defines a vector/array of I16 base type. The type starts with an I32 that defines the count of following I16 base type data. So a VecI16 is made up of one I32 followed by that number of I16. Note that it is valid for the I32 count number to be equal to —0 indicating no following I16.	
VecU16	The VecU16 type defines a vector/array of U16 base type. The type starts with an I32 that defines the count of following U16 base type data. So a VecU16 is made up of one I32 followed by that number of U16. Note that it is valid for the I32 count number to be equal to —0 indicating no following U16.	
VecI32	The VecI32 type defines a vector/array of I32 base type. The type starts with an I32 that defines the count of following I32 base type data. So a VecI32 is made up of one I32 followed by that number of I32. Note that it is valid for the I32 count number to be equal to —0 indicating no following I32.	
VecU32	The VecU32 type defines a vector/array of U32 base type. The type starts with an I32 that defines the count of following U32 base type data. So a VecU32 is made up of one I32 followed by that number of U32. Note that it is valid for the I32 count number to be equal to -0”, indicating no following U32.	

Empty field

When writing a JT file whose data did not originate from reading a previous JT file, an empty field should be set to a value of —0

When writing a JT file whose data originated from reading a previous JT file (i.e. rewriting a JT File), empty fields should be written with the same value that was read from the originating JT file.

Refer to best practice 14.4 Empty Field

5 File Format

All objects represented in the JT format are assigned an —object identifier” (e.g. see 6.1.1.1.1 Base Node Data, or 6.1.2.1.1 Base Attribute Data) and all references from one object to another object are represented in the JT format using the referenced object‘s —object identifier”. It is the responsibility of JT format readers/writers to maintain the integrity of these object references by doing appropriate pointer unswizzling/swizzling as JT format data is read into memory or written out to disk. Where —pointer swizzling” refers to the process of converting references based on object identifiers into direct memory pointer references and —pointer unswizzling” is the reverse operation (i.e. replacing references based on memory pointers with object identifier references).

5.1 File Structure

A JT file is structured as a sequence of blocks/segments. The File Header block is always the first block of data in the file. The File Header is followed (in no particular order) by a TOC Segment and a series of other Data Segments. The one Data Segment which shall always exist to have a reference compliant JT file is the 6 LSG Segment. The TOC Segment is located within the file using data stored in the File Header. Within the TOC Segment is information that locates all other Data Segments within the file.

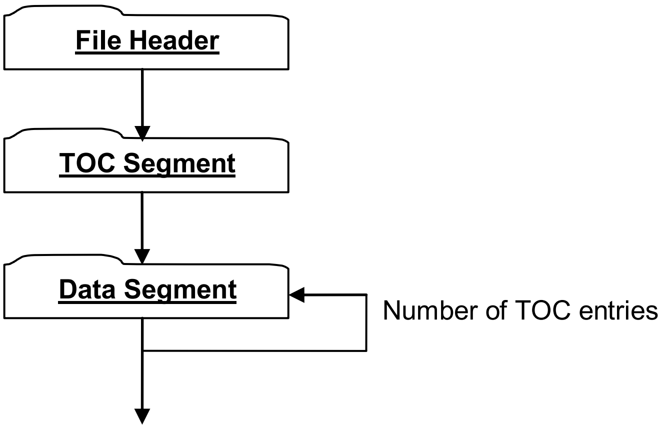


Figure 10 — JT File Structure

5.1.1 File Header

The File Header is always the first block of data in a JT file. The File Header contains information about the JT file version and TOC location, which Loaders use to determine how to read the file. The exact contents of the File Header are as follows:

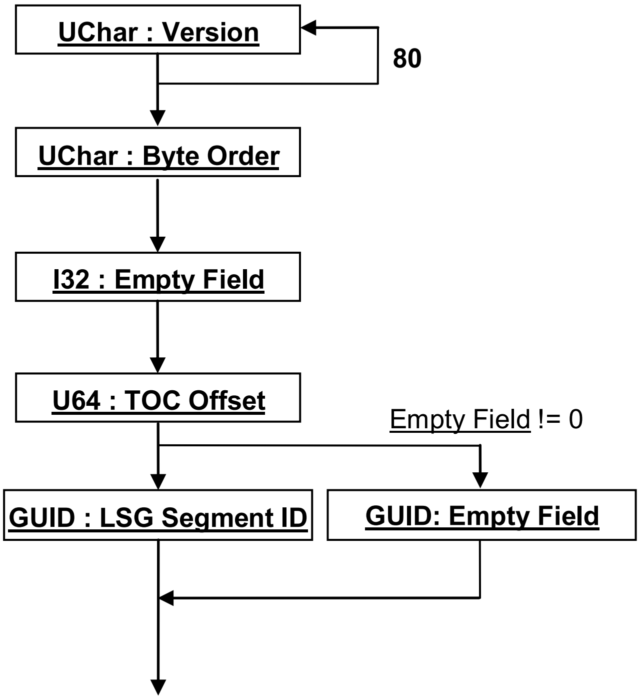


Figure 11 — File Header data collection

UChar : Version

An 80-character version string defining the version of the file format used to write this file. The Version string has the following format:

Version M.n Comment

Where ***M*** is replaced by the major version number, ***n*** is replaced by the minor version number, and ***Comment*** provides other information.

The version string is padded with spaces to a length of 75 ASCII characters and then the final five characters shall be filled with the following linefeed and carriage return character combination (shown using c-style syntax):

```
Version[75] = ‘ ‘
Version[76] = ‘\n‘
Version[77] = ‘\r‘
Version[78] = ‘\n‘
Version[79] = ‘ ‘
```

These final 5 characters (shown above and referred to as ASCII/binary translation detection bytes) can be used by JT file readers to validate that the JT files has not been corrupted by ASCII mode FTP transfers.

As an example, for a JT Version 10.0 file written by a Direct Model (DM) library version 8.0.5.0 this string will look as follows:

—Version 10.0 JT DM 8.0.5.0   \n\r\n —

UChar : Byte Order

Defines the file byte order and thus can be used by the loader to determine if there is a mismatch (thus byte swapping required) between the file byte order and the machine (on which the loader is being run) byte order. Valid values for Byte Order are:

0 – Least Significant byte first (LsbFirst)

1 – Most Significant byte first (MsbFirst)

I32 : Empty Field

Refer to best practice 14.4 Empty Field.

U64 : TOC Offset

Defines the byte offset from the top of the file to the start of the TOC Segment.

GUID : LSG Segment ID

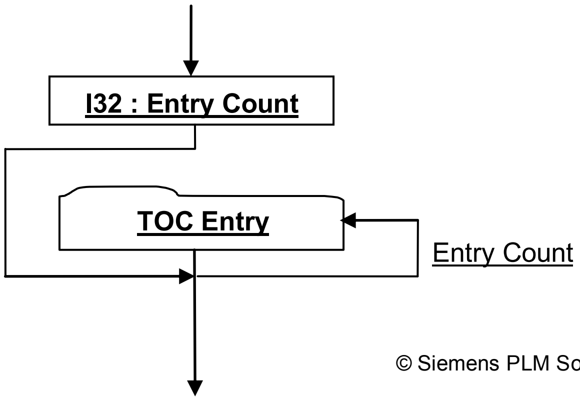
LSG Segment ID specifies the globally unique identifier for the Logical Scene Graph Data Segment in the file. This ID along with the information in the TOC Segment can be used to locate the start of LSG Data Segment in the file. This ID is needed because without it a loader would have no way of knowing the location of the root LSG Data Segment. All other Data Segments shall be accessible from the root LSG Data Segment.

GUID: Empty Field

Refer to best practice 14.4 Empty Field

5.1.2 TOC Segment

The TOC Segment contains information identifying and locating all individually addressable Data Segments within the file. A TOC Segment is always required to exist somewhere within a JT file. The actual location of the TOC Segment within the file is specified by the File Header segment‘s —TOC Offset” field. The TOC



Segment contains one TOC Entry for each individually addressable Data Segment in the file.

Figure 12 — TOC Segment data collection

I32 : Entry Count

Entry Count is the number of entries in the TOC.

TOC Entry

Each TOC Entry represents a Data Segment within the JT File. The essential function of a TOC Entry is to map a Segment ID to an absolute byte offset within the file.

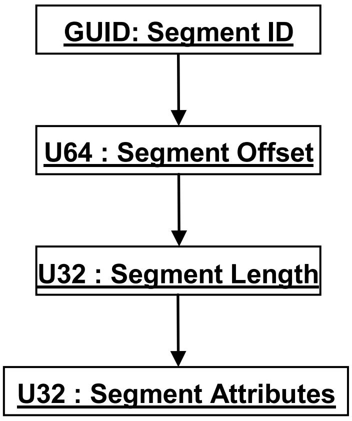


Figure 13 — TOC Entry data collection

GUID : Segment ID

Segment ID is the globally unique identifier for the segment.

U64 : Segment Offset

Segment Offset defines the byte offset from the top of the file to start of the segment.

U32 : Segment Length

Segment Length is the total size of the segment in bytes.

U32 : Segment Attributes

Segment Attributes is a collection of segment information encoded within a single U32 using the following bit allocation.

Table 5 — Segment attributes

Bits 0 - 23	Reserved for future use.
Bits 24 - 31	Segment type. Complete list of Segment types can be found in Table 6 — Segment Types.

5.1.3 Data Segment

All data stored in a JT file shall be defined within a Data Segment. Data Segments are ―typed” based on the general classification of data they contain. See Segment Type field description below for a complete list of the segment types.

Beyond specific data field compression/encoding, some Data Segment types also have compression conditionally applied to all the Data bytes of information persisted within the segment. Whetherthis compression is conditionally applied to a segment‘s Data bytes of information is indicated by information stored with the first ―Element” in the segment. AlsoTable 6 — Segment Types has a column indicating whether the Segment Type may have compression applied to its Data bytes.

All Data Segments have the same basic structure.

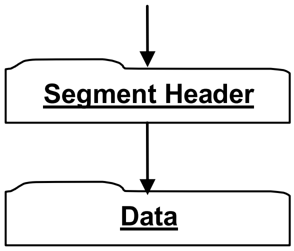


Figure 14 — Data Segment data collection

5.1.3.1 Segment Header

Segment Header contains information that determines how the remainder of the Segment is interpreted by the loader.

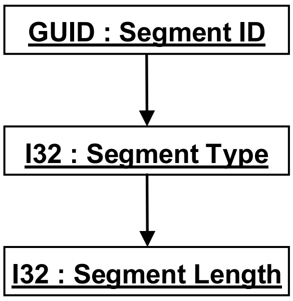


Figure 15 — Segment Header data collection

GUID : Segment ID

Global Unique Identifier for the segment.

I32 : Segment Type

Segment Type defines a broad classification of the segment contents. For example, a Segment Type of —"1 denotes that the segment contains Logical Scene Graph material; —"2denotes contents of a B-Rep, etc.

The complete list of segment and whether or not they support compression on all Data bytes in the paylaod is as follows: payload is as follows:

Table 6 — Segment Types

Type	Data Contents	Compression
1	Logical Scene Graph	Yes
2	JT B-Rep	Yes
3	PMI Data	Yes
4	Meta Data	Yes
6	Shape	No
7	Shape LOD0	No
8	Shape LOD1	No
9	Shape LOD2	No
10	Shape LOD3	No
11	Shape LOD4	No
12	Shape LOD5	No
13	Shape LOD6	No
14	Shape LOD7	No
15	Shape LOD8	No
16	Shape LOD9	No
17	XT B-Rep	Yes
18	Wireframe	Yes

Type	Data Contents	Compression
	Representation	
20	ULP	Yes
24	LWPA	Yes
30	XT B-Rep	Yes

NOTE 1 Segment Types 7-16 all identify the contents as LOD Shape data, where the increasing type number is intended to convey some notion of how high an LOD the specific shape segment represents. The lower the type in this 7-16 range the more detailed the Shape LOD (i.e. Segment Type 7 is the most detailed Shape LOD Segment). For the rare case when there are more than 10 LODs, LOD9 and greater are all assigned Segment Type 16.

NOTE 2 The more generic Shape Segment type (i.e. Segment Type 6) is used when the Shape Segment has one or more of the following characteristics:

— not a descendant of an LOD node;

— is referenced by (i.e. is a child of) more than one LOD node;

— Shape has its own built-in LODs;

— no way to determine what LOD a Shape Segment represents.

I32 : Segment Length

Segment Length is the total size of the segment in bytes. This length value includes all segment Data bytes plus the Segment Header bytes (i.e. it is the size of the complete segment) and should be equal to the length value stored with this segment‘s TOC Entry.

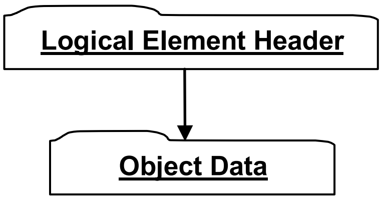
5.1.3.2 Data

The interpretation of the Data section depends on the Segment Type. See 5.2 Data Segments for complete description for all Data Segments that may be contained in a JT file.

Although the Data section is Segment Type dependent there is a common structure which often occurs within the Data section. This structure is a list or multiple lists of Elements where each Element has the same basic structure which consists of some fixed length header information describing the type of object contained in the Element, followed by some variable length object type specific data.

Individual data fields of an Element data collection (and its children data collections) may have advanced compression/encoding applied to them as indicated through compression related data values stored as part of the particular Element‘s storage format. In addition, another level of compression (i.e. LZMA compression) may be conditionally applied to all bytes of information stored for all Elements within a particular Segment. Not all Segment types support compression on all Segment data as indicated Table 8 — Segment Types_If a particular file Segment is of the type which supports compression on all the Segment data, whether this compression is applied or not is indicated by data values stored in the Logical Element Header Compressed data collection of the first Element within the Segment. An in-depth description of JT file compression/encoding techniques can be found in 13 Data Compression and Encoding.

For Segment Types that do **NOT** support compression on all Segment Data.
(see Table 6 — Segment Types.)



For Segment Types that do support compression on all Segment Data.
(see Table 6 — Segment Types.)

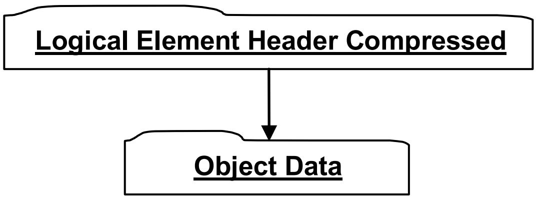


Figure 16 — Data collection

Logical Element Header

Logical Element Header contains data defining the length in bytes of the Element along with the Element Header.

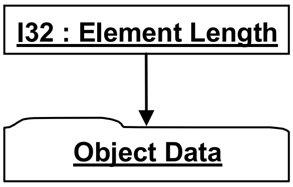


Figure 17 — Logical Element Header data collection

Complete description for Logical Element Header can be found in the Element Header.

I32 : Element Length

Element Length is the total length in bytes of the element Object Data.

Element Header

Element Header contains data describing the object type contained in the Element.

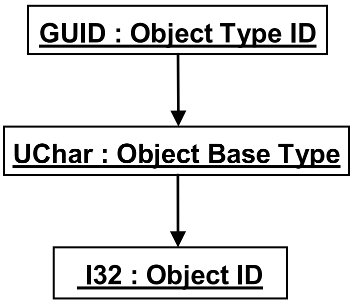


Figure 18 — Element Header data collection

GUID : Object Type ID

Object Type ID is the globally unique identifier for the object type. A complete list of the assigned GUID for all object types stored in a JT file can be found in Annex A. If the GUID is not found in Annex A, the reader should skip Element Length + 1 number of bytes

UChar : Object Base Type

Object Base Type identifies the base object type. If the Object Base Type is not present in **Table 7 — Object** Base Types then the loader should simply skip (read pass) Element Length number of bytes.

Valid Object Base Types include the following:

Table 7 — Object Base Types

Base Type	Description	Base Type's Data Format
255	None	None
0	Base Graph Node Object	6.1.1.1.1 Base Node Data
1	Group Graph Node Object	6.1.1.3.1 Group Node Data
2	Shape Graph Node Object	0 Base Shape Data
3	Base Attribute Object	6.1.2.1.1 Base Attribute Data
4	Shape LOD	None
5	Base Property Object	0 Base Property Atom Data
6	JT Object Reference Object	6.2.5 JT Object Reference Property Atom Element without the Logical Element Header Compressed

Base Type	Description	Base Type's Data Format
		data collection.
8	JT Late Loaded Property Object	6.2.7 Late Loaded Property Atom Element without the Logical Element Header Compressed data collection.
9	JtBase (none)	None

I32 : Object ID

Object ID is the identifier for this Object. Other objects referencing this particular object do so using the Object ID.

Logical Element Header Compressed

Logical Element Header Compressed data collection is the format of Element Header data used by all Elements within Segment Types that supportcompression on all data in the Segment. See Table 6 — Segment Typesfor information on whether a particular Segment Type supports compression on all data in the Segment.

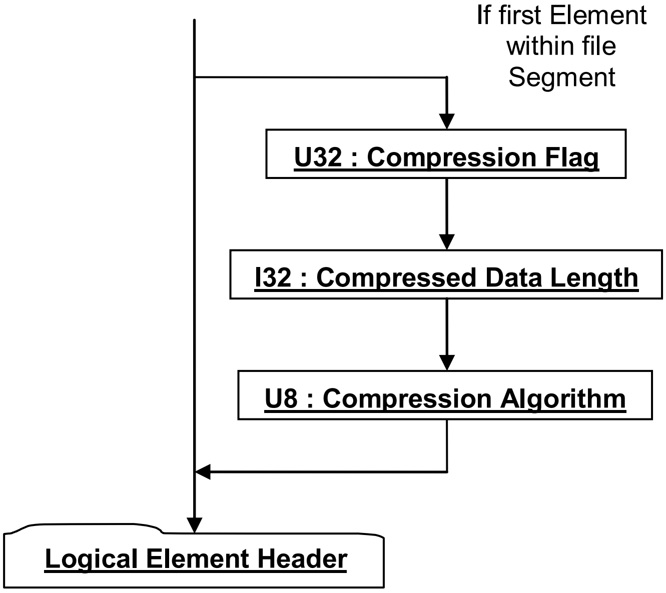

If first Element within file Segment
U32 : Compression Flag
I32 : Compressed Data Length
U8 : Compression Algorithm
Logical Element Header

Figure 19 — Logical Element Header Compressed data collection

Complete description for Logical Element Header can be found in Logical Element Header. Note that if Compression Flag indicates that compression is ON for all element data in the Segment, then the Logical Element Header data collection is also compressed accordingly.

U32 : Compression Flag

Compression Flag is a flag indicating whether compression is ON/OFF for all data elements in the file Segment. Valid values include the following:

Table 8 — Compression flag values

= 3	LZMA compression is ON
!= 3	LZMA No Compression

I32 : Compressed Data Length

Compressed Data Length specifies the compressed data length in number of bytes. Note that data field Compression Algorithm is included in this count.

U8 : Compression Algorithm

Compression Algorithm specifies the compression algorithm applied to all data in the Segment. Valid values include the following:

Table 9 — Compression algorithm values

= 1	No compression
= 3	LZMA compression

Object Data

The interpretation of the Object Data section depends upon the Object Type ID stored in the Logical Element Header (see Logical Element Header).

5.2 Data Segments

A JT file consists of the following segments of data:

— LSG Segment contains a collection of objects (i.e. elements) connected through directed references to form a directed acyclic graph structure (i.e. the LSG). The LSG is the graphical description of the model and contains graphics shapes and attributes representing the model‘s physical components, properties identifying arbitrary metadata (e.g. names, semantic roles) of those components, and a hierarchical structure expressing the component relationships.

— Shape LOD Segment segment contains an element that defines the geometric shape definition data (e.g. vertices, polygons, normals, etc) for a particular shape level of detail or alternative representation.

— JT B-Rep Segment contains an element that defines the precise geometric boundary representation data for a particular Part in JT B-Rep format.

— XT B-Rep Segment contains an element that defines the precise geometric boundary representation data for a particular part in boundary representation format.

— Wireframe Segment contains an element that defines the precise 3D wireframe data for a particular part.

— Meta Data Segments is used to store large collections of meta-data in separate addressable segments of the JT File. Storing meta-data in a separate addressable segment allows references (from within the JT file) to these segments to be constructed such that the meta-data can be late-loaded.

— JT ULP Segment contains an element that defines the semi-precise geometric boundary representation data for a particular part in JT ULP format.

— JT LWPA Segment contains an element that defines light weight precise analytic data for a particular part. More specifically LWPA contains the collection of analytic surfaces in the b-rep definition of the part.

— XT B-Rep Segment contains an element that defines the precise geometric boundary representation data for oneone or more parts in boundary representation format.

For completed information on all the segments of a JT file see the segment descriptions for each segment in their specific sections in this International Standard.

6 LSG Segment

LSG Segment contains a collection of objects (i.e. Elements) connected through directed references to form a directed acyclic graph structure (i.e. the LSG). The LSG is the graphical description of the model and contains graphics shapes and attributes representing the model‘s physical components, properties identifying arbitrary metadata (e.g. names, semantic roles) of those components, and a hierarchical structure expressing the component relationships. The —directed” nature of the LSG references implies that there is by default —state/attribute” inheritance from ancestor to descendant (i.e. predecessor to successor).

The first Graph Element in a LSG Segment should always be a Partition Node. The LSG Segment type supports compression on all element data, so all elements in LSG Segment use the Logical Element Header Compressed form of element header data.

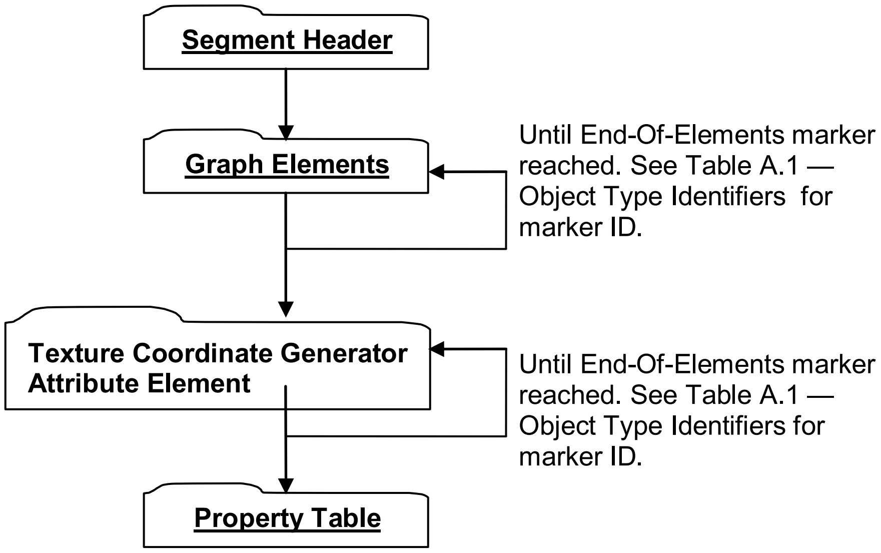

Segment Header
Graph Elements
Until End-Of-Elements marker reached. See Table A.1 — Object Type Identifiers for marker ID.
Texture Coordinate Generator Attribute Element
Until End-Of-Elements marker reached. See Table A.1 — Object Type Identifiers for marker ID.
Property Table

Figure 20 — LSG Segment data collection

Complete description for Segment Header can be found in 5.1.3.1 Segment Header.

6.1 Graph Elements

Graph Elements form the backbone of the LSG directed acyclic graph structure and in doing so serve as the JT model‘s fundamental description. There are two general classifications of Graph elements, Node Elements and Attribute Elements.

Node Elements are nodes in the LSG and in general can be categorized as either an internal or leaf node. The leaf nodes are typically shape nodes used to represent a model‘s physical components and as such either contain or reference some graphical representation or geometry. The internal nodes define the hierarchical organization of the leaf nodes, forming both spatial and logical model relationships, and often contain or reference information (e.g. Attribute Elements) that is inherited down the LSG to all descendant nodes.

Attribute Elements represent graphical data (like appearance characteristics (e.g. colour), or positional transformations) that can be attached to a node, and inherit down the LSG.

Each of these general Graph Element classifications (i.e. Node/Attribute Elements) is sub-typed into specific/concrete types based on data content and implied specialized behaviour. The following sub-sections describe each of the Node and Attribute Element types.

6.1.1 Node Elements

Node Elements represent the relationships of a model‘s components. The model‘s component hierarchy is formed via certain types of Node Elements containing collections of references to other Node Elements who in turn may reference other collections of Node Elements. Node Elements are also the holders (either directly or indirectly) of geometric shape, properties, and other information defining a model‘s components and representations.

6.1.1.1 Base Node Element

Object Type ID: 0x10dd1035, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Base Node Element represents the simplest form of a node that can exist within the LSG. The Base Node Element has no implied LSG semantic behaviour nor can it contain any children nodes.

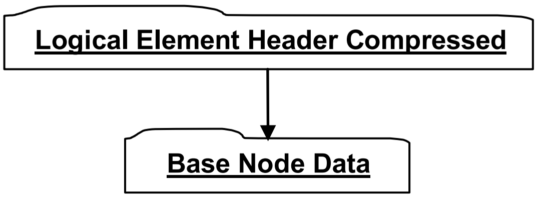


Figure 21 — Base Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

6.1.1.1.1 Base Node Data

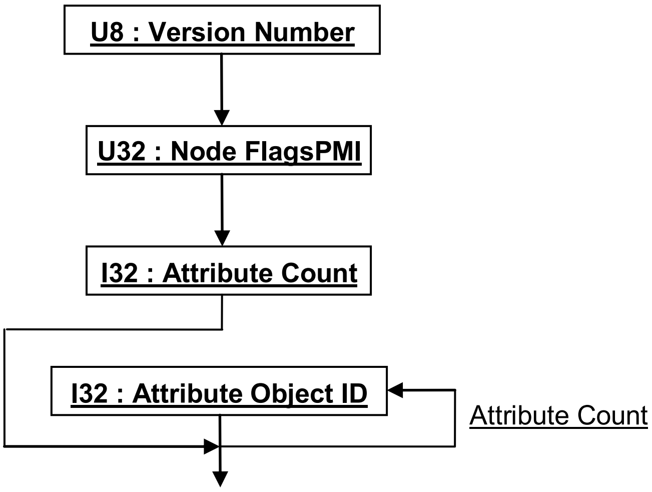


Figure 22 — Base Node Data collection

U8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers

U32 : Node FlagsPMI

Node Flags is a collection of flags. The flags are combined using the binary OR operator. These flags store various state information of the node object. All bits fields that are not defined as in use should be set to —0̕

Table 10 — Node Flag values

0x00000001	Ignore Flag = 0 – Algorithms traversing the LSG structure should include/process this node. = 1 – Algorithms traversing the LSG structure should skip the whole subgraph rooted at this node. Essentially the traversal should be pruned.

I32 : Attribute Count

Attribute Count indicates the number of Attribute Objects referenced by this Node Object. A node may have zero Attribute Object references.

I32 : Attribute Object ID

Attribute Object ID is the identifier for a referenced Attribute Object.

6.1.1.2 Partition Node Element

Object Type ID: 0x10dd103e, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

A partition in a JT file must always be either the root or leaf node. A leaf partition node represents an external JT file reference and provides a means to partition a model into multiple physical JT files (e.g. separate JT file per part in an assembly).

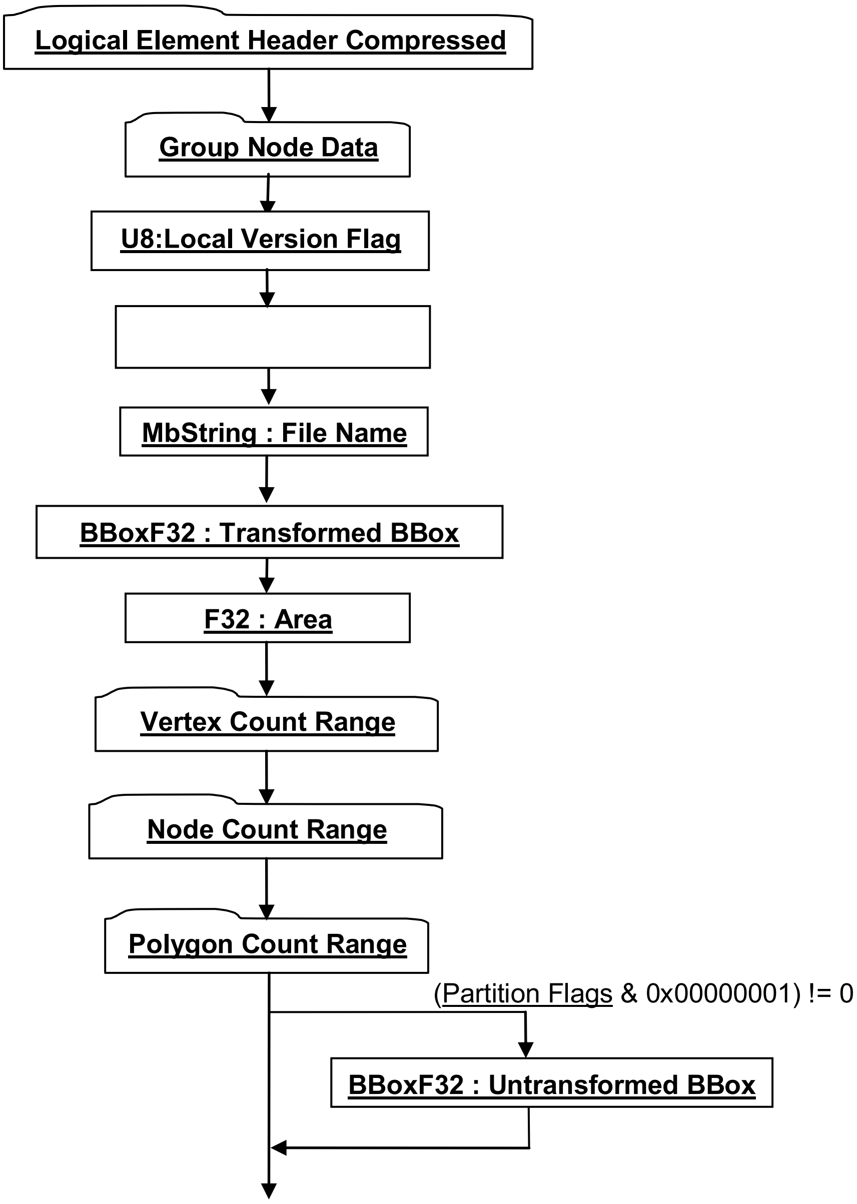

Logical Element Header Compressed
Group Node Data
U8:Local Version Flag
MbString : File Name
BBoxF32 : Transformed BBox
F32 : Area
Vertex Count Range
Node Count Range
Polygon Count Range
(Partition Flags & 0x00000001) != 0
BBoxF32 : Untransformed BBox

Figure 23 — Partition Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Group Node Data can be found in 6.1.1.3.1 Group Node Data.

U8:Local Version Flag

Local Version Flag is the version identifier for this partition node.

I32 : Partition Flags

Partition Flags is a collection of flags. The flags are combined using the binary OR operator. These flags store various state information of the Partition Node Object such as indicating the presence of optional data. All bits fields that are not defined as in use should be set to ―0‖

Table 11 — Partition flag bits

0x00000001	Untransformed bounding box is written.

MbString : File Name

File Name is the relative path portion of the Partition‘s file location. Where —relative path” should be interpreted to mean the string contains the file name along with any additional path information that locates the partition JT file relative to the location of the referencing JT file

BBoxF32 : Transformed BBox

The Transformed BBox is an MCS axis aligned bounding box and represents the transformed geometry extents for all geometry contained in the Partition Node. This bounding box information may be used by a renderer of JT data to determine whether to load the data contained within the Partition node (i.e. is any part of the bounding box within the view frustum).

F32 : Area

Area is the total surface area for this node and all of its descendents. This value is stored in MCS coordinate space (i.e. values scaled by MCS scaling).

BBoxF32 : Untransformed BBox

The Untransformed BBox is only present if Bit 0x00000001 of Partition Flags data field is ON. The Untransformed BBox is an LCS axis-aligned bounding box and represents the untransformed geometry extents for all geometry contained in the Partition Node. This bounding box information may be used by a renderer of JT data to determine whether to load the data contained within the Partition node (i.e. is any part of the bounding box within the view frustum).

6.1.1.2.1 Vertex Count Range

Vertex Count Range is the aggregate minimum and maximum vertex count for all descendants of the Partition Node. There is a minimum and maximum value to accommodate descendant branches having LOD nodes, which encompass a range of count values within the branch, and to accommodate nodes that can themselves generate varying representations. The minimum value represents the least vertex count that can be achieved by the Partition Node‘s descendants. The maximum value represents the greatest vertex count that can be achieved by the Partition Node‘s descendants.

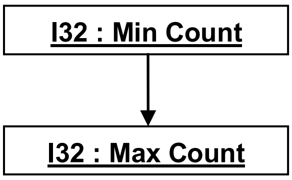


Figure 24 — Vertex Count Range data collection

I32 : Min Count

Min Count is the least vertex count that can be achieved by the Partition Node‘s descendants.

I32 : Max Count

Max Count is the maximum vertex count that can be achieved by the Partition Node‘s descendants.

6.1.1.2.2 Node Count Range

Node Count Range is the aggregate minimum and maximum count of all node descendants of the Partition Node. There is a minimum and maximum value to accommodate descendant branches having LOD nodes, which encompass a range of descendant node count values within the branch. The minimum value represents the least node count that can be achieved by the Partition Node‘s descendants. The maximum value represents the greatest node count that can be achieved by the Partition Node‘s descendants.

The data format for Node Count Range is the same as that described in 6.1.1.2.1 Vertex Count Range.

6.1.1.2.3 Polygon Count Range

Polygon Count Range is the aggregate minimum and maximum polygon count for all descendants of the Partition Node. There is a minimum and maximum value to accommodate descendant branches having LOD nodes, which encompass a range of count values within the branch, and to accommodate nodes that can themselves generate varying representations. The minimum value represents the least polygon count that can be achieved by the Partition Node's descendants. The maximum value represents the greatest polygon count that can be achieved by the Partition Node's descendants.

The data format for Polygon Count Range is the same as that described in 6.1.1.2.1 Vertex Count Range.

6.1.1.3 Group Node Element

Object Type ID: 0x10dd101b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Group Nodes contain an ordered list of references to other nodes, called the group's *children*. Group nodes may contain zero or more children; the children may be of any node type. Group nodes may not contain references to themselves or their ancestors.

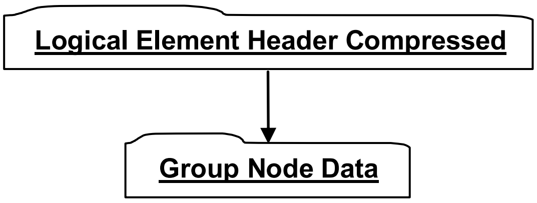


Figure 25 — Group Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

6.1.1.3.1 Group Node Data

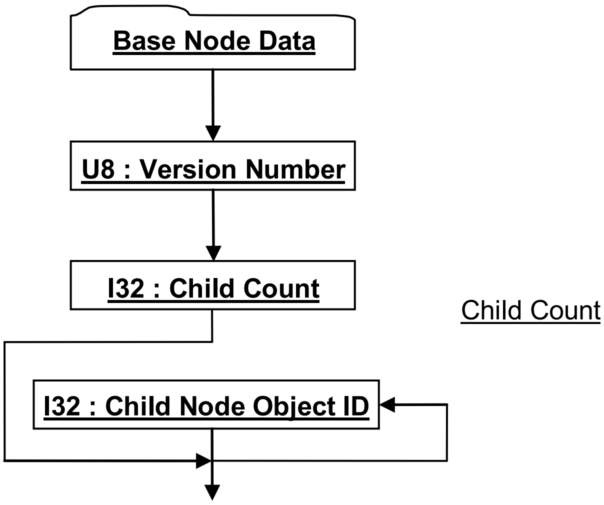


Figure 26 — Group Node Data collection

Complete description for Base Node Data can be found in 6.1.1.1.1 Base Node Data.

U8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

I32 : Child Count

Child Count indicates the number of child nodes for this Group Node Object. A node may have zero children.

I32 : Child Node Object ID

Child Node Object ID is the identifier for the referenced Node Object.

6.1.1.4 Instance Node Element

Object Type ID: 0x10dd102a, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

An Instance Node contains a single reference to another node. Their purpose is to allow sharing of nodes and assignment of instance-specific attributes for the instanced node. Instance Nodes may not contain references to themselves or their ancestors.

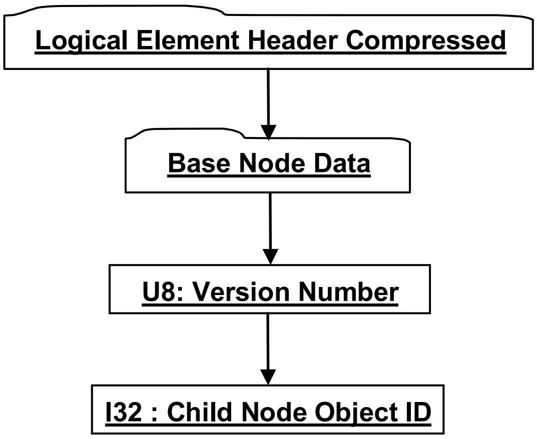


Figure 27 — Instance Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Node Data can be found in 6.1.1.1.1 Base Node Data.

U8: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

I32 : Child Node Object ID

Child Node Object ID is the identifier for the instanced Node Object.

6.1.1.5 Part Node Element

Object Type ID: 0xce357244, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1

A Part Node Element represents the root node for a particular Part within a LSG structure. Every unique Part represented within a LSG structure should have a corresponding Part Node Element. A Part Node Element typically references (using Late Loaded Property Atoms) additional Part specific geometric data and/or properties (e.g. B-Rep data, PMI data).

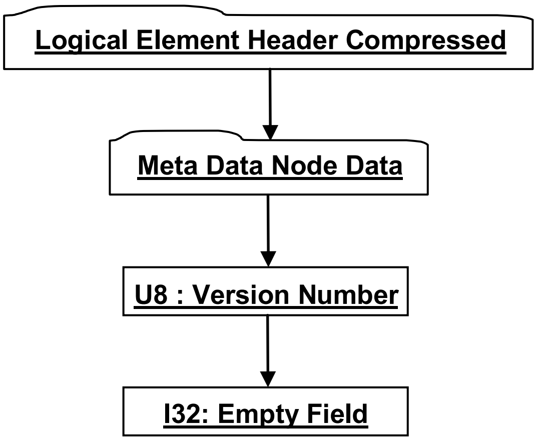


Figure 28 — Part Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Meta Data Node Data can be found in 6.1.1.6.1 Meta Data Node Data.

U8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

I32: Empty Field

Refer to best practice 14.4 Empty Field

6.1.1.6 Meta Data Node Element

Object Type ID: 0xce357245, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1

The Meta Data Node Element is a node type used for storing references to specific ―late loaded” meta-data (e.g. properties, PMI). The referenced meta-data is stored in a separate addressable segment of the JT File (see 11 Meta Data Segment) and thus the use of this Meta Data Node Element is in support of the JT file loader/reader ―best practice” of late loading data (i.e. storing the referenced meta-data in separate addressable segment of the JT file allows a JT file loader/reader to ignore this node‘s meta-data on initial load and instead late-load the node‘s meta-data upon demand so that the associated meta-data does not consume memory until needed).

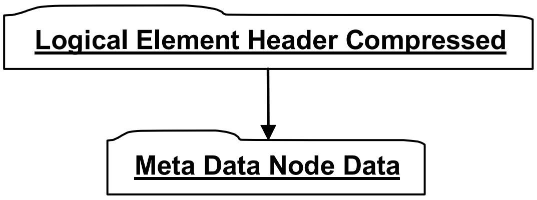


Figure 29 — Meta Data Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

6.1.1.6.1 Meta Data Node Data

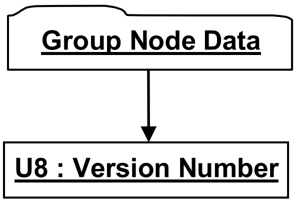


Figure 30 — Meta Data Node Data collection

Complete description for Group Node Data can be found in 6.1.1.3.1 Group Node Data.

U8 : Version Number

Version Number is the version identifier for this data. For information on local version numbers see best practice 14.5 Local version numbers.

6.1.1.7 LOD Node Element

Object Type ID: 0x10dd102c, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

An LOD Node holds a list of alternate representations. The list is represented as the children of a base group node, however, there are no implicit semantics associated with the ordering. Traversers of LSG may apply semantics to the ordering as part of alternative representation selection.

Each alternative representation could be a sub-assembly where the alternative representation is a group node with an assembly of children.

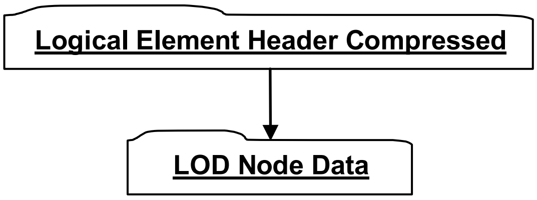


Figure 31 — LOD Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

6.1.1.7.1 LOD Node Data

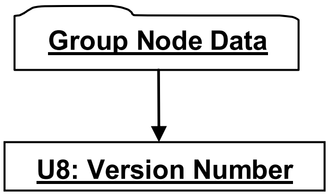


Figure 32 — LOD Node Data collection

Complete description for Group Node Data can be found in 6.1.1.3.1 Group Node Data.

U8: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

6.1.1.8 Range LOD Node Element

Object Type ID: 0x10dd104c, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Range LOD Nodes hold a list of alternate representations and the ranges over which those representations are appropriate. Range Limits indicate the distance between a specified centre point and the eye point, within which the corresponding alternate representation is appropriate. Traversers of LSG consult these range limit values when making an alternative representation selection.

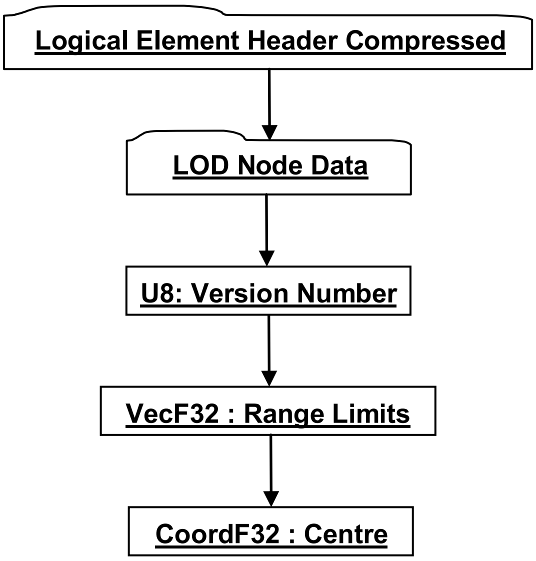


Figure 33 — Range LOD Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for LOD Node Data can be found in 6.1.1.7.1 LOD Node Data

U8: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers

VecF32 : Range Limits

Range Limits indicate the WCS distance between a specified centre point and the eye point, within which the corresponding alternate representation is appropriate. It is not required that the count of range limits is equivalent to the number of alternative representations. These values are considered —soft values” in that loaders/viewers of JT data are free to throw these values away and compute new values based on their desired LOD selection semantics.

Best practices suggest that LSG traversers apply the following strategy, at Range LOD Nodes, when making alternative representation selection decisions based on Range Limits: The first alternate representation is valid when the distance between the centre and the eye point is less than or equal to the first range limit (and when no range limits are specified). The second alternate representation is valid when the distance is greater than the first limit and less than or equal to the second limit, and so on. The last alternate representation is valid for all distances greater than the last specified limit.

CoordF32 : Centre

Centre specifies the X,Y,Z coordinates for the MCS centre point upon which alternative representation selection eye distance computations are based. Typically this location is the centre of the highest-detail alternative representation. These values are considered —soft values” in that loaders/viewers of JT data are free to throw these values away and compute new values based on their desired LOD selection semantics

6.1.1.9 Switch Node Element

Object Type ID: 0x10dd10f3, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

The Switch Node is very much like a Group Node in that it contains an ordered list of references to other nodes, called the *children* nodes. The difference is that a Switch Node also contains additional data indicating which child (one or none) a LSG traverser should process/traverse.

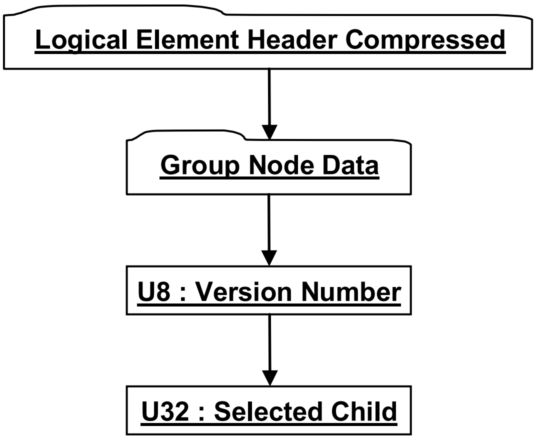


Figure 34 — Switch Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Group Node Data can be found in 6.1.1.3.1 Group Node Data.

U8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers

U32 : Selected Child

Selected Child is the index for the selected child node. Valid Selected Child values reside within the following range: ―1 < Selected Child < Child Count”. Where ―1” indicates that no child is to be selected and ―Child Count” is the data field value from 6.1.1.3.1 Group Node Data.

6.1.1.10 Shape Node Elements

Shape Node Elements are ―leaf” nodes within the LSG structure and contain or reference the geometric shape definition data (e.g. vertices, polygons, normals, etc.).

Typically Shape Node Elements do not directly contain the actual geometric shape definition data, but instead reference (using Late Loaded Property Atoms) Shape LOD Segments within the file for the actual geometric shape definition data. Storing the geometric shape definition data within separate independently addressable data segments in the JT file, allows a JT file reader to be structured to support the ―best practice” of delaying the loading/reading of associated data until it is actually needed. Complete descriptions for Late Loaded Property Atom Elements and Shape LOD Segments can be found in 6.2.7 Late Loaded Property Atom Element and 6.2 Property Atom Elements respectively.

There are several types of Shape Node Elements which the JT format supports. The following sub-sections document the various Shape Node Element types.

6.1.1.10.1 Base Shape Node Element

Object Type ID: 0x10dd1059, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Base Shape Node Element represents the simplest form of a shape node that can exist within the LSG.

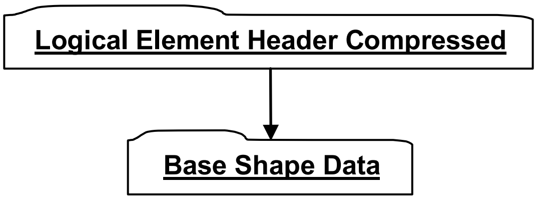


Figure 35 — Base Shape Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Base Shape Data

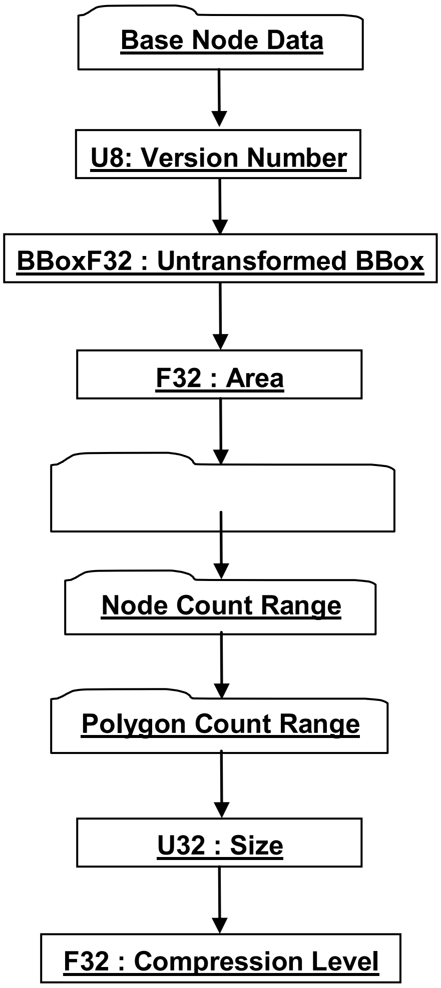


Figure 36 — Base Shape Data collection

Complete description for Base Node Data can be found in 6.1.1.1.1 Base Node Data

U8: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

BBoxF32 : Untransformed BBox

The Untransformed BBox is an axis-aligned LCS bounding box and represents the untransformed geometry extents for all geometry contained in the Shape Node.

F32 : Area

Area is the total surface area for this node and all of its descendents. This value is stored in MCS coordinate space (i.e. values scaled by MCS scaling).

U32 : Size

Size specifies the in memory length in bytes of the associated/referenced Shape LOD Element

. This Size value has no relevancy to the on-disk (JT File) size of the associated/referenced Shape LOD Element

. A value of zero indicates that the in memory size is unknown. See 7.1 Shape LOD Element

for complete description of Shape LOD Elements. JT file loaders/readers can leverage this Size value during late load processing to help pre-determine if there is sufficient memory to load the Shape LOD Element

.

F32 : Compression Level

Compression Level specifies the qualitative compression level applied to the associated/referenced Shape LOD Element

. See 7.1 Shape LOD Element

for complete description of Shape LOD Elements. This compression level value is a qualitative representation of the compression applied to the Shape LOD Element

. The absolute compression (derived from this qualitative level) applied to the Shape LOD Element

is physically represented in the JT format by other data stored with both the Shape Node and the Shape LOD Element

(e.g.), and thus it's not necessary to understand how to map this qualitative value to absolute compression values in order to uncompress/decode the data

Table 12 — Compression level values

= 0.0	—Lossless” compression used.
= 0.1	—Minimally Lossy” compression used. This setting generally results in modest compression ratios with little if any visual difference when compared to the same images rendered from —Lossless” compressed Shape LOD Element.
= 0.5	—Moderate Lossy” compression used. The setting results in more data loss than —Minimally Lossy” and thus higher compression ratio is obtained. Some visual difference will likely be noticeable when compared to the same images rendered from —Lossless” compressed Shape LOD Element.
= 1.0	—Aggressive Lossy” compression used. With this setting as much data as possible will be thrown away, resulting in highest compression ratio, while still maintaining a modestly useable representation of the underlying data. Visual differences may be evident when compared to the same images rendered from —Lossless” compressed Shape LOD Element.

Vertex Count Range

Vertex Count Range is the aggregate minimum and maximum vertex count for this Shape Node. There is a minimum and maximum value to accommodate shape types that can themselves generate varying representations. The minimum value represents the least vertex count that can be achieved by the Shape Node. The maximum value represents the greatest vertex count that can be achieved by the Shape Node.

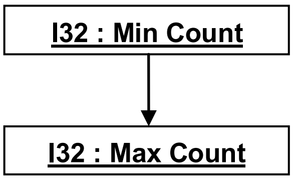


Figure 37 —

Vertex Count Range data collection

I32 : Min Count

Min Count is the least vertex count that can be achieved by this Shape Node.

I32 : Max Count

Max Count is the maximum vertex count that can be achieved by this Shape Node. A value of —1" indicates maximum vertex count is unknown.

Node Count Range

Node Count Range is the aggregate minimum and maximum count of all node descendants of the Shape Node. The minimum value represents the least node count that can be achieved by the Shape Node's descendants. The maximum value represents the greatest node count that can be achieved by Shape Node's descendants. For Shape Nodes the minimum and maximum count values should always be equal to —1"

The data format for Node Count Range is the same as that described in

Vertex Count Range.

Polygon Count Range

Polygon Count Range is the aggregate minimum and maximum polygon count for this Shape Node. There is a minimum and maximum value to accommodate shape types that can themselves generate varying representations. The minimum value represents the least polygon count that can be achieved by the Shape Node. The maximum value represents the greatest polygon count that can be achieved by the Shape Node.

The data format for Polygon Count Range is the same as that described in

Vertex Count Range.

6.1.1.10.2 Vertex Shape Node Element

Object Type ID: 0x10dd107f, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Vertex Shape Node Element represents shapes defined by collections of vertices.

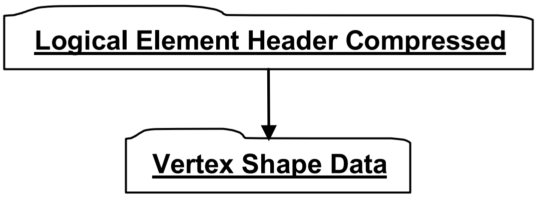


Figure 38 — Vertex Shape Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Vertex Shape Data

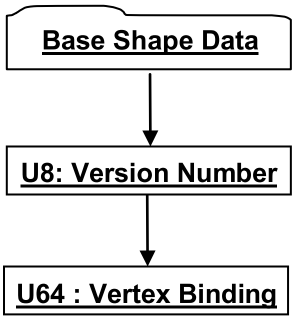


Figure 39 — Vertex Shape Data collection

Complete description for Base Shape Data can be found in Base Shape Data.

U8: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

U64 : Vertex Binding

Vertex Bindings is a collection of normal, texture coordinate, and colour binding information encoded within a single U64. All bits fields that are not defined as in use should be set to —‟0 For more information see Vertex Shape LOD Data U64 : Vertex Bindings.

6.1.1.10.3 Tri-Strip Set Shape Node Element

Object Type ID: 0x10dd1077, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

A Tri-Strip Set Shape Node Element defines a collection of independent and unconnected triangle strips. Each strip constitutes one primitive of the set and is defined by one list of vertex coordinates.

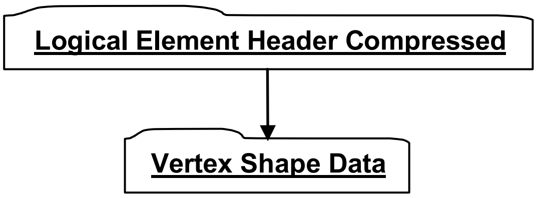


Figure 40 — Tri-Strip Set Shape Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Vertex Shape Data can be found in Vertex Shape Data.

6.1.1.10.4 Polyline Set Shape Node Element

Object Type ID: 0x10dd1046, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

A Polyline Set Shape Node Element defines a collection of independent and unconnected polylines. Each polyline constitutes one primitive of the set and is defined by one list of vertex coordinates.

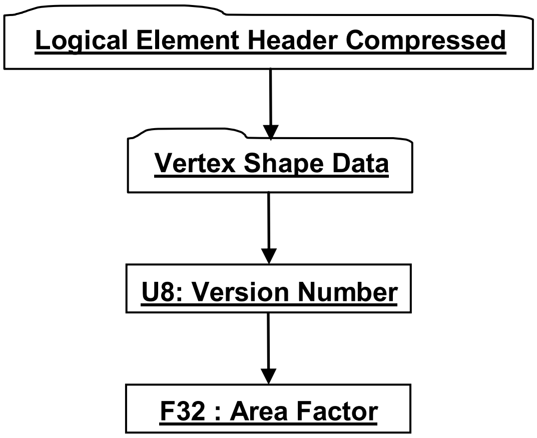


Figure 41 — Polyline Set Shape Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Vertex Shape Data can be found in Vertex Shape Data.

U8: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

F32 : Area Factor

Area Factor specifies a multiplier factor applied to a Polyline Set computed surface area. In JT data viewer applications there may be LOD selection semantics that are based on screen coverage calculations. The so-called ”surface area” of a polyline is computed as if each line segment were a square. This Area Factor turns each edge into a narrow rectangle. Valid Area Factor values lie in the range (0,1].

6.1.1.10.5 Point Set Shape Node Element

Object Type ID: 0x98134716, 0x0010, 0x0818, 0x19, 0x98, 0x08, 0x00, 0x09, 0x83, 0x5d, 0x5a

A Point Set Shape Node Element defines a collection of independent and unconnected points. Each point constitutes one primitive of the set and is defined by one vertex coordinate.

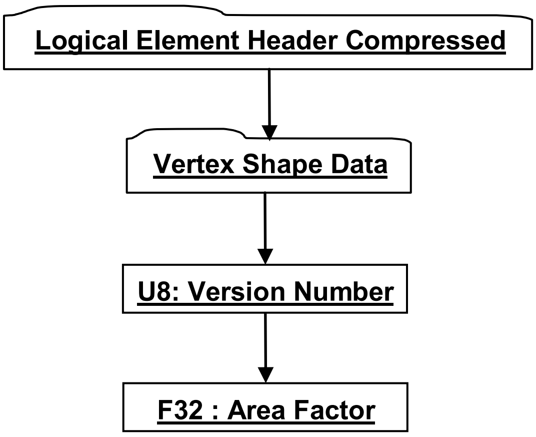


Figure 42 — Point Set Shape Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Vertex Shape Data can be found in Vertex Shape Data.

U8: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers

F32 : Area Factor

Area Factor specifies a multiplier factor applied to the Point Set computed surface area. In JT data viewer applications there may be LOD selection semantics that are based on screen coverage calculations. The computed ―surface area” of a Point Set is equal to the larger (i.e. whichever is greater) of either the area of the Point Set‘s bounding box, or ―1.0”. Area Factor scales the result of this ―surface area” computation..

U64: Vertex Bindings

Vertex Bindings is a collection of normal, texture coordinate, and colour binding information encoded within a single U64. All bits fields that are not defined as in use should be set to ―0‘ For more information see Vertex Shape LOD Data U64 : Vertex Bindings.

6.1.1.10.6 Polygon Set Shape Node Element

Object Type ID: 0x10dd1048, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

A Polygon Set Shape Node Element defines a collection of independent and unconnected polygons. Each polygon constitutes one primitive of the set and is defined by one list of vertex coordinates.

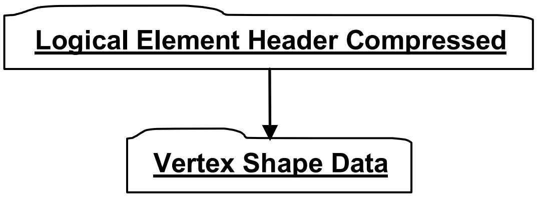


Figure 43 — Polygon Set Shape Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Vertex Shape Data can be found in Vertex Shape Data.

6.1.1.10.7 NULL Shape Node Element

Object Type ID: 0xd239e7b6, 0xdd77, 0x4289, 0xa0, 0x7d, 0xb0, 0xee, 0x79, 0xf7, 0x94, 0x94

A NULL Shape Node Element defines a shape which has no direct geometric primitive representation (i.e. it is empty/NULL). NULL Shape Node Elements are often used as ―proxy/placeholder” nodes within the serialized LSG when the actual Shape LOD data is run time generated (i.e. not persisted).

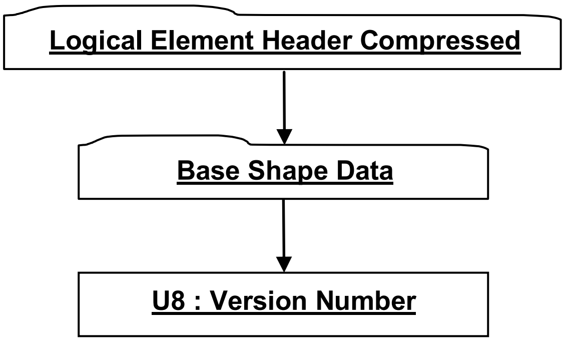


Figure 44 — NULL Shape Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Shape Data can be found in Base Shape Data.

U8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

6.1.1.10.8 Primitive Set Shape Node Element

Object Type ID: 0xe40373c1, 0x1ad9, 0x11d3, 0x9d, 0xaf, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2

A Primitive Set Shape Node Element represents a list/set of primitive shapes (e.g. box, cylinder, sphere, etc.) whose LODs can be procedurally generated. —Procedurally generate” means that the raw geometric shape definition data (e.g. vertices, polygons, normals, etc) for LODs is not directly stored; instead some basic shape information is stored (e.g. sphere centre and radius) from which LODs can be generated.

Primitive Set Shape Node Elements actually do not even directly contain this basic shape definition data, but instead reference (using Late Loaded Property Atoms) Primitive Set Shape Node Element within the file for the actual basic shape definition data. Storing the basic shape definition data within separate independently addressable data segments in the JT file, allows a JT file reader to be structured to support the —best practice” of delaying the loading/reading of associated data until it is actually needed. Complete descriptions for Late Loaded Property Atom Elements and Primitive Set Shape Element can be found in 6.2.7 Late Loaded Property Atom Element and 7.2 Primitive Set Shape Element respectively.

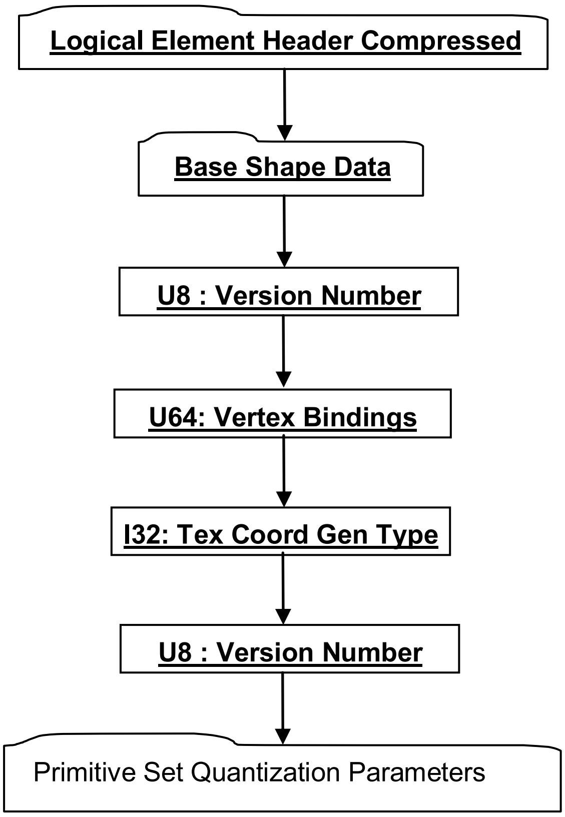


Figure 45 — Primitive Set Shape Node Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Shape Data can be found in Base Shape Data.

U8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

U64: Vertex Bindings

Vertex Bindings is a collection of normal, texture coordinate, and colour binding information encoded within a single U64. All bits fields that are not defined as in use should be set to —'0 For more information see Vertex Shape LOD Data U64 : Vertex Bindings.

I32: Tex Coord Gen Type

Texture Coord Gen Type specifies how a texture is applied to each face of the primitive. Single tile means one copy of the texture will be stretched to fit the face, isotropic means that the texture will be duplicated on the longer dimension of the face in order to maintain the texture's aspect ratio

Table 13 — Texture Coord Gen Type values

= 0	Single Tile…Indicates that a single copy of a texture image will be applied to significant primitive features (i.e. cube face, cylinder wall, end cap) no matter how eccentrically shaped.
= 1	Isotropic…Implies that multiple copies of a texture image may be mapped onto eccentric surfaces such that a mapped texel stays approximately square.

U8 : Version Number

Version Number is the version identifier for this element. The value of this Version Number indicates the format of data fields to follow.

Table 14 — Version Number values

= 0	Version 0 Format
= 1	Version 1 Format

Primitive Set Quantization ParametersPrimitive Set Quantization Parameters specifies for the two shape data type grouping (i.e. Vertex, Colour) the number of quantization bits used for given qualitative compression level. Although these .

values are saved in the associated/referenced Shape LOD Element, they are also saved here so that a JT File loader/reader does not have to load the Shape LOD Element in order to determine the Shape quantization level. See 7.1 Shape LOD Element

for complete description of Shape LOD Elements.

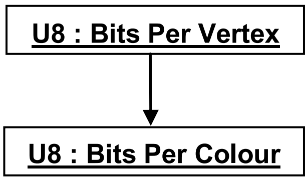


Figure 46 — Primitive Set Quantization Parameters data collection

U8 : Bits Per Vertex

Bits Per Vertex specifies the number of quantization bits per vertex coordinate component. Value shall be within range [0:24] inclusive.

U8 : Bits Per Colour

Bits Per Colour specifies the number of quantization bits per colour component. Value shall be within range [0:24] inclusive.

6.1.2 Attribute Elements

Attribute Elements (e.g. colour, texture, material, lights, etc.) are placed in LSG as objects associated with nodes. Attribute Elements are not nodes themselves, but can be associated with any node.

For applications producing or consuming JT format data, it is important that the JT format semantics of how attributes are meant to be applied and accumulated down the LSG are followed. If not followed, then consistency between the applications in terms of 3D positioning and rendering of LSG model data will not be achieved.

To that end each attribute type defines its own application and accumulation semantics, but in general attributes at lower levels in the LSG take precedence and replace or accumulate with attributes set at higher levels. Nodes without associated attributes inherit those of their parents. Attributes inherit only from their parents, thus a node‘s attributes do not affect that node‘s siblings. The root of a partition inherits the attributes in effect at the referring partition node.

In previous version of the JT file format, Attributes held a single –final” bit denoting that no further accumulations were to take place into that attribute type by Attributes of the same type lying below it in the scene graph. JT v10 replaces this single bit with separate –field final” bits for each field within the Attribute. Different Attributes have different fields, and are documented accordingly in the following sections. Only three Attributes define more than one internal field (i.e. Material Attribute Element, Texture Image Attribute Element, and Draw Style Attribute Element). All other Attributes merely define a single default field the encompasses their entire state.

In addition to –field final” bits, each Attribute also defines a parallel set of –field inhibit” bits. These bits denote, on a field-by-field basis, whether a field is allowed to accumulate. Said differently, if a field inhibit bit is set to 0, the field accumulates normally; if the bit is set to 1, then the field will not accumulate, and is ignored.

Descendants can explicitly do a one-shot override of –final” using the attribute –force” flag (see 6.1.2.1.1 Base Attribute Data), but do not by default. Note that –force” does not turn OFF –final” – it is simply a one-shot override of –final” for the specific attribute marked as –forcing.” Note that the –force” flag is attribute-wide – not on a field-by-field basis like field-finals and field-inhibits. An analogy for this –force” and –final” interaction is that –final” is a back-door in the attribute accumulation semantics, and that –force” is a doggy-door in the back-door!

6.1.2.1 Common Attribute Data Containers

6.1.2.1.1 Base Attribute Data

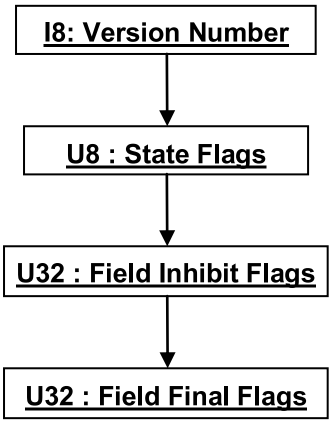


Figure 47 — Base Attribute Data collection

I8: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers.

U8 : State Flags

State Flags is a collection of flags. The flags are combined using the binary OR operator and store various state information for Attribute Elements; such as indicating that the attributes accumulation is final. All bits fields that are not defined as in use should be set to —'0

Table 15 — State Flag values

0x01	Unused
0x02	Accumulation Force flag. Provides a way to assign nodes in LSG, attributes that shall not be overridden by ancestors. = 0 – Accumulation of this attribute obeys ancestor‘s Final flag setting. = 1 – Accumulation of this attribute is forced (overrides ancestor‘s Final flag setting)
0x04	Accumulation Ignore Flag. Provides a way to indicate that the attribute is to be ignored (not accumulated). = 0 – Attribute is to be accumulated normally (subject to values of Force/Final flags) = 1 – Attribute is to be ignored.
0x08	Attribute Persistable Flag. Provides a way to indicate that the attribute is to be persistable to a JT file. = 0 – Attribute is to be non-persistable. = 1 – Attribute is to be persistable.

U32 : Field Inhibit Flags

Field Inhibit Flags is a collection of flags, each flag corresponding to a collection of state data within a particular Attribute type. Each value (or semantically related set of values) present present in an Attribute Element is given a field number ranging from 0 to 31. If the field‘s corresponding bit in Inhibit Flags is set, then the field should not participate in attribute accumulation. All bits are reserved.

See each particular Attribute Element (e.g. Material Attribute Element) for a description of bit field assignments for each attribute value.

U32 : Field Final Flags

Field Final Flags is a collection of flags, each flag being parallel to the corresponding flag in the Field Inhibit Flags. If the field‘s bit in Field Final Flags is set, then that field within the Attribute will become —final” and will not allow any subsequent accumulation into the specified field. All bits are reserved.

See each particulare Attribute Element for a description of bit field assignments for each Attribute value.

6.1.2.2 Material Attribute Element

Object Type ID: 0x10dd1030, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Material Attribute Element defines the material properties of a object. JT format LSG traversal semantics state that material attributes accumulate down the LSG by replacement.

The Field Inhibit flag (see 6.1.2.1.1 Base Attribute Data) bit assignments for the Material Attribute Element data fields, are as follows:

Table 16 — Material Attribute data field inihibit values

Field Inhibit Flag Bit	Data Field(s) Bit Applies To
0	Ambient Common RGB Value, Ambient Colour
1	Specular Common RGB Value, Specular Colour
2	Emission Common RGB Value, Emission Colour
3	Blending Flag, Source Blending Factor, Destination Blending Factor
4	Override Vertex Colour Flag
5	Material Reflectivity
6	Diffuse Colour

7	Diffuse Alpha
8	

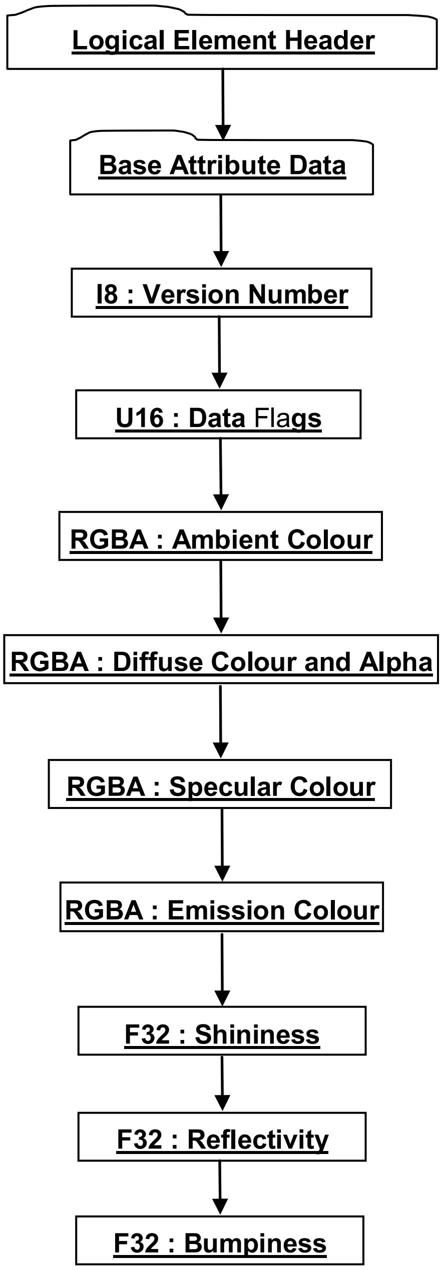


Figure 48 — Material Attribute Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Attribute Data can be found in 6.1.2.1.1 Base Attribute Data.

I8 : Version Number

Version Number is the version identifier for this element. The value of this Version Number indicates the format of data fields to follow.

Table 17 — Material Attribute Version number value

= 1	Version-1 Format

U16 : Data Flags

Data Flags is a collection of flags and factor data. The flags and factor data are combined using the binary OR operator. The flags store information to be used for interpreting how to read subsequent Material data fields. All bits fields that are not defined as in use should be set to —'0

Table 18 — Material Attribute Data Flag values

0x0010	Blending Flag. Blending is a colour combining operation in the graphics pipeline that happens just before writing a colour to the framebuffer. If Blending is ON then incoming fragment RGBA colour values are used (based on Source Blend Factor) and existing framebuffer‘s RGBA colour values are used (based on Destination Blend Factor) to blend between the incoming fragment RGBA and the current frame buffer RGBA to arrive at a new RGBA colour to write into the framebuffer. If Blending is OFF then incoming fragment RGBA colour is written directly into framebuffer unmodified (i.e. completely overriding existing framebuffer RGBA colour). = 0 – Blending OFF. = 1 – Blending ON
0x0020	Override Vertex Colours Flag. If ON, then a shape‘s per vertex colours are to be overridden by the accumulated Material colour. = 0 – Override OFF = 1 – Override ON
0x07C0	Source Blend Factor (stored in bits 6 – 10 or in binary notation 0000011111000000). If Blending Flag enabled, this value indicates how the incoming fragment‘s (i.e. the source) RGBA colour values are to be used to blend with the current framebuffer‘s (i.e. the destination) RGBA colour values. Additional information on the interpretation of the Blending Factor values and how one might leverage them to render an image can be found in reference [1] listed in the bibliography section. = 0 – Interpret same as OpenGL **GL_ZERO** Blending Factor = 1 – Interpret same as OpenGL **GL_ONE** Blending Factor = 2 – Interpret same as OpenGL **GL_DST_COLOUR** Blending Factor = 3 – Interpret same as OpenGL **GL_SRC_COLOUR** Blending Factor = 4 – Interpret same as OpenGL **GL_ONE_MINUS_DST_COLOUR** Blending Factor = 5 – Interpret same as OpenGL **GL_ONE_MINUS_SRC_COLOUR** Blending Factor = 6 – Interpret same as OpenGL **GL_SRC_ALPHA** Blending Factor = 7 – Interpret same as OpenGL **GL_ONE_MINUS_SRC_ALPHA** Blending Factor = 8 – Interpret same as OpenGL **GL_DST_ALPHA** Blending Factor = 9 – Interpret same as OpenGL **GL_ONE_MINUS_DST_ALPHA** Blending Factor = 10 – Interpret same as OpenGL **GL_SRC_ALPHA_SATURATE** Blending Factor
0xF800	Destination Blend Factor (stored in bits 11 – 15 or in binary notation 1111100000000000).). If Blending Flag enabled, this value indicates how the current framebuffer‘s (the destination) RGBA colour values are to be used to blend with the incoming fragment‘s (the source) RGBA colour values. Additional information on the interpretation of the Blending Factor values and how one might leverage them to render an image can be found in reference [1] listed the bibliography section. = 0 – Interpret same as OpenGL **GL_ZERO** Blending Factor = 1 – Interpret same as OpenGL **GL_ONE** Blending Factor = 2 – Interpret same as OpenGL **GL_DST_COLOUR** Blending Factor = 3 – Interpret same as OpenGL **GL_SRC_COLOUR** Blending Factor = 4 – Interpret same as OpenGL **GL_ONE_MINUS_DST_COLOUR** Blending Factor = 5 – Interpret same as OpenGL **GL_ONE_MINUS_SRC_COLOUR** Blending Factor = 6 – Interpret same as OpenGL **GL_SRC_ALPHA** Blending Factor = 7 – Interpret same as OpenGL **GL_ONE_MINUS_SRC_ALPHA** Blending Factor

	= 8 – Interpret same as OpenGL **GL_DST_ALPHA** Blending Factor = 9 – Interpret same as OpenGL **GL_ONE_MINUS_DST_ALPHA** Blending Factor = 10 – Interpret same as OpenGL **GL_SRC_ALPHA_SATURATE** Blending Factor

RGBA : Ambient Colour

Ambient Colour specifies the ambient red, green, blue, alpha colour values of the material.

RGBA : Diffuse Colour and Alpha

Diffuse Colour and Alpha specify the diffuse red, green, blue colour components, and alpha value of the material.

RGBA : Specular Colour

Specular Colour specifies the specular red, green, blue, alpha colour values of the material.

RGBA : Emission Colour

Emission Colour specifies the emissive red, green, blue, alpha colour values of the material.

F32 : Shininess

Shininess is the exponent associated with specular reflection and highlighting of the Phong specular lighting model. Shininess controls the degree with which the specular highlight decays. Only values in the range [1,128] are valid.

F32 : Reflectivity

Reflectivity specifies the material reflectivity of the material. It represents the fraction of light reflected in the mirror direction by the material. Only values in the range [0.0, 1.0] are valid.

F32 : Bumpiness

Bumpiness is used to control bump mapping, and specifies the degree to which bump mapping modifies the local normal vector. A value of 1.0 is the default. Values larger than 1.0 are intended to make the shaded object look as if it is more highly embossed; values between 0.0 and 1.0 make it look less so. Negative values are legal and make the object appear to be *engraved* rather than embossed.

6.1.2.3 Texture Image Attribute Element

Object Type ID: 0x10dd1073, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Texture Image Attribute Element defines a texture image and its mapping environment. JT format LSG traversal semantics state that texture image attributes accumulate down the LSG by replacement on a *per texture channel* basis. See below for more information on texture image channels.

Note that additional information on the interpretation of the various Texture Image Attribute Element data fields can be found in the OpenGL references listed in the bibliography section [1GUID].

The Field Inhibit and Field Final flag (see 6.1.2.1.1 Base Attribute Data) bit assignments for the Texture Image Attribute Element data fields, are as follows:

Table 19 — Texture Image Attribute data field inihibit values

Field Inhibit Flag Bit	Data Field(s) Bit Applies To
0	I32 : Texture Type, Mipmap Image Texel Data, **Error! Reference source not found.**, Shared Image Flag
1	Border Mode, Border Colour
2	Mipmap Minification Filter, Mipmap Magnification Filter
3	S-Dimen Wrap Mode, T-Dimen Wrap Mode, R-Dimen Wrap Mode
4	Blend Type, Blend Colour
5	Texture Transform
6	Tex Coord Gen Mode, Tex Coord Reference Plane

7	Internal Compression Level

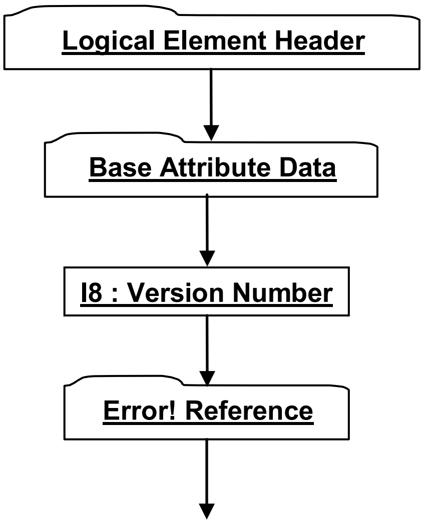


Figure 49 — Texture Image Attribute Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Attribute Data can be found in 6.1.2.1.1 Base Attribute Data.

Complete description for Texture Vers-3 Data can be found in 6.1.2.3.1 Texture Vers-1 Data.

I8 : Version Number

Version Number is the version identifier for this element. The value of this Version Number indicates the format of data fields to follow.

Table 20 — Texture Image Version Number values

= 1	Version-1 Format

When a data element in the JT file is versioned, it is for the purpose of adding a few pieces of new data onto the end of the existing data format. In this way, older viewers and readers of the JT file that do not yet know about higher local versions will naturally read the lower-numbered version blocks and ignore the higher-numbered ones they do not know how to read. At present, this mechanism is not being used in JT v10, but experience has shown from previous versions of JT that it probably will become useful at some point during the life of JT v10.

6.1.2.3.1 Texture Vers-1 Data

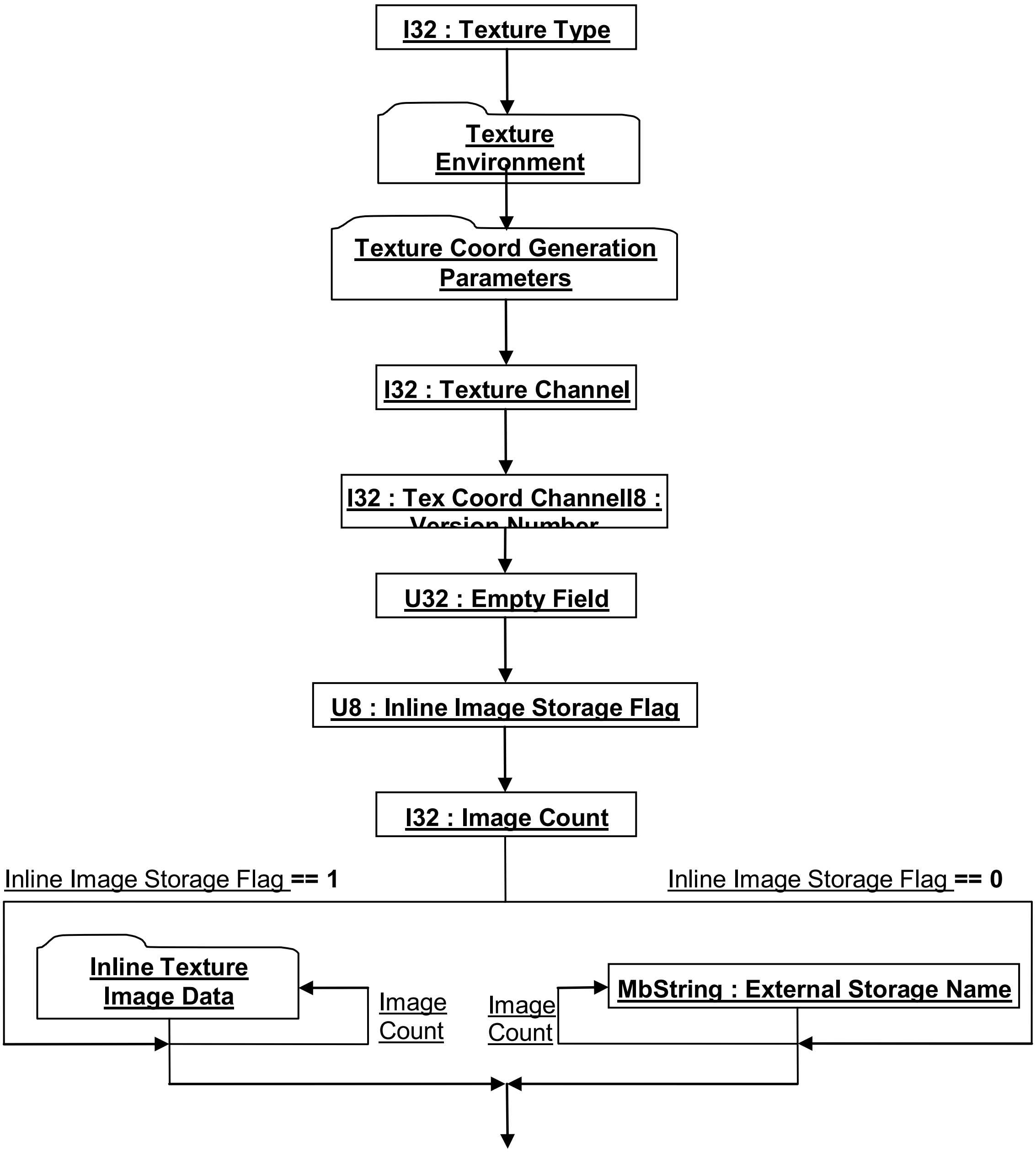

I32 : Texture Type
Texture Environment
Texture Coord Generation Parameters
I32 : Texture Channel
I32 : Tex Coord ChannelI8 : Version Number
U32 : Empty Field
U8 : Inline Image Storage Flag
I32 : Image Count
Inline Image Storage Flag == 1
Inline Image Storage Flag == 0
Inline Texture Image Data
Image Count
Image Count
MbString : External Storage Name

Figure 50 — Texture Vers-1 Data collection

Complete details for Texture Environment can be found in Texture Environment.

Complete details for Texture Coord Generation Parameters can be found in Texture Coord Generation Parameters.

Complete details for Inline Texture Image Data can be found in Inline Texture Image Data.

I32 : Texture Type

Texture Type specifies the type of texture. A new texture type, separator texture, is defined in Texture Vers-1 Data to support resetting the texture accumulation state mid-graph. Shadow maps and prefiltered light maps, however, are a general exception to this rule. In the following list, ―image” refers to an image texture, ―pre-lit” indicates that the image texture is to be applied before lighting when rendering the object to which it is applied, and ―post-lit" indicates that the image texture is to be applied after lighting. A gloss map is a pre-lit texture that applies itself to the specular material component of lighting instead of the diffuse component. A light map is an environment texture (texture at infinity surrounding the whole model) that serves as a source of illumination during shading calculations.

Table 21 — Texture Vers-1 Texture Type values

Texture Type	Description	Explicit Channel	Auto Channel
= 0	None.	N/A	N/A
= 1	One-Dimensional post-lit image texture.	Yes	No
= 2	Two-Dimensional post-lit image texture.	Yes	No
= 3	Three-Dimensional post-lit image texture.	Yes	No
= 4	Two-Dimensional 3-component tangent-space normal map.	No	Yes
= 5	Cube post-lit image texture.	Yes	No
= 7	Cube pre-lit image texture.	Yes	No
= 8	One-Dimensional pre-lit image texture.	Yes	No
= 9	Two-Dimensional pre-lit image texture.	Yes	No
= 10	Three-Dimensional pre-lit image texture.	Yes	No
= 11	Cube environment map.	No	Yes
= 12	One-Dimensional gloss map (specular) texture.	No	Yes
= 13	Two-Dimensional gloss map (specular) texture.	No	Yes
= 14	Three-Dimensional gloss map (specular) texture.	No	Yes
= 15	Cube gloss map (specular) texture.	No	Yes
= 16	Two-Dimensional 1-component bumpmap.	No	Yes
= 17	Two-Dimensional 3-component world-space normal map.	No	Yes
= 18	Two-Dimensional sphere environment map.	No	Yes
= 19	Two-Dimensional latitude/longitude environment map.	No	Yes
= 20	Two-Dimensional spherical diffuse light map.	No	Yes
= 21	Cube diffuse light map.	No	Yes
= 22	Two-Dimensional latitude/longitude diffuse light map.	No	Yes
= 23	Two-Dimensional spherical specular light map.	No	Yes
= 24	Cube specular light map.	No	Yes
= 25	Two-Dimensional latitude/longitude specular light map.	No	Yes
= 26	Resets texture state except shadow map and light maps.	N/A	N/A

I32 : Texture Channel

Texture Channel specifies the texture channel number for the Texture Image Element. For purposes of multi-texturing, the JT concept of a texture channel corresponds to the OpenGL concept of a ―texture unit." The Texture Channel value shall be between -1 and 2,147,483,647 inclusive. The value -1 is accepted to denote a texture whose channel number is to be automatically assigned. This assignment will never displace another texture with an explicit texture channel assignment from its slot. Best practices suggest that a renderer of JT data ignore all but channel-0 if the renderer does not support multi-textured geometry. Also for purposes of blending, any renderer of JT data should ensure that higher numbered texture channels ―blend over" lower numbered ones.

Pre- and post-lit image textures shall specify an explicit texture channel. All other texture types shall specify -1 for their texture channel.

U32 : Empty Field

Refer to best practice 14.4 Empty Field

U8 : Inline Image Storage Flag

Inline Image Storage Flag is a flag that indicates whether the texture image is stored within the JT File (i.e. inline) or in some other external file.

Table 22 — Texture Vers-1 Inline Image Storage Flag values

= 0	Texture image stored in an external file.
= 1	Texture image stored inline in this JT file.

I32 : Image Count

Image Count specifies the number of texture images. A ―Cube Map” I32 : Texture Type shall have six images while all other Texture Types should only have one image.

MbString : External Storage Name

External Storage Name is a string identifying the name of an external texture image storage. External Storage Name is only present if data field **Error! Reference source not found.** equals ―0. If present there will be data field **Error! Reference source not found.** number of External Storage Name instances. This External Storage Name string is a relative path based name for the texture image file. Where ―relative path” should be interpreted to mean the string contains the file name along with any additional path information that locates the texture image file relative to the location of the referencing JT file.

I32 : Tex Coord Channel

Tex Coord Channel specifies the channel number for texture coordinate generation. Value shall be within range [-1, 2147483647] inclusive.

Texture Environment

The Texture Environment is a collection of data defining various aspects of how a texture image is to be mapped/applied to a surface.

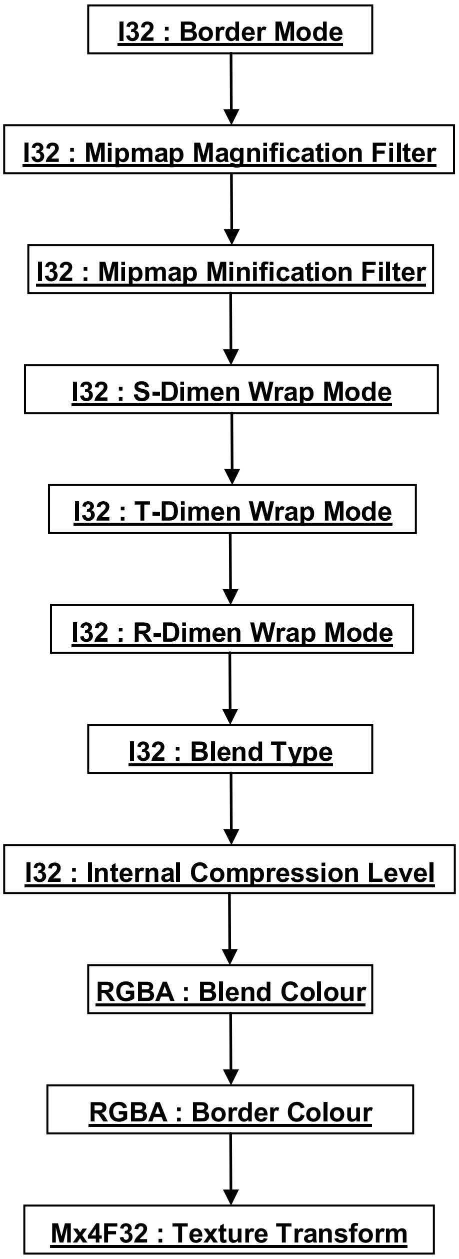


Figure 51 — Texture Environment data collection

I32 : Border Mode

Border Mode specifies the texture border mode.

Table 23 — Texture Vers-1 Texture Environment Border Mode values

= 0	No border.
= 1	Constant Border Colour. Indicates that the texture has a constant border colour whose value is defined in data field Border Colour.
= 2	Explicit. Indicates that a border texel ring is present in the texture image definition.

I32 : Mipmap Magnification Filter

Mipmap Magnification Filter specifies the texture filtering method to apply when a single pixel on screen maps to a tiny portion of a texel.

Table 24 — Texture Vers-1 Texture Environment Mipmap Magnification Filter values

= 0	None.
= 1	Nearest. Texel with coordinates nearest the centre of the pixel is used.
= 2	Linear. A weighted linear average of the 2 x 2 array of texels nearest to the centre of the pixel is used. For one-dimensional texture is average of 2 texels. For three dimensional texel is 2 x 2 x 2 array.

I32 : Mipmap Minification Filter

Mipmap Minification Filter specifies the texture filtering method to apply when a single pixel on screen maps to a large collection of texels.

Table 25 — Texture Vers-1 Texture Environment Mipmap Minification Filter values

= 0	None.
= 1	Nearest. Texel with coordinates nearest the centre of the pixel is used.
= 2	Linear. A weighted linear average of the 2 x 2 array of texels nearest to the centre of the pixel is used. For one-dimensional texture is average of 2 texels. For three-dimensional texture is 2 x 2 x 2 array.
= 3	Nearest in Mipmap. Within an individual mipmap, the texel with coordinates nearest the centre of the pixel is used.
= 4	Linear in Mipmap. Within an individual mipmap, a weighted linear average of the 2 x 2 array of texels nearest to the centre of the pixel is used. For one-dimensional texture is average of 2 texels. For three-dimensional texture is 2 x 2 x 2 array
= 5	Nearest between Mipmaps. Within each of the adjacent two mipmaps, selects the texel with coordinates nearest the centre of the pixel and then interpolates linearly between these two selected mipmap values.
= 6	Linear between Mipmaps. Within each of the two adjacent mipmaps, computes value based on a weighted linear average of the 2 x 2 array of texels nearest to the centre of the pixel and then interpolates linearly between these two computed mipmap values.

I32 : S-Dimen Wrap Mode

S-Dimen Wrap Mode specifies the mode for handling texture coordinates S-Dimension values outside the range [0, 1].

Table 26 — Texture Vers-1 Texture Environment S-Dimen Wrap Mode values

= 0	None.
= 1	Clamp. Any values greater than 1.0 are set to 1.0; any values less than 0.0 are set to 0.0
= 2	Repeat Integer parts of the texture coordinates are ignored (i.e. retains only the fractional component o texture coordinates greater than 1.0 and only one-minus the fractional component of values less than zero). Resulting in copies of the texture map tiling the surface
= 3	Mirror Repeat. Like Repeat, except the surface tiles –flip-flop” resulting in an alternating mirror pattern of surface tiles.
= 4	Clamp to Edge. Border is always ignored and instead texel at or near the edge is chosen for coordinates outside the range [0, 1]. Whether the exact nearest edge texel or some average of the nearest edge texels is used is dependent upon the mipmap filtering value.
= 5	Clamp to Border. Nearest border texel is chosen for coordinates outside the range [0, 1]. Whether the exact nearest border texel or some average of the nearest border texels is used is dependent upon the mipmap filtering value.

I32 : T-Dimen Wrap Mode

T-Dimen Wrap Mode specifies the mode for handling texture coordinates T-Dimension values outside the range [0, 1]. Same mode values as documented for S-Dimen Wrap Mode.

I32 : R-Dimen Wrap Mode

R-Dimen Wrap Mode specifies the mode for handling texture coordinates R-Dimension values outside the range [0, 1]. Same mode values as documented for S-Dimen Wrap Mode.

I32 : Blend Type

Blend Type contains information indicating how the values in the texture map are to be modulated/combined/blended with the original colour of the surface or some other alternative colour to compute the final colour to be painted on the surface. Additional information on the interpretation of the Blend Type values and how one might leverage them to render an image can be found in reference [1] listed in the bibliography section.

Table 27 — Texture Vers-1 Texture Environment Blend Type values

= 0	None.
= 1	Decal. Interpret same as OpenGL **GL_DECAL** environment mode.
= 2	Modulate. Interpret same as OpenGL **GL_MODULATE** environment mode.
= 3	Replace. Interpret same as OpenGL **GL_REPLACE** environment mode.
= 4	Blend. Interpret same as OpenGL **GL_BLEND** environment mode.
= 5	Add. Interpret same as OpenGL **GL_ADD** environment mode.
= 6	Combine. Interpret same as OpenGL **GL_COMBINE** environment mode.

I32 : Internal Compression Level

Internal Compression Level specifies a data compression hint/recommendation that a JT file loader is free to follow for internally (in memory) storing texel data. This setting does not affect how image texel data is actually stored in JT files or other externally referenced files.

Table 28 — Texture Vers-1 Texture Environment Internal Compression Level values

= 0	None. No compression of texel data.
= 1	Conservative. Lossless compression of texel data.
= 2	Moderate. Texel components truncated to 8-bits each.
= 3	Aggressive. Texel components truncates to 4-bits each (or 5 bits for RGB images).

RGBA : Blend Colour

Blend Colour specifies the colour to be used for the —Blend” mode of Blend Type operations.

RGBA : Border Colour

Border Colour specifies the constant border colour to use for —Clamp to Border” style wrap modes when the texture itself does not have a border.

Mx4F32 : Texture Transform

Texture Transform defines the texture coordinate transformation matrix. A renderer of JT data would typically apply this transform to texture coordinates prior to applying the texture.

Texture Coord Generation Parameters

Texture Coord Generation Parameters contains information indicating if and how texture coordinate components should be automatically generated for each of the 4 components (S, T, R, Q) of a texture coordinate.

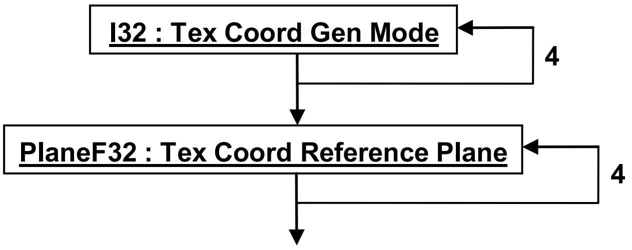


Figure 52 — Texture Coord Generation Parameters data collection

I32 : Tex Coord Gen Mode

Tex Coord Gen Mode specifies the texture coordinate generation mode for each component (S, T, R, Q) of texture coordinate. There are four mode values stored, one for each component of texture coordinate. The mode values are stored in S, T, R, Q order.

Table 29 — Texture Vers-1 Texture Coord Generation Gen Mode values

= 0	None. No texture coordinates automatically generated.
= 1	Model Coordinate System Linear. Texture coordinates computed as a distance from a reference plane specified in model coordinates.
= 2	View Coordinate System Linear. Texture coordinates computed as a distance from a reference plane specified in view coordinates.
= 3	Sphere Map. Texture coordinates generated based on spherical environment mapping.
= 4	Reflection Map. Texture coordinates generated based on cubic environment mapping.
= 5	Normal Map. Texture coordinates computed/set by copying vertex normal in view coordinates to S, T, R.

PlaneF32 : Tex Coord Reference Plane

Reference Plane specifies the reference plane used for ―Model Coordinate System Linear" and ―View Coordinate System Linear" texture coordinate generation modes. There are four Reference Planes stored, one for each component of texture coordinate. The Reference Planes are stored in S, T, R, Q order. Even if a components ―Tex Coord Gen Mode" is one that does not require a reference plane, dummy reference planes are still stored in JT file.

Inline Texture Image Data

Inline Texture Image Data is a collection of data defining the texture format properties and image texel data for one texture image. Inline Texture Image Data is only present if data field **Error! Reference source not found.** equals ―1.‖ If present there will be data field **Error! Reference source not found.** number of Inline Texture Image Data instances.

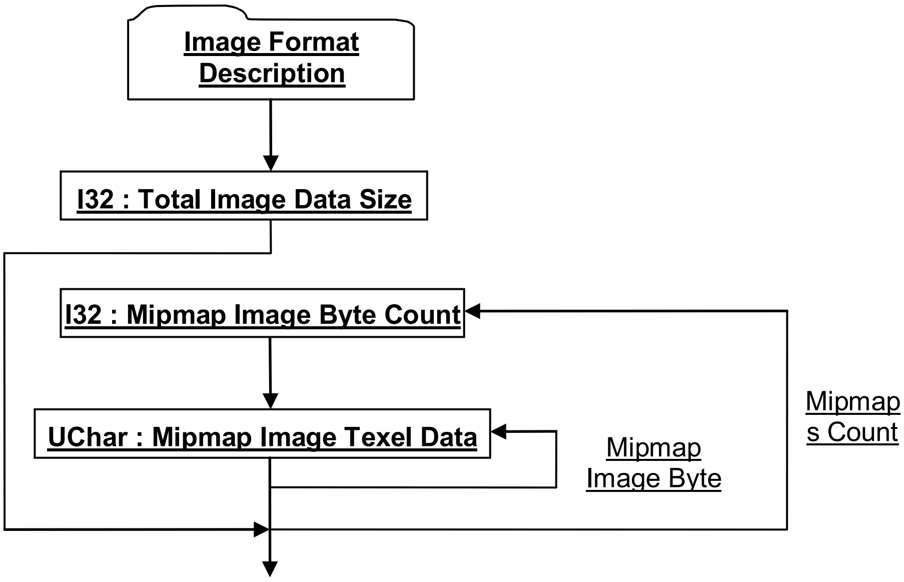


Image Format Description
I32 : Total Image Data Size
I32 : Mipmap Image Byte Count
UChar : Mipmap Image Texel Data
Mipmap Image Byte
Mipmaps Count

Figure 53 — Inline Texture Image Data collection

Complete description for Image Format Description can be found in Image Format Description.

I32 : Total Image Data Size

Total Image Data Size specifies the total length, in bytes, of the on-disk representation for all mipmap images. This byte total does not include the I32 : Mipmap Image Byte Count data field storage (4 bytes per) for each mipmap.

I32 : Mipmap Image Byte Count

Mipmap Image Byte Count specifies the length, in bytes, of the on-disk representation of the next mipmap image.

UChar : Mipmap Image Texel Data

Mipmap Image Texel Data is the mipmap‘s block of image data. The length of this field in bytes is specified by the value of data field Mipmap Image Byte Count.

Image Format Description

The Image Format Description is a collection of data defining the pixel format, data type, size, and other miscellaneous characteristics of the texel image data.

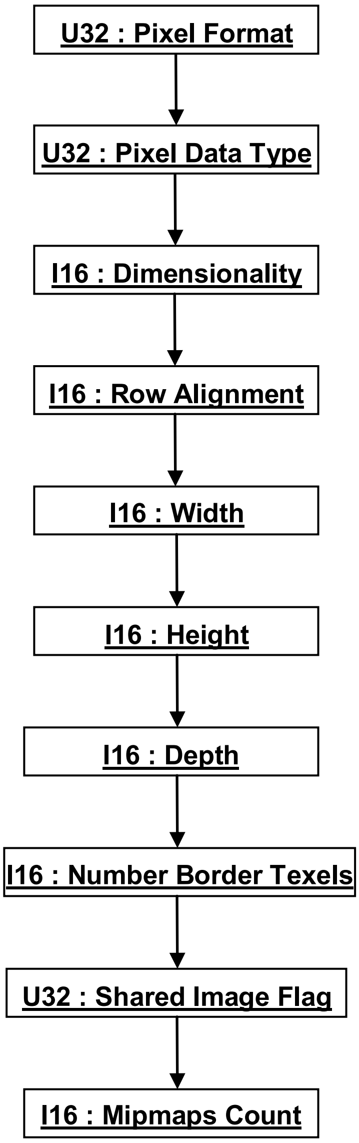


Figure 54 — Image Format Description data collection

U32 : Pixel Format

Pixel format specifies the format of the texture image pixel data. Depending on the format, anywhere from one to four elements of data exists per texel.

Table 30 — Texture Vers-1 Image Format Description Pixel Format values

= 0	No format specified. Texture mapping is not applied.

= 1	RGB: A red colour component followed by green and blue colour components
= 2	RGBA: A red colour component followed by green, blue, and alpha colour components
= 3	LUM: A single luminance component
= 4	LUMA: A luminance component followed by an alpha colour component.
= 5	A single stencil index.
= 6	A single depth component
= 7	A single red colour component
= 8	A single green colour component
= 9	A single blue colour component
= 10	A single alpha colour component
= 11	A blue colour component, followed by green and red colour components
= 12	A blue colour component, followed by green , red, and alpha colour components
= 13	A depth component, followed by a stencil component

U32 : Pixel Data Type

Pixel Data Type specifies the data type used to store the per texel data. If the Pixel Format represents a multi component value (e.g. red, green, blue) then each value requires the Pixel Data Type number of bytes of storage (e.g. a Pixel Format Type of —"1with Pixel Data Type of —"3would require 3 bytes of storage for each texel).

Table 31 — Texture Vers-1 Image Format Description Pixel Data values

= 0	No type specified. Texture mapping is not applied.
= 1	Signed 8-bit integer
= 2	Single-precision 32-bit floating point
= 3	Unsigned 8-bit integer
= 4	Single bits in unsigned 8-bit integers
= 5	Unsigned 16-bit integer
= 6	Signed 16-bit integer
= 7	Unsigned 32-bit integer
= 8	Signed 32-bit integer
= 9	16-bit floating point according to IEEE-754 format (i.e. 1 sign bit, 5 exponent bits, 10 mantissa bits)

I16 : Dimensionality

Dimensionality specifies the number of dimensions the texture image has. Valid values include:

Table 32 — Texture Vers-1 Image Format Description Dimensionality values

= 1	One-dimensional texture
= 2	Two-dimensional texture
= 3	Three-dimensional texture

I16 : Row Alignment

Row Alignment specifies the byte alignment for image data rows. This data field shall have a value of 1, 2, 4, or 8. If set to 1 then all bytes are used (i.e. no bytes are wasted at end of row). If set to 2, then if necessary, an extra wasted byte(s) is/are stored at the end of the row so that the first byte of the next row has an address that is a multiple of 2 (multiple of four for Row Alignment equal 4 and multiple of 8 for row alignment equal 8). The actual formula (using C syntax) to determine number of bytes per row is as follows:

$$\text{BytesPerRow} = (\text{numBytesPerPixel} * \text{ImageWidth} + \text{RowAlignment} - 1)\ \&\ \sim(\text{RowAlignment} - 1)$$

I16 : Width

Width specifies the width dimension (number of texel columns) of the texture image in number of pixels.

I16 : Height

Height specifies the height dimension (number of texel rows) of the texture image in number of pixels. Height is 1 for one-dimensional images.

I16 : Depth

Depth specifies the depth dimension (number of texel slices) of the texture image in number of pixels. Depth is 1 for one-dimensional and two-dimensional images.

I16 : Number Border Texels

Number Border Texels specifies the number of border texels in the texture image definition. Valid values are 0 and 1.

U32 : Shared Image Flag

Shared Image Flag is a flag indicating whether this texture image is shareable with other Texture Image Element attributes.

Table 33 — Texture Vers-1 Image Format Description Shared Image Flag values

= 0	Image is not shareable with other Texture Image Elements.
= 1	Image is shareable with other Texture Image Elements.

I16 : Mipmaps Count

Mipmaps Count specifies the number of mipmap images. A value of 1 indicates that no mipmaps are used. A value greater than 1 indicates that mipmaps are present all the way down to a 1-by-1 texel.

6.1.2.4 Draw Style Attribute Element

Object Type ID: 0x10dd1014, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Draw Style Attribute Element contains information defining various aspects of the graphics state/style that should be used for rendering associated geometry. JT format LSG traversal semantics state that draw style attributes accumulate down the LSG by replacement.

The Field Inhibit flag (see 6.1.2.1.1Base Attribute Data) bit assignments for the Draw Style Attribute Element data fields, are as follows:

Table 34 — Draw Style Attribute Field Inhibit flag values

Field Inhibit Flag Bit	Data Field(s) Bit Applies To
0	Two Sided Lighting Flag
1	Back-face Culling Flag
2	Outlined Polygons Flag
3	Lighting Enabled Flag
4	Flat Shading Flag
5	Separate Specular Flag

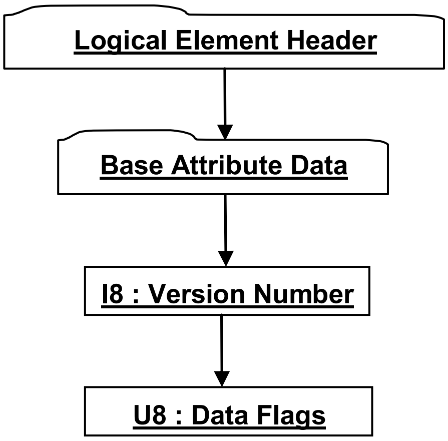


Figure 55 — Draw Style Attribute Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Attribute Data can be found in 6.1.2.1.1 Base Attribute Data.

I8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers. Only version 1 is defined for JT v10.0.

U8 : Data Flags

Data Flags is a collection of flags. The flags are combined using the binary OR operator and store various state settings for Draw Style Attribute Elements. All bits fields that are not defined as in use should be set to ―0‖

Table 35 — Draw Style Attribute Data Flag values

0x01	Back-face Culling Flag. Indicates if back-facing polygons should be discarded (culled). = 0 – Back-facing polygons not culled. = 1 – Back-facing polygons culled.
0x02	Two Sided Lighting Flag. Indicates if two sided lighting should be enabled to insure that polygons are illuminated on both sides. = 0 – Disable two sided lighting. = 1 – Enable two sided lighting.
0x04	Outlined Polygons Flag. Indicates if polygons should be draw as ―wireframes" i.e. not filled. = 0 – Polygons drawn as filled. = 1 – Only polygon‘s outline drawn.
0x08	Lighting Enabled Flag. Indicates if lighting should be enabled. If lighting disabled, then renderer should perform no calculations concerning normals, light sources, material properties, etc. = 0 – Disable lighting. = 1 – Enable lighting.
0x10	Flat Shading Flag. Indicates if the geometry should be rendered with single colour (flat shading) or with many different colour (smooth/Gouraud) shading. = 0 – Disable flat shading (i.e. use smooth/Gouraud shading). = 1 – Enable flat shading.
0x20	Separate Specular Flag. Indicates if the application of the specular colour should be delayed until after texturing. If no texture mapping then this flag setting is irrelevant. = 0 – Apply specular colour contribution before texture mapping. = 1 – Apply specular colour contribution after texture mapping.

6.1.2.5 Light Set Attribute Element

Object Type ID: 0x10dd1096, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Light Set Attribute Element holds an unordered list of Lights. JT format LSG traversal semantics state that light set attributes accumulate down the LSG through addition of lights to an attribute list.

Light Set Attribute Element does not have any Field Inhibit flag (see 6.1.2.1.1 Base Attribute Data) bit assignments.

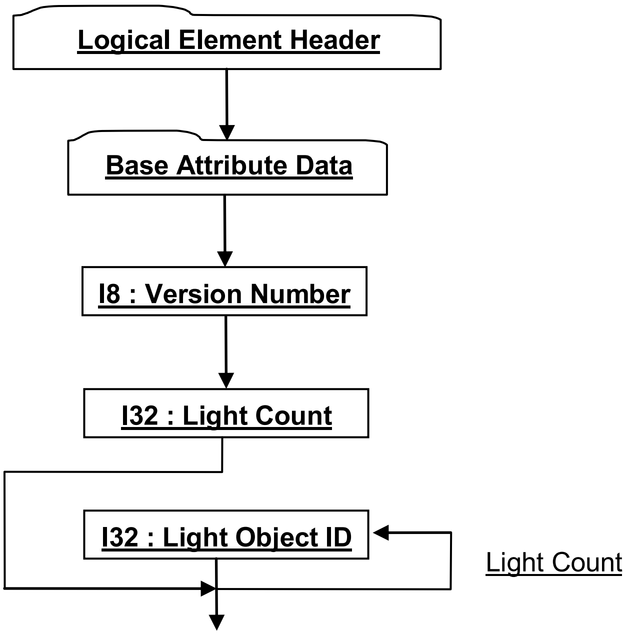


Figure 56 — Light Set Attribute Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Attribute Data can be found in 6.1.2.1.1 Base Attribute Data.

I8 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice 14.5 Local version numbers. Only version 1 is defined for JT v10.0.

I32 : Light Count

Light Count specifies the number of lights in the Light Set.

I32 : Light Object ID

Light Object ID is the identifier for a referenced Light Object.

6.1.2.6 Infinite Light Attribute Element

Object Type ID: 0x10dd1028, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Infinite Light Attribute Element specifies a light source emitting unattenuated light in a single direction from every point on an infinite plane. The infinite location indicates that the rays of light can be considered parallel by the time they reach an object.

JT format LSG traversal semantics state that infinite light attributes accumulate down the LSG through addition of lights to an attribute list.

Infinite Light Attribute Element does not have any Field Inhibit flag (see 6.1.2.1.1 Base Attribute Data) bit assignments.

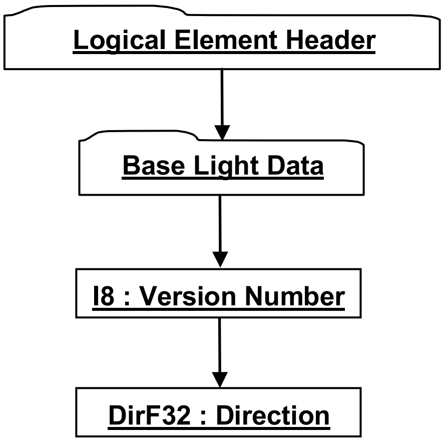


Figure 57 — Infinite Light Attribute Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

Complete description for Base Light Data can be found in 6.1.2.6.1 Base Light Data.

I8 : Version Number

Version Number is the version identifier for this element. The value of this Version Number indicates the format of data fields to follow.

Table 36 — Light Set Attribute Version Number values

= 1	Version-1 Format

DirF32 : Direction

Direction specifies the direction the light is pointing in.

6.1.2.6.1 Base Light Data

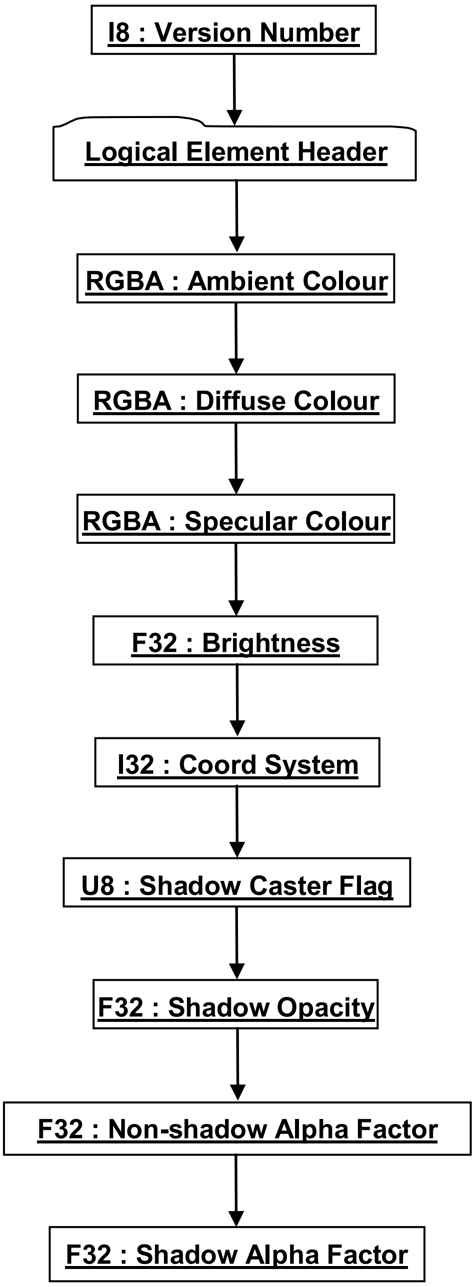


Figure 58 — Base Light Data collection

I8 : Version Number

Version number is the version identifier for this element. For information on local version numbers see best practice 14.5 Local version numbers. Only version 1 is defined for JT v10.0.

RGBA : Ambient Colour

Ambient Colour specifies the ambient red, green, blue, alpha colour values of the light.

RGBA : Diffuse Colour

Diffuse Colour specifies the diffuse red, green, blue, alpha colour values of the light.

RGBA : Specular Colour

Specular Colour specifies the specular red, green, blue, alpha colour values of the light.

F32 : Brightness

Brightness specifies the Light brightness. The Brightness value shall be greater than or equal to —1”.

I32 : Coord System

Coord System specifies the coordinate space in which Light source is defined. Valid values include the following:

Table 37 — Base Light Data Cood System values

= 1	Viewpoint Coordinate System. Light source is to move together with the viewpoint
= 2	Model Coordinate System. Light source is affected by whatever model transforms that are current when the light source is encountered in LSG.
= 3	World Coordinate system. Light source is not affected by model transforms in the LSG.

U8 : Shadow Caster Flag

Shadow Caster Flag is a flag that indicates whether the light is a shadow caster or not.

Table 38 — Base Light Data Shadow Caster Flag values

= 0	Light source is not a shadow caster.
= 1	Light source is a shadow caster.

F32 : Shadow Opacity

Shadow Opacity specifies the shadow opacity factor on Light source. Value shall be within range [0.0, 1.0] inclusive. Shadow Opacity is intended to convey how dark a shadow cast by this light source are to be rendered. A value of 1.0 means that no light from this light source reaches a shadowed surface, resulting in a black shadow.

F32 : Non-shadow Alpha Factor

Non-shadow Alpha Factor is one of a matched pair of fields intended to govern how a shadowing light source (one whose Shadow Caster Flag is set) casts "alpha light" into areas that it directly illuminates (i.e. are not in shadow). Those fragments directly lit by this light source will have their alpha values scaled by Non-shadow Alpha Factor. Non-shadow Alpha Factor value shall lie on the range [0.0, 1.0] inclusive.

This field can be used to create "drop shadows" by setting its value to 0. The effect being that all geometry illuminated by the light source will be "burned away," leaving behind only those parts lying in shadow. Naturally, implementing this intended behaviour implies extensive viewer support.

F32 : Shadow Alpha Factor

Shadow Alpha Factor is one of a matched pair of fields intended to govern how a shadowing light source (one whose Shadow Caster Flag is set) casts "alpha light" into areas that it does not illuminate (i.e. are in shadow). Those fragments in shadow from this light source will have their alpha values scaled by Shadow Alpha Factor. Shadow Alpha Factor value shall lie on the range [0.0, 1.0] inclusive.

This field has the opposite effect of Non-shadow Alpha Factor. If set to a value of 0, for example, it will cause all geometry shadowed from the light source to be burned away, leaving behind only those parts directly illuminated by the light source. Naturally, implementing this intended behaviour implies extensive viewer support.

6.1.2.7 Point Light Attribute Element

Object Type ID: 0x10dd1045, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Point Light Attribute Element specifies a light source emitting light from a specified position, along a specified direction, and with a specified spread angle

JT format LSG traversal semantics state that point light attributes accumulate down the LSG through addition of lights to an attribute list.

Point Light Attribute Element does not have any Field Inhibit flag (see 6.1.2.1.1 Base Attribute Data) bit assignments.

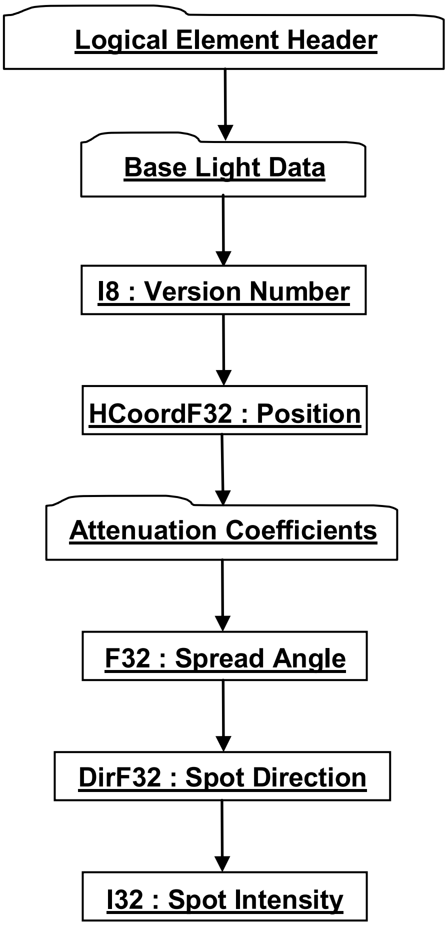


Figure 59 — Point Light Attribute ElementPoint Light Attribute Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

Complete description for Base Light Data can be found in 6.1.2.6.1 Base Light Data.

Complete description for Attenuation Coefficients can be found in 6.1.2.7.1 Attenuation Coefficients.

I8 : Version Number

Version Number is the version identifier for this element. The value of this Version Number indicates the format of data fields to follow.

Table 39 — Point Light Attribute Version Number values

= 1	Version-1 Format

HCoordF32 : Position

Position specifies the light position in homogeneous coordinates.

F32 : Spread Angle

Spread Angle, as shown in Figure 60 — Spread Angle value with respect to the light cone below, specifies in degrees the half angle of the light cone. Valid Spread Angle values are clamped and interpreted as follows:

Table 40 — Point Light Attribute Spread Angle values

angle = = 180.0	Simple point light
0.0 >= angle <= 90.0	Spot Light

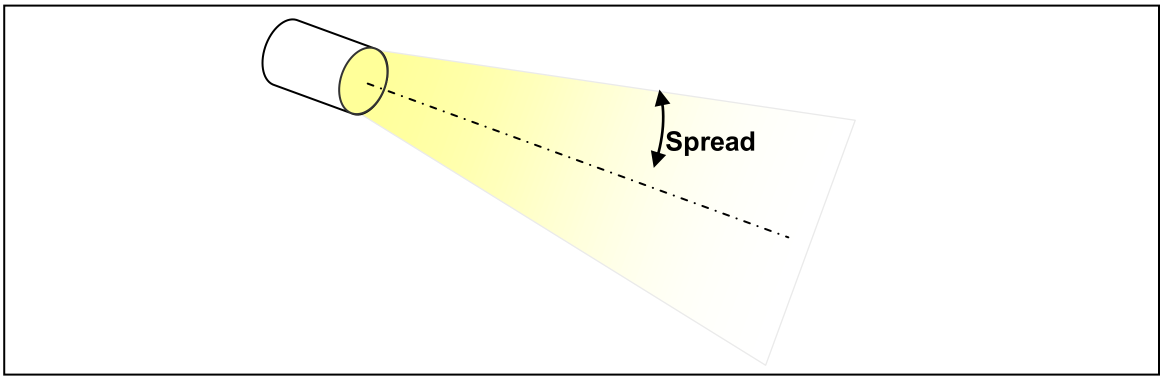


Figure 60 — Spread Angle value with respect to the light cone

DirF32 : Spot Direction

Spot Direction specifies the direction the spot light is pointing in.

I32 : Spot Intensity

Spot Intensity specifies the intensity distribution of the light within the spot light cone. Spot Intensity is really a —spot exponent" in a lighting equation and indicates how focused the light is at the centre. The larger the value, the more focused the light source. Only non-negative Spot intensity values are valid.

6.1.2.7.1 Attenuation Coefficients

Attenuation Coefficients data collection contains the coefficients for how light intensity decreases with distance.

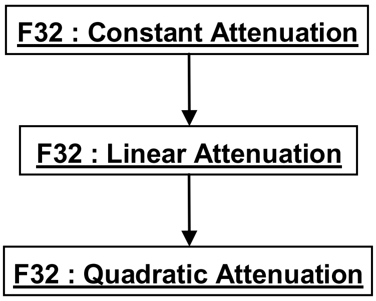


Figure 61 — Attenuation Coefficients data collection

F32 : Constant Attenuation

Constant Attenuation specifies the constant coefficient for how light intensity decreases with distance. Value shall be greater than or equal to —0'

F32 : Linear Attenuation

Linear Attenuation specifies the linear coefficient for how light intensity decreases with distance. Value shall be greater than or equal to —0'

F32 : Quadratic Attenuation

Quadratic Attenuation specifies the quadratic coefficient for how light intensity decreases with distance. Value shall be greater than or equal to —0'

6.1.2.8 Linestyle Attribute Element

Object Type ID: 0x10dd10c4, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Linestyle Attribute Element contains information defining the graphical properties to be used for rendering polylines. JT format LSG traversal semantics state that Linestyle attributes accumulate down the LSG by replacement.

Linestyle Attribute Element does not have any Field Inhibit flag (see 6.1.2.1.1 Base Attribute Data) bit assignments.

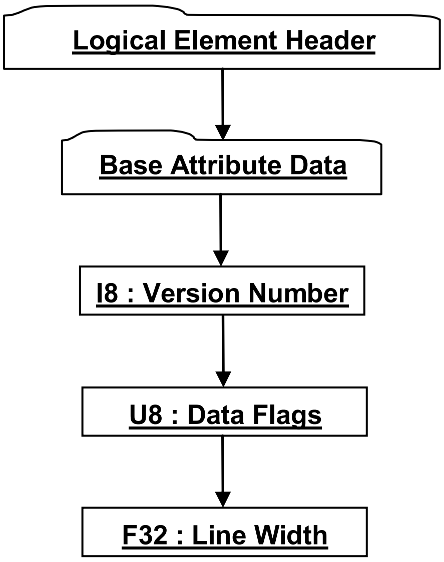


Figure 62 — Linestyle Attribute Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Attribute Data can be found in 6.1.2.1.1 Base Attribute Data.

I8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers

U8 : Data Flags

Data Flags is a collection of flags and line type data. The flags and line type data are combined using the binary OR operator and store various polyline rendering attributes. All bits fields that are not defined as in use should be set to —'0

Table 41 — Linestyle Attribute Data Flag values

0x0F	Line Type (stored in bits 0 – 3 or in binary notation 00001111) Line type specifies the polyline rendering stipple-pattern. = 0 - Solid ———————————— = 1 – Dash - - - - - - - - - - - - - - - - = 2 – Dot ······························ = 3 – Dash_Dot -·-·-·-·-·-·-·-·-·-·-·-·- = 4 – Dash_Dot_Dot —··—··—··—··—··—··— = 5 – Long_Dash — — — — — — — — — — — = 6 – Centre_Dash —·—·—·—·—·—·—·—·—·— = 7 – Centre_Dash_Dash —··—··—··—··—··—··—
0x10	Antialiasing Flag (stored in bit 4 or in binary notation 00010000) Indicates if antialiasing should be applied as part of rendering polylines. = 0 – Antialiasing disabled. = 1 – Antialiasing enabled.

F32 : Line Width

Line Width specifies the width in pixels that should be used for rendering polylines. The value of this field shall be greater than 0.0.

6.1.2.9 Pointstyle Attribute Element

Object Type ID: 0x8d57c010, 0xe5cb, 0x11d4, 0x84, 0xe, 0x00, 0xa0, 0xd2, 0x18, 0x2f, 0x9d

Pointstyle Attribute Element contains information defining the graphical properties that should be used for rendering points. JT format LSG traversal semantics state that Pointstyle attributes accumulate down the LSG by replacement.

Pointstyle Attribute Element does not have any Field Inhibit flag (see 6.1.2.1.1 Base Attribute Data) bit assignments.

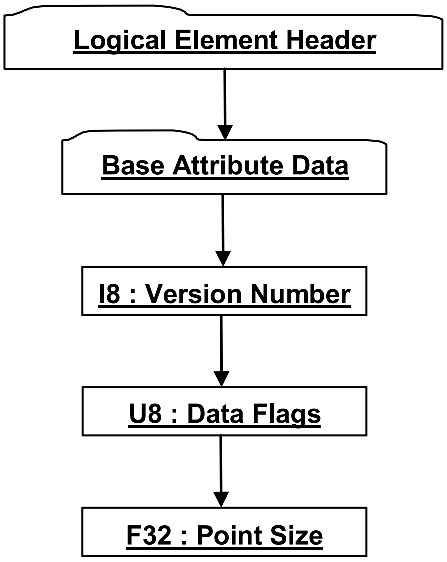


Figure 63 — Pointstyle Attribute Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Attribute Data can be found in 6.1.2.1.1 Base Attribute Data.

I8 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice 14.5 Local version numbers

U8 : Data Flags

Data Flags is a collection of flags and point type data. The flags and point type data are combined using the binary OR operator and store various point rendering attributes. All bits fields that are not defined as in use should be set to —0

Table 42 — Pointstyle Attribute Data Flag values

0x0F	Point Type (stored in bits 0 – 3 or in binary notation 00001111) These bits are reserved for future expansion of the format to support Point Types.
0x10	Antialiasing Flag (stored in bit 4 or in binary notation 00010000) Indicates if antialiasing should be applied as part of rendering points. = 0 – Antialiasing disabled. = 1 – Antialiasing enabled.

F32 : Point Size

Point Size specifies the size in pixels that should be used for rendering points. The value shall be greater than 0.0.

6.1.2.10 Geometric Transform Attribute Element

Object Type ID: 0x10dd1083, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Geometric Transform Attribute Element contains a 4x4 homogeneous transformation matrix that positions the associated LSG node's coordinate system relative to its parent LSG node. JT format LSG traversal semantics state that geometric transform attributes accumulate down the LSG through matrix multiplication as follows:

p' = pAM

Where *p* is a point of the model, *p'* is the transformed point, *M* is the current modeling transformation matrix inherited from ancestor LSG nodes and previous Geometric Transform Attribute Element, and *A* is the transformation matrix of this Geometric Transform Attribute Element. The matrix is allowed to contain translation, rotation, and uniform- and non-uniform scaling factors, including negative scales. It is not allowed to contain shearing or projective components, or scaling factors of zero (which would make the matrix singular).

Geometric Transform Attribute Element does not have any Field Inhibit flag (see 6.1.2.1.1 Base Attribute Data) bit assignments.

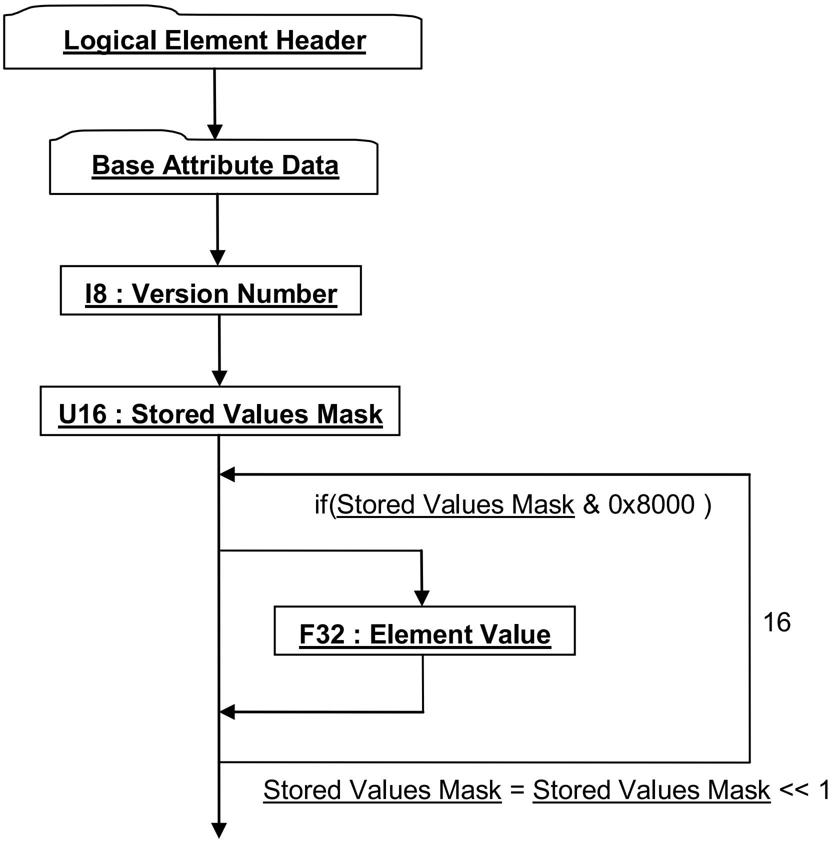


Figure 64 — Geometric Transform Attribute Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

Complete description for Base Attribute Data can be found in 6.1.2.1.1 Base Attribute Data.

I8 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 14.5 Local version numbers

U16 : Stored Values Mask

Stored Values mask is a 16-bit mask where each bit is a flag indicating whether the corresponding element in the matrix is different from the identity matrix. Only elements which are different from the identity matrix are actually stored. The bits are assigned to matrix elements as follows:

Bit15	Bit14	Bit13	Bit12
Bit11	Bit10	Bit9	Bit8

Bit7 Bit6 Bit5 Bit4

Bit3 Bit2 Bit1 Bit0

The individual bit-flag values are interpreted as follows:

Table 43 — Geometric Transform Attribute Stored Value Mask individual bit-flag values

= 0	Value not stored (matrix value same as corresponding element in identity matrix)
= 1	Value stored

F32 : Element Value

Element Value specifies a particular matrix element value.

6.1.2.11 Texture Coordinate Generator Attribute Element

Object Type ID: 0xaa1b831d, 0x6e47, 0x4fee, 0xa8, 0x65, 0xcd, 0x7e, 0x1f, 0x2f, 0x39, 0xdc

Texture Coordinate Generator Attribute Element defines texture coordinate generation for texture mapping. Multiple texture coordinate generation at a given node is supported by way of the -texture coordinate channel" concept. JT format LSG traversal semantics state that Texture Coordinate Generator attributes accumulate down the LSG by replacement on a per-channel basis.

Texture Coordinate Generator Attribute Element does not have any Field Inhibit flag (see 6.1.2.1.1 Base Attribute Data) bit assignments.

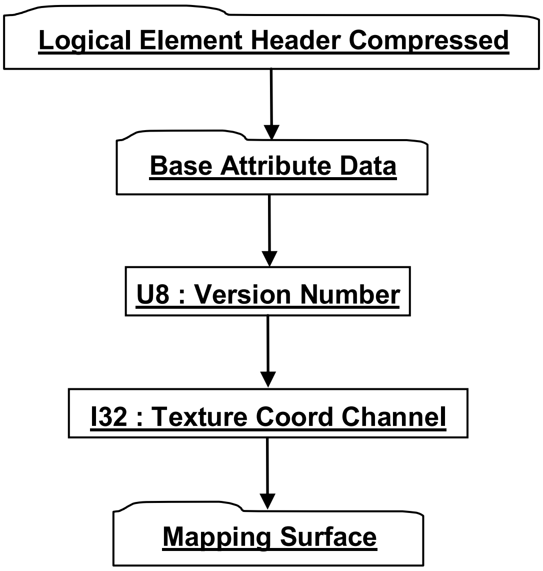

Logical Element Header Compressed
Base Attribute Data
U8 : Version Number
I32 : Texture Coord Channel
Mapping Surface

Figure 65 — Texture Coordinate Generator Attribute Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Attribute Data can be found in 6.1.2.1.1 Base Attribute Data.

Complete description for Mapping Surface can be found in 6.1.2.11.1 Mapping Surface.

U8 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice 14.5 Local version numbers.

I32 : Texture Coord Channel

Tex Coord Channel specifies the channel number for texture coordinate generation. Value shall be within range [0, 2147483647] inclusive. This number is intended to match up with the I32 : Tex Coord Channel field on Texture Image Attribute Element in order to associate a specific Texture Coordinate Generator with a Specific Texture Image.

6.1.2.11.1 Mapping Surface

Mapping Surface defines the mapping surface for texture coordinate generation. Four kinds of mapping surfaces, Mapping Plane Element, Mapping Cylinder Element,

Mapping Sphere Element, and Mapping TriPlanar Element, are defined to support texture coordinate generation.

Mapping Plane Element

Object Type ID: 0xa3cfb921, 0xbdeb, 0x48d7, 0xb3, 0x96, 0x8b, 0x8d, 0xe, 0xf4, 0x85, 0xa0

Mapping Plane Element defines the mapping plane for texture coordinate generation.

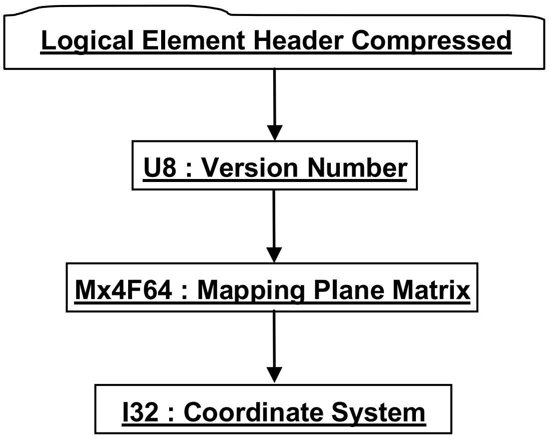

Logical Element Header Compressed
U8 : Version Number
Mx4F64 : Mapping Plane Matrix
I32 : Coordinate System

Figure 66 — Mapping Plane Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8 : Version Number

Version Number is the version identifier for this element. For informaiton on local version numbers see best practice 14.5 Local version numbers.

Mx4F64 : Mapping Plane Matrix

Mx4F64 : Mapping Plane Matrix specifies the transformation matrix and mapping parameters for the mapping plane. The transformation matrix defines the mapping coordinate system transformed from I32 : Coordinate System. The mapping parameters specifies the width and height of the mapping plane. The mapping plane is defined in the + xy-plane of the mapping coordinate system. In the mapping process, the geometry vertex coordinates in Model Coordinate System are transformed to the mapping coordinate system at first, and then the transformed vertex coordinates are mapped to texture coordinates as following:

s-coordinate = x-coordinate of the transformed vertex / the width of the mapping plane
t-coordinate = y-coordinate of the transformed vertex / the height of the mapping plane

I32 : Coordinate System

Coordinate system specifies the coordinate space in which mapping plane is defined. Valid values include the following

Table 44 — Mapping Plane Matrix Coordinate System values

= 0	Undefined Coordinate System.
= 1	Viewpoint Coordinate System. Mapping plane is to move together with the viewpoint.
= 2	Model Coordinate System. Mapping plane is affected by whatever model transforms that are current when the mapping plane is encountered in LSG.
= 3	World Coordinate system. Mapping plane is not affected by model transforms in the LSG.

Mapping Cylinder Element

Object Type ID: 0x3e70739d, 0x8cb0, 0x41ef, 0x84, 0x5c, 0xa1, 0x98, 0xd4, 0x0, 0x3b, 0x3f

Mapping Cylinder Element defines the mapping cylinder for texture coordinate generation.

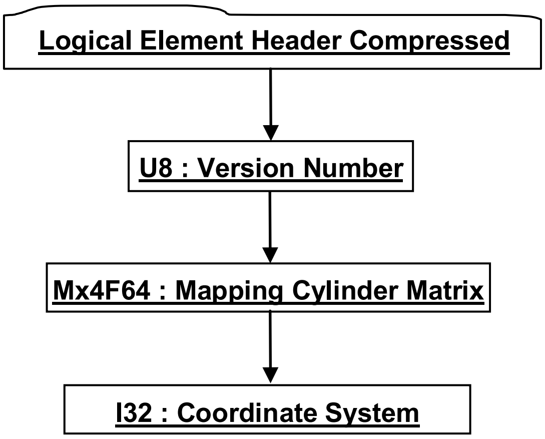

Logical Element Header Compressed
U8 : Version Number
Mx4F64 : Mapping Cylinder Matrix
I32 : Coordinate System

Figure 67 — Mapping Cylinder Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8 : Version Number

Version Number is the version identifier for this element. For informaiton on local version numbers see best practice 14.5 Local version numbers.

Mx4F64 : Mapping Cylinder Matrix

Mx4F64 : Mapping Cylinder Matrix specifies the transformation matrix and mapping parameters for the mapping cylinder. The transformation matrix defines the mapping coordinate system transformed from I32 : Coordinate System. The mapping parameters specifies the horizontal sweep angle and height of the mapping cylinder. The mapping cylinder's axis is parallel to the z-axis of the mapping coordinate system, and the horizontal sweep angle starts from the +x-axis in a counter clockwise direction. In the mapping process, the geometry vertex coordinates in Model Coordinate System are transformed to the mapping coordinate system at first, and then the transformed vertex coordinates are mapped to texture coordinates as following:

s-coordinate = the horizontal sweep angle of the vertex / the horizontal sweep angle of the mapping cylinder

t-coordinate = the z-coordinate of the vertex / height of the mapping cylinder

Mapping Cylinder Element implements the strategy to handle texture coordinates who cross the seam of the texture in the mapping process.

I32 : Coordinate System

Coordinate system specifies the coordinate space in which mapping cylinder is defined. Valid values include the following

Table 45 — Mapping Cylinder Matrix Coordinate System values

= 0	Undefined Coordinate System.
= 1	Viewpoint Coordinate System. Mapping cylinder is to move together with the viewpoint.
= 2	Model Coordinate System. Mapping cylinder is affected by whatever model transforms that are current when the mapping cylinder is encountered in LSG.
= 3	World Coordinate system. Mapping cylinder is not affected by model transforms in the LSG.

Mapping Sphere Element

Object Type ID: 0x72475fd1, 0x2823, 0x4219, 0xa0, 0x6c, 0xd9, 0xe6, 0xe3, 0x9a, 0x45, 0xc1

Mapping Sphere Element defines the mapping sphere for texture coordinate generation.

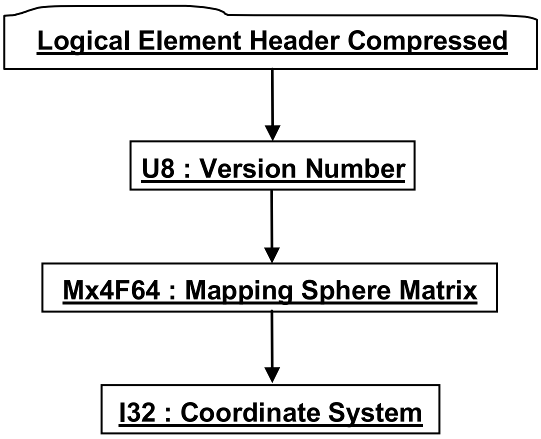

Logical Element Header Compressed
U8 : Version Number
Mx4F64 : Mapping Sphere Matrix
I32 : Coordinate System

Figure 68 — Mapping Sphere Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8 : Version Number

Version Number is the version identifier for this element. For informaiton on local version numbers see best practice 14.5 Local version numbers.

Mx4F64 : Mapping Sphere Matrix

Mx4F64 : Mapping Sphere Matrix specifies the transformation matrix and mapping parameters of the mapping sphere. The transformation matrix defines the mapping coordinate system transformed from I32 : Coordinate System. The mapping parameters specify the horizontal sweep angle and vertical sweep angle of the mapping sphere. The mapping sphere's centre is at the origin of the mapping coordinate system, and the poles of the sphere are parallel to the z-axis of the coordinate system. The horizontal sweep angle starts from the +x-axis in a counter clockwise direction, and the vertical sweep angle is from the +z-axis to the –z-axis. In the mapping process, the geometric vertex coordinates in Model Coordinate System are transformed to the mapping coordinate system at first, and then the transformed vertex coordinates are mapped to texture coordinates as following:

s-coordinate = the horizontal sweep angle of the vertex / the horizontal sweep angle of the mapping sphere

t-coordinate = the vertical sweep angle of the vertex / the vertical sweep angle of the mapping sphere

Mapping Sphere Element implements the strategy to handle texture coordinates who cross the seam of the texture in the mapping process.

I32 : Coordinate System

Coordinate system specifies the coordinate space in which mapping sphere is defined. Valid values include the following

Table 46 — Mapping Sphere Matrix Coordinate System values

= 0	Undefined Coordinate System.
= 1	Viewpoint Coordinate System. Mapping sphere is to move together with the viewpoint.
= 2	Model Coordinate System. Mapping sphere is affected by whatever model transforms that are current when the mapping sphere is encountered in LSG.
= 3	World Coordinate system. Mapping sphere is not affected by model transforms in the LSG.

Mapping TriPlanar Element

Object Type ID: 0x92f5b094, 0x6499, 0x4d2d, 0x92, 0xaa, 0x60, 0xd0, 0x5a, 0x44, 0x32, 0xcf

Mapping TriPlanar Element defines the mapping triplanar surface for texture coordinate generation.

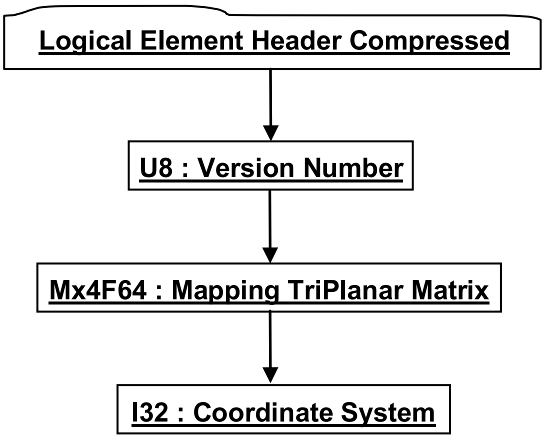

Logical Element Header Compressed
U8 : Version Number
Mx4F64 : Mapping TriPlanar Matrix
I32 : Coordinate System

Figure 69 — Mapping TriPlanar Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8 : Version Number

Version Number is the version identifier for this element. For informaiton on local version numbers see best practice 14.5 Local version numbers.

Mx4F64 : Mapping TriPlanar Matrix

Mx4F64 : Mapping TriPlanar Matrix specifies the transformation matrix and mapping parameter for the mapping triplanar. The transformation matrix defines the mapping coordinate system transformed from I32 : Coordinate System. The mapping parameter specifies the planar length of the triplanar. The left bottom corner of the triplanar is located at the origin of the mapping coordinate system, and the three planes are in the + xy-plane, + yz-plane, and + xz-plane respectively. In the mapping process, the geometry vertex coordinates in Model Coordinate System are transformed to the mapping coordinate system at first, and then the transformed vertex coordinates are projected to the corresponding plane based on the maximum component of its normals, and at last the projected vertex coordinates are mapped to texture coordinates as following:

s-coordinate = the first-coordinate of the projected vertex / the planar length of the triplanar

t-coordinate = the second-coordinate of the projected vertex / the planar length of the triplanar

I32 : Coordinate System

Coordinate system specifies the coordinate space in which mapping triplanar surface is defined. Valid values include the following

Table 47 — Mapping TriPlanar Matrix Coordinate System values

= 0	Undefined Coordinate System.
= 1	Viewpoint Coordinate System. Mapping triplanar surface is to move together with the viewpoint.
= 2	Model Coordinate System. Mapping triplanar surface is affected by whatever model transforms that are current when the mapping triplanar surface is encountered in LSG.
= 3	World Coordinate system. Mapping triplanar surface is not affected by model transforms in the LSG.

6.2 Property Atom Elements

Property Atom Elements are meta-data objects associated with nodes or Attributes. Property Atom Elements are not nodes or attributes themselves, but can be associated with any node or Attribute to maintain arbitrary application- or enterprise information (meta-data) pertaining to that node or Attribute. Each Node Element or Attribute Element in an LSG may hold zero or more Property Atom Elements and this relationship information is stored within 6.3 Property Table section of a JT file.

An individual property is specified as a *key/value* Property Atom Element pair, where the *key* identifies the type and meaning of the *value*. The JT format supports many different Property Atom Element key/value object types. The different Property Atom Element key/value object types are documented in the following subsections.

Some —Best Practices” for placing application or enterprise properties/meta-data on Nodes in JT files can be found in 14.8 Metadata Conventions section of this reference.

6.2.1 Base Property Atom Element

Object Type ID: 0x10dd104b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Base Property Atom Element represents the simplest form of a property that can exist within the LSG and has no type specific value data associated with it.

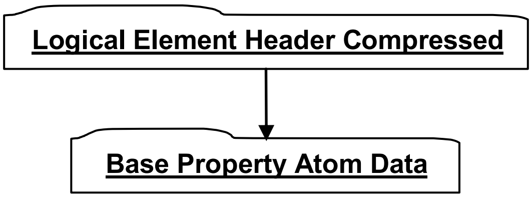


Figure 70 — Base Property Atom Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Base Property Atom Data

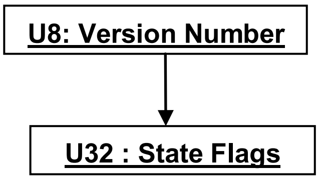


Figure 71 — Base Property Atom Data collection

U8: Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers

U32 : State Flags

State Flags is a collection of flags. The flags are combined using the binary OR operator and store various state information for property atoms. Bits 0 – 7 are freely available for an application to store whatever property atom information desired. All other bits are reserved for future expansion of the file format.

6.2.2 String Property Atom Element

Object Type ID: 0x10dd106e, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

String Property Atom Element represents a character string property atom.

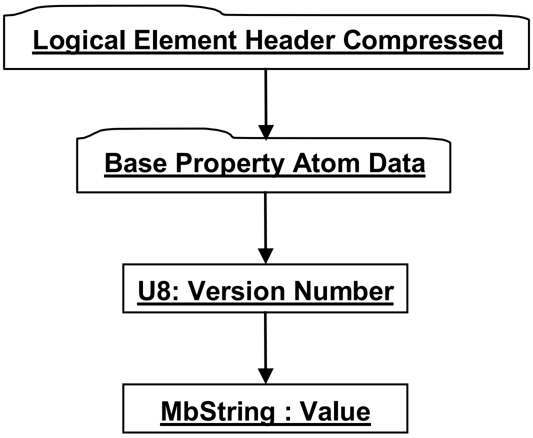


Figure 72 — String Property Atom Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Property Atom Data can be found in Base Property Atom Data.

U8: Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers.

MbString : Value

Value contains the character string value for this property atom.

6.2.3 Integer Property Atom Element

Object Type ID: 0x10dd102b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Integer Property Atom Element represents a property atom whose value is of I32 data type.

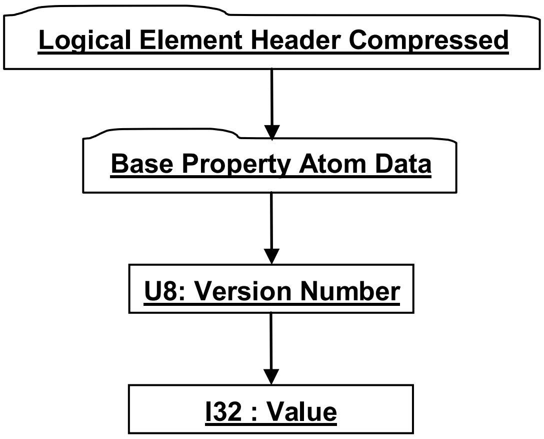


Figure 73 — Integer Property Atom Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Property Atom Data can be found in Base Property Atom Data.

U8: Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers.

I32 : Value

Value contains the integer value for this property atom.

6.2.4 Floating Point Property Atom Element

Object Type ID: 0x10dd1019, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Floating Point Property Atom Element represents a property atom whose value is of F32 data type.

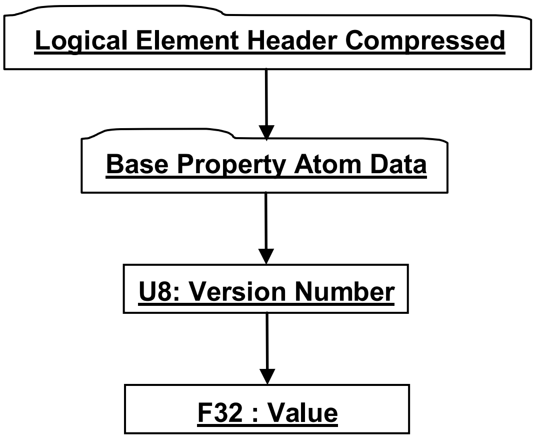


Figure 74 — Floating Point Property Atom Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Property Atom Data can be found in Base Property Atom Data.

U8: Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers.

F32 : Value

Value contains the floating point value for this property atom.

6.2.5 JT Object Reference Property Atom Element

Object Type ID: 0x10dd1004, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

JT Object Reference Property Atom Element represents a property atom whose value is an object ID for another object within the JT file.

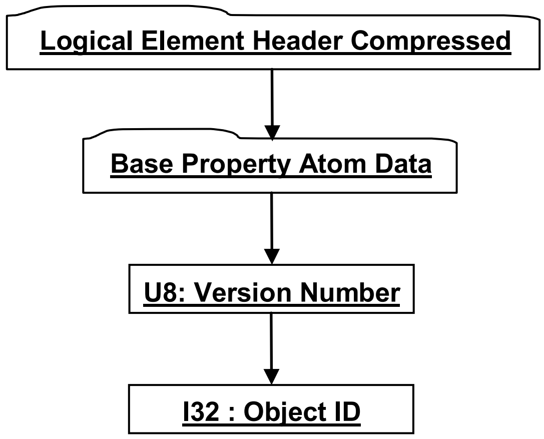


Figure 75 — JT Object Reference Property Atom Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Property Atom Data can be found in Base Property Atom Data.

U8: Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers.

I32 : Object ID

Object ID specifies the identifier within the JT file for the referenced object.

6.2.6 Date Property Atom Element

Object Type ID: 0xce357246, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1

Date Property Atom Element represents a property atom whose value is a ―date”.

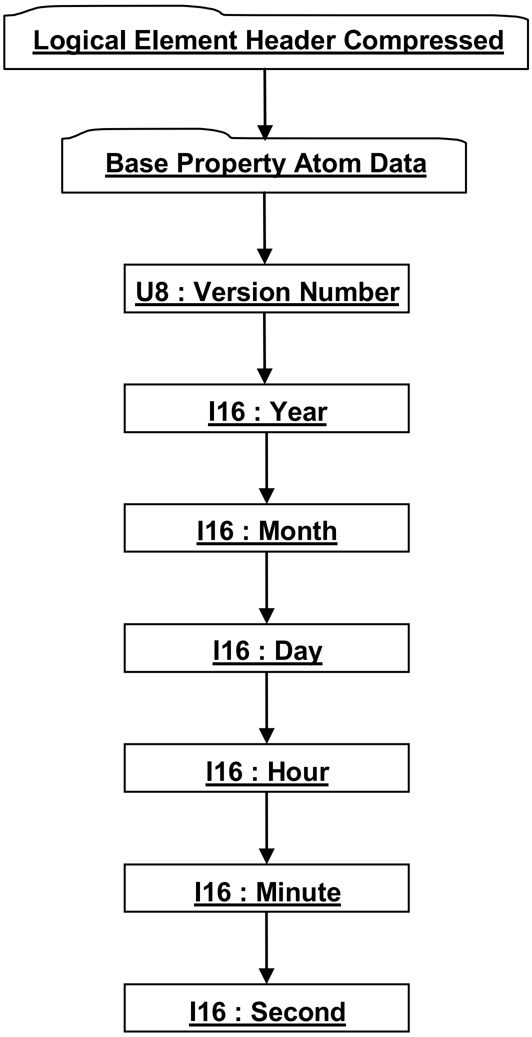


Figure 76 — Date Property Atom Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Property Atom Data can be found in Base Property Atom Data.

U8 : Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers.

I16 : Year

Year specifies the date year value. Valid values are [1900, 2999] inclusive.

I16 : Month

Month specifies the date month value. Valid values are [0, 11] inclusive.

I16 : Day

Day specifies the date day value. Valid values are [1, 31] inclusive.

I16 : Hour

Hour specifies the date hour value. Valid values are [0, 23] inclusive.

I16 : Minute

Minute specifies the date minute value. Valid values are [0, 59] inclusive.

I16 : Second

Second specifies the date Second value. Valid values are [0, 59] inclusive.

6.2.7 Late Loaded Property Atom Element

Object Type ID: 0xe0b05be5, 0xfbbd, 0x11d1, 0xa3, 0xa7, 0x00, 0xaa, 0x00, 0xd1, 0x09, 0x54

Late Loaded Property Atom Element is a property atom type used to reference an associated piece of atomic data in a separate addressable segment of the JT file. The ―Late Loaded" connotation derives from the associated data being stored in a separate addressable segment of the JT file, and thus a JT file reader can be structured to support the ―best practice" of delaying the loading/reading of the associated data until it is actually needed.

Late Loaded Property Atom Elements are used to store a variety of data, including, but not limited to, Shape LOD Segments and B-Rep Segments (see 6.2 Shape LOD Element

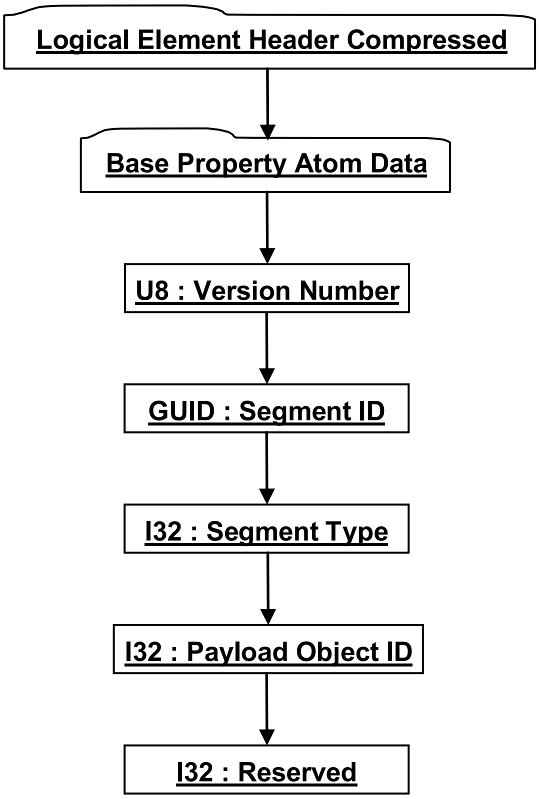


Figure 77 — Late Loaded Property Atom Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Property Atom Data can be found in Base Property Atom Data.

U8 : Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers.

GUID : Segment ID

Segment ID is the globally unique identifier for the associated data segment in the JT file. See 5.1.2 TOC Segment for additional information on how this Segment ID can be used in conjunction with the file TOC Entries to locate the associated data in the JT file.

The complete list of segment types can be found Table 8 — Segment Types.

I32 : Segment Type

Segment Type defines a broad classification of the associated data segment contents. For example, a Segment Type of ―1‖ denotes that the segment contains Logical Scene Graph material; ―2‖ denotes contents of a B-Rep, etc.

I32 : Payload Object ID

Object ID is the identifier for the payload. Other objects referencing this particular payload will do so using the Object ID.

I32 : Reserved

Reserved data field that is guaranteed to always be greater than or equal to 1

6.2.8 Vector4f Property Atom Element

Object Type ID: 0x2e7db4be, 0xc71a, 0x4b18, 0x9d, 0x7, 0xc7, 0x22, 0x7e, 0x9f, 0xef, 0x76

Vector4f Property Atom Element represents a property atom whose value is of VecF32 data type with the length to be equal to 4 .

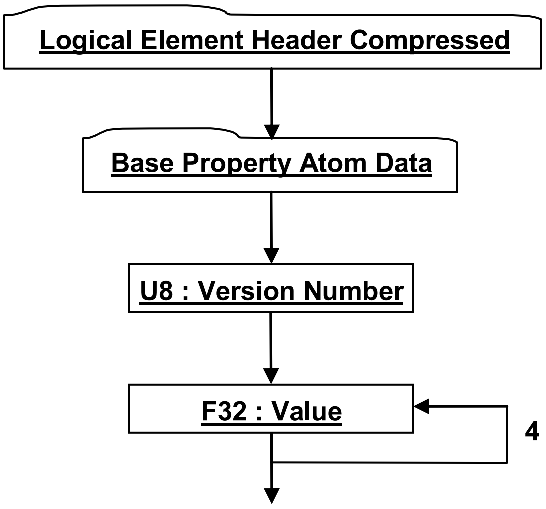


Figure 78 — Vector4f Property Atom Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

Complete description for Base Property Atom Data can be found in Base Property Atom Data.

U8 : Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers.

F32 : Value

Value contains the floating point value for this property atom.

6.3 Property Table

The Property Table is where the data connecting Node Elements and Attribute Elements with their associated Properties is stored. The Property Table contains an Element Property Table for each element in the JT File which has associated Properties. An Element Property Table is a list of key/value Property Atom Element pairs for all Properties associated with a particular Node Element Object or Attribute Element Object.

For a reference compliant JT File all Node Elements, Attribute Elements, and Property Atom Elements contained in a JT file should have been read by the time a JT file reader reaches the Property Table section of the file. This means that all Node Objects, Attribute Objects, and Property Atom Objects referenced in the Property Table (through Object IDs), should have already been read, and if not, then the file is corrupt (i.e. not reference compliant).

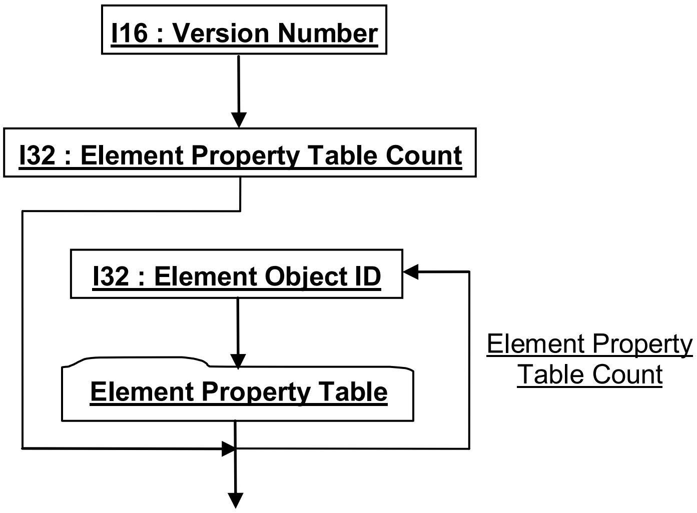


Figure 79 — Property Table data collection

I16 : Version Number

Version Number is the version identifier for this Property Table. For informaiton on local version numbers see best practice 14.5 Local version numbers.

I32 : Element Property Table Count

Element Property Table Count specifies the number of Element Property Tables to follow. This value is equivalent to the total number of Node Elements (see 6.1.1 Node Elements) and Attribute Elements (see 6.1.2 Attribute Elements) that have associated Property Atom Elements (see 6.2 Property Atom Elements).

I32 : Element Object ID

Element Object ID is the identifier for the Node Element object (see 6.1.1 Node Elements) or the Attribute Element object (see 6.1.2 Attribute Elements) that the following Element Property Table is for (i.e. Node Element or Attribute Element that all properties in the following Element Property Table are associated with).

6.3.1 Element Property Table

The Element Property Table is a list of key/value Property Atom Element pairs for all properties associated with a particular Node Element Object or Attribute Element Object. The list is terminated by a —0̓value for Key Property Atom Object ID.

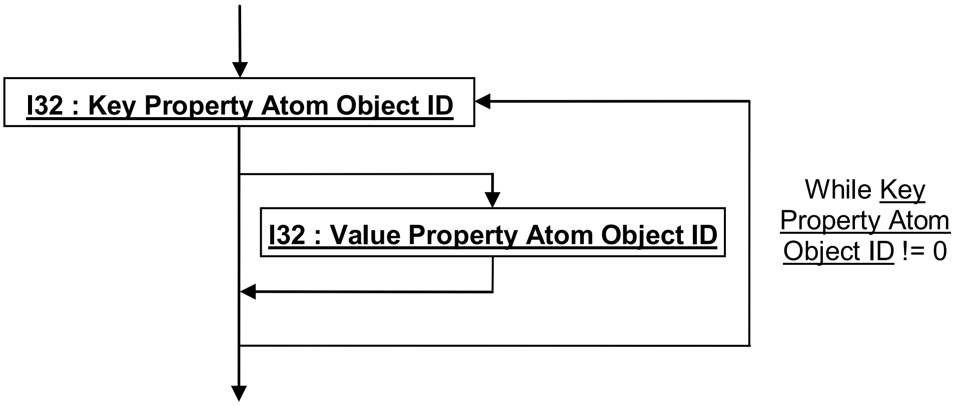

I32 : Key Property Atom Object ID
I32 : Value Property Atom Object ID
While Key Property Atom Object ID != 0

Figure 80 — Element Property Table data collection

I32 : Key Property Atom Object ID

Key Property Atom Object ID is the identifier for the Property Atom Element object (see Property Atom Elements) representing the ―key” part of the property key/value pair. A value of ―0‖indicates the end of the Node Property Table.

I32 : Value Property Atom Object ID

Value Property Atom Object ID is the identifier for the Property Atom Element object (see Property Atom Elements) representing the ―value” part of the property key/value pair. A value is not stored if I32 : Key Property Atom Object ID has a value of ―0‖

7 Shape LOD Segment

Shape LOD Segment contains an Element that defines the geometric shape definition data (e.g. vertices, polygons, normals, etc) for a particular shape Level Of Detail or alternative representation. Shape LOD Segments are typically referenced by Shape Node Elements using Late Loaded Property Atom Elements (see 6.1.1.10 Shape Node Elements and 6.2.7 Late Loaded Property Atom Element respectively).

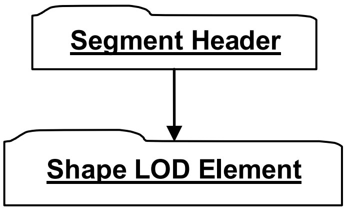


Figure 81 —Shape LOD Segmentdata collection

Complete description for Segment Header can be found in 5.1.3.1 Segment Header.

7.1 Shape LOD Element

A Shape LOD Element is the holder/container of the geometric shape definition data (e.g. vertices, polygons, normals, etc.) for a single LOD. Much of the ―heavyweight” data contained within a Shape LOD Element may be optionally compressed and/or encoded. The compression and/or encoding state is indicated through other data stored in each Shape LOD Element.

There are several types of Shape LOD Elements which the JT format supports. The following sub-sections document the various Shape LOD Element types.

7.1.1 Base Shape LOD Element

Object Type ID: 0x10dd10a4, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Base Shape LOD Element serves as the underlying representation for all LODs.

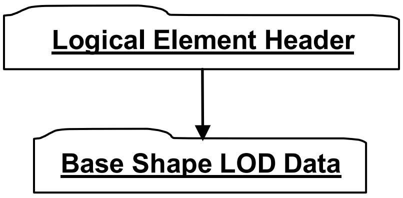


Figure 82 — Base Shape LOD Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

7.1.1.1 Base Shape LOD Data

Base shape LOD data contains the common items to all shape LODs.

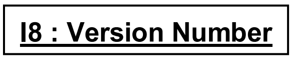


Figure 83 — Base Shape LOD Data collection

I8 : Version Number

Version Number is the version identifier for this Base Shape LOD Data. For informaiton on local version numbers see best practice 14.5 Local version numbers

7.1.2 Vertex Shape LOD Element

Object Type ID: 0x10dd10b0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Vertex Shape LOD Element represents LODs defined by collections of vertices.

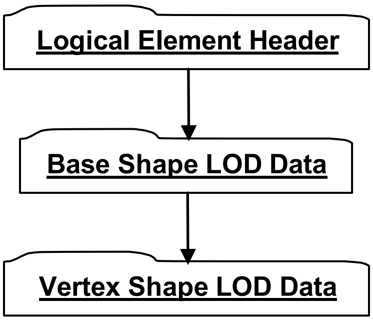


Figure 84 — Vertex Shape LOD Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

7.1.2.1 Vertex Shape LOD Data

Vertex Shape LOD Data collection is an abstract container for geometric *primitives* such as triangles, line strips, or points, depending on the specific type of Vertex Shape. The set of primitives are further partitioned into so-called "face groups." The Vertex Shape LOD Data also contains the vertex attribute bindings and quantization settings used to store the vertex records referenced by the primitives.

One use for face groups is to establish a correspondence between Brep faces and their triangle representation. A convention for mapping JTBrep and XTBrep faces to face groups is described in section 14.12 Brep Face Group Associations.

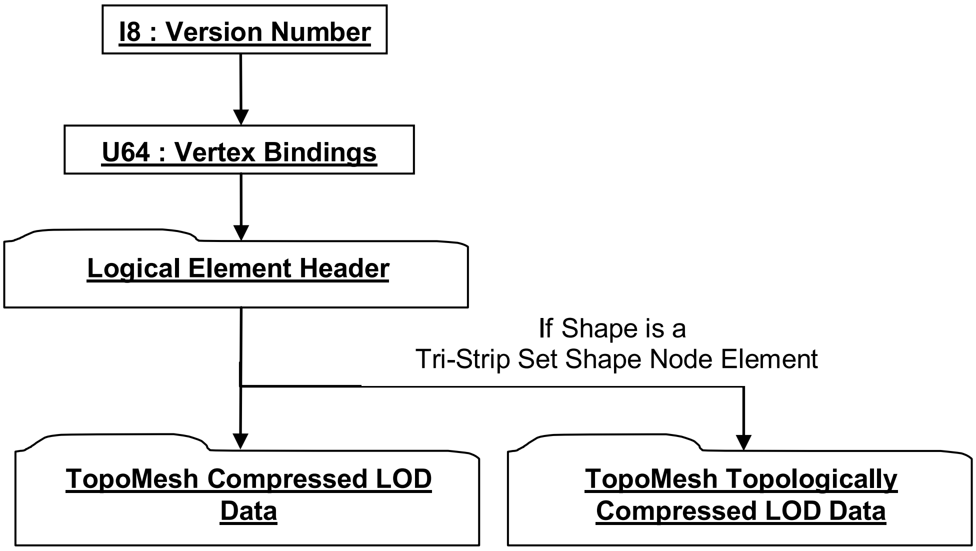

I8 : Version Number
U64 : Vertex Bindings
Logical Element Header
If Shape is a
Tri-Strip Set Shape Node Element
TopoMesh Compressed LOD Data
TopoMesh Topologically Compressed LOD Data

Figure 85 — Vertex Shape LOD Data collection

Complete description for TopoMesh Compressed LOD Data and TopoMesh Topologically Compressed LOD Data can be found in TopoMesh Compressed LOD Data and TopoMesh Topologically Compressed LOD Data.

I8 : Version Number

Version Number is the version identifier for this Vertex Shape LOD Data. For informaiton on local version numbers see best practice 14.5 Local version numbers

U64 : Vertex Bindings

Binding Attributes is a collection of normal, texture coordinate, and colour binding information encoded within a single U64 using the following bit allocation. All bits fields that are not defined as in use should be set to —’0.

Table 48 — Vertex Shape LOD Bindings values

Bits 1-3	Vertex Coordinate Binding. The Vertex Coordinate Binding denotes per vertex coordinate field data is present when one of the bits is set. Bit 1 - 2 Component Vertex Coordinates Bit 2 - 3 Component Vertex Coordinates Bit 3 - 4 Component Vertex Coordinates
Bit 4	Normal Binding. The Normal Binding denotes per vertex normal field data is present when the bit is set. Normal field data is always stored in 3 Component Normals when present.
Bits 5 -6	Colour Binding. The Colour Binding denotes per vertex colour field data is present when one of the bits is set. Bit 5 - 3 Component Colours Bit 6 - 4 Component Colour
Bit 7	Vertex Flag Binding. The Vertex Flag Binding denotes the per vertex flag field is present on the shape when the bit is set.
Bits 9-12	Texture Coordinate 0 Binding. The Texture Coordinate 0 binding denotes per vertex texture coordinates field data is present when one of the bits is set: Bit 9 - 1 Component Texture Coordinates Bit 10 - 2 Component Texture Coordinates Bit 11 - 3 Component Texture Coordinates Bit 12 - 4 Component Texture Coordinates
Bits 13-16	Texture Coordinate 1 Binding. The Texture Coordinate 1 binding denotes per vertex texture coordinates field data is present when one of the bits is set:

	Bit 13 - 1 Component Texture Coordinates Bit 14 - 2 Component Texture Coordinates Bit 15 - 3 Component Texture Coordinates Bit 16 - 4 Component Texture Coordinates
Bits 17-20	Texture Coordinate 2 Binding. The Texture Coordinate 2 binding denotes per vertex texture coordinates field data is present when one of the bits is set: Bit 17 - 1 Component Texture Coordinates Bit 18 - 2 Component Texture Coordinates Bit 19 - 3 Component Texture Coordinates Bit 20 - 4 Component Texture Coordinates
Bits 21-24	Texture Coordinate 3 Binding. The Texture Coordinate 3 binding denotes per vertex texture coordinates field data is present when one of the bits is set: Bit 21 - 1 Component Texture Coordinates Bit 22 - 2 Component Texture Coordinates Bit 23 - 3 Component Texture Coordinates Bit 24 - 4 Component Texture Coordinates
Bits 25-28	Texture Coordinate 4 Binding. The Texture Coordinate 4 binding denotes per vertex texture coordinates field data is present when one of the bits is set: Bit 25 - 1 Component Texture Coordinates Bit 26 - 2 Component Texture Coordinates Bit 27 - 3 Component Texture Coordinates Bit 28 - 4 Component Texture Coordinates
Bits 29-32	Texture Coordinate 5 Binding. The Texture Coordinate 5 binding denotes per vertex texture coordinates field data is present when one of the bits is set: Bit 29 - 1 Component Texture Coordinates Bit 30 - 2 Component Texture Coordinates Bit 31 - 3 Component Texture Coordinates Bit 32 - 4 Component Texture Coordinates
Bits 33-36	Texture Coordinate 6 Binding. The Texture Coordinate 6 binding denotes per vertex texture coordinates field data is present when one of the bits is set: Bit 33 - 1 Component Texture Coordinates Bit 34 - 2 Component Texture Coordinates Bit 35 - 3 Component Texture Coordinates Bit 36 - 4 Component Texture Coordinates
Bits 37-40	Texture Coordinate 7 Binding. The Texture Coordinate 7 binding denotes per vertex texture coordinates field data is present when one of the bits is set: Bit 37 - 1 Component Texture Coordinates Bit 38 - 2 Component Texture Coordinates Bit 39 - 3 Component Texture Coordinates Bit 40 - 4 Component Texture Coordinates
Bits 41-62	Unused
Bit 63	
Bit 64	Auxiliary Vertex Field Binding. The Auxiliary Vertex Field Binding denotes per vertex auxiliary field data is present on the shape when the bit is set.

7.1.2.2 TopoMesh LOD Data

TopoMesh LOD Data collection contains the common items to all TopoMesh LOD elements.

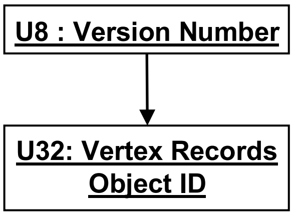


Figure 86 — TopoMesh LOD Data collection

U8 : Version Number

Version Number is the version identifier for this TopoMesh LOD Data. For information on local version numbers see best practice 14.5 Local version numbers

U32: Vertex Records Object ID

Vertex Records Object ID is the identifier for the vertex records associated with this Object. Other objects referencing these vertex records will do so using this Object ID. It is via this mechanism that multiple TopMeshes are able to reference the same set of vertex records.

7.1.2.2.1 TopoMesh Compressed LOD Data

TopoMesh Compressed LOD Data collection contains the common items to all TopoMesh Compressed LOD data elements.

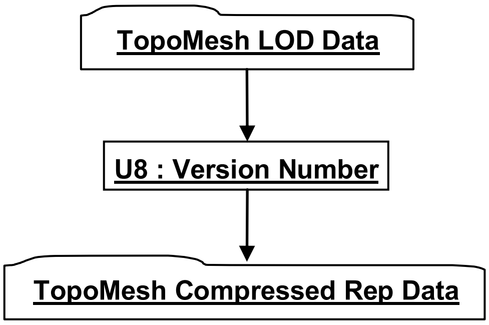


Figure 87 — TopoMesh Compressed LOD Data collection

Complete description for TopoMesh LOD Data, and TopoMesh Compressed Rep Data, can be found in TopoMesh LOD Data, TopoMesh Compressed Rep Data .

U8 : Version Number

Version Number is the version identifier for this TopoMesh Compressed LOD Data. For informaiton on local version numbers see best practice 14.5 Local version numbers

7.1.2.2.2 TopoMesh Topologically Compressed LOD Data

TopoMesh Topologically Compressed LOD Data collection contains the common items to all TopoMesh Topologically Compressed LOD data elements such as Tri-Strip Set Shape LOD Element.

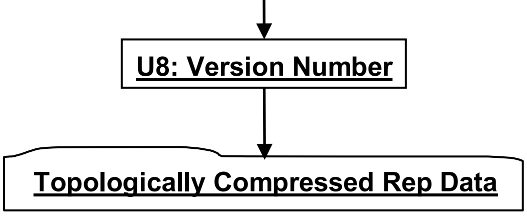


Figure 88 — TopoMesh Topologically Compressed LOD Data collection

Complete description for TopoMesh LOD Data and Topologically Compressed Rep Data can be found in TopoMesh LOD Data and Topologically Compressed Rep Data.

U8: Version Number

Version Number is the version identifier for this TopoMesh Topologically Compressed LOD Data. For informaiton on local version numbers see best practice 14.5 Local version numbers

7.1.2.2.3 Topologically Compressed Rep Data

JT v10, like JT v9, represents triangle strip data very differently than it does in the JT v8 format. The scheme stores the triangles from a TriStripSet or polygons from a PolygonSet as a topologically-connected mesh. Even though *more* information is stored to the JT file, the additional structure provided by storing the full topological adjacency information actually provides a handsome reduction in the number of bytes needed to encode the triangles or polygons. More importantly, however, the topological information aids us in a more significant respect -- that of only storing the *unique* vertex records used by the TriStripSet or PolygonSet. Combined, these two effects reduce the typical storage footprint of TriStripSet data by approximately half relative to the JT v8 format.

The tristrip information itself is not stored in the JT file -- only the triangles themselves. The reader is expected to re-tristrip (or not) as she sees fit, as tristrips may no longer provide a performance advantage during rendering. There may, however, remain some memory savings for tristripping, and so the decision to tristrip is left to the user.

To begin the decoding process, first read the compressed data fields shown in Figure 89. These fields provide all the information necessary to reconstruct the per face-group organized sets of triangles. The first 22 fields represent the topological information, and the remaining fields constitute the set of unique vertex records to be used. The next step is to run the topological decoder algorithm detailed in Annex D on this data to reconstruct the topologically connected representation of the triangle mesh in a so-called "Dual VFMesh." The triangles or polygons in this heavy-weight data structure can then be exported to a lighter-weight form, and the dual VFMesh discarded if desired.

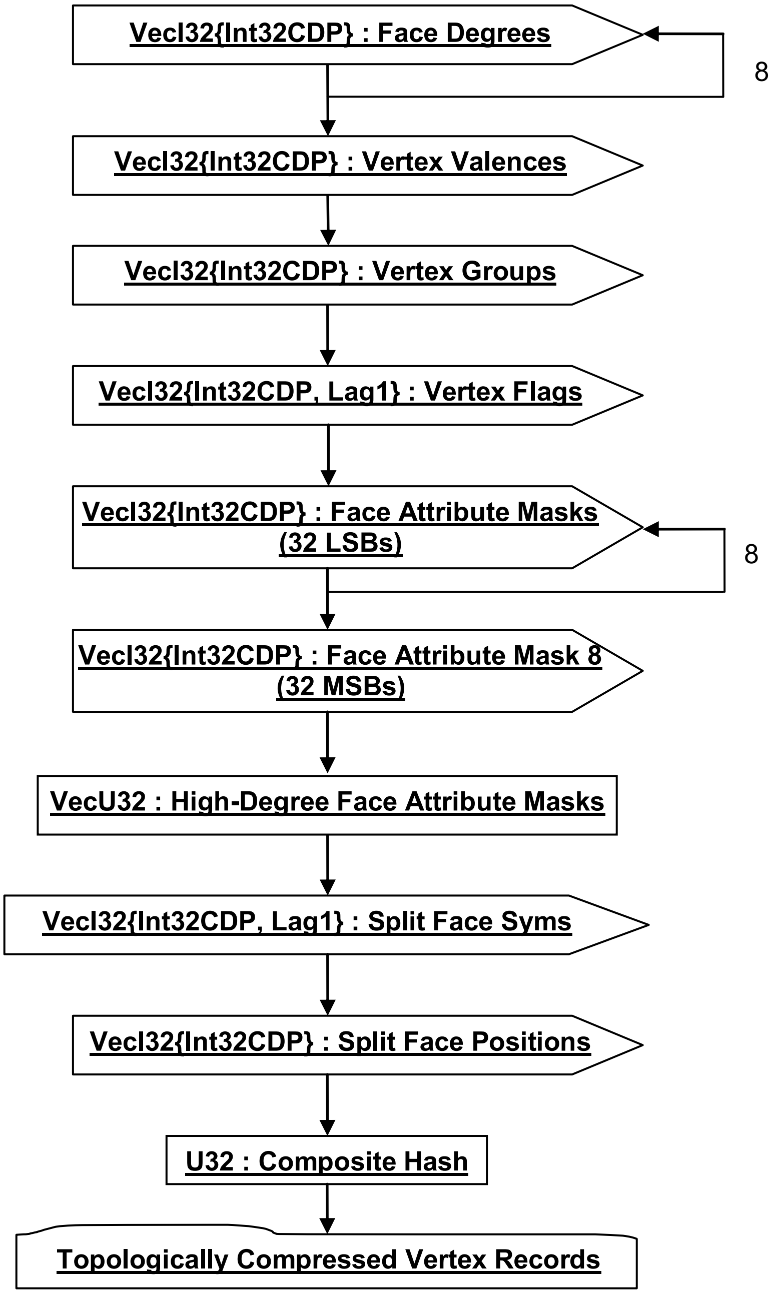

VecI32{Int32CDP} : Face Degrees
8
VecI32{Int32CDP} : Vertex Valences
VecI32{Int32CDP} : Vertex Groups
VecI32{Int32CDP, Lag1} : Vertex Flags
VecI32{Int32CDP} : Face Attribute Masks (32 LSBs)
8
VecI32{Int32CDP} : Face Attribute Mask 8 (32 MSBs)
VecU32 : High-Degree Face Attribute Masks
VecI32{Int32CDP, Lag1} : Split Face Syms
VecI32{Int32CDP} : Split Face Positions
U32 : Composite Hash
Topologically Compressed Vertex Records

Figure 89 — Topologically Compressed Rep Data Collection

VecI32{Int32CDP} : Face Degrees

Similarly to the way valences are encoded, the topology encoder emits the *degree* (number of incident vertices) of each face *in the order they were visited*. The number of face degrees in this array is equal to the number of faces in the mesh.

VecI32{Int32CDP} : Vertex Valences

As the coder visits each vertex in the mesh, it emits the *valence* (number of incident faces) of each vertex. These valences are collect *in the order they were visited* into this array. The number of valences in this array is equal to the number of (topological) vertices in the mesh.

VecI32{Int32CDP} : Vertex Groups

This array is parallel to the Vertex Valences array above. As the coder emits the valence of each vertex, it also emits the face group number to which the dual vertex belongs into this array.

VecI32{Int32CDP, Lag1} : Vertex Flags

This array is also parallel to the Vertex Valences array, and contains a value of 0 when the dual face was present in the original triangle mesh, and a value of 1 if the dual face is a *cover face* that was added to artificially close the original mesh.

VecI32{Int32CDP} : Face Attribute Masks (32 LSBs)

This field is written 8 times – once for each of the 8 context groups listed above – and encodes the face attribute bit vector associated with a single face.

VecI32{Int32CDP} : Face Attribute Mask 8 (32 MSBs)

This field encodes the 32 most significant bits of the 8th context group of face attribute bit vectors.

VecU32 : High-Degree Face Attribute Masks

This field encodes all remaining face attribute bit vectors, adjoined end-to-end, and encoded as a single array of unsigned integers.

VecI32{Int32CDP, Lag1} : Split Face Syms

Encodes the list of ―split face" ID numbers in the order the coder encountered them.

VecI32{Int32CDP} : Split Face Positions

Encodes the list of ―split face" positions in the active vertex queue in the order the code encountered them.

U32 : Composite Hash

This field is a hash value computed on all of the above data using the hash function described in Annex C. It is written into the JT file so that a reader can perform the same hash on the above data and compare against this value in order to guarantee that it has read and decoded correct data from the JT file. It is *highly* encouraged that all readers perform this check, as even a single bit error in the topology information above can have catastrophic consequences on the topology decoder and the resulting mesh. Any writers are *required* to write this field using the method provided so that other readers may validate the data they read.

```
UInt32 uHash        = 0;
UInt32 anDegSyms[8] = {0},
       nValSyms = 0,
       nVGrpSyms = 0,
       nVtxFlags = 0,
       anAttrMasks[8] = {0},
       nLrgAttrMasks = 0,
       nSplitVtxSyms = 0,
       nSplitVtxPos = 0;
VecI32 vFaceDegreeSymbols[8], vviValenceSymbols, vFaceGroupSyms,
       vvuAttrMasks[8], viSplitVtxSyms, viSplitVtxPos;
VecI16 vFaceFlags;
VecU32 vuTmp, vuAttrMasksLrg;
...
for (i=0 ; i<8 ;i++)
  uHash = hash32((UInt32*) vFaceDegreeSymbols[i].ptr(), anDegSyms[i], uHash );
uHash = hash32((UInt32*) vviValenceSymbols.ptr(), nValSyms, uHash );
uHash = hash32((UInt32*)vVtxGroupSyms.ptr(), nVGrpSyms, uHash );
uHash = hash16((UInt16*)vVtxFlags.ptr(), nFlags, uHash );
for (i=0 ; i<7 ;i++)
  uHash = hash32((UInt32*)vvuAttrMasks[i].ptr(), anAttrMasks[i], uHash );
vuTmp = vvuAttrMasks[7] & 0xffffffff;  // Lower 32 bits of each element
uHash = hash32(vuTmp.ptr(), anAttrMasks[7], uHash );
vuTmp = (vvuAttrMasks[7] >> 32) & 0xffffffff; // Next 32 bits of each element
uHash = hash32(vuTmp.ptr(), anAttrMasks[7], uHash );
uHash = hash32(vuAttrMasksLrg.ptr(), nLrgAttrMasks, uHash );
uHash = hash32((UInt32*)viSplitVtxSyms.ptr(), nSplitVtxSyms, uHash );
uHash = hash32((UInt32*)viSplitVtxPos.ptr(), nSplitVtxPos, uHash );
```

7.1.2.3 Topologically Compressed Vertex Records

Documented here is the format of the vertex data written by the topological encoder from Annex E. Some additional explanation is necessary, however, because only the *unique* vertex coordinates are written to the

JT file, while the remaining vertex attributes (normals, colours, texture coordinates, vertex flags) may not be unique.

Vertex coordinates are written to the file in the order that they were visited by the topology encoder. Note that this means that the number of vertex coordinates written is equal to the number of topological vertices in the mesh (i.e. all vertex coordinates are unique).

By contrast one set of vertex attribute records is written to the file corresponding to each 1 bit across all encoded dual Face Attribute Masks. The vertex attribute records are written in the order that the topology encoder visited them. The reader shall then use the topology decoder's output to correctly associate each vertex attribute record to the correct vertex coordinate using the dual Face Attribute Masks.

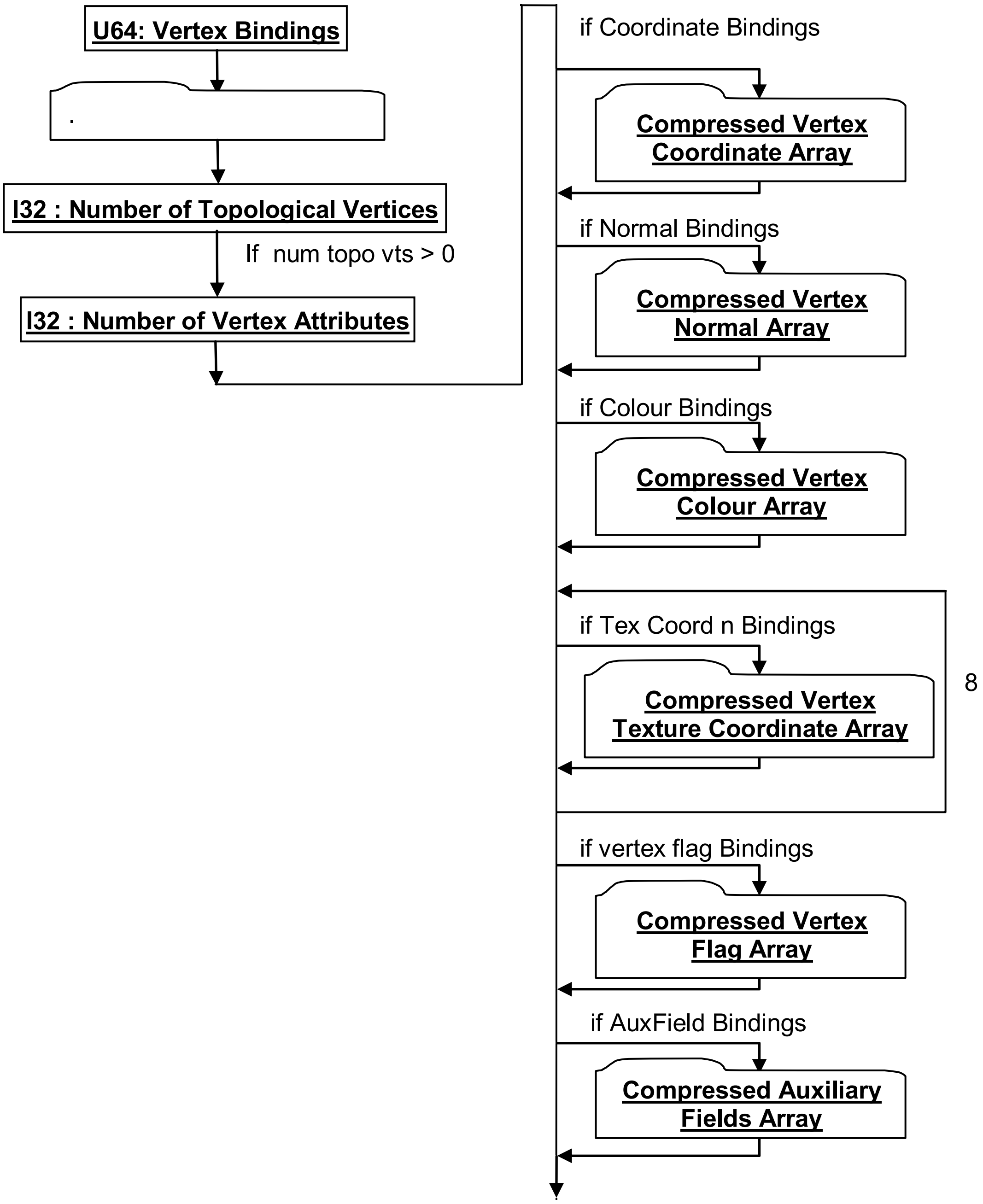

U64: Vertex Bindings
.
I32 : Number of Topological Vertices
If num topo vts > 0
I32 : Number of Vertex Attributes
if Coordinate Bindings
Compressed Vertex Coordinate Array
if Normal Bindings
Compressed Vertex Normal Array
if Colour Bindings
Compressed Vertex Colour Array
if Tex Coord n Bindings
Compressed Vertex Texture Coordinate Array
8
if vertex flag Bindings
Compressed Vertex Flag Array
if AuxField Bindings
Compressed Auxiliary Fields Array

Figure 90 — Topologically Compressed Vertex Records data collection

U64: Vertex Bindings

Vertex Bindings is a collection of normal, texture coordinate, and colour binding information encoded within a single U64. All bits fields that are not defined as in use should be set to ―0̓. For more information see Vertex Shape LOD Data U64 : Vertex Bindings.

I32 : Number of Topological Vertices

This field is the number of topological vertices encoded by the topology encoder. This is the number of unique vertex coordinates that will be written in the later Compressed Vertex Coordinate Array field.

I32 : Number of Vertex Attributes

One set of vertex attribute records is written to the file corresponding to each 1 bit across all encoded dual Face Attribute Masks. The vertex attribute records are written in the order that the topology encoder visited them. The reader shall then use the topology decoder's output to correctly associate each vertex attribute record to the correct vertex coordinate using the dual Face Attribute Masks.

7.1.2.4 TopoMesh Compressed Rep Data

TopoMesh Compressed Rep Data contains the geometric shape definition data (e.g. vertices, colours, normals, etc.) in a lossy or lossless compressed format. This format is used when the shape type is Polyline Set Shape Node Element, or Point Set Shape Node Element. For Tri-Strip Set Shape Node Element and Polygon Set Shape Node Element, please refer to 7.1.2.2.3 Topologically Compressed Rep Data.

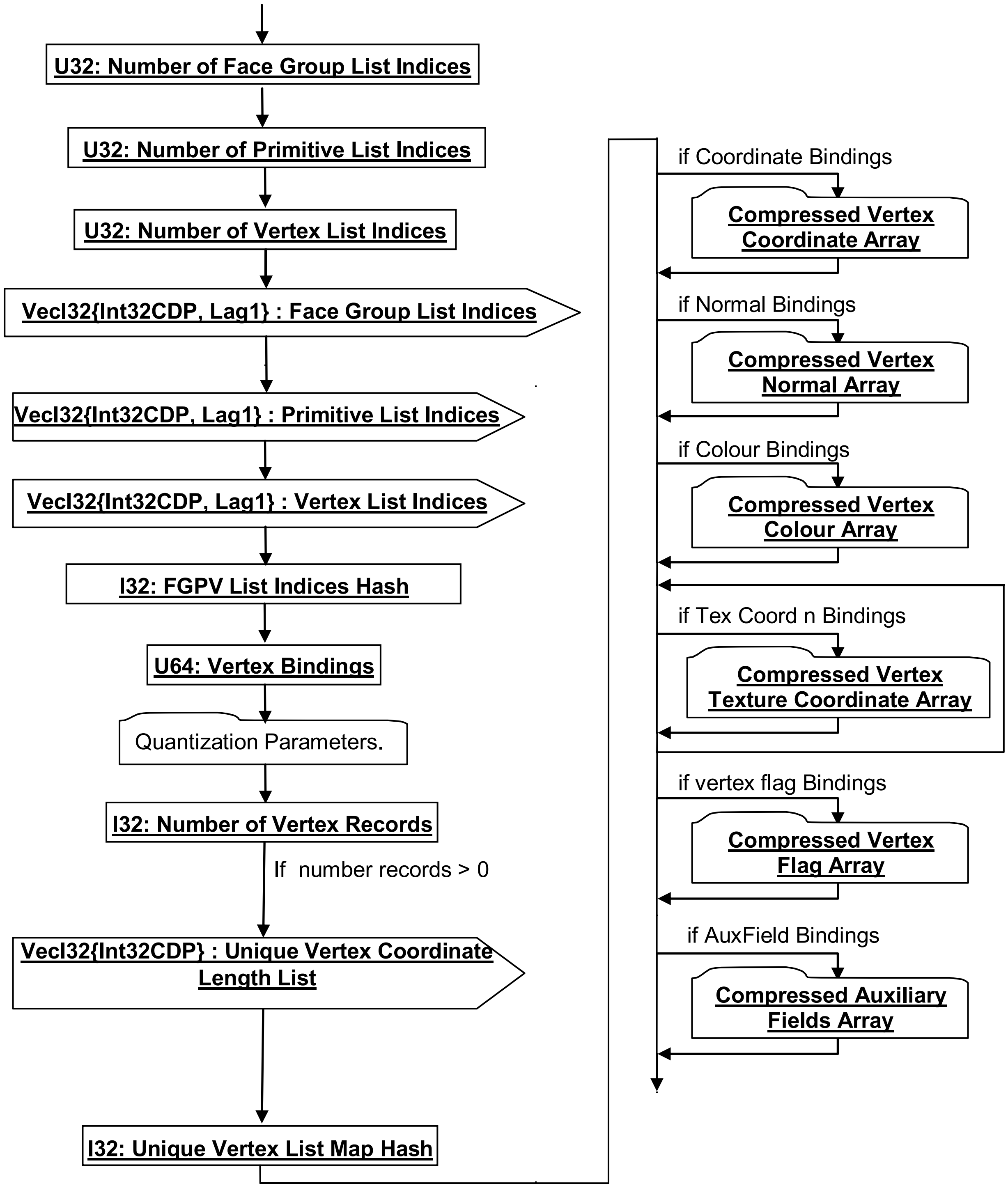

U32: Number of Face Group List Indices
U32: Number of Primitive List Indices
U32: Number of Vertex List Indices
VecI32{Int32CDP, Lag1} : Face Group List Indices
VecI32{Int32CDP, Lag1} : Primitive List Indices
VecI32{Int32CDP, Lag1} : Vertex List Indices
I32: FGPV List Indices Hash
U64: Vertex Bindings
Quantization Parameters.
I32: Number of Vertex Records
If number records > 0
VecI32{Int32CDP} : Unique Vertex Coordinate Length List
I32: Unique Vertex List Map Hash
if Coordinate Bindings
Compressed Vertex Coordinate Array
if Normal Bindings
Compressed Vertex Normal Array
if Colour Bindings
Compressed Vertex Colour Array
if Tex Coord n Bindings
Compressed Vertex Texture Coordinate Array
8
if vertex flag Bindings
Compressed Vertex Flag Array
if AuxField Bindings
Compressed Auxiliary Fields Array

Figure 91 — TopoMesh Compressed Rep Data data collection

Complete description for Quantization Parameters can be found in Quantization Parameters.

U32: Number of Face Group List Indices

Number of Face Group List Indices.

U32: Number of Primitive List Indices

Number of Primitive List Indices.

U32: Number of Vertex List Indices

Number of Vertex List Indices.

VecI32{Int32CDP, Lag1} : Face Group List Indices

Face Group List Indices is a vector of indices into the uncompressed Raw Primitive Data marking the start/beginning of Faces. Face Group List Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Primitive List Indices

Primitive List Indices is a vector of indices into the uncompressed Raw Vertex Data marking the start/beginning of primitives. Primitive List Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Vertex List Indices

Vertex List Indices is a vector of indices (one per vertex) into the uncompressed/dequantized unique vertex data arrays (Vertex Coords, Vertex Normals, Vertex Texture Coords, Vertex Colours) identifying each Vertex's data (i.e. for each Vertex there is an index identifying the location within the unique arrays of the particular Vertex's data). The Compressed Vertex Index List uses the Int32 version of the CODEC to compress and encode data.

I32: FGPV List Indices Hash

The FGPV Hash is the combined hash value of the Face Group List Indices (if Polyline), Primitive List Indices, and Vertex List Indices. Refer to section 14.6 for a more detailed description on hashing.

```
UInt32 uHash    = 0;
UInt32 nFGIdx   = 0,
       nPrimIdx = 0,
       nVtxIdx  = 0;
vecI32 vFGIndices, vPrimIdices, vVertexIndices;
...
uHash = hash32( (UInt32*)(& vFGIndices), nFGIdx+1, uHash );
uHash = hash32( (UInt32*)(& vPrimIdices), nPrimIdx+1, uHash );
uHash = hash32( (UInt32*)(& vVertexIndices), nVtxIdx  , uHash );
```

U64: Vertex Bindings

Vertex Bindings is a collection of normal, texture coordinate, and colour binding information encoded within a single U64. All bits fields that are not defined as in use should be set to —0 For more information see Vertex Shape LOD Data U64 : Vertex Bindings.

I32: Number of Vertex Records

Number of vertex records.

VecI32{Int32CDP} : Unique Vertex Coordinate Length List

The Unique Vertex Length List contains the number of vertex records containing each of the unique vertex coordinates and should sum to the number of vertex records. When read in the Compressed Vertex Coordinate Array only contains a single value for each unique vertex coordinate value and is therefore parallel to the Unique Vertex Length List. In order to expand its coordinates into the vertex record space it unique coordinate value will need to be smeared out such that each unique vertex coordinate is repeated the number of times specified in the Unique Vertex Length List. The Compressed Vertex Normal, Colour, Texture, and Flag Arrays do not require the same expansion.

I32: Unique Vertex List Map Hash

The Unique Vertex List Map Hash is the hash value of Unique Vertex Coordinate Length List. Refer to section 14.6 for a more detailed description on hashing.

```
UInt32 uHash    = 0;
UInt32 nUniqVtx = 0;
vecF32 vUniqVtxIndices;
...
uHash = hash32( (UInt32*)(&vUniqVtxIndices), nUniqVtx, uHash );
```

Quantization Parameters

Quantization Parameters specifies for each shape data type grouping (i.e. Vertex, Normal, Texture Coordinates, Colour) the number of quantization bits used for given qualitative compression level. Although these Quantization Parameters values are saved in the associated/referenced Shape LOD Element, they are also saved here so that a JT File loader/reader does not have to load the Shape LOD Element in order to determine the Shape quantization level. See 7.1 Shape LOD Element

for complete description of Shape LOD Elements.

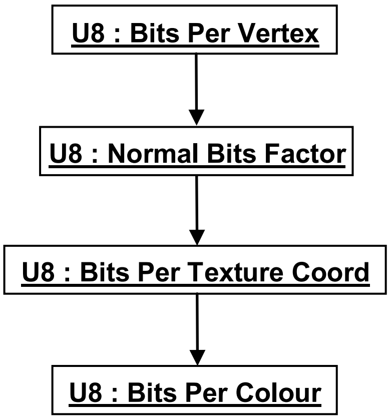


Figure 92 — Quantization Parameters data collection

U8 : Bits Per Vertex

Bits Per Vertex specifies the number of quantization bits per vertex coordinate component. Value shall be within range [0:24] inclusive.

U8 : Normal Bits Factor

Normal Bits Factor is a parameter used to calculate the number of quantization bits for normal vectors. Value shall be within range [0:13] inclusive. The actual number of quantization bits per normal is computed using this factor and the following formula: –BitsPerNormal = 6 + 2 * Normal Bits Factor”

U8 : Bits Per Texture Coord

Bits Per Texture Coord specifies the number of quantization bits per texture coordinate component. Value shall be within range [0:24] inclusive.

U8 : Bits Per Colour

Bits Per Colour specifies the number of quantization bits per colour component. Value shall be within range [0:24] inclusive.

7.1.3 Tri-Strip Set Shape LOD Element

Object Type ID: 0x10dd10ab, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

A Tri-Strip Set Shape LOD Element contains the geometric shape definition data (e.g. vertices, polygons, normals, etc.) for a single LOD of a collection of independent and unconnected triangle strips. Each strip constitutes one primitive of the set and the ordering of the vertices in forming triangles, is the same as OpenGL‘s triangle strip definition [1].

A Tri-Strip Set Shape LOD Element is typically referenced by a Tri-Strip Set Shape Node Element using Late Loaded Property Atom Elements (see 6.1.1.10.3 Tri-Strip Set Shape Node Element and 6.2.7 Late Loaded Property Atom Element respectively).

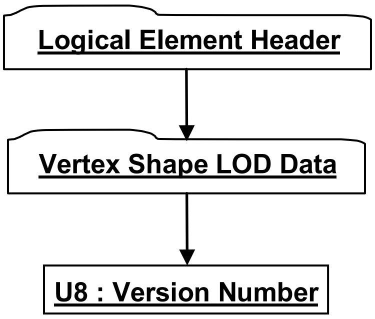


Figure 93 — Tri-Strip Set Shape LOD Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

Complete description for Vertex Shape LOD Data can be found in Vertex Shape LOD Data.

U8 : Version Number

Version Number is the version identifier for this Tri-Strip Set Shape LOD. For informaiton on local version numbers see best practice 14.5 Local version numbers

7.1.4 Polyline Set Shape LOD Element

Object Type ID: 0x10dd10a1, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

A Polyline Set Shape LOD Element contains the geometric shape definition data (e.g. vertices, normals, etc.) for a single LOD of a collection of independent and unconnected polylines. Each polyline constitutes one primitive of the set.

A Polyline Set Shape LOD Element is typically referenced by a Polyline Set Shape Node Element using Late Loaded Property Atom Elements (see 6.1.1.10.5 Polyline Set Shape Node Element and 6.2.7 Late Loaded Property Atom Element respectively).

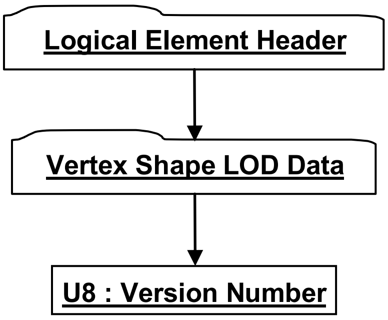


Figure 94 — Polyline Set Shape LOD Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

Complete description for Vertex Shape LOD Data can be found in Vertex Shape LOD Data.

U8 : Version Number

Version Number is the version identifier for this Polyline Set Shape LOD. For informaiton on local version numbers see best practice 14.5 Local version numbers

7.1.5 Point Set Shape LOD Element

Object Type ID: 0x98134716, 0x0011, 0x0818, 0x19, 0x98, 0x08, 0x00, 0x09, 0x83, 0x5d, 0x5a

A Point Set Shape LOD Element contains the geometric shape definition data (e.g. coordinates, normals, etc.) for a collection of independent and unconnected points. Each point constitutes one primitive of the set.

A Point Set Shape LOD Element is typically referenced by a Point Set Shape Node Element using Late Loaded Property Atom Elements (see 6.1.1.10.5 Point Set Shape Node Element and 6.2.7 Late Loaded Property Atom Element respectively).

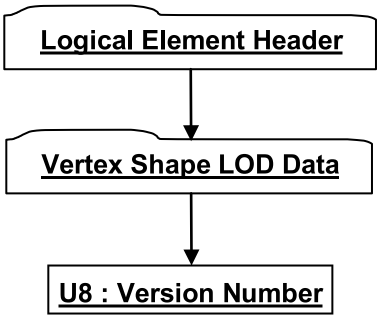


Figure 95 — Point Set Shape LOD Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

Complete description for Vertex Shape LOD Data can be found in Vertex Shape LOD Data.

U8 : Version Number

Version Number is the version identifier for this Point Set Shape LOD. For informaiton on local version numbers see best practice 14.5 Local version numbers

7.1.6 Null Shape LOD Element

Object Type ID: 0x3e637aed, 0x2a89, 0x41f8, 0xa9, 0xfd, 0x55, 0x37, 0x37, 0x3, 0x96, 0x82

A Null Shape LOD Element represents the pseudo geometric shape definition data for a NULL Shape Node Element. Although a NULL Shape Node Element has no real geometric primitive representation (i.e. is empty), its usage as a ―proxy/placeholder" node within the LSG still supports the concept of having a defined bounding box and thus the existence of this Null Shape LOD Element type.

A Null Shape LOD Element is typically referenced by a NULL Shape Node Element using Late Loaded Property Atom Elements (see 6.1.1.10.7 NULL Shape Node Element and 6.2.7 Late Loaded Property Atom Element respectively).

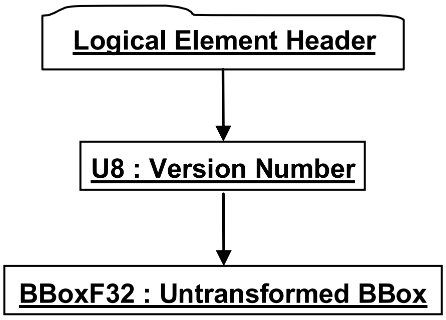


Figure 96 — Null Shape LOD Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

U8 : Version Number

Version Number is the version identifier for this Null Shape LOD Element. For informaiton on local version numbers see best practice 14.5 Local version numbers

BBoxF32 : Untransformed BBox

The Untransformed BBox is an axis-aligned LCS bounding box and represents the untransformed extents for this Null Shape LOD Element.

7.1.7 Polygon Set LOD Element

Object Type ID: 0x10dd109f, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

A Polygon Set LOD Element contains the geometric shape definition data (e.g. vertices, polygons, normals, etc.) for a single LOD of a collection of independent and unconnected polygons. Each polygon constitutes one primitive of the set and the ordering of the vertices in forming polygons, is the same as OpenGL's polygon definition [1].

A Polygon Set LOD Element is typically referenced by a Polygon Set Shape Node Element using Late Loaded Property Atom Elements (see 6.1.1.10.6 Polygon Set Shape Node Element and 6.2.7 Late Loaded Property Atom Element respectively).

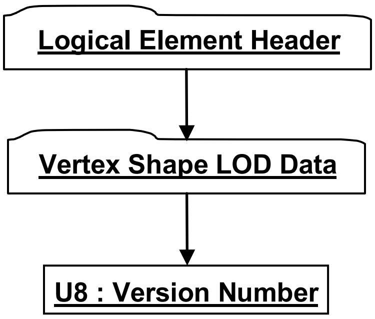


Figure 97 —Polygon Set LOD Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

Complete description for Vertex Shape LOD Data can be found in Vertex Shape LOD Data.

U8 : Version Number

Version Number is the version identifier for this Polygon Set LOD Element. For information on local version numbers see best practice 14.5 Local version numbers

7.2 Primitive Set Shape Element

Object Type ID: 0xe40373c2, 0x1ad9, 0x11d3, 0x9d, 0xaf, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2

A Primitive Set Shape Element defines the minimum data necessary to procedurally generate LODs for a list of primitive shapes (e.g. box, cylinder, sphere, etc.). —Prcedurally generate" means that the raw geometric shape definition data (e.g. vertices, polygons, normals, etc) for LODs is not directly stored; instead some basic shape information is stored (e.g. sphere centre and radius) from which LODs can be generated.

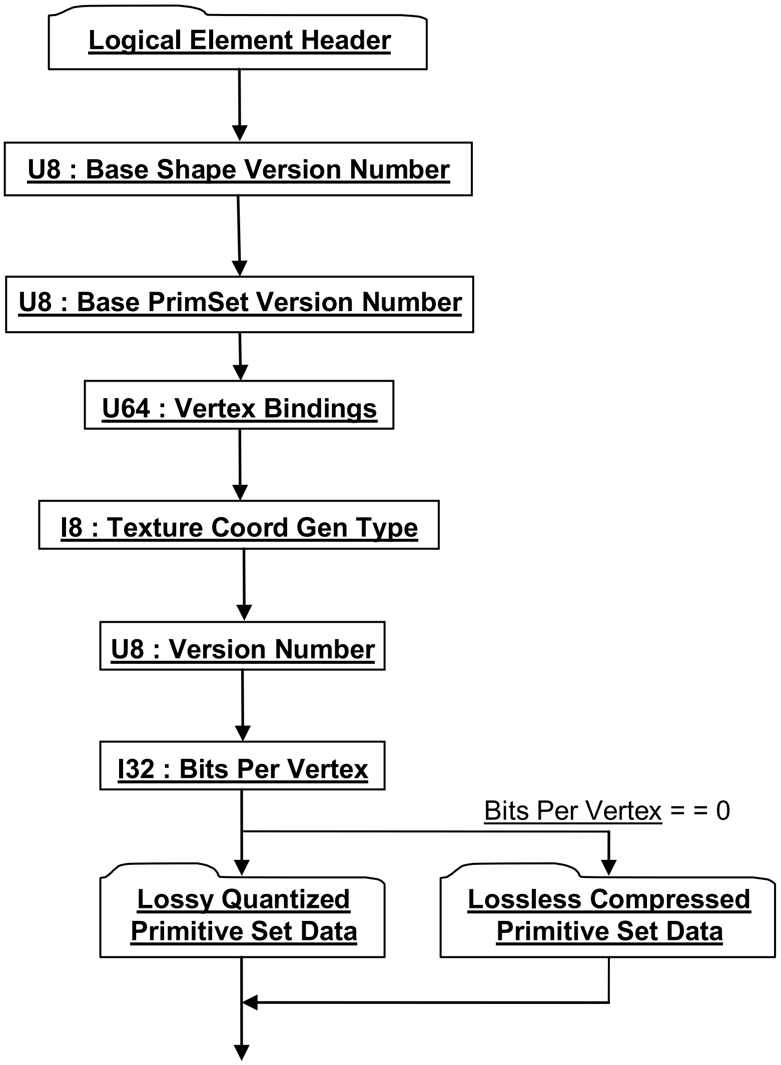

Logical Element Header
U8 : Base Shape Version Number
U8 : Base PrimSet Version Number
U64 : Vertex Bindings
I8 : Texture Coord Gen Type
U8 : Version Number
I32 : Bits Per Vertex
Bits Per Vertex = = 0
Lossy Quantized Primitive Set Data
Lossless Compressed Primitive Set Data

Figure 98 —Primitive Set Shape Element data collection

Complete description for Logical Element Header can be found in Logical Element Header.

U8 : Base Shape Version Number

Base Shape Version Number is the version identifier for the 2-level base class of this element. For informaiton on local version numbers see best practice 14.5 Local version numbers

U8 : Base PrimSet Version Number

Base PrimSet Version Number is the version identifier for the immediate base class of element. For informaiton on local version numbers see best practice 14.5 Local version numbers.

U64 : Vertex Bindings

Vertex Bindings is a collection of normal, texture coordinate, and colour binding information encoded within a single U64. All bits fields that are not defined as in use should be set to ―0‖ For more information see Vertex Shape LOD Data U64 : Vertex Bindings.

U8 : Version Number

Version Number is the version identifier for this element. The value of this Version Number indicates the format of data fields to follow.

Table 49 — Primitive Set Shape Version Number values

= 1	Version-1 Format

I32 : Bits Per Vertex

Bits Per Vertex specifies the number of quantization bits per vertex coordinate component. Value shall be within range [0:32] inclusive.

I8 : Texture Coord Gen Type

Texture Coord Gen Type specifies how a texture is applied to each face of the primitive. Single tile means one copy of the texture will be stretched to fit the face, isotropic means that the texture will be duplicated on the longer dimension of the face in order to maintain the texture's aspect ratio

Table 50 — Primitive Set Shape Texture Coord Gen Type values

= 0	Single Tile…Indicates that a single copy of a texture image will be applied to significant primitive features (i.e. cube face, cylinder wall, end cap) no matter how eccentrically shaped.
= 1	Isotropic…Implies that multiple copies of a texture image may be mapped onto eccentric surfaces such that a mapped texel stays approximately square.

7.2.1 Lossless Compressed Primitive Set Data

The Lossless Compressed Primitive Set Data collection contains all the per-primitive information stored in a ―lossless” compression format for all primitives in the Primitive Set. The Lossless Compressed Primitive Set Data collection is only present when the Bits Per Vertex data field equals ―0‖(see 7.2 Primitive Set Shape Element for complete description).

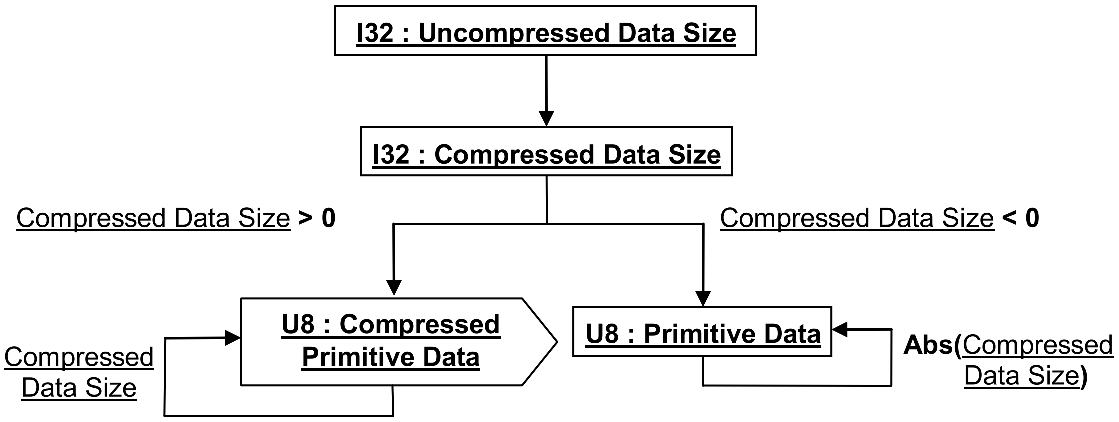

I32 : Uncompressed Data Size
I32 : Compressed Data Size
Compressed Data Size > 0
Compressed Data Size < 0
U8 : Compressed Primitive Data
U8 : Primitive Data
Compressed Data Size
Abs(Compressed Data Size)

Figure 99 — Lossless Compressed Primitive Set Data collection

I32 : Uncompressed Data Size

Uncompressed Data size specifies the uncompressed size of Primitive Data or Compressed Primitive Data in bytes.

I32 : Compressed Data Size

Compressed Data Size specifies the compressed size of Primitive Data or Compressed Primitive Data in bytes. If the Compressed Data Size is negative, then the Compressed Primitive Data field is not present (i.e. data is not compressed) and the absolute value of Compressed Data Size should be equal to Uncompressed Data Size value.

U8 : Primitive Data

The Primitive Data field is a packed array of the raw per primitive data (i.e. reserved, params1, params2, params3, colour, type) sequentially for all primitives in the set. The Primitive Data field is only present if Compressed Data Size value is less than zero.

The per primitive data is packed into Primitive Data array using an interleaved data schema/format as follows:

{[reserved], [params1], [params2], [params3], [colour], [type]}, …, **for all primitives**

Where the data elements have the following size and meaning:

Table 51 — Lossless Compressed Primitive Set Data Field values

Element	Data Type	Description
reserved	I32	This is a field reserved for future expansion of the JT Format.
params1	CoordF32	Interpretation is Primitive Type specific (see below table)
params2	DirF32	Interpretation is Primitive Type specific (see below table)
params3	Quaternion	Interpretation is Primitive Type specific (see below table)
Colour	RGB	Red, Green, Blue colour component values
Type	I32	Primitive Type = 0 – Box = 1 – Cylinder = 2 – Pyramid = 3 – Sphere = 4 – Tri-Prism

Given this format of the Primitive Data, and the previously read size fields, a reader can then implicitly compute the data stride (length of one primitive entry in Primitive Data), and number of primitives.

The interpretation of the three —params#” data fields is primitive type dependent as follows:

Table 52 — Primitive Set —params#” Data Fields Interpretation

Primitive Type	params1			params2			params3			
	[0]	[1]	[2]	[0]	[1]	[2]	[0]	[1]	[2]	[3]
Box	min X	min Y	min Z	length X	length Y	length Z	orientation in Quaternion form			
Cylinder	base centre X	base centre Y	base centre Z	spine X	spine Y	spine Z	radius 1	radius 2	N/A	N/A
Pyramid	base centre X	base centre Y	base centre Z	length X	length Y	length Z	orientation in Quaternion form			
Sphere	centre X	centre Y	centre Z	radius	N/A	N/A	N/A	N/A	N/A	N/A
Tri-Prism	bottom front X	bottom front Y	bottom front Z	length X (to right)	length Y (to back)	length Z (to top)	orientation in Quaternion form			

U8 : Compressed Primitive Data

The Compressed Primitive Data field represents the same data as documented in Primitive Data field above except that the data is compressed using the general —LZMA” method. The Compressed Primitive Data field is only present if Compressed Data Size value is greater than zero. See 13 Data Compression and Encoding for more details on LZMA compression and LZMA library version used.

7.2.2 Lossy Quantized Primitive Set Data

The Lossy Quantized Primitive Set Data collection contains all the per-primitive information (i.e. reserved, params1, params2, params3, colour, type) stored in a —lossy” encoding/compression format for all primitives in the Primitive Set. The Lossy Quantized Primitive Set Data collection is only present when the Bits Per Vertex data field is NOT equal to —0”(See 7.2 Primitive Set Shape Element for compete description).

The interpretation of the three per-primitive —params#” data fields is primitive type dependent. See Table 52 — Primitive Set —params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the —params#” data fields.

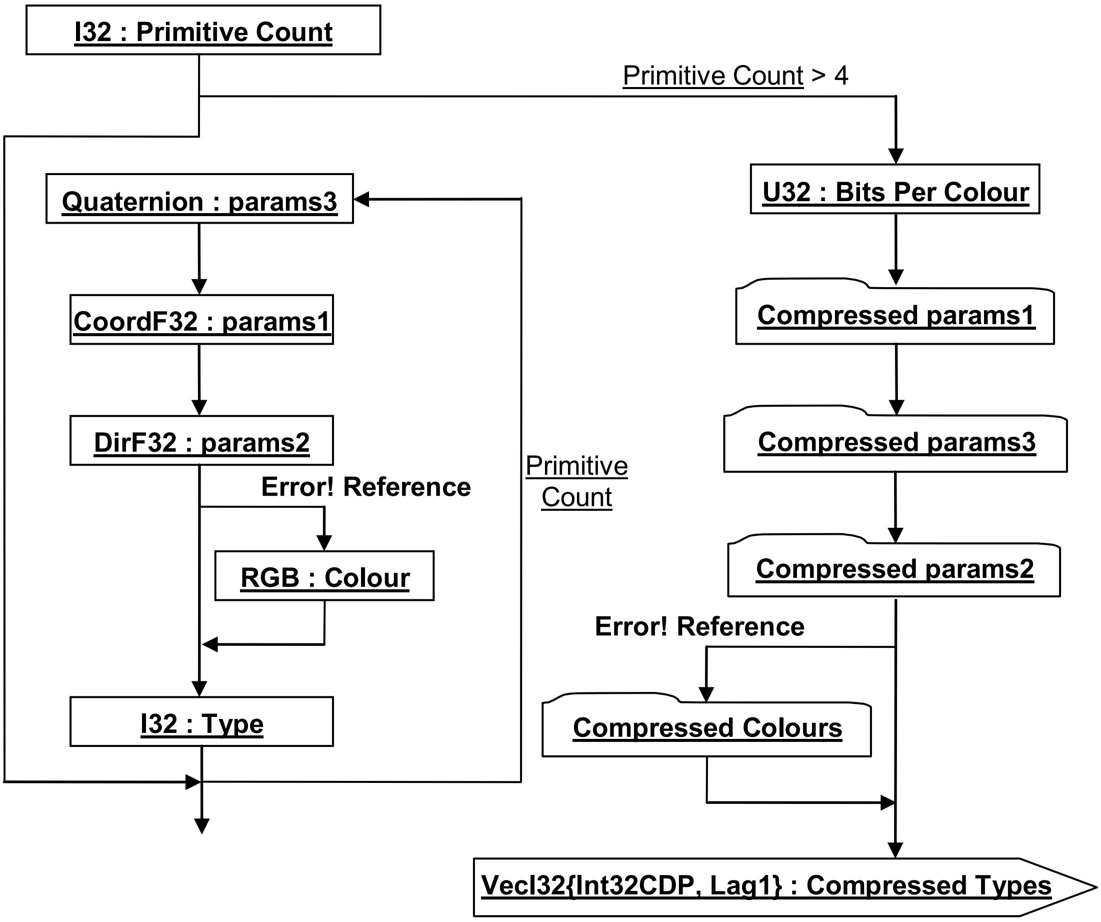


Figure 100 — Lossy Quantized Primitive Set Data collection

I32 : Primitive Count

Primitive Count specifies the number of primitives in the Primitive Set.

Quaternion : params3

Interpretation of params3 data field is primitive Type dependent. See Table 52 — Primitive Set —params#" Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the params3 data fields.

CoordF32 : params1

Interpretation of params1 data field is primitive Type dependent. See Table 52 — Primitive Set —params#" Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the params1 data fields.

DirF32 : params2

Interpretation of params1 data field is primitive Type dependent. See Table 52 — Primitive Set —params#" Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the params1 data fields.

RGB : Colour

Colour specifies the Red, Green Blue colour components for the primitive. This data field is only present if previously read Colour Binding (see 7.2 Primitive Set Shape Element) is not equal to —0

I32 : Type

Type specifies the primitive type. See Table 51 — Lossless Compressed Primitive Set Data Field values in Lossless Compressed Primitive Set Data for valid primitive Type values.

U32 : Bits Per Colour

Bits Per Colour specifies the number of quantization bits per colour component. Value shall be within range [0:32] inclusive.

VecI32{Int32CDP, Lag1} : Compressed Types

The Compressed Types data field is a vector of Type data for all the primitives in the Primitive Set. Compressed Types uses the Int32 version of the CODEC to compress and encode data. In an uncompressed form the valid primitive Type vales are as documented in Table 51 — Lossless Compressed Primitive Set Data Field values in Lossless Compressed Primitive Set Data.

7.2.2.1 Compressed params1

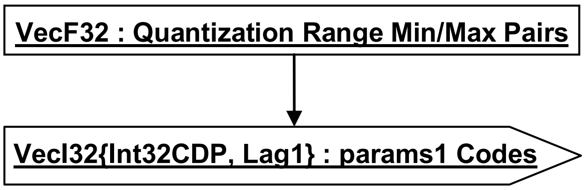
Compressed params1 is the compressed representation of the *params1* data for all the primitives in the Primitive Set. Note that the interpretation of the uncompressed *params1* data is primitive Type dependent. See Table 52 — Primitive Set —params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the *params1* data fields

The *params1* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Vertex number of quantization bits) for each collection of ordinate values. Since *params1* is of type —CoordF32”, it has three ordinate values (three F32 values), and thus three Uniform Quantizers (where a Uniform Quantizer is a scalar quantizer/encoder whose range is divided into levels of equal spacing). See 13 Data Compression and Encoding for more complete description of Uniform Quantizer.

The JT Format packs all the *params1* data for all primitives into a single array using an ordinate dependent order (as shown below) and then encodes each of the lists of ordinate values using a separate Uniform Quantizer per ordinate list.

```
{prim1 params1[0], prim2 params1[0],…primN params1[0],
 prim1 params1[1], prim2 params1[1],…primN params1[1],
 prim1 params1[2], prim2 params1[2],…primN params1[2]}
```

The result of the Uniform Quantizer encoding is a range min and max floating point value pairs for each ordinate value collection, and an integer array of *params1* quantization codes that corresponds to the above described —ordinate dependent order” packed array of *params1* data.



VecF32 : Quantization Range Min/Max Pairs
VecI32{Int32CDP, Lag1} : params1 Codes

Figure 101 — Compressed params1 data collection

VecF32 : Quantization Range Min/Max Pairs

Quantization Range Min/Max Pairs is a vector of Uniform Quantizer range min/max value pairs. There shall be a min/max pair for each ordinate value collection (i.e. each Uniform Quantizer). Thus the length of this vector is —2* num_ordinates” (so vector length would be —6”for *params1* data).

VecI32{Int32CDP, Lag1} : params1 Codes

The params1 Codes data field is a vector of quantizer —codes” for the *params1* data of all the primitives in the Primitive Set. The params1Codes also uses the Int32 version of the CODEC to compress and encode data.

7.2.2.2 Compressed params3

Compressed params3 is the compressed representation of the *params3* data for all the primitives in the Primitive Set. Note that the interpretation of the uncompressed *param31* data is primitive Type dependent. See Table 52 — Primitive Set —params#" Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the *params3* data fields

The *params3* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Vertex number of quantization bits) for each collection of ordinate values. Since *params1* is of type —Quaternion", it has four ordinate values (four F32 values), and thus four Uniform Quantizers (where a Uniform Quantizer is a scalar quantizer/encoder whose range is divided into levels of equal spacing). See 13 Data Compression and Encoding for more complete description of Uniform Quantizer.

The JT Format packs all the *params3* data for all primitives into a single array using an ordinate dependent order (as shown below) and then encodes each of the lists of ordinate values using a separate Uniform Quantizer per ordinate list.

```
{prim1 params3[0], prim2 params3[0],…primN params3[0],
 prim1 params3[1], prim2 params3[1],…primN params3[1],
 prim1 params3[2], prim2 params3[2],…primN params3[2],
 prim1 params3[3], prim2 params3[3],…primN params3[3]}
```

The result of the Uniform Quantizer encoding is a range min and max floating point value pairs for each ordinate value collection, and an integer array of *params3* quantization codes that corresponds to the above described —ordinate dependent order" packed array of *params3* data.

The storage format of Compressed params3 is exactly the same as that documented in Figure 101 — Compressed params1 data collection.

7.2.2.3 Compressed params2

Compressed params2 is the compressed representation of the *params2* data for all the primitives in the Primitive Set. Note that the interpretation of the uncompressed *params2* data is primitive Type dependent. See Table 52 — Primitive Set —params#" Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the *params2* data fields

The *params2* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Vertex number of quantization bits) for each collection of ordinate values. Since *params2* is of type —DF32", it has three ordinate values (three F32 values), and thus three Uniform Quantizers (where a Uniform Quantizer is a scalar quantizer/encoder whose range is divided into levels of equal spacing). See 13 Data Compression and Encoding for more complete description of Uniform Quantizer.

The JT Format packs all the *params2* data for all primitives into a single array using an ordinate dependent order (as shown below) and then encodes each of the lists of ordinate values using a separate Uniform Quantizer per ordinate list.

```
{prim1 params2[0], prim2 params2[0],…primN params2[0],
 prim1 params2[1], prim2 params2[1],…primN params2[1],
 prim1 params2[2], prim2 params2[2],…primN params2[2]}
```

The result of the Uniform Quantizer encoding is a range min and max floating point value pairs for each ordinate value collection, and an integer array of *params2* quantization codes that corresponds to the above described —ordinate dependent order" packed array of *params2* data.

The storage format of Compressed params2 is exactly the same as that documented in Figure 101 — Compressed params1 data collection.

7.2.2.4 Compressed Colours

Compressed Colours is the compressed representation of the *colour* data for all the primitives in the Primitive Set. This data collection is only present if previously read Colour Binding (see 7.2 Primitive Set Shape Element) is not equal to —0'

The *colour* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Colour number of quantization bits) for each collection of ordinate values. Since *colour* is of type ―RGB", it has three ordinate values (three F32 values), and thus three Uniform Quantizers (where a Uniform Quantizer is a scalar quantizer/encoder whose range is divided into levels of equal spacing). See 13 Data Compression and Encoding for more complete description of Uniform Quantizer.

The JT Format packs all the *colour* data for all primitives into a single array using an ordinate dependent order (as shown below) and then encodes each of the lists of ordinate values using a separate Uniform Quantizer per ordinate list.

```
{prim1 colour[0], prim2 colour[0],…primN colour[0],
 prim1 colour[1], prim2 colour[1],…primN colour[1],
 prim1 colour[2], prim2 colour[2],…primN colour[2]}
```

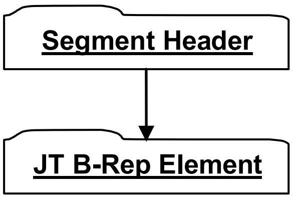
The result of the Uniform Quantizer encoding is a range min and max floating point value pairs for each ordinate value collection, and an integer array of *colour* quantization codes that corresponds to the above described ―ordinate dependent order" packed array of *colour* data.

The storage format of Compressed Colours is exactly the same as that documented in Figure 101 — Compressed params1 data collection.

8 JT B-Rep Segment

JT B-Rep Segment contains an Element that defines the precise geometric Boundary Representation data for a particular Part in JT B-Rep format. Note that there is also another Boundary Representation format (i.e. XT B-Rep) supported by the JT file format within a different file Segment Type. Complete description for the XT B-Rep can be found in 9 XT B-Rep Segment.

JT B-Rep Segments are typically referenced by Part Node Elements (see 6.1.1.5 Part Node Element) using Late Loaded Property Atom Elements (see 6.2.7 Late Loaded Property Atom Element). The JT B-Rep



Segment type supports compression on all element data, so all elements in JT B-Rep Segment use the Logical Element Header Compressed form of element header data.

Figure 102 — JT B-Rep Segment data collection

Complete description for Segment Header can be found in 5.1.3.1 Segment Header.

8.1 JT B-Rep Element

Object Type ID: 0x873a70c0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

JT B-Rep Element represents a particular Part's precise data in JT boundary representation format. Much of the ―heavyweight" data contained within a JT B-Rep Element is compressed and/or encoded. The compression and/or encoding state is indicated through other data stored in each JT B-Rep Element.

Two important aspects of a Part are its geometry and its topology. The geometry describes the shape of a Part: this Surface is a plane, that Surface is a cylinder, this Curve is an arc, etc. The topology describes the connectivity of the Part: this Point is inside the Part, these Surfaces are next to each other, etc. The 0, 1, and 2 dimensional building blocks of geometry are Points, Curves, and Surfaces. The corresponding topological building blocks are Vertices, Edges, and Faces. Topology also uses Shells and Regions to conceptually divide up the three dimensional space.

Parts may have the same topology, but wildly different geometry. Imagine the Surfaces of a Part being composed of rubber. The topology of the Part does not change as we deform the Part by bending or stretching the surfaces, as long as we do not cut or glue them (we call this a ―nice" deformation). A Part's topology can be classified as being ―manifold” or ―non-manifold”; where ―manifold” implies that the Part has the property that each Edge, excluding seams and poles, has exactly two faces using it.

Similarly, Parts may have nearly identical geometry but different topology. The topology of a Part depends on how the geometry is put together. A Part may be manifold or non-manifold simply depending on how the geometry is put together. In addition to describing connectivity in space, topology is used to describe areas of interest (active areas) on Surfaces. These active Surface areas are used in defining a complex Part. The areas are specified by oriented Loops and often referred to as trimmed Surfaces which are exactly the 2-dimensional topological building block called a Face.

Readers desiring/needing a more in-depth exploration of boundary representation theory in order to understand the significance/meaning of some of the JT B-Rep data fields are referred to references [5] and [6] listed in the bibliography section of this International Standard.

Since the topology is a convenient way to describe or organize the Part, it is also convenient to store the geometry of the Part in the topological structures. The following sub-sections document the JT B-Rep format for storing the topology and geometry of a Part in a JT file.

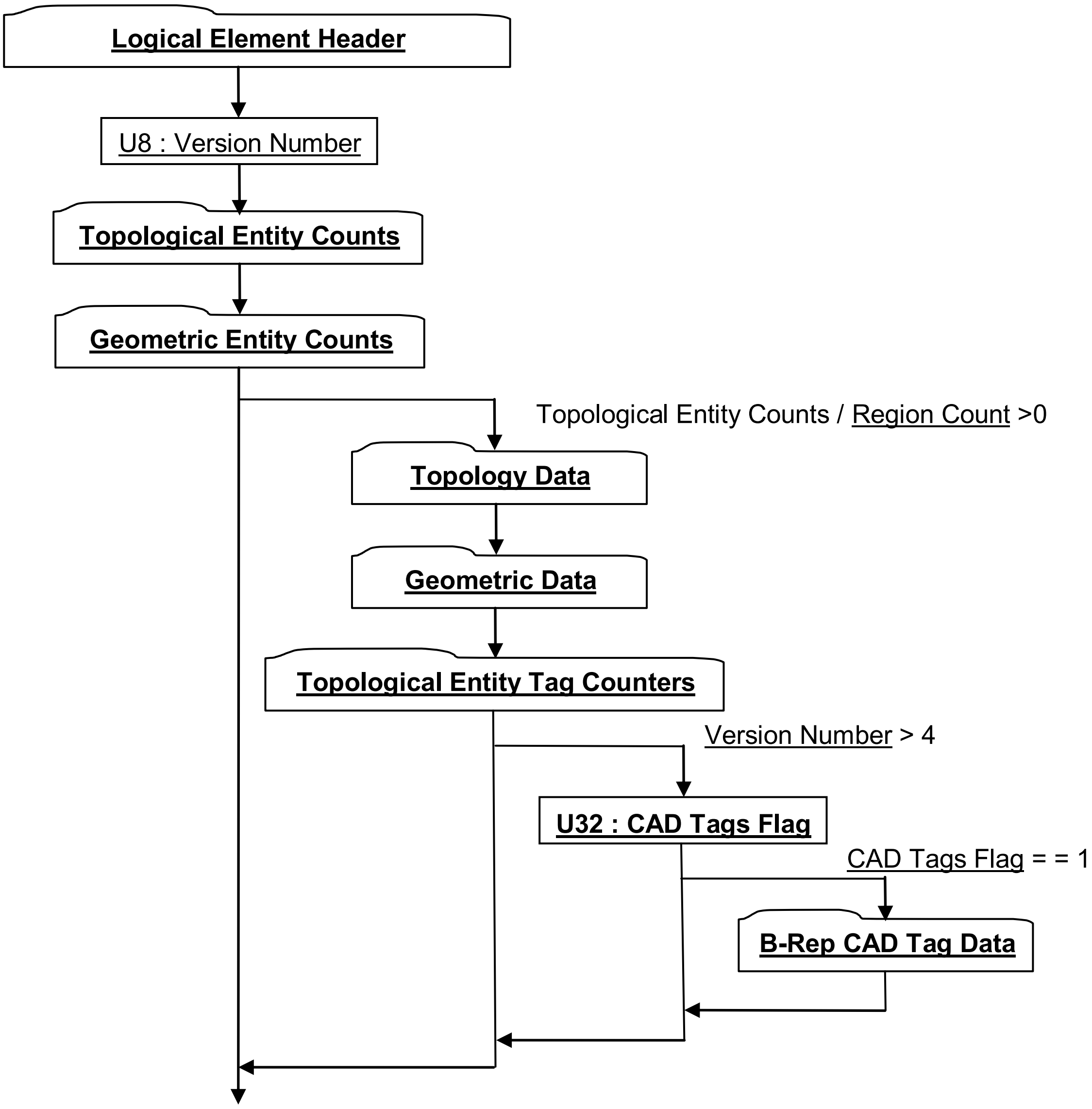


Figure 103 — JT B-Rep Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8 : Version Number

Version Number is the version identifier for this JT B-Rep Element. For informaiton on local version numbers see best practice 14.5 Local version numbers

U32 : CAD Tags Flag

CAD Tags Flag is a flag indicating whether CAD Tag data exist for the JT B-Rep.

8.1.1 Topological Entity Counts

Topological Entity Counts data collection defines the counts for each of the various topological entities within a B-Rep.

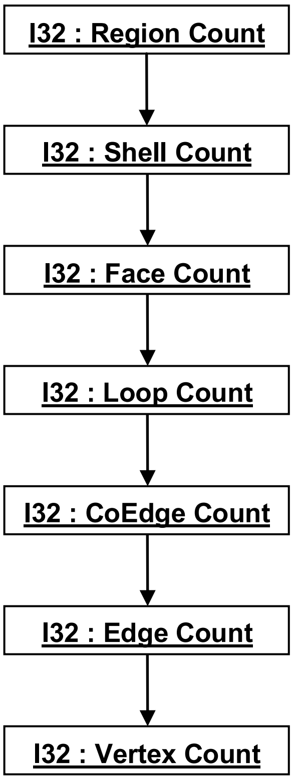


Figure 104 — Topological Entity Counts data collection

I32 : Region Count

Region Count indicates the number of topological region entities in the B-Rep.

I32 : Shell Count

Shell Count indicates the number of topological shell entities in the B-Rep

I32 : Face Count

Face Count indicates the number of topological face entities in the B-Rep

I32 : Loop Count

Loop Count indicates the number of topological loop entities in the B-Rep

I32 : CoEdge Count

CoEdge Count indicates the number of topological coedge entities in the B-Rep

I32 : Edge Count

Edge Count indicates the number of topological edge entities in the B-Rep

I32 : Vertex Count

Vertex Count indicates the number of topological vertex entities in the B-Rep

8.1.2 Geometric Entity Counts

Geometric Entity Counts data collection defines the counts for each of the various geometric entities within a B-Rep.

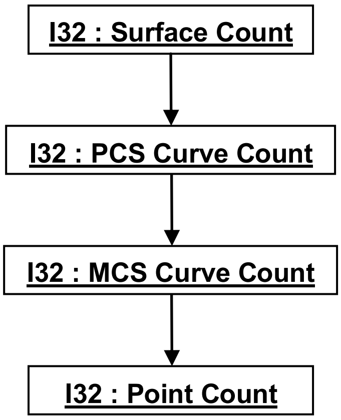


Figure 105 — Geometric Entity Counts data collection

I32 : Surface Count

Surface Count indicates the number of distinct geometric surface entities in the B-Rep

I32 : PCS Curve Count

PCS Curve Count indicates the number of distinct geometric Parameter Coordinate Space curves (i.e. UV curve) entities in the B-Rep

I32 : MCS Curve Count

MCS Curve Count indicates the number of distinct geometric (Model Coordinate Space) curves (i.e. XYZ curve) entities in the B-Rep.

I32 : Point Count

Point Count indicates the number of distinct geometric point entities in the B-Rep.

8.1.3 Topology Data

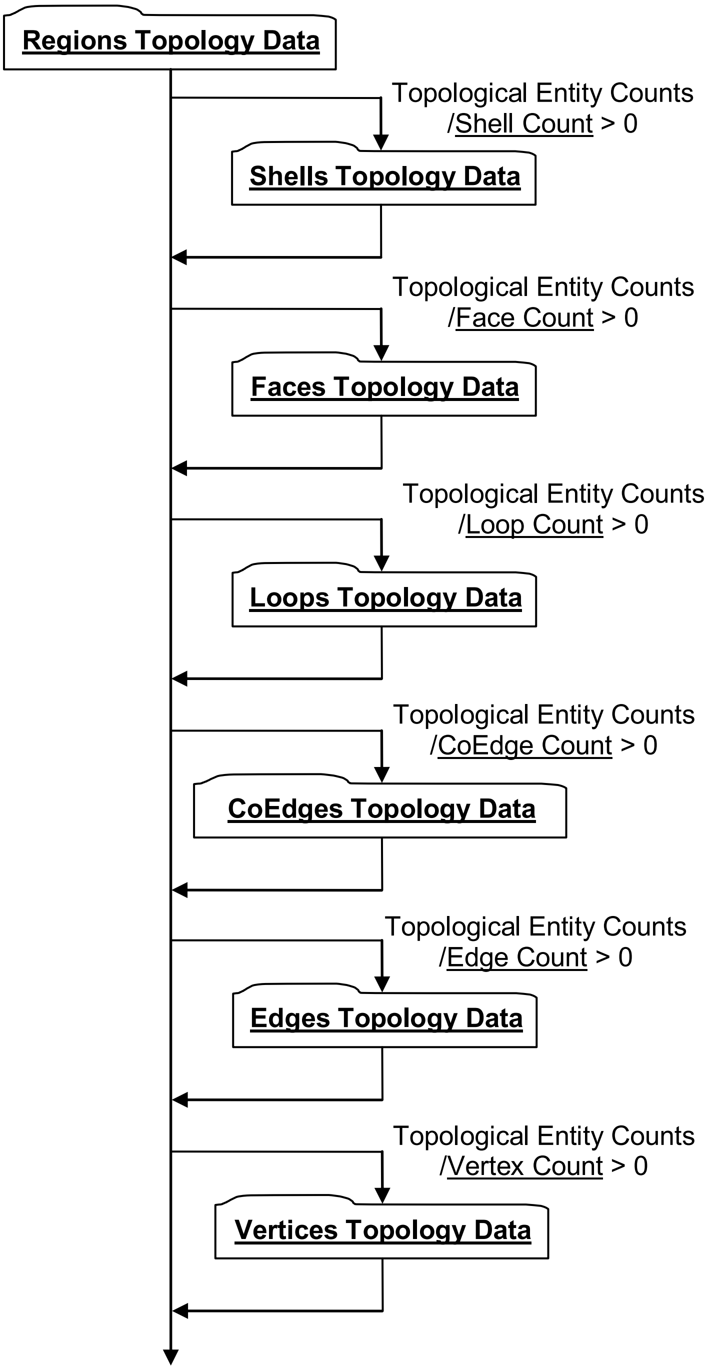

Regions Topology Data
Topological Entity Counts /Shell Count > 0
Shells Topology Data
Topological Entity Counts /Face Count > 0
Faces Topology Data
Topological Entity Counts /Loop Count > 0
Loops Topology Data
Topological Entity Counts /CoEdge Count > 0
CoEdges Topology Data
Topological Entity Counts /Edge Count > 0
Edges Topology Data
Topological Entity Counts /Vertex Count > 0
Vertices Topology Data

Figure 106 — Topology Data collection

8.1.3.1 Regions Topology Data

Regions Topology Data defines the set of non-overlapping Shells comprising each Region. The volume of a Region is that volume lying inside each ―anti-hole Shell" and outside each simply-contained ―hole Shell" belonging to the particular Region. A Region is analogous to a dimensionally elevated face where Region corresponds to Face and Shell corresponds to Trim Loop.

Each Region's defining Shells are identified in a list of Shells by an index for both the first Shell and the last Shell in each Region (i.e. all Shells inclusive between the specified first and last Shell list index define the particular Region).

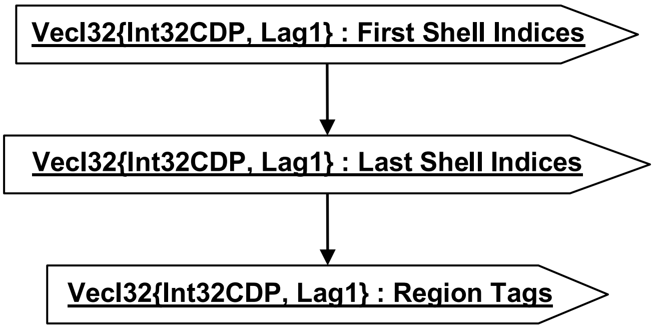

VecI32{Int32CDP, Lag1} : First Shell Indices
VecI32{Int32CDP, Lag1} : Last Shell Indices
VecI32{Int32CDP, Lag1} : Region Tags

Figure 107 — Regions Topology Data collection

VecI32{Int32CDP, Lag1} : First Shell Indices

First Shell Indices is a vector of indices representing the index of the first Shell in each Region. First Shell Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Last Shell Indices

Last Shell Indices is a vector of indices representing the index of the last Shell in each Region. Last Shell Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Region Tags

Each Region has an identifier tag. Region Tags is a vector of identifier tags for a set of Regions. Region Tags uses the Int32 version of the CODEC to compress and encode data.

8.1.3.2 Shells Topology Data

Shells Topology Data defines the set of topological adjacent Faces making up each Shell. A Shell‘s set of topological adjacent Faces define a single (usually closed) two manifold solid that in turn defines the boundary between the finite volume of space enclosed within the Shell and the infinite volume of space outside the Shell. Additional, each Shell has a flag that denotes whether the Shell refers to the finite interior volume (i.e. a ―hole Shell”) or the infinite exterior volume (i.e. an ―anti-hole Shell”).

Each Shell‘s defining Faces are identified in a list of Faces by an index for both the first Face and the last Face in each Shell (i.e. all Faces inclusive between the specified first and last Face list index define the particular Shell).

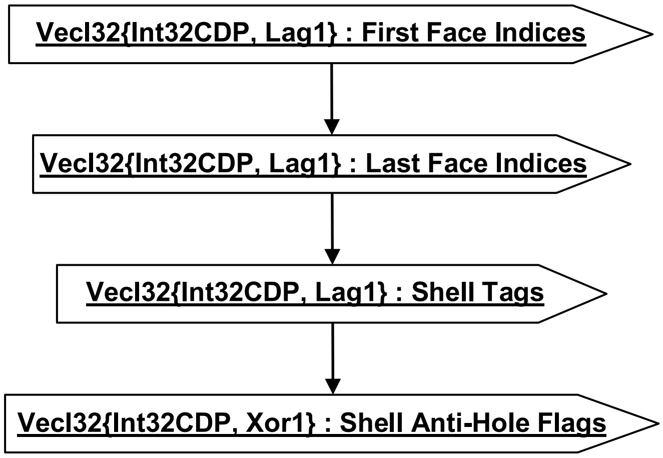

VecI32{Int32CDP, Lag1} : First Face Indices
VecI32{Int32CDP, Lag1} : Last Face Indices
VecI32{Int32CDP, Lag1} : Shell Tags
VecI32{Int32CDP, Xor1} : Shell Anti-Hole Flags

Figure 108 — Shells Topology Data collection

VecI32{Int32CDP, Lag1} : First Face Indices

First Face Indices is a vector of indices representing the index of the first Face in each Shell. First Face Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Last Face Indices

Last Face Indices is a vector of indices representing the index of the last Face in each Shell. Last Face Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Shell Tags

Each Shell has an identifier tag. Shell Tags is a vector of identifier tags for a set of Shells. Shell Tags uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Xor1} : Shell Anti-Hole Flags

Each Shell has a flag identifying whether the Shell is an anti-hole Shell. Shell Anti-Hole Flags is a vector of anti-hole flags for a set of Shells.

In an uncompressed/decoded form the flag values have the following meaning:

Table 53 — JT B-Rep Shell Topology Anti-Hole Flag values

= 0	Shell is not an anti-hole Shell
= 1	Shell is an anti-hole Shell

Shell Anti-Hole Flags uses the Int32 version of the CODEC to compress and encode data.

8.1.3.3 Faces Topology Data

A Face is a two-dimensional topological building block defined as the active (that portion to be used in the model) regions/areas of a Geometric Surface; where active regions/areas of a Geometric Surface are indicated using oriented Trim Loops. Faces Topology Data specifies the underlying Geometric Surface and Trim Loops making up each Face along with a —reverse normal” flag and identifier tag for each Face.

A Face shall be trimmed with at least one —anti-hole” Trim Loop and zero or more —hole” Trim Loops. Thus the area of the Geometric Surface defined as the Face, is the area inside the —anti-hole” Trim Loops and outside each —hole” Trim Loop. No Trim Loops (—hole‘ or —anti-hole”) may intersect/cross or be tangent at any point. —Anti-Hole” Trim Loops shall be defined with a counter-clockwise orientation in the underlying surface's parameter space whereas —hole” Trim Loops shall be defined with a clockwise orientation. With this Trim Loop orientation definition, as one traverses a Trim Loop of a Face, the material or —active region” is always to one‘s left. Figure 109 gives an example in parameter space of proper trim loop definition and orientation (as indicated by the arrows on the loop‘s CoEdges) for a face with two holes. —L1” represents the face —anti-hole” Trim Loop while —L2” and L3” represent the two —hole” Trim Loops. Note that each hole is always represented by a separate distinct —hole” Trim Loop.

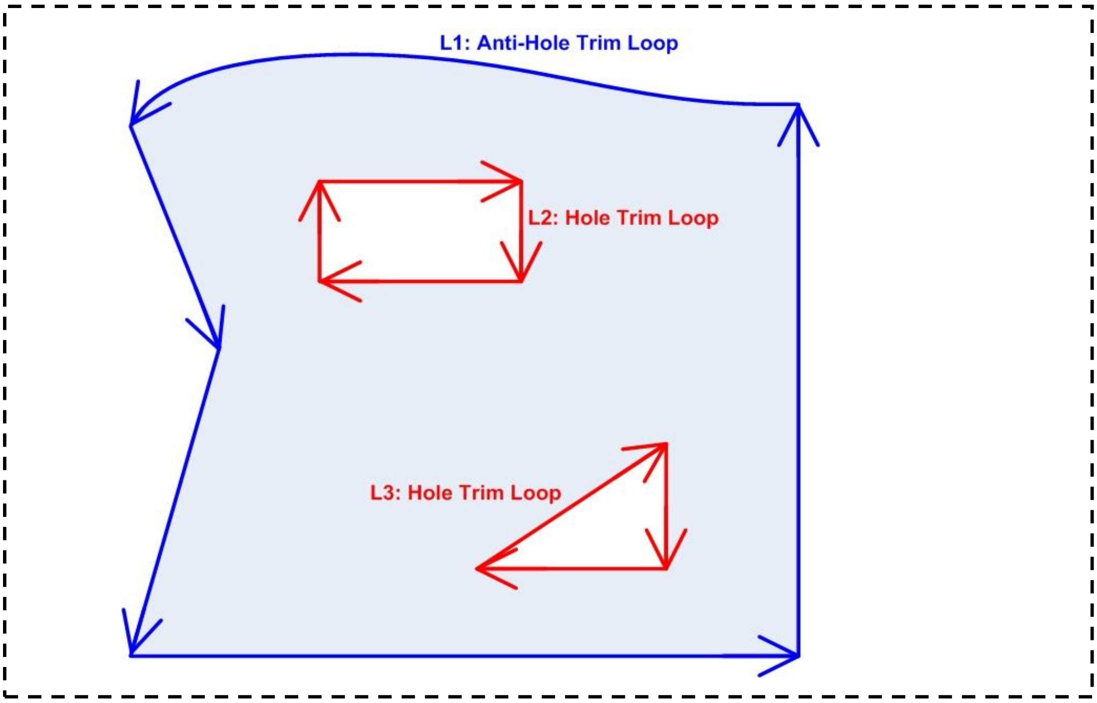


Figure 109 — Trim Loop example in parameter Space - One Face with 2 Holes

Each Face's underlying Geometric Surface is identified by an index into a list of Geometric Surfaces. Each Face's defining Trim Loops are identified in a list of trim Loops by an index for both the first Trim Loop and the last Trim Loop in each Face (i.e. all Trim Loops inclusive between the specified first and last Trim Loop list index define the particular Face).

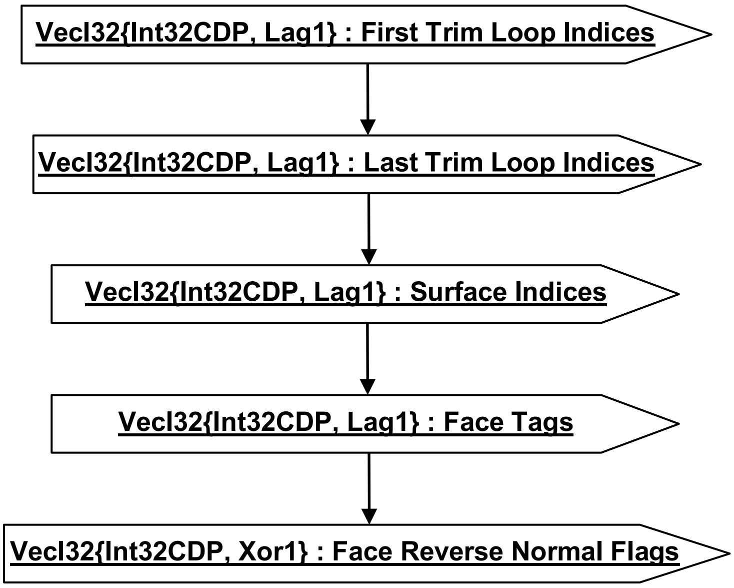

VecI32{Int32CDP, Lag1} : First Trim Loop Indices
VecI32{Int32CDP, Lag1} : Last Trim Loop Indices
VecI32{Int32CDP, Lag1} : Surface Indices
VecI32{Int32CDP, Lag1} : Face Tags
VecI32{Int32CDP, Xor1} : Face Reverse Normal Flags

Figure 110 — Faces Topology Data collection

VecI32{Int32CDP, Lag1} : First Trim Loop Indices

First Trim Loop Indices is a vector of indices representing the index of the first Trim Loop in each Face. First Trim Loop Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Last Trim Loop Indices

Last Trim Loop Indices is a vector of indices representing the index of the last Trim Loop in each Face. Last Trim Loop Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Surface Indices

Surface Indices is a vector of indices representing the index of the underlying Geometric Surface for each Face. Surface Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Face Tags

Each Face has an identifier tag. Face Tags is a vector of identifier tags for a set of Faces. Face Tags uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Xor1} : Face Reverse Normal Flags

Each Face has a flag identifying whether the Face's normal(s) should be interpreted to point in the direction opposite of the usual U cross V normal (note that these flags do not imply any sort of parameter reversal, the flag only implies that the material is on the other side of the surface).

Face Reverse Normal Flags is a vector of reverse-normal flags for a set of Faces.

In an uncompressed/decoded form the flag values have the following meaning:

Table 54 — JT B-Rep Face Reverse Normal Flag values

= 0	Face normal is not reversed
= 1	Face normal is reversed.

Face Reverse Normal Flags uses the Int32 version of the CODEC to compress and encode data.

8.1.3.4 Loops Topology Data

A Loop (often called Trimming Loop) defines in parameter space a 1D boundary around which geometric surfaces are trimmed to form a Face. Loops Topology Data specifies the CoEdges making up each Loop along with an anti-hole flag and identifier tag for each Loop.

A Loop is composed of one or more CoEdges and the Loop shall be closed and non-self-intersecting.

Each Loop‘s defining CoEdges are identified in a list of CoEdges by an index for both the first CoEdge and the last CoEdge in each Loop (i.e. all CoEdges inclusive between the specified first and last CoEdge list index define the particular Loop).

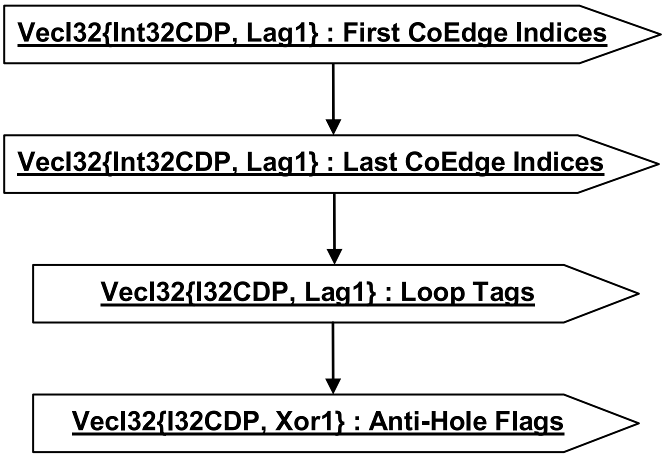

VecI32{Int32CDP, Lag1} : First CoEdge Indices
VecI32{Int32CDP, Lag1} : Last CoEdge Indices
VecI32{I32CDP, Lag1} : Loop Tags
VecI32{I32CDP, Xor1} : Anti-Hole Flags

Figure 111 — Loops Topology Data collection

VecI32{Int32CDP, Lag1} : First CoEdge Indices

First CoEdge Indices is a vector of indices representing the index of the first CoEdge in each Loop. First CoEdge Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Last CoEdge Indices

Last CoEdge Indices is a vector of indices representing the index of the last CoEdge in each Loop. Last CoEdge Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{I32CDP, Lag1} : Loop Tags

Each Loop has an identifier tag. Loop Tags is a vector of identifier tags for a set of Loops. Loop Tags uses the Int32 version of the CODEC to compress and encode data.

VecI32{I32CDP, Xor1} : Anti-Hole Flags

Each Loop has a flag identifying whether the Loop is an anti-hole Loop. Anti-Hole Flags is a vector of anti-hole flags for a set of Loops

In an uncompressed/decoded form the flag values have the following meaning:

Table 55 — JT B-Rep Loops Topology Data Anti-Hole Flag values

= 0	Loop is not an anti-hole Loop
= 1	Loop is an anti-hole Loop

Anti-Hole Flags uses the Int32 version of the CODEC to compress and encode data.

8.1.3.5 CoEdges Topology Data

A CoEdge defines a parameter space edge trim Loop segment (i.e. the projection of an Edge into the parameter space of the Face). CoEdges Topology Data specifies the underlying Edge and PCS Curve making up each CoEdge along with a MCS curve reversed flag and tag for each CoEdge.

The —Co” portion of the CoEdge name derives from the manifold topology definition that each Edge has exactly two Faces containing it; thus a CoEdge defines one Face‘s —use” of an Edge and the adjoining Face also has a CoEdge (—edge use” in some other terminologies) for the same underlying Edge. This sharing of the same underlying Edge by two adjoining Faces requires an —MCS Curve Reversed Flag” on each CoEdge to indicate the edge traversal direction (i.e. for a proper manifold topology definition each CoEdge shall traverse the Edge in opposite directions).

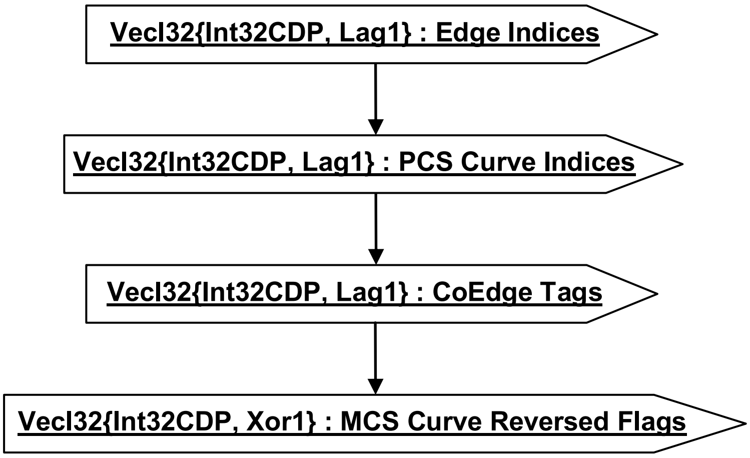

VecI32{Int32CDP, Lag1} : Edge Indices
VecI32{Int32CDP, Lag1} : PCS Curve Indices
VecI32{Int32CDP, Lag1} : CoEdge Tags
VecI32{Int32CDP, Xor1} : MCS Curve Reversed Flags

Figure 112 — CoEdges Topology Data collection

VecI32{Int32CDP, Lag1} : Edge Indices

Edge Indices is a vector of indices representing the index of the underlying Edge for each CoEdge. Edge Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : PCS Curve Indices

PCS Curve Indices is a vector of indices representing the index of the PCS Curve (UV Curve) for each CoEdge. PCS Curve Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : CoEdge Tags

Each CoEdge has an identifier tag. CoEdge Tags is a vector of identifier tags for a set of CoEdges. CoEdge Tags uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Xor1} : MCS Curve Reversed Flags

Each CoEdge has a flag indicating whether the directional sense of the associated Edge‘s MCS curve should be interpreted as opposite the direction its parameterization implies. MCS Curve Reversed Flags is a vector of reverse flags for a set of CoEdges.

In an uncompressed/decoded form the flag values have the following meaning:

Table 56 — JT B-Rep MCS Curve Reversed Flag values

= 0	Directional sense of associated edges MCS curve should not be interpreted as opposite the direction its parameterization implies.
= 1	Directional sense of associated edges MCS curve should be interpreted as opposite the direction its parameterization implies.

MCS Curve Reversed Flags uses the Int32 version of the CODEC to compress and encode data.

8.1.3.6 Edges Topology Data

An Edge defines a model space trim Loop segment. Edges Topology Data specifies the underlying MCS Curve and start and end Vertex making up each Edge along with an identification tag for each Edge.

If manifold topology, then two faces join at a single model Edge and thus an edge is shared/referenced by two CoEdges (one per Face).

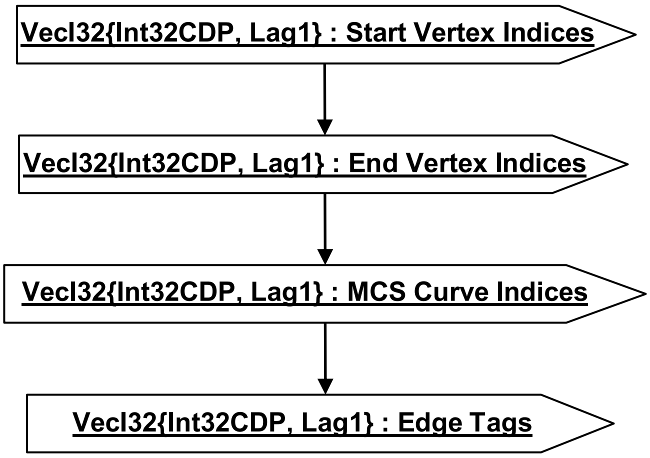

VecI32{Int32CDP, Lag1} : Start Vertex Indices
VecI32{Int32CDP, Lag1} : End Vertex Indices
VecI32{Int32CDP, Lag1} : MCS Curve Indices
VecI32{Int32CDP, Lag1} : Edge Tags

Figure 113 — Edges Topology Data collection

VecI32{Int32CDP, Lag1} : Start Vertex Indices

Start Vertex Indices is a vector of indices representing the index of the start Vertex in each Edge. Start Vertex Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : End Vertex Indices

End Vertex Indices is a vector of indices representing the index of the end Vertex in each Edge. End Vertex Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : MCS Curve Indices

MCS Curve Indices is a vector of indices representing the index of the MCS Curve (Model Space curve) for each Edge. MCS Curve Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Edge Tags

Each Edge has an identifier Tag. Edge Tags is a vector of identifier Tags for a set of Edges. Edge Tags uses the Int32 version of the CODEC to compress and encode data.

8.1.3.7 Vertices Topology Data

A Vertex is the simplest topological entity and is basically made up of a geometric Point. Vertices Topology Data specifies the underlying geometric Point making up each Vertex along with an identification tag for each Vertex.

The presence of Vertices Topology Data in a JT B-Rep topology definition is optional. Vertex data is optional because unlike most topological entities, no connectivity information is contained in a Vertex structure and Vertex data is also not necessary for performing operations such as tessellation or mass properties calculations.

A Vertex is usually shared/referenced by two or more Edges (e.g. if the corners of four rectangular Faces touches at a common point, this point is represented by a Vertex and is shared by four Edges).

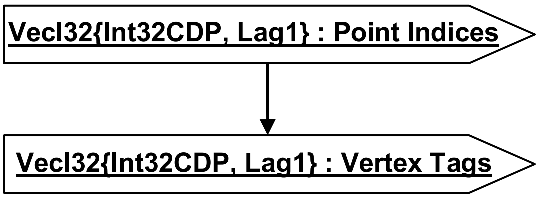

VecI32{Int32CDP, Lag1} : Point Indices
VecI32{Int32CDP, Lag1} : Vertex Tags

Figure 114 — Vertices Topology Data collection

VecI32{Int32CDP, Lag1} : Point Indices

Point Indices is a vector of indices representing the index of the geometric point for each Vertex. Point Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Vertex Tags

Each Vertex has an identifier Tag. Vertex Tags is a vector of identifier Tags for a set of Vertices. Vertex Tags uses the Int32 version of the CODEC to compress and encode data.

8.1.4 Geometric Data

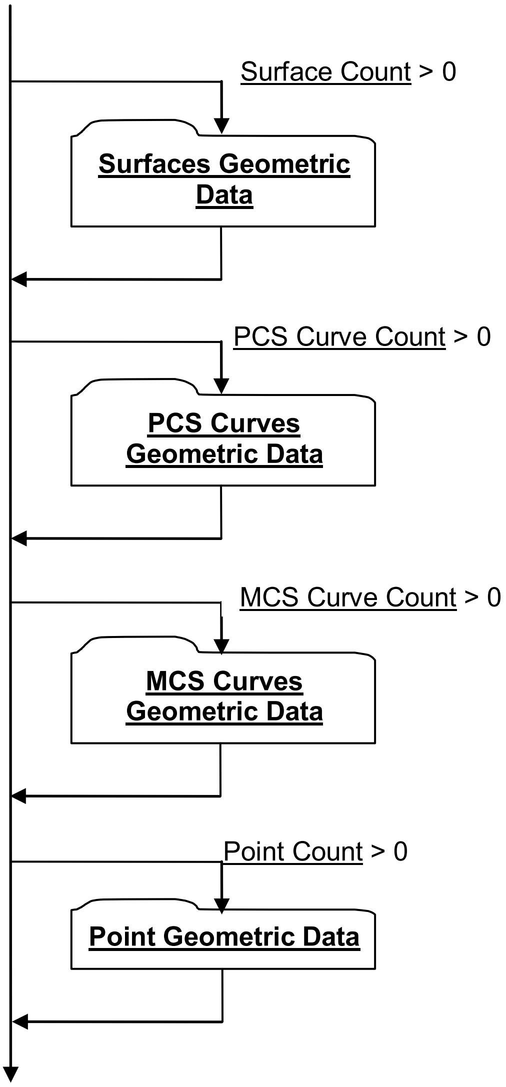


Figure 115 — Geometric Data collection

8.1.4.1 Surfaces Geometric Data

Surfaces Geometric Data collection contains the JT B-Rep's geometric Surface data. Currently only NURBS Surface types are supported within a JT B-Rep. The count/number of Surfaces within a JT B-Rep is indicated by data field Surface Count documented in 8.1.2 Geometric Entity Counts.

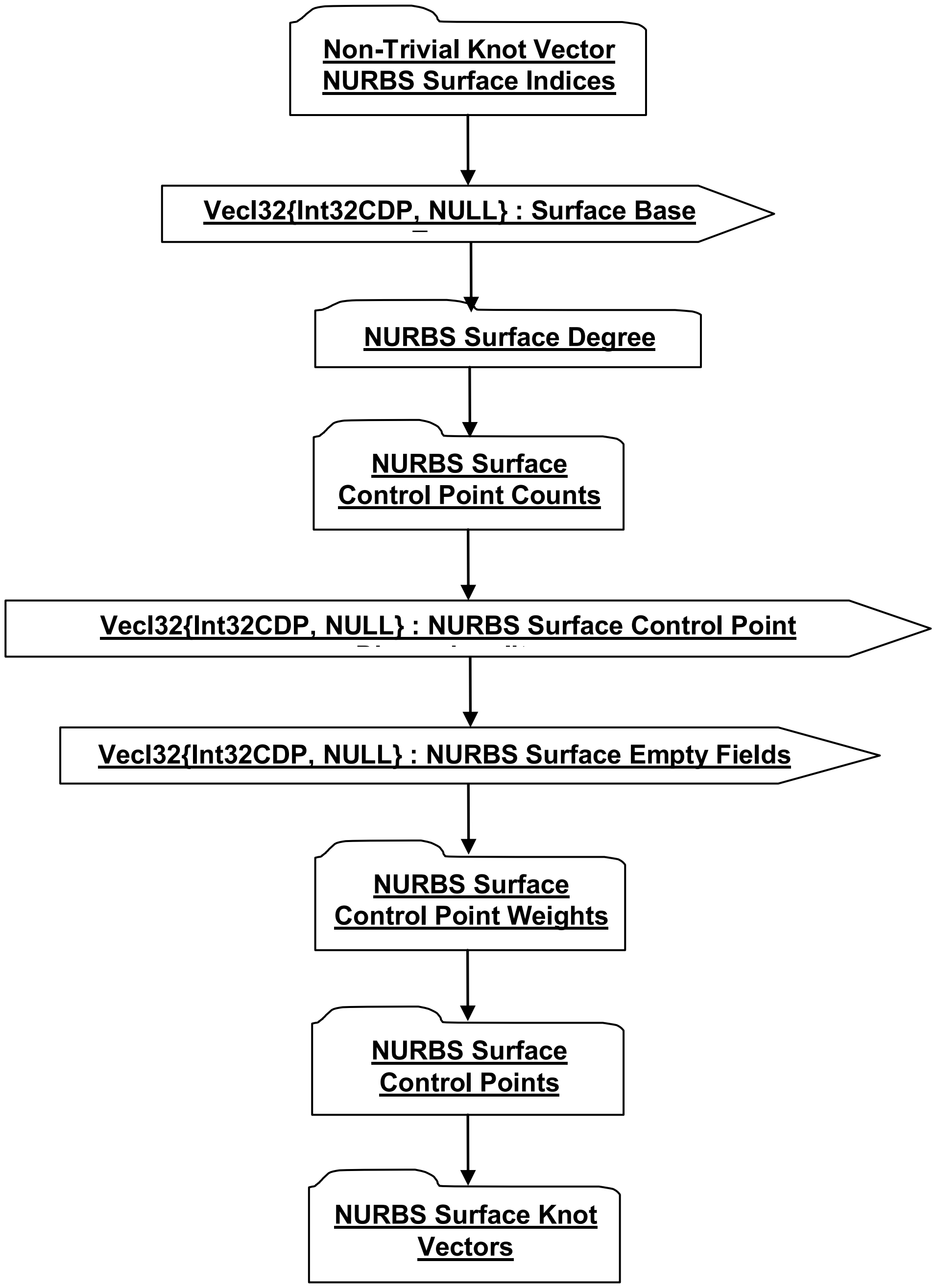

Non-Trivial Knot Vector NURBS Surface Indices
VecI32{Int32CDP, NULL} : Surface Base
NURBS Surface Degree
NURBS Surface Control Point Counts
VecI32{Int32CDP, NULL} : NURBS Surface Control Point
VecI32{Int32CDP, NULL} : NURBS Surface Empty Fields
NURBS Surface Control Point Weights
NURBS Surface Control Points
NURBS Surface Knot Vectors

Figure 116 — Surfaces Geometric Data collection

VecI32{Int32CDP, NULL} : Surface Base Types

Each Surface is assigned a base type identifier. Surface Base Types is a vector of base type identifiers for each Surface in a list of Surfaces. Currently only NURBS Surface Base Type is supported, but a type identifier is still included in the specification to allow for future expansion of the JT Format to support other surface types within a JT B-Rep.

In an uncompressed/decoded form the Surface base type identifier values have the following meaning:

Table 57 — JT B-Rep Surface Base Type value

= 1	Surface is a NURBS surface

Surface Base Types uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : NURBS Surface Control Point Dimensionality

NURBS Surface Control Point Dimensionality is a vector of control point dimensionality values for each NURBS Surface in a list of Surfaces (i.e. there is a stored values for each NURBS Surface in the list).

In an uncompressed/decoded form dimensionality values have the following meaning:

Table 58 — JT B-Rep NURBS Surface Control Point Demensionality values

= 3	Non-Rational (each control point has 3 coordinates)
= 4	Rational (each control point has 4 coordinates)

NURBS Surface Control Point Dimensionality uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : NURBS Surface Empty Fields

NURBS Surface Empty Fields is a vector of data. Each NURBS Surface in a list of Surfaces has one empty data field entry in this NURBS Surface Empty Fields vector. NURBS Surface Empty Fields uses the Int32 version of the CODEC to compress and encode data. Refer to best practice 14.4 Empty Field.

8.1.4.1.1 Non-Trivial Knot Vector NURBS Surface Indices

Non-Trivial Knot Vector NURBS Surface Indices data collection specifies for both U and V directions the Surface index identifiers (i.e. indices to particular NURBS Surfaces within a list of Surfaces) for all NURBS Surfaces containing non-trivial knot vectors. A description/definition for ―non-trivial knot vector” can be found in 13.1.13 Compressed Entity List for Non-Trivial Knot Vector.

This Surface index data is stored in a compressed format.

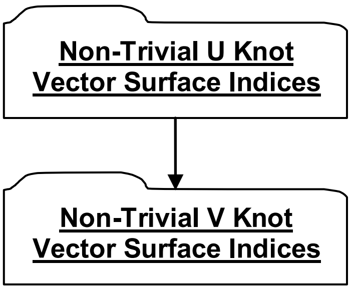

Non-Trivial U Knot Vector Surface Indices
Non-Trivial V Knot Vector Surface Indices

Figure 117 — Non-Trivial Knot Vector NURBS Surface Indices data collection

Both Non-Trivial U Knot Vector Surface Indices and Non-Trivial V Knot Vector Surface Indices have the same data format as that documented for data collection 13.1.13 Compressed Entity List for Non-Trivial Knot Vector.

8.1.4.1.2 NURBS Surface Degree

NURBS Surface Degree data collection defines the Surface degree in both U and V directions for each NURBS Surface in a list of Surfaces (i.e. there are stored values for each NURBS Surface in the list). This degree data for the list of Surfaces is stored in a compressed format.

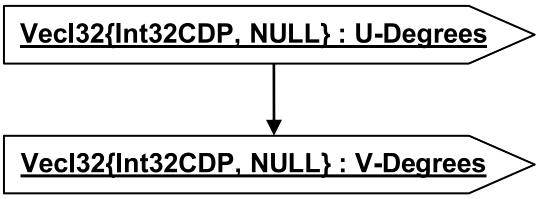

VecI32{Int32CDP, NULL} : U-Degrees
VecI32{Int32CDP, NULL} : V-Degrees

Figure 118 — NURBS Surface Degree data collection

VecI32{Int32CDP, NULL} : U-Degrees

U-Degrees is a vector of Surface degree values in U direction for each NURBS Surface in a list of Surfaces. U-Degrees uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : V-Degrees

V -Degrees is a vector of Surface degree values in V direction for each NURBS Surface in a list of Surfaces. V-Degrees uses the Int32 version of the CODEC to compress and encode data.

8.1.4.1.3 NURBS Surface Control Point Counts

NURBS Surface Control Point Counts defines the number of NURBS Surface control points for both U and V directions for each NURBS Surface in a list of Surfaces (i.e. there are stored values for each NURBS Surface in the list). The control point count data for the list of Surfaces in stored in a compressed format.

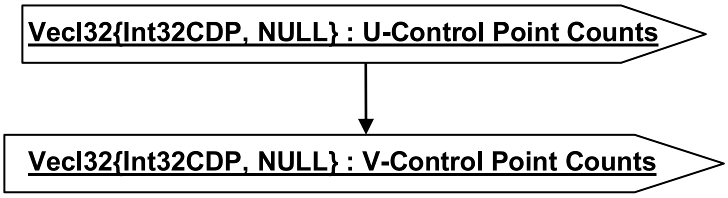

VecI32{Int32CDP, NULL} : U-Control Point Counts
VecI32{Int32CDP, NULL} : V-Control Point Counts

Figure 119 — NURBS Surface Control Point Counts data collection

VecI32{Int32CDP, NULL} : U-Control Point Counts

U-Control Point Counts is a vector of control point counts in U direction for each NURBS Surface in a list of Surfaces. U-Control Point Counts uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : V-Control Point Counts

V-Control Point Counts is a vector of control point counts in V direction for each NURBS Surface in a list of Surfaces. V-Control Point Counts uses the Int32 version of the CODEC to compress and encode data.

8.1.4.1.4 NURBS Surface Control Point Weights

NURBS Surface Control Point Weights data collection defines the Weight values for a conditional set of Control Points for a list of NURBS Surfaces. The storing of the Weight value for a particular Control Point is conditional, because if NURBS Surface Control Point Dimension is —non-rational" or the actual Control Point's Weight value is —1" then no Weight value is stored for the Control Point (i.e. Weight value can be inferred to be —1").

The NURBS Surface Control Point Weights data is stored in a compressed format.

Figure 120 — NURBS Surface Control Point Weights data collection

Complete description for Compressed Control Point Weights Data can be found in 13.1.14 Compressed Control Point Weights Data.

8.1.4.1.5 NURBS Surface Control Points

NURBS Surface Control Points is the compressed and/or encoded representation of the Control Point coordinates for each NURBS Surface in a list of Surfaces (i.e. there are stored values for each NURBS Surface in the list). Note that these are non-homogeneous coordinates (i.e. Control Point coordinates have been divided by the corresponding Control Point Weight values).

VecF64{Int64CDP, NULL} : Control Points

Figure 121 — NURBS Surface Control Points data collection

VecF64{Int64CDP, NULL} : Control Points

Control Points is a vector of Control Point coordinates for all the NURBS Surfaces in a list of Surfaces. All the NURBS Surfaces Control Point coordinates are cumulated into this single vector in the same order as the Surface appears in the Surface list (i.e. Surface-1 U Control Points, Surface-1 V Control Points, Surface-2 U Control Points, Surface-2 V Control Points, etc.). Control Points uses the Int64 version of the CODEC to compress and encode data in a ―lossless” manner. Each deserialized 64 bit integer number should be converted to bit wise equivalent 64 bit floating number.

8.1.4.1.6 NURBS Surface Knot Vectors

NURBS Surface Knot Vectors defines the knot vectors for both U and V directions for each NURBS Surface having non-trivial knot vectors in a list of Surfaces (i.e. there are stored values for each non-trivial knot vector NURBS Surface in the list). The NURBS Surfaces for which knot vectors are stored (i.e. those containing non-trivial knot vectors) are identified in data collection Non-Trivial Knot Vector NURBS Surface Indices documented in 8.1.4.1.1 Non-Trivial Knot Vector NURBS Surface Indices.

The knot vector data for the list of Surfaces is stored in a compressed format.

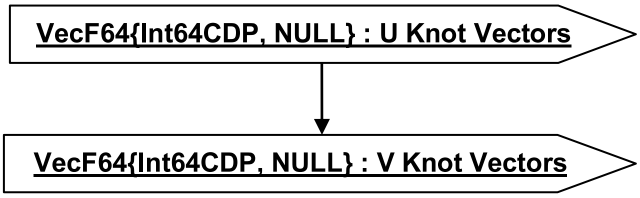

VecF64{Int64CDP, NULL} : U Knot Vectors
VecF64{Int64CDP, NULL} : V Knot Vectors

Figure 122 — NURBS Surface Knot Vectors data collection

VecF64{Int64CDP, NULL} : U Knot Vectors

U Knot Vectors is a list of knot vector values in U direction for each NURBS Surface having non-trivial knot vectors in a list of Surfaces. All these NURBS Surface U direction non-trivial knot vectors are cumulated into this single list in the same order as the Surface appears in the Surface list (i.e. Surface-N Non-Trivial U Knot Vector, Surface-M Non-Trivial U Knot Vector, etc.). U Knot Vectors uses the Int64 version of the CODEC to compress and encode data. Each deserialized 64 bit integer number should be converted to bit wise equivalent 64 bit floating number.

VecF64{Int64CDP, NULL} : V Knot Vectors

V Knot Vectors is a list of knot vector values in V direction for each NURBS Surface having non-trivial knot vectors in a list of Surfaces. All these NURBS Surface V direction non-trivial knot vectors are cumulated into this single list in the same order as the Surface appears in the Surface list (i.e. Surface-N Non-Trivial V Knot Vector, Surface-M Non-Trivial V Knot Vector, etc.). V Knot Vectors uses the Int64 version of the CODEC to compress and encode data. Each deserialized 64 bit integer number should be converted to bit wise equivalent 64 bit floating number.

8.1.4.2 PCS Curves Geometric Data

PCS Curves Geometric Data collection contains the JT B-Rep‘s Parameter Coordinate Space geometric Curve data (i.e. UV Curve data). This geometric PCS Curve data is divided up into two collection types; one data collection for what are considered ―trivial" PCS curves and one data collection for compressed/encoded PCS NURBS Curve data.

―Trivial" PCS Curves are those UV Curves whose definition is such that the actual UV Curve definition can be derived from the parametric domain definition by storing a limited amount of descriptive data for each UV curve (i.e. do not have to store the complete NURBS UV Curve definition).

The count/number of PCS Curves within a JT B-Rep is indicated by data field PCS Curve Count documented in 8.1.2 Geometric Entity Counts.

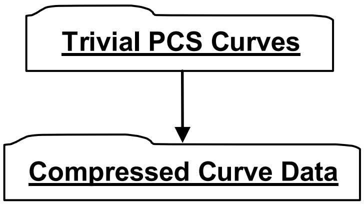


Figure 123 — PCS Curves Geometric Data collection

Complete description for Compressed Curve Data can be found in 13.1.15 Compressed Curve Data.

8.1.4.2.1 Trivial PCS Curves

Trivial PCS Curves data collection represents those UV curves whose definition is such (i.e. ―trivial" enough) that the actual UV curve definition can be derived from the parametric domain definition by storing a limited amount of descriptive data for each UV curve (i.e. do not have to store the complete UV curve definition). These Trivial PCS Curves are grouped into three classifications (Trivial Domain Loop, Trivial Box Loop, or Trivial Domain UV Curve) and stored as described in the following sub-sections.

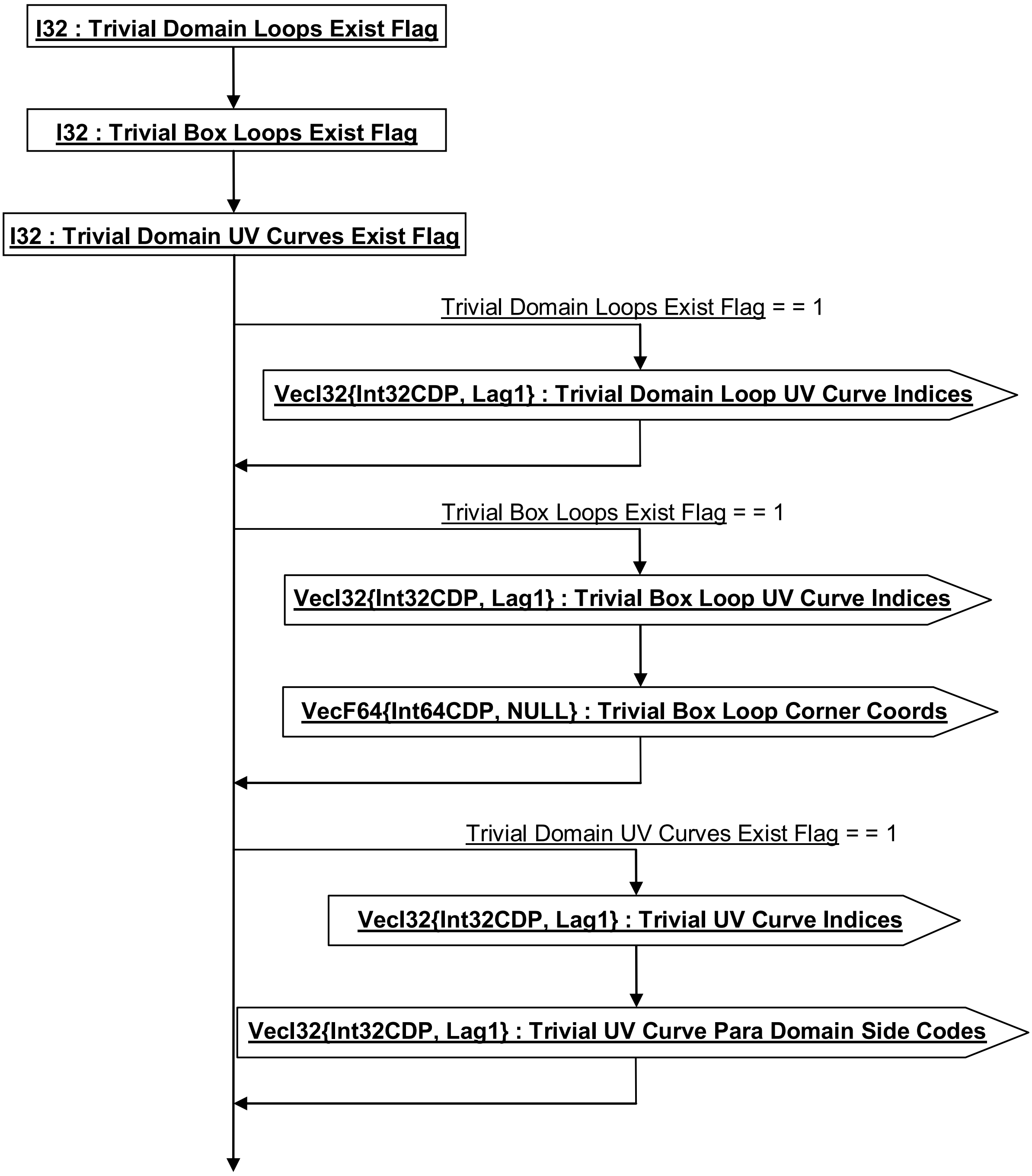

I32 : Trivial Domain Loops Exist Flag
I32 : Trivial Box Loops Exist Flag
I32 : Trivial Domain UV Curves Exist Flag
Trivial Domain Loops Exist Flag = = 1
VecI32{Int32CDP, Lag1} : Trivial Domain Loop UV Curve Indices
Trivial Box Loops Exist Flag = = 1
VecI32{Int32CDP, Lag1} : Trivial Box Loop UV Curve Indices
VecF64{Int64CDP, NULL} : Trivial Box Loop Corner Coords
Trivial Domain UV Curves Exist Flag = = 1
VecI32{Int32CDP, Lag1} : Trivial UV Curve Indices
VecI32{Int32CDP, Lag1} : Trivial UV Curve Para Domain Side Codes

Figure 124 — Trivial PCS Curves data collection

I32 : Trivial Domain Loops Exist Flag

Trivial Domain Loops Exist Flag is a flag indicating whether ―trivial" domain loops exist/follow. A Trivial Domain Loop is a Loop that encloses the entire parametric domain. (i.e. all UV Curves of the Loop span the entire length of the Surface parametric domain). Given this criteria a Trivial Domain Loop shall always be made up of four Trivial Domain UV curves.

Table 59 — Trivial Domain Loops Exist Flag values

= 0	Trivial Domain Loops do not exist.
= 1	Trivial Domain Loops exist.

I32 : Trivial Box Loops Exist Flag

Trivial Box Loops Exist Flag is a flag indicating whether ―trivial" box loops exist/follow. A trivial Box Loop is a Loop that forms a rectangle (i.e. corresponding curve end coordinates of opposite sides of the box are equal). Given this criteria a Trivial Box Loop shall always be made up of four UV curves

Table 60 — Trivial Box Loops Exist Flag values

= 0	Trivial Box Loops do not exist.
= 1	Trivial Box Loops exist.

—Equality of corresponding curve end coordinates of opposite sides of the box” is represented graphically in Figure 125.

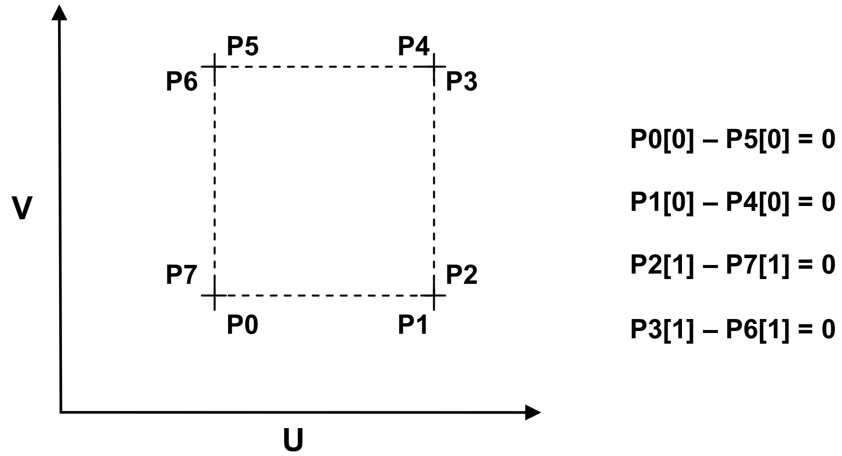


Figure 125 — Equality of corresponding curve end coordinates of opposite sides of the box

I32 : Trivial Domain UV Curves Exist Flag

Trivial Domain UV Curves Exist Flag is a flag indicating whether —trivial” domain UV curves (Loop CoEdges) exist/follow that are not part of a Trivial Domain Loop or Trivial Box Loop (i.e. a Loop contains some UV curves that span the entire length of the Surface parametric domain but not all the Loop UV curves meet this criteria and thus not captured as part of the Trivial Domain Loop data).

Table 61 — Trivial Domain UV Curves Exist Flag values

= 0	Trivial Domain UV Curves do not exist.
= 1	Trivial Domain UV Curves exist.

VecI32{Int32CDP, Lag1} : Trivial Domain Loop UV Curve Indices

Trivial Domain Loop UV Curve Indices is a vector of all UV curve indices that are part of a Trivial Domain Loop. Note that each Trivial Domain Loop is always made up of four UV curves (thus four UV curve indices per Loop). Trivial Domain Loop UV Curve Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Trivial Box Loop UV Curve Indices

Trivial Box Loop UV Curve Indices is a vector of all UV Curve indices that are part of a Trivial Box Loop. Note that each Trivial Box Loop is always made up of four UV Curves (thus four UV Curve indices per Loop). Trivial Box Loop UV Curve Indices uses the Int32 version of the CODEC to compress and encode data.

VecF64{Int64CDP, NULL} : Trivial Box Loop Corner Coords

Trivial Box Loop Corner Coords is a vector of box corner coordinates for all Trivial Box Loops (i.e. each Box Loop will store two box coroner coordinates). A Box Loop‘s set of —box corner coordinates” are the coordinates of the two min/max diagonally opposite corners of the box. Note that if the Box Loop is a —hole”, then the max and min corners are the other ends of the respective box sides that contain the max and min corners. Trivial Box Loop Corner Coords uses the Int64 version of the CODEC to compress and encode data. Each deserialized 64 bit integer number should be converted to bit wise equivalent 64 bit floating number.

VecI32{Int32CDP, Lag1} : Trivial UV Curve Indices

Trivial UV Curve Indices is a vector of all Loop UV Curve indices that are not part of a Trivial Domain Loop or Trivial Box Loop. Trivial UV Curve Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Trivial UV Curve Para Domain Side Codes

Trivial UV Curve Para Domain Side Codes is a vector containing a ―side code" for each Trivial UV Curve indicating which parametric domain side the UV Curve lies on.

In an uncompressed/decoded form the parametric domain side values have the following meaning:

Table 62 — Trivial UV Curve Para Domain Side Codes values

= 0	Bottom side of parametric domain
= 1	Right side of parametric domain
= 2	Top side of parametric domain
= 3	Left side of parametric domain

Trivial UV Curve Para Domain Side Codes uses the Int32 version of the CODEC to compress and encode data.

8.1.4.2.2 MCS Curves Geometric Data

MCS Curves Geometric Data collection contains the JT B-Rep‘s Model Coordinate System geometric Curve data (i.e. XYZ Curve data). Currently only NURBS Curve types are supported within a JT B-Rep. The count/number of MCS Curves within a JT B-Rep is indicated by data field MCS Curve Count documented in 8.1.2 Geometric Entity Counts.

Figure 126 — MCS Curves Geometric Data collection

Complete description for Compressed Curve Data can be found in 13.1.15 Compressed Curve Data.

8.1.4.2.3 Point Geometric Data

Point Geometric Data collection contains the JT B-Rep‘s geometric Point data. Each Point is simply represented by a CoordF32 for the Point‘s coordinate components. The count/number of Points within a JT B-Rep is indicated by data field Point Count documented in 8.1.2 Geometric Entity Counts.

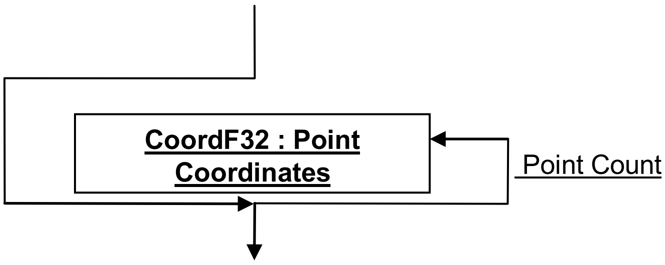


Figure 127 — Point Geometric Data collection

CoordF32 : Point Coordinates

Point Coordinates specifies the XYZ coordinate components for a Point.

8.1.5 Topological Entity Tag Counters

Topological Entity Tag Counters data collection specifies the next available ―unique” tag value for each entity type in a JT B-Rep. These are rolling tag counters that are meant to be used for assigning a unique tag when a new entity is added to a JT B-Rep.

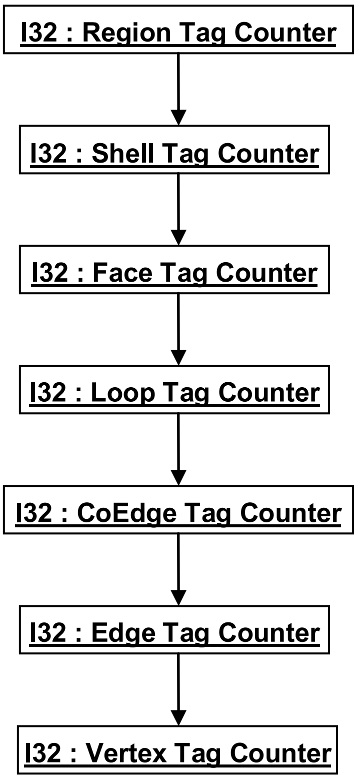


Figure 128 — Topological Entity Tag Counters data collection

I32 : Region Tag Counter

Region tag Counter specifies the next available ―unique‘ tag value for Region entity.

I32 : Shell Tag Counter

Shell Tag Counter specifies the next available ―unique‘ tag value for Shell entity.

I32 : Face Tag Counter

Face Tag Counter specifies the next available ―unique‘ tag value for Face entity.

I32 : Loop Tag Counter

Loop Tag Counter specifies the next available ―unique‘ tag value for Loop entity.

I32 : CoEdge Tag Counter

CoEdge Tag Counter specifies the next available ―unique‘ tag value for CoEdge entity.

I32 : Edge Tag Counter

Edge Tag Counter specifies the next available ―unique‘ tag value for Edge entity.

I32 : Vertex Tag Counter

Vertex Tag Counter specifies the next available ―unique‘ tag value for Vertex entity.

8.1.6 B-Rep CAD Tag Data

The B-Rep CAD Tag Data collection contains the list of persistent IDs, as defined in the CAD System, to uniquely identify individual Faces and Edges in the JT B-Rep. The existence of this B-Rep CAD Tag Data collection is dependent upon the value of previously read data field CAD Tags Flag as documented in 8.1 JT B-Rep Element.

If B-Rep CAD Tag Data collection is present, there will be a CAD Tag for every Face and every Edge in the JT B-Rep and the list order will be Face CAD Tags followed by Edge CAD Tags. Therefore the total number of CAD Tags in the list should be equal to –Face Count + Edge Count” as documented in 8.1.1 Topological Entity Counts.

Figure 129 — B-Rep CAD Tag Data collection

Complete description for Compressed CAD Tag Data can be found in 13.1.16 Compressed CAD Tag Data.

9 XT B-Rep Segment

XT B-Rep Segment contains an Element that defines the precise geometric Boundary Representation data for a particular Part in XT boundary representation format. Note that there is also another Boundary Representation format (i.e. JT B-Rep) supported by the JT file format within a different file Segment Type. Complete description for the JT B-Rep can be found in 8 JT B-Rep Segment.

XT B-Rep Segments are typically referenced by Part Node Elements (see 6.1.1.5 Part Node Element) using Late Loaded Property Atom Elements (see 6.2.7 Late Loaded Property Atom Element). The XT B-Rep Segment type supports LZMA compression on all element data, so all elements in XT B-Rep Segment use the Logical Element Header Compressed form of element header data.

9.1 XT B-Rep Element

Object Type ID: 0x873a70e0, 0x2ac9, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

XT B-Rep Element represents a particular part‘s precise data in XT boundary representation) format.

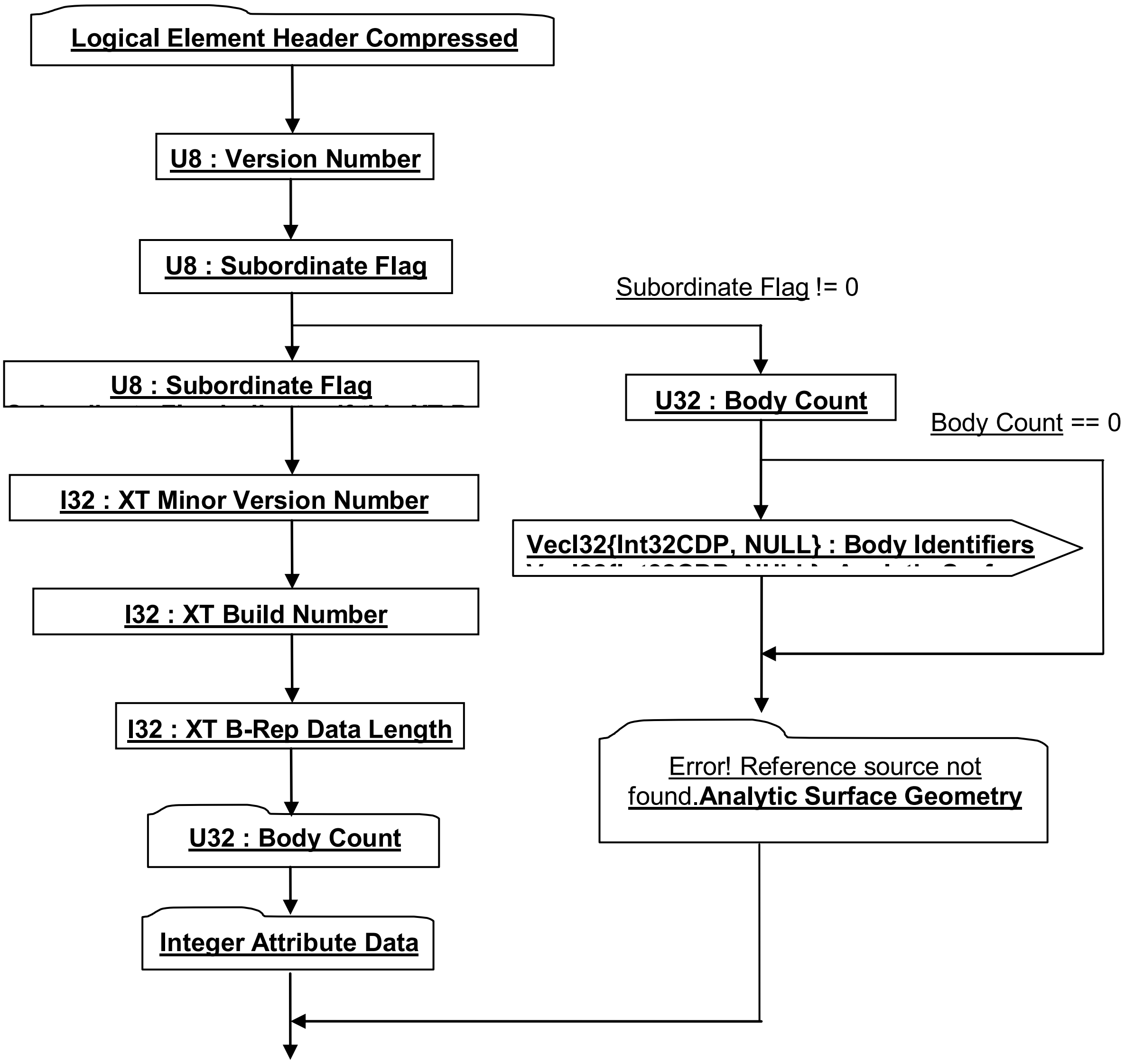

Logical Element Header Compressed
U8 : Version Number
U8 : Subordinate Flag
Subordinate Flag != 0
U8 : Subordinate Flag
U32 : Body Count
Body Count == 0
I32 : XT Minor Version Number
VecI32{Int32CDP, NULL} : Body Identifiers
I32 : XT Build Number
I32 : XT B-Rep Data Length
Error! Reference source not found.Analytic Surface Geometry
U32 : Body Count
Integer Attribute Data

Figure 130 — XT B-Rep Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8 : Version Number

Version Number is the version identifier for this XT B-Rep Element. For information on local version numbers see best practice 14.5 Local version numbers

U8 : Subordinate Flag

Subordinate Flag indicates if this XT B-Rep segment is a subordinate of a XT B-Rep segment. If its value is set to be 0, then this XT B-Rep segment contains complete Parasolid data representation and therefore is not a subordinate. Otherwise if this XT B-Rep segment is a subordinate, meaning that its Parasolid data representation resides in a XT B-Rep segment.

I32 : XT Major Version Number

XT Major Version Number specifies the major version number for the for the XT B-Rep data in the JT File. Major version number is an informative field which can be set to 0 without negative impact to implementation.

I32 : XT Minor Version Number

XT Minor Version Number specifies the minor version number for the the XT B-Rep data in the JT File. Minor version number is an informative field which can be set to 0 without negative impact to implementation.

I32 : XT Build Number

XT Build Number specifies the build number for the XT B-Rep data in the JT File. XT build number is an informative field which can be set to 0 without negative impact to implementation.

I32 : XT B-Rep Data Length

XT B-Rep Data Length specifies the length in bytes of the XT B-Rep Element collection. A JT file loader/reader may use this information to compute the end position of the XT B-Rep Data within the JT file and thus skip (for whatever reason) reading the remaining XT B-Rep Data

.

U32 : Body Count

Body Count specifies the number of XT bodies in this XT B-Rep Element.

VecI32{Int32CDP, NULL} : Body Identifiers

Body Identifiers is an integer array with its length equal to Body Count. The value of each element in this array represents the persisted identifier of the corresponding XT entity. This array is a subset of Body Identifiers array described in **Error! Reference source not found.**, and is used to indicate which XT bodies in **Error! Reference source not found.** belong to this XT B-Rep Element.

9.1.1 XT B-Rep Data

The U32 : Body Count

Body Count specifies the number of XT bodies in this XT B-Rep Element.

VecI32{Int32CDP, NULL} : Body Identifiers

Body Identifiers is an integer array with its length equal to Body Count. The value of each element in this array represents the persisted identifier of the corresponding XT entity. This array is a subset of Body Identifiers array described in Error! Reference source not found., and is used to indicate which XT bodies in Error! Reference source not found. belong to this XT B-Rep Element.

XT B-Rep Data collection specifies the raw stream of bytes used to represent a Part‘s XT B-Rep Body(s). Complete documentation for the XT —Neutral Binary” encoding format can be found in Annex E.

9.1.2 Integer Attribute Data

Integer Attribute Data represents the collection of integer values that may be associated with each face and edge in the XT B-Rep representation.

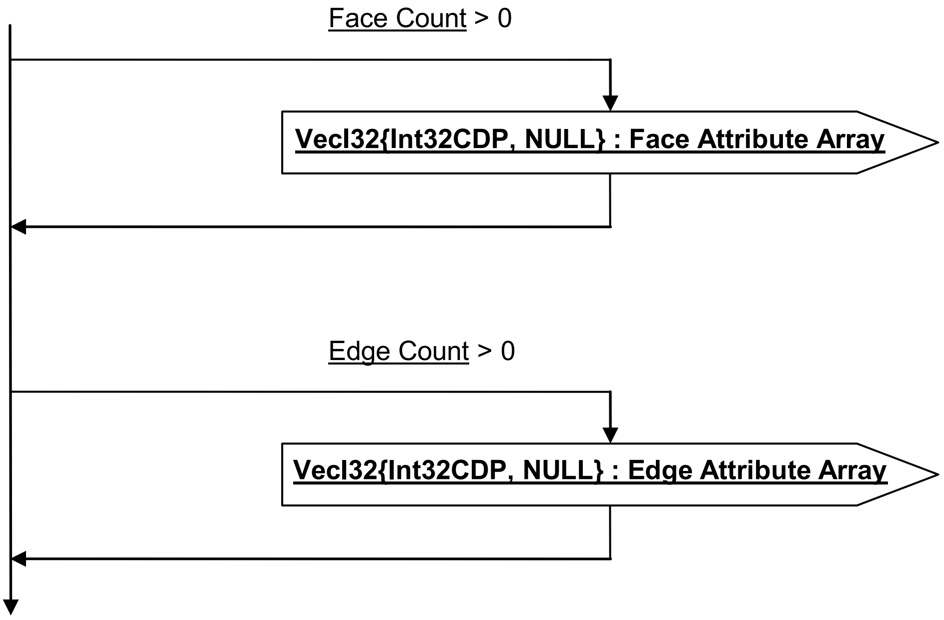

Face Count > 0
Vecl32{Int32CDP, NULL} : Face Attribute Array
Edge Count > 0
Vecl32{Int32CDP, NULL} : Edge Attribute Array

Figure 131 — Integer Attribute Data collection

Face Count

Face Count specifies the number of XT B-Rep faces, which can be found from U32 : Body Count

Body Count specifies the number of XT bodies in this XT B-Rep Element.

VecI32{Int32CDP, NULL} : Body Identifiers

Body Identifiers is an integer array with its length equal to Body Count. The value of each element in this array represents the persisted identifier of the corresponding XT entity. This array is a subset of Body Identifiers array described in **Error! Reference source not found.**, and is used to indicate which XT bodies in **Error! Reference source not found.** belong to this XT B-Rep Element.

XT B-Rep Data.

VecI32{Int32CDP, NULL} : Face Attribute Array

Face Attribute Array specifies the collection of integer attributes, one for each XT B-Rep face. The attributes are arranged according to XT B-Rep face index as describedin 14.12 Brep Face Group Associations.

Edge Count

Edge Count specifies the number of XT B-Rep edges, which can be found from U32 : Body Count

Body Count specifies the number of XT bodies in this XT B-Rep Element.

VecI32{Int32CDP, NULL} : Body Identifiers

Body Identifiers is an integer array with its length equal to Body Count. The value of each element in this array represents the persisted identifier of the corresponding XT entity. This array is a subset of Body Identifiers array described in **Error! Reference source not found.**, and is used to indicate which XT bodies in **Error! Reference source not found.** belong to this XT B-Rep Element.

XT B-Rep Data.

VecI32{Int32CDP, NULL} : Edge Attribute Array

Edge Attribute Array specifies the collection of integer attributes, one for each XT B-Rep edge. The attributes are arranged in the same sequence according XT B-Rep edge index, decided in a similar way as XT B-Rep face index as describedin 14.12 Brep Face Group Associations.

10 Wireframe Segment

Wireframe Segment contains an Element that defines the precise 3D wireframe data for a particular Part. A Wireframe Segment is typically referenced by a Part Node Element (see 6.1.1.5 Part Node Element) using a Second specifies the date Second value. Valid values are [0, 59] inclusive.

Late Loaded Property Atom Element (see 6.2.7 Late Loaded Property Atom Element). The Wireframe Segment type supports LZMA compression on all element data, so all elements in Wireframe Segment use the Logical Element Header Compressed form of element header data.

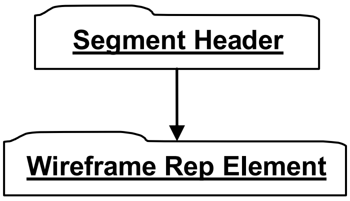


Figure 132 —Wireframe Segment data collection

Complete description for Segment Header can be found in 5.1.3.1 Segment Header.

10.1 Wireframe Rep Element

Object Type ID: 0x873a70d0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

A Wireframe Rep Element represents a particular Part's precise 3D wireframe data (e.g. reference curves, section curves). Much of the ―heavyweight" data contained within a Wireframe Rep Element is compressed and/or encoded. The compression and/or encoding state is indicated through other data stored in each Wireframe Rep Element.

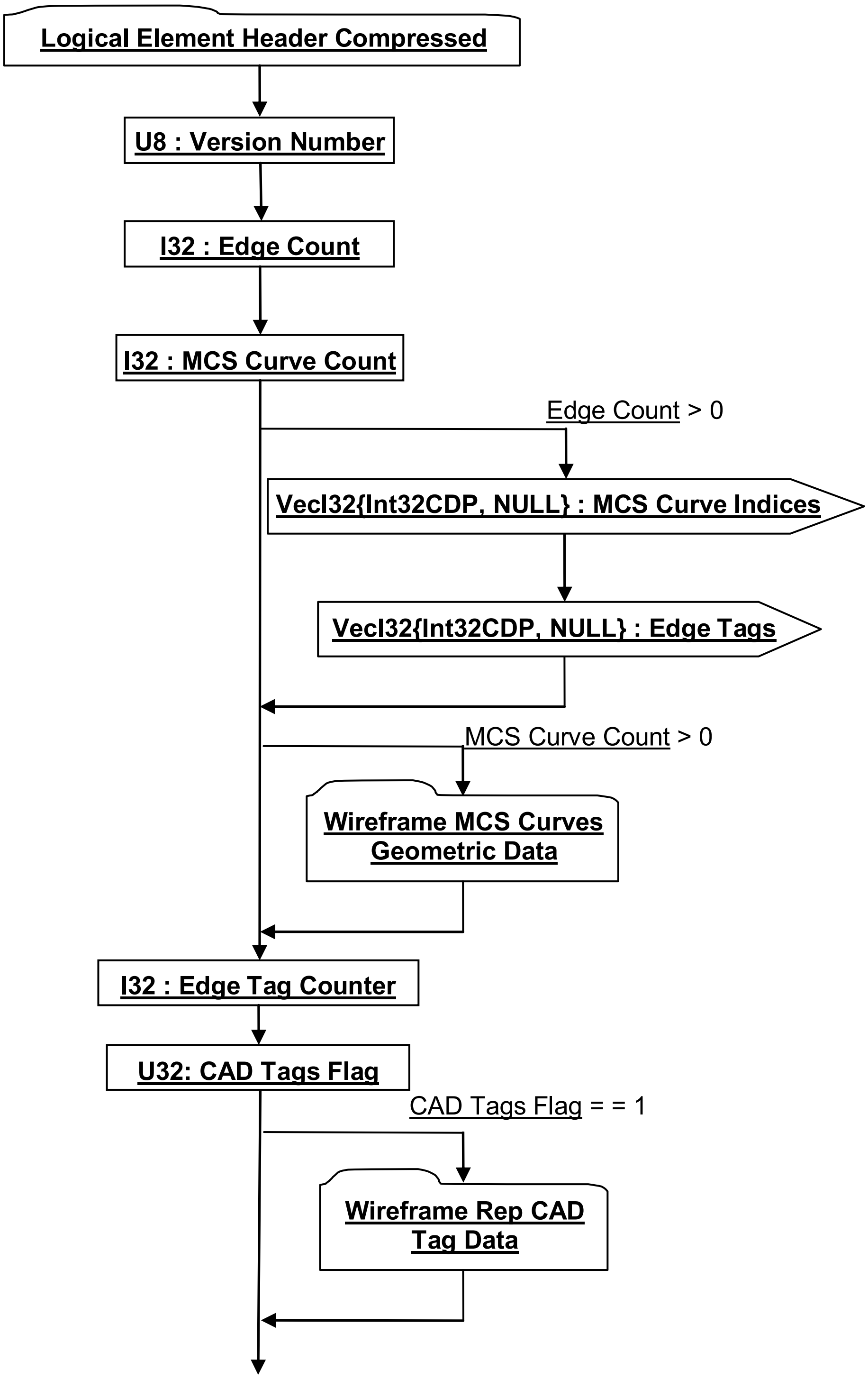

Logical Element Header Compressed
U8 : Version Number
I32 : Edge Count
I32 : MCS Curve Count
Edge Count > 0
VecI32{Int32CDP, NULL} : MCS Curve Indices
VecI32{Int32CDP, NULL} : Edge Tags
MCS Curve Count > 0
Wireframe MCS Curves Geometric Data
I32 : Edge Tag Counter
U32: CAD Tags Flag
CAD Tags Flag = = 1
Wireframe Rep CAD Tag Data

Figure 133 — Wireframe Rep Element data collection

Complete description for Logical Element Header Compressed can be found in_Logical Element Header Compressed.

U8 : Version Number

Version Number is the version identifier for this JT Wireframe Rep Element. For informaiton on local version numbers see best practice 14.5 Local version numbers

I32 : Edge Count

Edge Count indicates the number of topological Edge entities in the Wireframe Rep

I32 : MCS Curve Count

MCS Curve Count indicates the number of distinct geometric (Model Coordinate Space) curves (i.e. XYZ curve) entities in the Wireframe Rep.

VecI32{Int32CDP, NULL} : MCS Curve Indices

MCS Curve Indices is a vector of indices representing the index of the MCS Curve (Model Space curve) for each Edge. MCS Curve Indices uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : Edge Tags

Each Edge has an identifier Tag. Edge Tags is a vector of identifier Tags for a set of Edges. Edge Tags uses the Int32 version of the CODEC to compress and encode data.

I32 : Edge Tag Counter

Edge Tag Counter specifies the next available —unique‘ tag value for Edge entity.

U32: CAD Tags Flag

CAD Tags Flag is a flag indicating whether CAD Tag data exist for the Wireframe Rep.

10.1.1 Wireframe MCS Curves Geometric Data

Wireframe MCS Curves Geometric Data collection contains the Wireframe Rep‘s Model Coordinate System geometric Curve data (i.e. XYZ Curve data). Currently only NURBS Curve types are supported within a Wireframe Rep. The count/number of MCS Curves within a Wireframe Rep is indicated by data field MCS Curve Count documented in 10.1 Wireframe Rep Element.

Figure 134 — Wireframe MCS Curves Geometric Data collection

Complete description for Compressed Curve Data can be found in 13.1.15 Compressed Curve Data.

10.1.2 Wireframe Rep CAD Tag Data

The Wireframe Rep CAD Tag Data collection contains the list of persistent IDs, as defined in the CAD System, to uniquely identify individual Edges in the Wireframe Rep. The existence of this Wireframe Rep CAD Tag Data collection is dependent upon the value of previously read data field CAD Tags Flag as documented in 10.1 Wireframe Rep Element.

If Wireframe Rep CAD Tag Data collection is present, there will be a CAD Tag for every Edge in the Wireframe Rep. Therefore the total number of CAD Tags in the list should be equal to —Edge Count” as documented in 10.1 Wireframe Rep Element.

Figure 135 — Wireframe Rep CAD Tag Data collection

Complete description for Compressed CAD Tag Data can be found in 13.1.16 Compressed CAD Tag Data.

11 Meta Data Segment

Meta Data Segments are used to store large collections of meta-data in separate addressable segments of the JT File. Storing meta-data in a separate addressable segment allows references (from within the JT file) to these segments to be constructed such that the meta-data can be late-loaded (i.e. JT file reader can be structured to support the ―best practice" of delaying the loading/reading of the referenced meta-data segment until it is actually needed).

Meta Data Segments are typically referenced by Part Node Elements (see 6.1.1.5 Part Node Element) using Late Loaded Property Atom Elements (see 6.2.7 Late Loaded Property Atom Element).

The Meta Data Segment type supports compression on all element data, so all elements in Meta Data Segment use the Logical Element Header Compressed form of element header data.

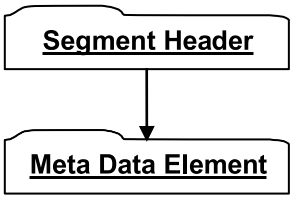


Figure 136 — Meta Data Segment data collection

Complete description for Segment Header can be found in 5.1.3.1 Segment Header.

The following sub-sections document the various Meta Data Element types.

11.1 Property Proxy Meta Data Element

Object Type ID: 0xce357247, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1

A Property Proxy Meta Data Element serves as a ―proxy" for all meta-data properties associated with a particular Meta Data Node Element (see 6.1.1.6 Meta Data Node Element). The proxy is in the form of a list of key/value property pairs where the *key* identifies the type and meaning of the *value*. Although the property *key* is always in the form of a String data type, the *value* can be one of several data types.

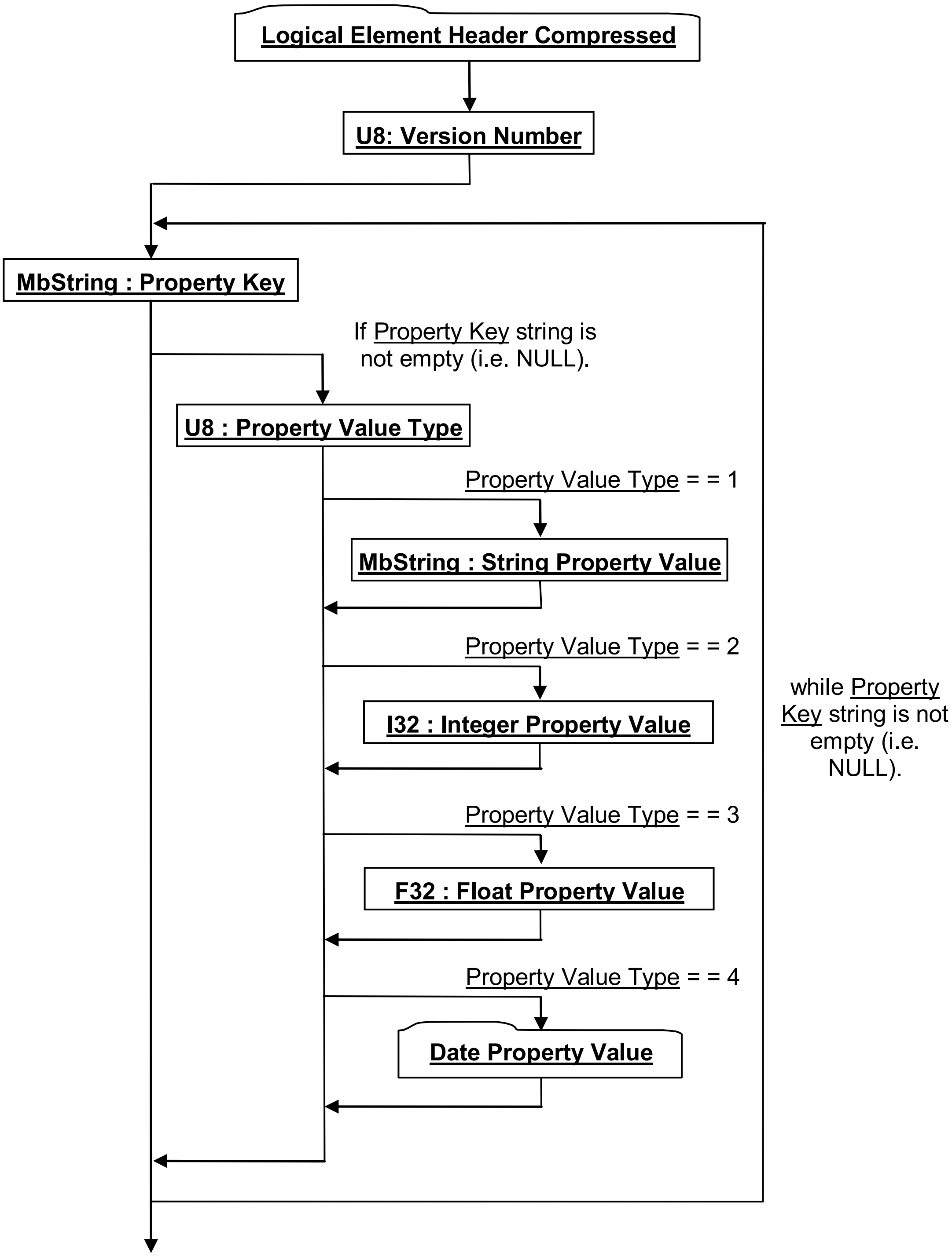

Logical Element Header Compressed
U8: Version Number
MbString : Property Key
If Property Key string is not empty (i.e. NULL).
U8 : Property Value Type
Property Value Type = = 1
MbString : String Property Value
Property Value Type = = 2
I32 : Integer Property Value
Property Value Type = = 3
F32 : Float Property Value
Property Value Type = = 4
Date Property Value
while Property Key string is not empty (i.e. NULL).

Figure 137 —Property Proxy Meta Data Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8: Version Number

Version Number is the version identifier for this data collection. For informaiton on local version numbers see best practice 14.5 Local version numbers

MbString : Property Key

Property Key specifies the *key* string for the property.

U8 : Property Value Type

Property Value Type specifies the data type for the Property Value. If the type equals ―0‖ then no Property Value is written. Valid types include the following:

Table 63 — Property Proxy Meta Data Property Value Type values

= 0	Unknown
= 1	MbString data type value
= 2	I32 data type value
= 3	F32 data type value
= 4	Date value

MbString : String Property Value

String Property Value represents the property value when Property Value Type = = 1.

I32 : Integer Property Value

Integer Property Value represents the property value when Property Value Type = = 2.

F32 : Float Property Value

Float Property Value represents the property value when Property Value Type = = 3.

11.1.1 Date Property Value

Date Property Value represents the property value when Property Value Type = = 4. Date Property Value data collection represents a date as a combination of year, month, day, hour, minute, and second data fields.

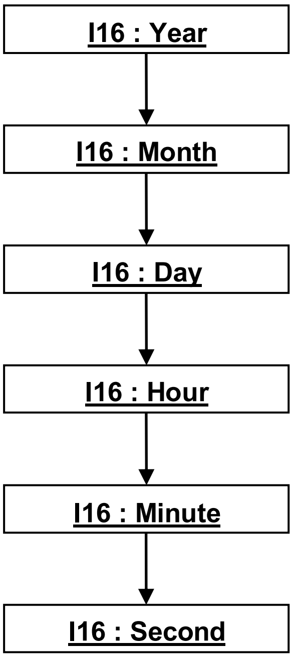


Figure 138 — Date Property Value data collection

I16 : Year

Year specifies the date year value.

I16 : Month

Month specifies the date month value.

I16 : Day

Day specifies the date day value.

I16 : Hour

Hour specifies the date hour value.

I16 : Minute

Minute specifies the date minute value.

I16 : Second

Second specifies the date Second value.

11.2 PMI Manager Meta Data Element

Object Type ID: 0xce357249, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1

The PMI Manager Meta Data Element data collection is a type of Meta Data Element which contains the Product and Manufacturing Information for a part/assembly.

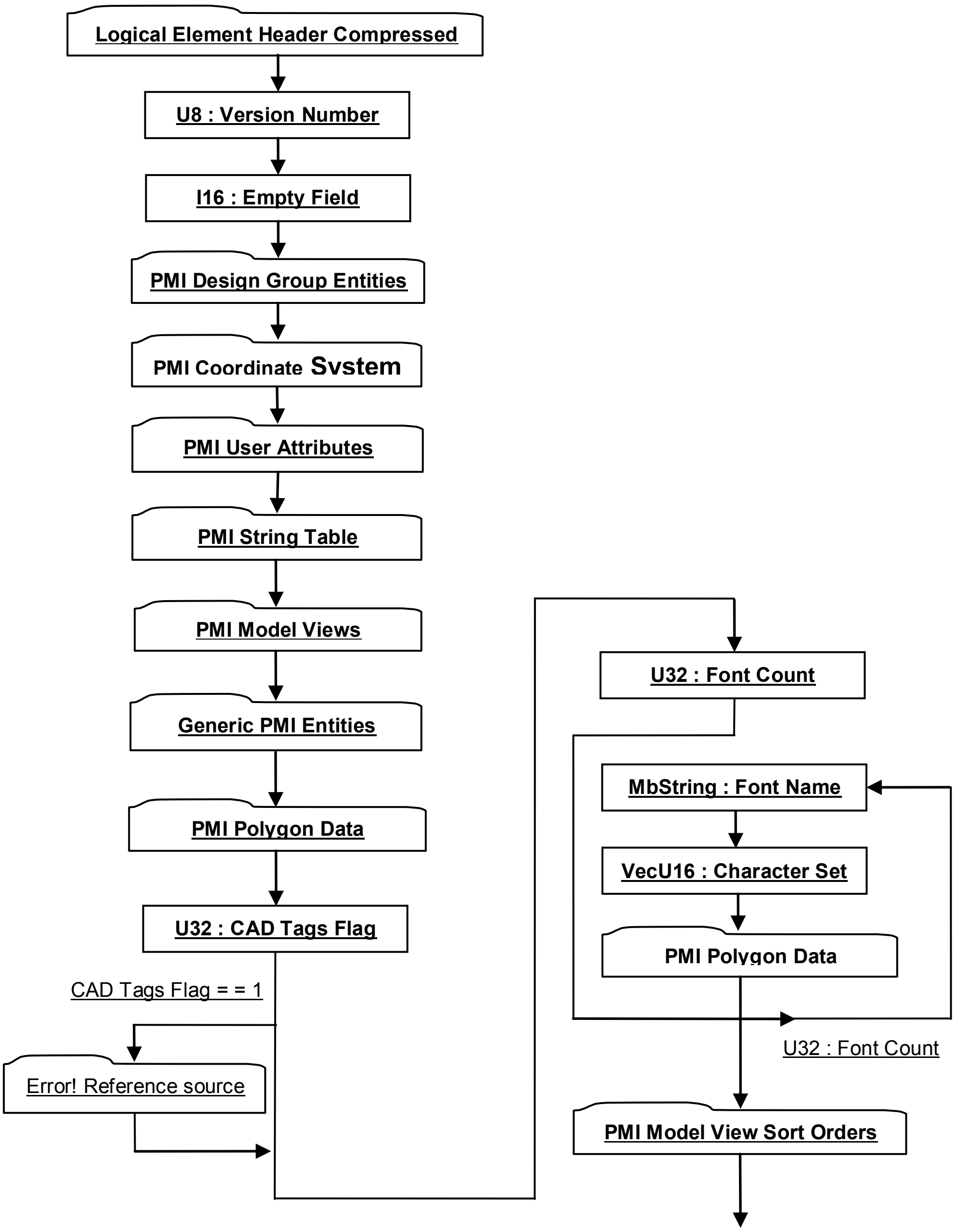


Figure 139 — PMI Manager Meta Data Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8 : Version Number

Version Number is the version identifier for this PMI Manager Element. For information on local version numbers see best practice 14.5 Local version numbers.

I16 : Empty Field

Refer to best practice 14.4 Empty Field.

U32 : CAD Tags Flag

CAD Tags Flag is a flag indicating whether CAD Tag data exist for the PMI.

U32 : Font Count

U32 : Font Count specifies the number of sets of glyph definitions. Each set of glyphs represents a single font definition that consists of a name, a character set and polygonal glyph definition for each character in the set.

MbString : Font Name

Font name specifies a representative name for the font set.

VecU16 : Character Set

VecU16 : Character Set contains the unsigned 16-bit integer identifiers for each character whose symbol is defined in the ensuing PolygonData segment.

11.2.1 PMI Entities

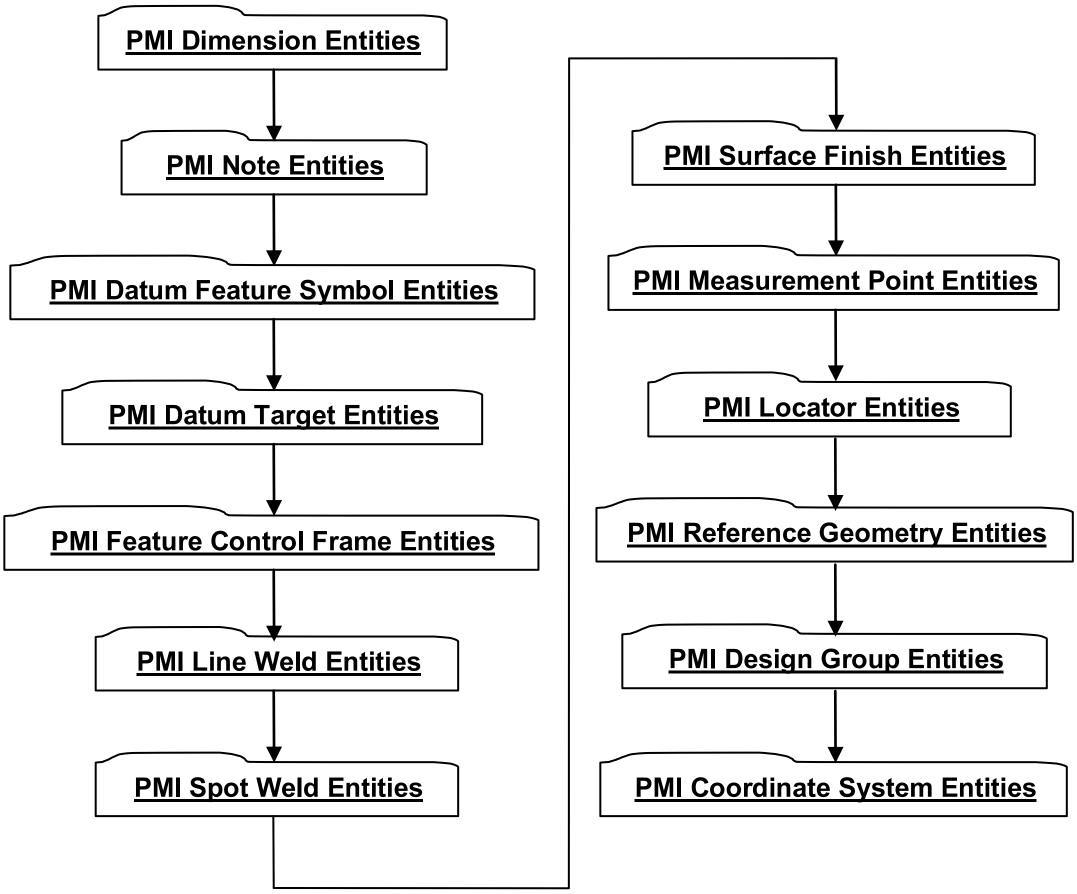

PMI Dimension Entities
PMI Note Entities
PMI Datum Feature Symbol Entities
PMI Datum Target Entities
PMI Feature Control Frame Entities
PMI Line Weld Entities
PMI Spot Weld Entities
PMI Surface Finish Entities
PMI Measurement Point Entities
PMI Locator Entities
PMI Reference Geometry Entities
PMI Design Group Entities
PMI Coordinate System Entities

Figure 140 — PMI data collection

11.2.1.1 PMI Dimension Entities

The PMI Dimension Entities data collection defines data for a list of Dimensions.

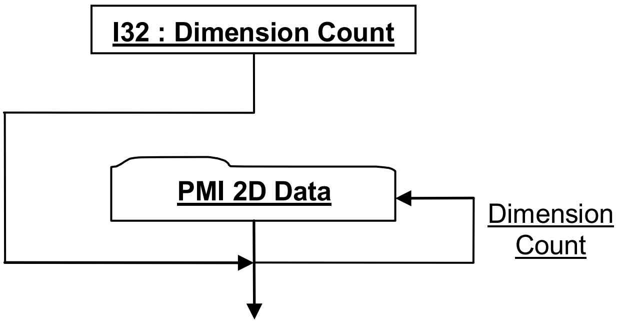


Figure 141 — PMI Dimension Entities data collection

I32 : Dimension Count

Dimension Count specifies the number of Dimension entities.

11.2.1.1.1 PMI 2D Data

The PMI 2D Data collection defines a data format common to all 2D based PMI entities.

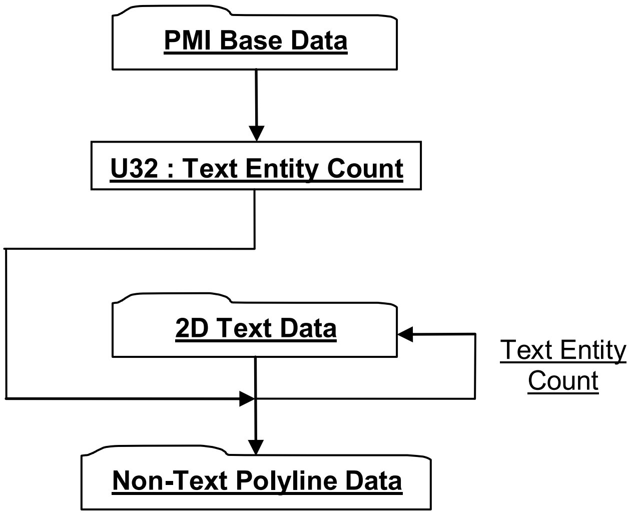


Figure 142 — PMI 2D Data collection

I32 : Text Entity Count

Text Entity Count specifies the number of Text entities in the particular PMI entity.

PMI Base Data

The PMI Base Data collection defines the basic/common data that every 2D and 3D PMI entity contains.

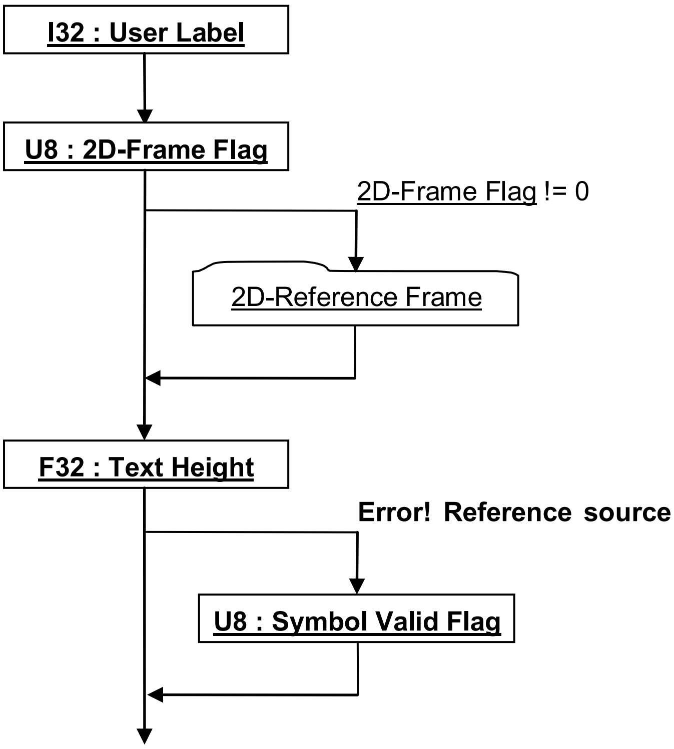


Figure 143 — PMI Base Data collection

I32 : User Label

User Label specifies the particular PMI entity identifier.

U8 : 2D-Frame Flag

2D-Frame Flag is a flag specifying whether 2D-Reference Frame data is stored. If 2D-Frame Flag has a non-zero value then 2D-Reference Frame data is included. If 2D-Frame Flag has a value of —2”, then dummy (i.e. all zeros) 2D-Reference Frame data is written. The -2D-Frame Flag = = 2” case is used by 11.2.8 Generic PMI Entities because for Generic PMI Entities all the Non-Text Polyline Data is already in 3D form (i.e. XYZ coordinate data).

F32 : Text Height

Text Height specifies the PMI text height in WCS.

U8 : Symbol Valid Flag

Symbol Valid Flag is a flag specifying whether the particular PMI entity is valid. If Symbol Valid Flag has a non-zero value then PMI entity is valid. This flag is only stored if the Version Number as defined in 11.2 PMI Manager Meta Data Element is greater than —4”.

2D-Reference Frame

The 2D-Reference Frame data collection defines a reference frame (2D coordinate system) where the PMI entity is displayed in 3D space. All the PMI entity‘s 2D and 3D polyline data is assumed to lie on the defined plane.

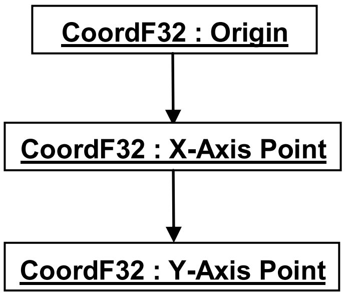


Figure 144 — 2D-Reference Frame data collection

CoordF32 : Origin

Origin defines the origin (min-corner) of the 2D coordinate system

CoordF32 : X-Axis Point

X-Axis Point defines a point along the X-Axis of the 2D coordinate system.

CoordF32 : Y-Axis Point

Y-Axis Point defines a point along the Y-Axis of the 2D coordinate system.

2D Text Data

The 2D Text Data collection defines a 2D text entity/primitive.

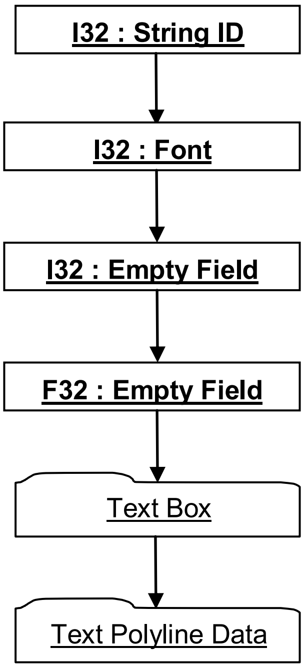


Figure 145 — 2D Text Data collection

I32 : String ID

String ID specifies the identifier for the character string. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of —1" indicates no string.

I32 : Font

Font identifies the font to be used for this text. Valid values include the following:

Table 64 — PMI 2D Base Data Font values

= 1	Simplex
= 2	Din
= 3	Military
= 4	ISO
= 5	Lightline
= 6	IGES 1001
= 7	Century
= 8	IGES 1002
= 9	IGES 1003
= 101	Japanese JISX 0208 coded character set
= 102	Japanese Extended Unix Codes JISX 0208 coded character set
= 103	Chinese GB 2312.1980 Simplified coded character set
= 104	Korean KSC 5601 coded character set
= 105	Chinese Big5 Traditional coded character set

I32 : Empty Field

Refer to best practice 14.4 Empty Field

F32 : Empty Field

Refer to best practice 14.4 Empty Field

Text Box

The Text Box data collection specifies a 2D box that particular text fits within. All values are with respect to 2D-Reference Frame documented in 0 2D-Reference Frame.

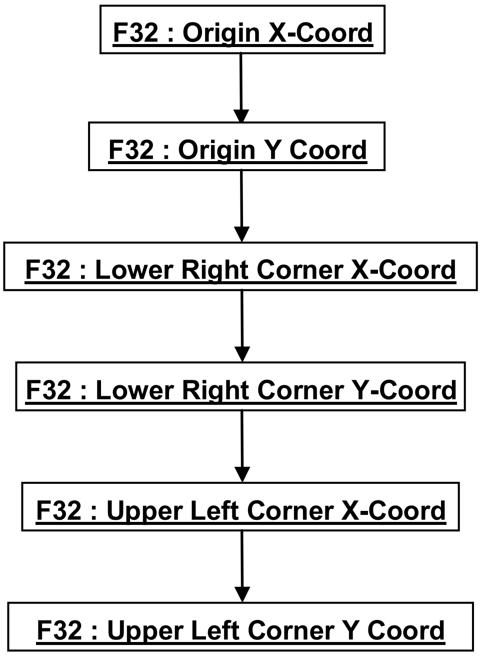

F32 : Origin X-Coord
F32 : Origin Y Coord
F32 : Lower Right Corner X-Coord
F32 : Lower Right Corner Y-Coord
F32 : Upper Left Corner X-Coord
F32 : Upper Left Corner Y Coord

Figure 146 — Text Box data collection

F32 : Origin X-Coord

Origin X-Coord defines the 2D X-coordinate of the text origin with respect to 2D-Reference Frame.

F32 : Origin Y Coord

Origin Y-Coord defines the 2D Y-coordinate of the text origin with respect to 2D-Reference Frame.

F32 : Lower Right Corner X-Coord

Lower Right Corner X-Coord defines the 2D X-coordinate of the lower right corner of the text with respect to 2D-Reference Frame.

F32 : Lower Right Corner Y-Coord

Lower Right Corner Y-Coord defines the 2D Y-coordinate of the lower right corner of the text with respect to 2D-Reference Frame.

F32 : Upper Left Corner X-Coord

Upper Left Corner X-Coord defines the 2D X-coordinate of the upper left corner of the text with respect to 2D-Reference Frame.

F32 : Upper Left Corner Y Coord

Upper Left Corner Y-Coord defines the 2D Y-coordinate of the upper left corner of the text with respect to 2D-Reference Frame.

Text Polyline Data

The Text Polyline Data collection defines any polyline segments which are part of the text representation. This existence of this polyline data is conditional (i.e. not all text has it) and is made up of an array of indices into an array of polyline segments packed as 2D vertex coordinates, specifying where each polyline segment begins and ends. Polylines are constructed from these arrays of data as follows:

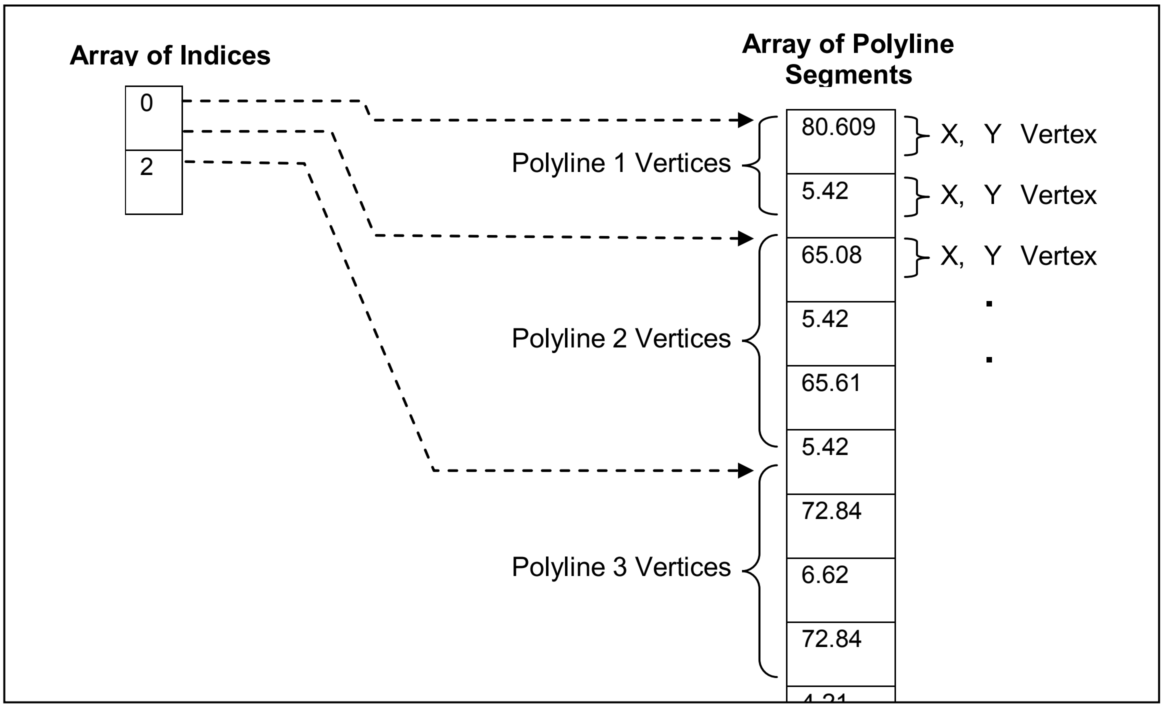


Figure 147 — Constructing Text Polylines from data arrays

This data is represented in JT file in the following format:

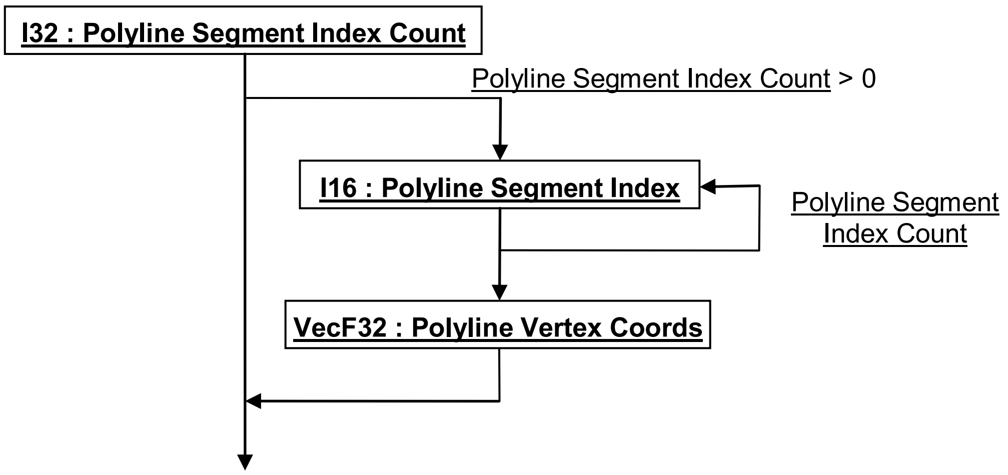

I32 : Polyline Segment Index Count
Polyline Segment Index Count > 0
I16 : Polyline Segment Index
Polyline Segment Index Count
VecF32 : Polyline Vertex Coords

Figure 148 — Text Polyline Data collection

I32 : Polyline Segment Index Count

Polyline Segment Index Count specifies the number of polyline segment indices.

I16 : Polyline Segment Index

Polyline Segment Index is an index into the Polyline Vertex Coords array specifying where polyline segment begins or ends. This index is a vertex coordinate index so the absolute index into the Polyline Vertex Coords array is computed by multiplying the index value by —”2(i.e. for 2D coordinates).

VecF32 : Polyline Vertex Coords

Polyline Vertex Coords is an array of polyline segments packed as 2D point coordinates. These 2D point coordinates are with respect to the 2D-Reference Frame documented in 0 2D-Reference Frame.

Non-Text Polyline Data

The Non-Text Polyline Data collection contains all the non-text polylines making up the particular PMI entity. Examples of non-text polylines include line attachments, text boxes, symbol box dividers, etc. The Non-Text Polyline Data collection is made up of an array of indices into an array of polyline segments packed as either 2D or 3D vertex coordinates, specifying where each polyline segment begins and ends. Whether vertex coordinates are 2D or 3D is dependent upon the PMI entity type using this data collection. If it is a 11.2.8 Generic PMI Entities type then the packed coordinate data is 3D; for all other PMI entity types the packed coordinate data is 2D. Also for Version Number, as defined in 11.2 PMI Manager Meta Data Element, greater than —”4an array of values that sequentially specify the polyline type in the polyline segments array is included.

Figure 180 below shows how Polylines are constructed from these arrays of data for the packed 2D coordinates case. The packed 3D coordinates case is interpreted the same except that the coordinates array includes a Z component and is thus packed as —X[YZ][XYZ][XYZ]…”

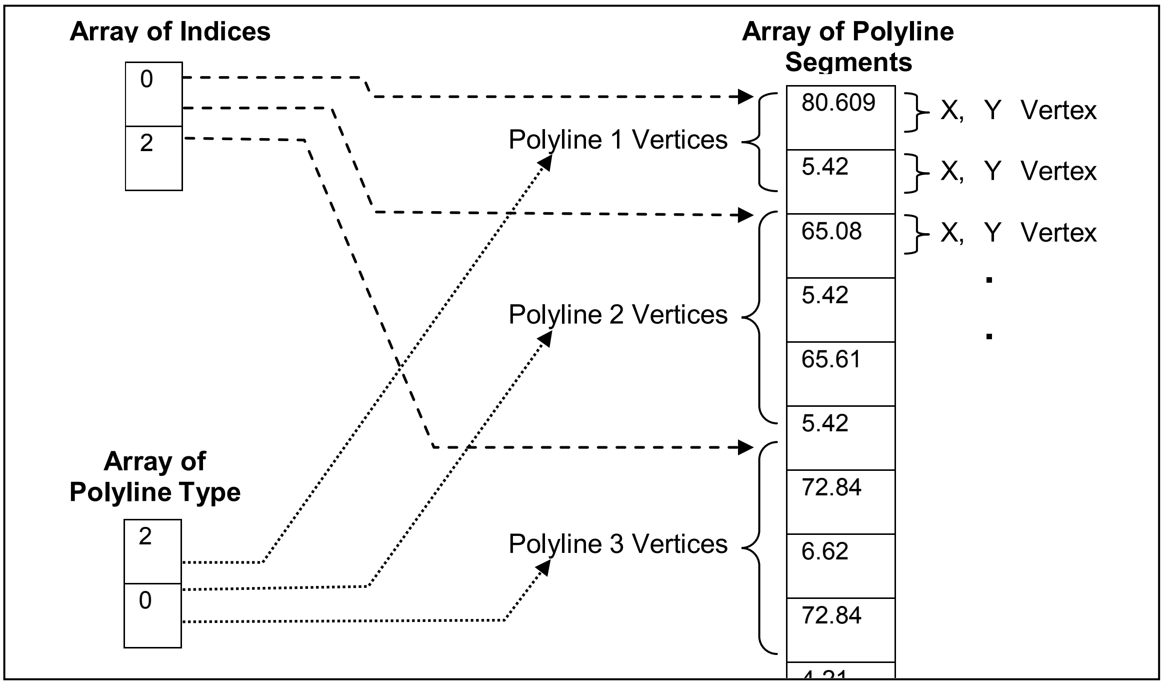


Figure 149 — Constructing Non-Text Polylines from packed 2D data arrays

This data is represented in the JT format as follows:

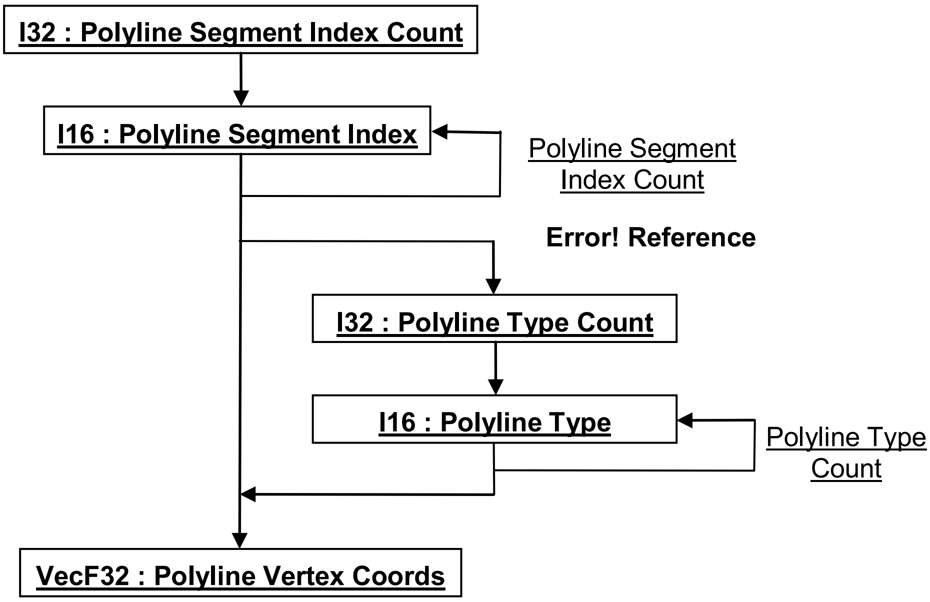

I32 : Polyline Segment Index Count
I16 : Polyline Segment Index
Polyline Segment Index Count
Error! Reference
I32 : Polyline Type Count
I16 : Polyline Type
Polyline Type Count
VecF32 : Polyline Vertex Coords

Figure 150 — Non-Text Polyline Data collection

I32 : Polyline Segment Index Count

Polyline Segment Index Count specifies the number of polyline segment indices.

I16 : Polyline Segment Index

Polyline Segment Index is an index into the Polyline Vertex Coords array specifying where polyline segment begins or ends. This index is a vertex/coordinate index so the absolute index into the Polyline Vertex Coords array is computed by multiplying the index value by —2"(i.e. for 2D coordinates).

I32 : Polyline Type Count

Polyline Type Count specifies the number of polyline type values.

I16 : Polyline Type

Polyline Type specifies the type of polyline segment in Polyline Vertex Coords array. See Figure 180 — Constructing Non-Text Polylines from packed 2D data arrays for interpretation of this array of type values relative to the defined polylines. Valid values include the following:

Table 65 — PMI 2D Non-Text Polyline Type values

= 0	General line
= 1	General arrow
= 2	General circle
= 3	General arc
= 4	Extended line 1
= 5	Extended line 2
= 6	Extended arc
= 7	Extended circle
= 8	Text line (used in text boxes and symbol box dividers)
= 9	Text string

VecF32 : Polyline Vertex Coords

Polyline Vertex Coords is an array of polyline segments packed as 2D point coordinates. These 2D point coordinates are with respect to the 2D-Reference Frame documented in 0 2D-Reference Frame.

11.2.1.2 PMI Note Entities

The PMI Note Entities data collection defines data for a list of Notes. Notes are used to connect textual information to specific Part entities.

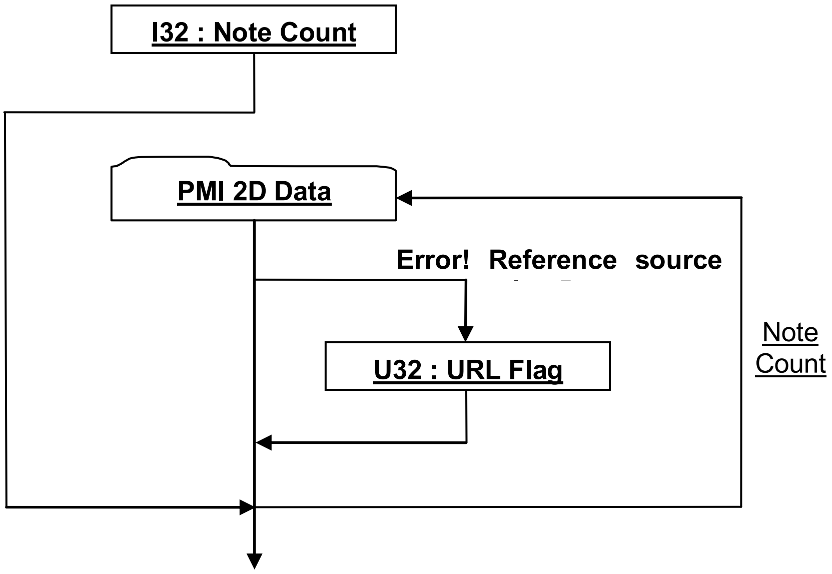


Figure 151 — PMI Note Entities data collection

Complete description for PMI 2D Data can be found in 11.2.8.1 PMI 2D Data.

I32 : Note Count

Note Count specifies the number of Note entities.

U32 : URL Flag

URL Flag specifies whether Note is an URL. This data field is only present if Version Number, as defined in 11.2 PMI Manager Meta Data Element, is greater than —’5 The URL is the actual text of the note as specified in PMI 2D Data.

11.2.1.3 PMI Datum Feature Symbol Entities

The PMI Datum Feature Symbol Entities data collection defines data for a list of Datum Feature Symbols. A Datum Feature Symbol is a Geometric Dimensioning and Tolerancing (GD&T) symbol that provides a ―label" for a part feature which is referenced by a Feature Control Frame.

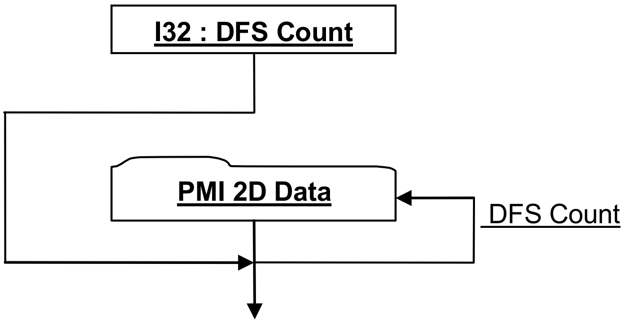


Figure 152 — PMI Datum Feature Symbol Entities data collection

Complete description for PMI 2D Data can be found in 11.2.8.1 PMI 2D Data.

I32 : DFS Count

DFS Count specifies the number of Datum Feature Symbol entities.

11.2.1.4 PMI Datum Target Entities

The PMI Datum Target Entities data collection defines data for a list of Datum Targets. A Datum Target is a Geometric Dimensioning and Tolerancing (GD&T) symbol that specifies a point, a line, or an area on a part to define a ―datum" for manufacturing and inspection operations.

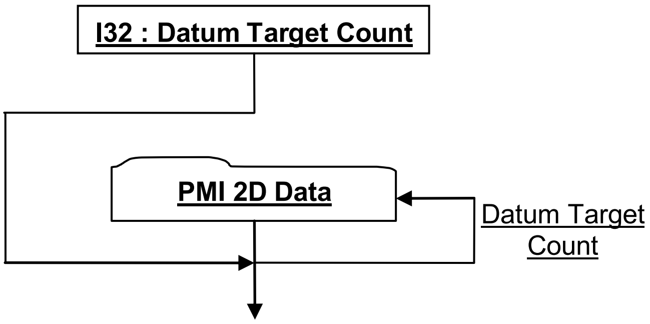


Figure 153 — PMI Datum Target Entities data collection

Complete description for PMI 2D Data can be found in 11.2.8.1 PMI 2D Data.

I32 : Datum Target Count

Datum Target Count specifies the number of Datum Target entities.

11.2.1.5 PMI Feature Control Frame Entities

The PMI Feature Control Frame Entities data collection defines data for a list of Feature Control Frames. A Feature Control Frame is a Geometric Dimensioning and Tolerancing (GD&T) symbol used for expressing the geometric characteristics, form tolerance, runout or location tolerance, and relationships between the geometric features of a part. If necessary, Datum Feature and/or Datum Target references may be included in the Feature Control Frame symbol.

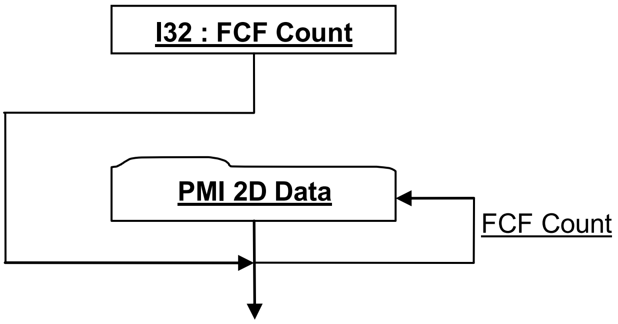


Figure 154 — PMI Feature Control Frame Entities data collection

Complete description for PMI 2D Data can be found in 11.2.8.1 PMI 2D Data.

I32 : FCF Count

FCF Count specifies the number of Feature Control Frame entities.

11.2.1.6 PMI Line Weld Entities

The PMI Line Weld Entities data collection defines data for a list of Line Weld symbols. A Line Weld symbol describes the specifications for welding a joint.

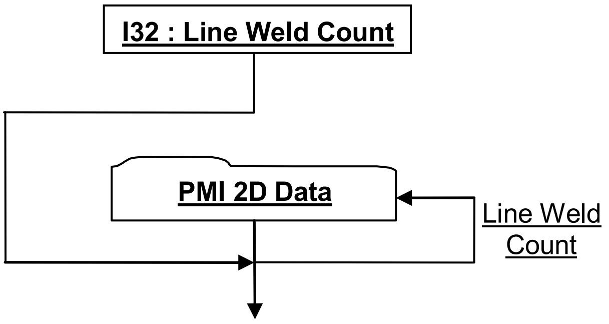


Figure 155 — PMI Line Weld Entities data collection

Complete description for PMI 2D Data can be found in 11.2.8.1 PMI 2D Data.

I32 : Line Weld Count

Line Weld Count specifies the number of Line Weld entities.

11.2.1.7 PMI Spot Weld Entities

The PMI Spot Weld Entities data collection defines data for a list of Spot Weld Symbols. Spot Weld symbols describe the specifications for welding sheet metal.

Several data fields of the PMI Spot Weld Entities data collection are only present if Version Number, as defined in 11.2 PMI Manager Meta Data Element, is greater than or equal to ―4‖

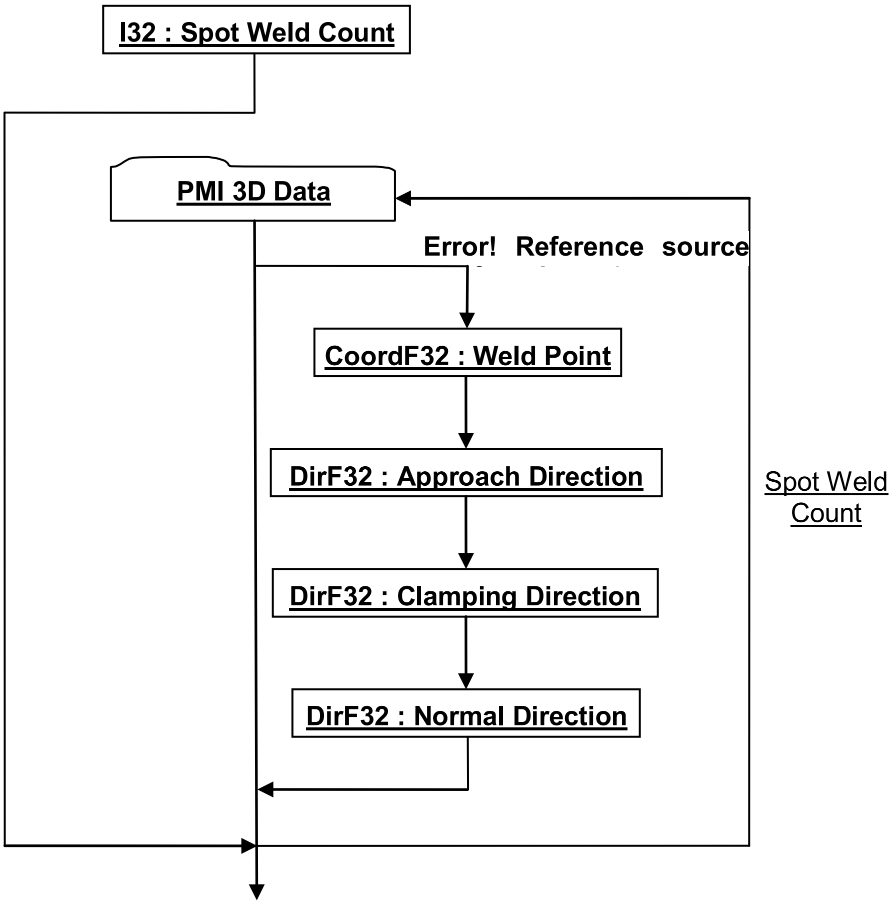


Figure 156 — PMI Spot Weld Entities data collection

I32 : Spot Weld Count

Spot Weld Count specifies the number of Spot Weld entities.

CoordF32 : Weld Point

Weld Point specifies the coordinates of the weld point.

DirF32 : Approach Direction

Approach Direction specifies the components of the direction vector from which the weld gun approaches the part.

DirF32 : Clamping Direction

Clamping Direction specifies the components of the clamping force direction vector.

DirF32 : Normal Direction

Normal Direction specifies the components of the direction vector normal to the actual spot weld.

11.2.1.7.1 PMI 3D Data

The PMI 3D Data collection defines a data format common to all 3D based PMI entities.

Along with the PMI Base Data and String identifier, this data collection also includes non-text polyline data defined by an array of indices into an array of polyline segments packed as 2D/3D vertex coordinates, specifying where each polyline segment begins and ends. How polylines are constructed from this index array and packed vertex coordinates array is the same as that illustrated in Figure 178 of 0 Text Polyline Data.

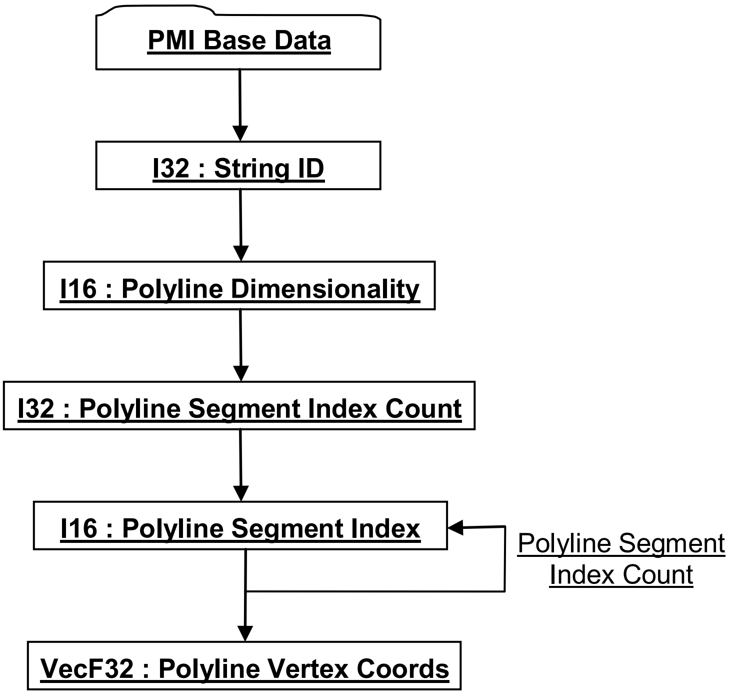

PMI Base Data
I32 : String ID
I16 : Polyline Dimensionality
I32 : Polyline Segment Index Count
I16 : Polyline Segment Index
Polyline Segment Index Count
VecF32 : Polyline Vertex Coords

Figure 157 — PMI 3D Data collection

Complete description for PMI Base Data can be found in PMI Base Data.

I32 : String ID

String ID specifies the identifier for the character string. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―1" indicates no string.

I16 : Polyline Dimensionality

Polyline Dimensionality specifies the dimensionality of the polyline coordinates packed in Polyline Vertex Coords. Valid values include the following:

Table 66 — PMI 3D Data Polyline Dimensionality values

= 2	Indicates 2-dimensioanl (xyxy…) data packing..
= 3	Indicates 3-dimensional (xyzxyz…) data packing.

I32 : Polyline Segment Index Count

Polyline Segment Index Count specifies the number of polyline segment indices.

I16 : Polyline Segment Index

Polyline Segment Index is an index into the Polyline Vertex Coords array specifying where polyline segment begins or ends. This index is a vertex coordinate index so the absolute index into the Polyline Vertex Coords array is computed by multiplying the index value by Polyline Dimensionality.

VecF32 : Polyline Vertex Coords

Polyline Vertex Coords is an array of polyline segments packed as Polyline Dimensionality point coordinates.

11.2.1.8 PMI Surface Finish Entities

The PMI Surface Finish Entities data collection defines data for a list of Surface Finish symbols. Surface Finish symbols indicate surface quality and generally are only specified where finish quality affects function (e.g. bearings, pistons, gears).

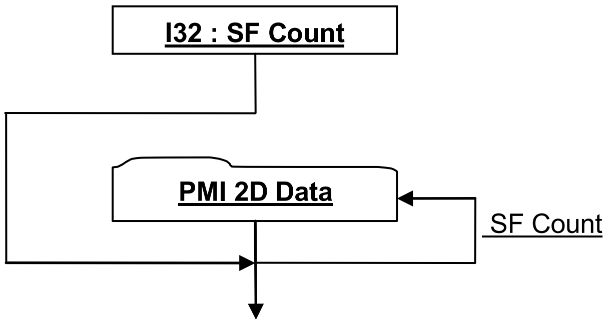


Figure 158 — PMI Surface Finish Entities data collection

Complete description for PMI 2D Data can be found in 11.2.8.1 PMI 2D Data.

I32 : SF Count

SF Count specifies the number of Surface Finish symbol entities.

11.2.1.9 PMI Measurement Point Entities

The PMI Measurement Point Entities data collection defines data for a list of Measurement Point symbols. Measurement Points are predefined locations (i.e. geometric entities or theoretical, but measurable points, such as surface locations) which are measured on manufactured parts to verify the accuracy of the manufacturing process.

Several data fields of the PMI Measurement Point Entities data collection are only present if Version Number, as defined in 11.2 PMI Manager Meta Data Element, is greater than or equal to —”4

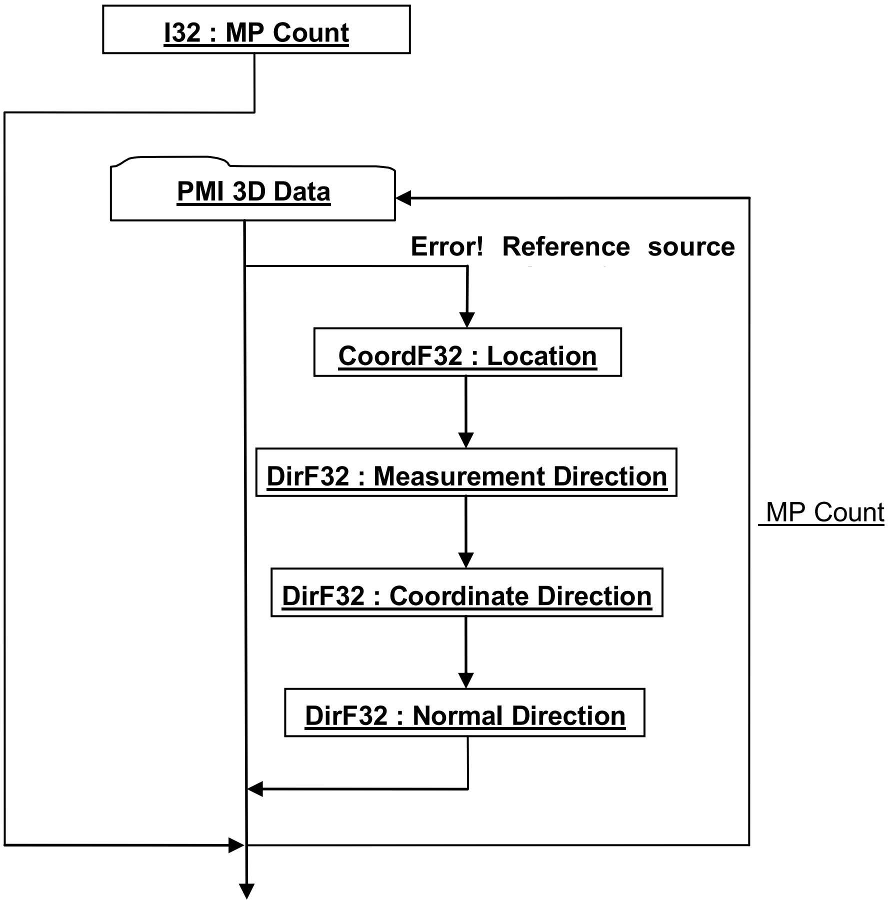

I32 : MP Count
PMI 3D Data
Error! Reference source
CoordF32 : Location
DirF32 : Measurement Direction
DirF32 : Coordinate Direction
DirF32 : Normal Direction
MP Count

Figure 159 — PMI Measurement Point Entities data collection

Complete description for PMI 3D Data can be found in 11.2.1.7.1 PMI 3D Data.

I32 : MP Count

MP Count specifies the number of Measurement Point entities.

CoordF32 : Location

Location specifies the coordinates of the Measurement Point.

DirF32 : Measurement Direction

Measurement Direction specifies the components of the direction vector from which a CCM (Coordinate Measuring Machine) approaches when taking a measurement.

DirF32 : Coordinate Direction

Coordinate Direction specifies the components of the direction vector another Measurement Point on a mating part would like to align with a Measurement Point on the first part.

DirF32 : Normal Direction

Normal Direction specifies the components of the direction vector normal to the actual Measurement Point.

11.2.1.10 PMI Locator Entities

The PMI Locator Entities data collection defines data for a list of Locator symbols. Locator symbols are used to accurately locate components with respect to each other and the manufacturing tooling.

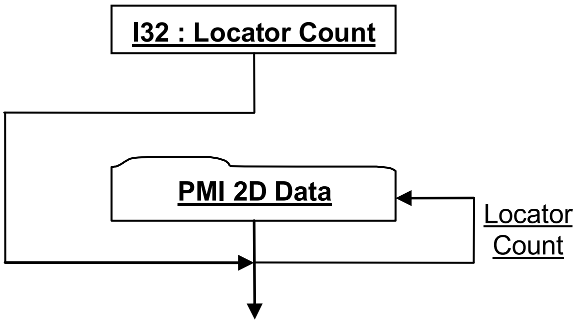


Figure 160 — PMI Locator Entities data collection

Complete description for PMI 2D Data can be found in 11.2.8.1 PMI 2D Data.

I32 : Locator Count

Locator Count specifies the number of Locator symbol entities.

11.2.1.11 PMI Reference Geometry Entities

The PMI Reference Geometry Entities data collection defines data for a list of Reference Geometry. Reference Geometry can be thought of as user-definable datums, which are positioned relative to the topology of an existing entity. Each reference geometry type (point, polyline, polygon) can be implicitly determined by the value of Polyline Segment Index[1] (see 11.2.1.7.1 PMI 3D Data) as follows:

Table 67 — PMI Reference Geometry Entity values

Polyline Segment Index[1]	Implied Reference Geometry Type
= 1	Point
= 2	Polyline
> 2	Polygon

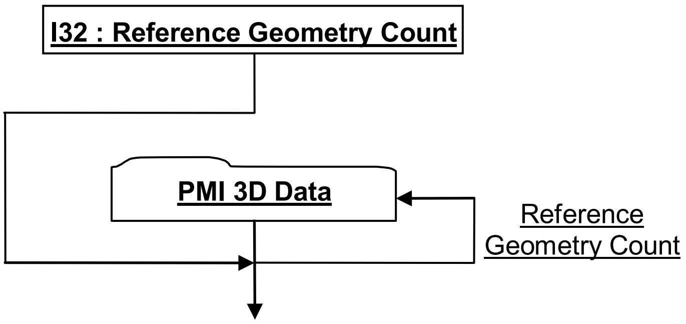


Figure 161 — PMI Reference Geometry Entities data collection

Complete description for PMI 3D Data can be found in 11.2.1.7.1 PMI 3D Data.

I32 : Reference Geometry Count

Reference Geometry Count specifies the number of Reference Geometry entities.

11.2.2 PMI Design Group Entities

The PMI Design Group Entities data collection defines data for a list of Design Groups. Design Groups are collections of PMI created to organize a model into smaller subsets of information. This organization is achieved via PMI Associations (see 11.2.3 PMI Associations), where specific PMI entities are associated as ―destinations" to a ―source" PMI Design Group.

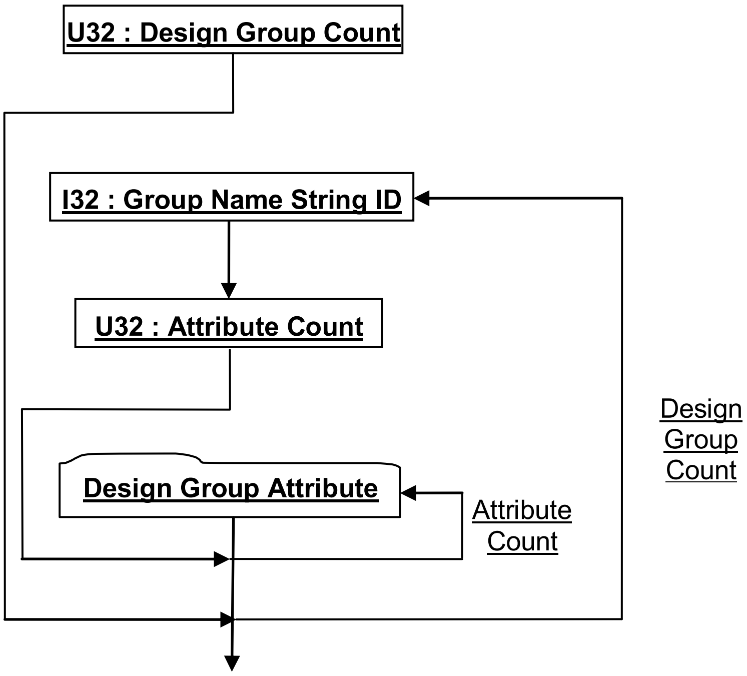


Figure 162 — PMI Design Group Entities data collection

U32 : Design Group Count

Design Group Count specifies the number of Design Group entities.

I32 : Group Name String ID

Group Name String ID specifies the identifier for the group name character string. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―-1" indicates no string.

U32 : Attribute Count

Attribute Count specifies the number of Design Group Attribute data collections

11.2.2.1 Design Group Attribute

The Design Group Attribute data collection defines a group property/attribute.

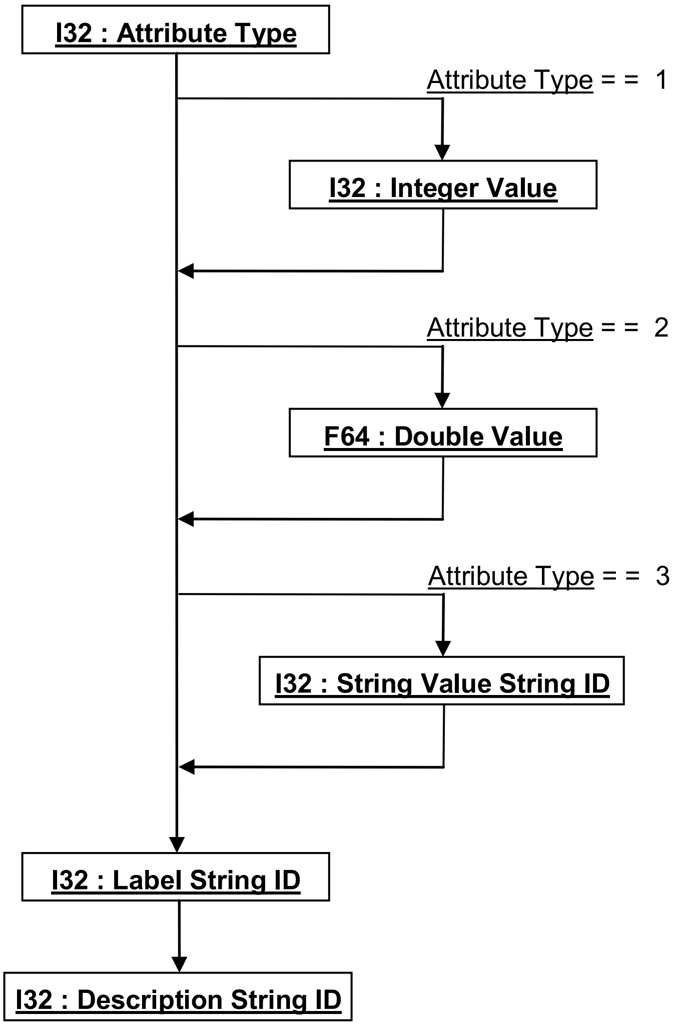


Figure 163 — Design Group Attribute data collection

I32 : Attribute Type

Attribute Type specifies the attribute type. Valid types include the following:

Table 68 — PMI Design Group Attribute Type values

= 1	Integer
= 2	Double
= 3	String

I32 : Integer Value

Integer Value specifies the value for ―integer” Attribute Types.

F64 : Double Value

Double Value specifies the value for ―double” Attribute Types.

I32 : String Value String ID

String Value String ID specifies the string identifier value for ―string" Attribute Types. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―-1" indicates no string.

I32 : Label String ID

Label String ID specifies the string identifier for the attribute label. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―-1" indicates no string.

I32 : Description String ID

Description String ID specifies the string identifier for the attribute description. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―-1" indicates no string.

11.2.2.2 PMI Coordinate System Entities

The PMI Coordinate System Entities data collection defines data for a list of Coordinate Systems.

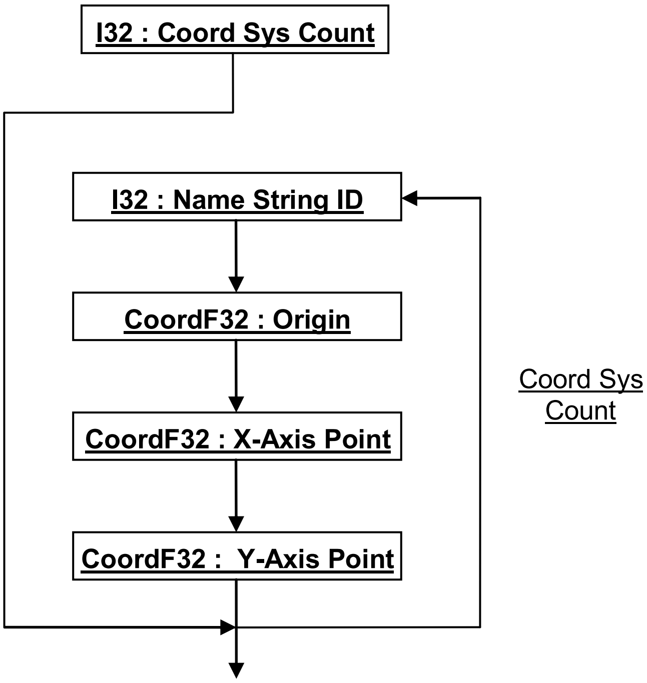


Figure 164 — PMI Coordinate System Entities data collection

I32 : Coord Sys Count

Coord Sys Count specifies the number of Coordinate System entities.

I32 : Name String ID

Name String ID specifies the string identifier for the Coordinate System name. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―-1" indicates no string.

CoordF32 : Origin

Origin defines the origin of the coordinate system.

CoordF32 : X-Axis Point

X-Axis Point defines a point along the X-Axis of the coordinate system.

CoordF32 : Y-Axis Point

Y-Axis Point defines a point along the Y-Axis of the coordinate system.

11.2.3 PMI Associations

The PMI Associations data collection defines data for a list of associations. An association defines a link (—relationship”) between two PMI, B-Rep, or Wireframe Rep entities where one entity is defined as the —source” and the other entity is defined as the —destination”.

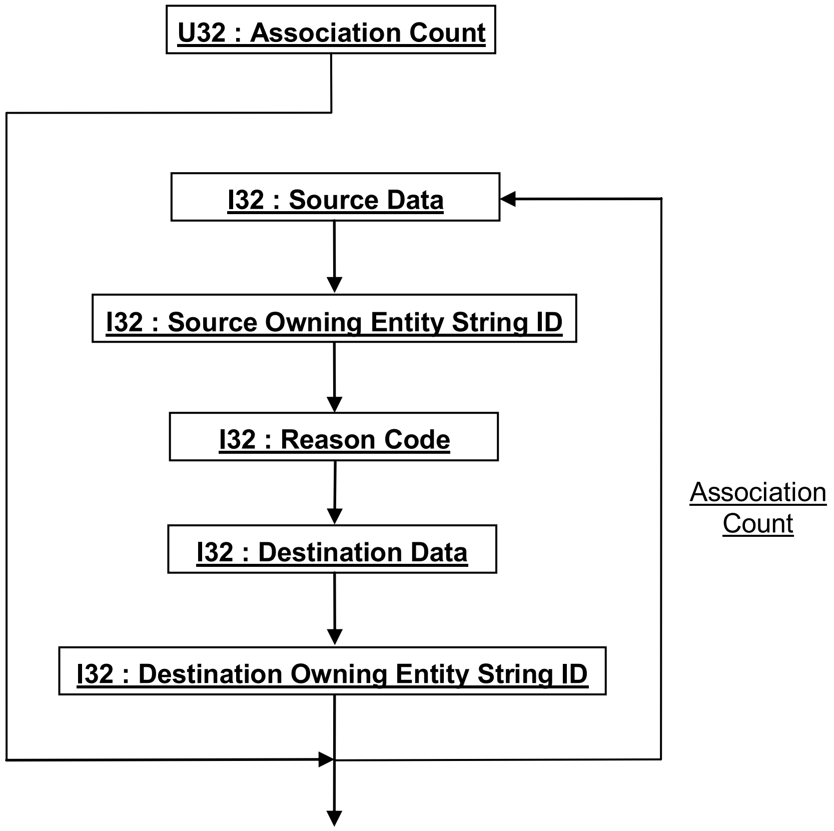


Figure 165 — PMI Associations data collection

U32 : Association Count

Association Count specifies the number of associations.

I32 : Source Data

Source Data is a collection of source entity information encoded/packed within a single I32 using the following bit allocation. All bits fields that are not defined as in use should be set to —0’

Table 69 — PMI Associations Source Data values

Bits 0 - 23	Source Entity Identifier. The interpretation of this identifier data is dependent upon the value of Bit 31 documented below.
Bits 24 -30	Source Entity PMI or B-Rep type. Valid types include the following: = 0 PMI - Dimension = 1 PMI - Note = 2 PMI - Datum Feature Symbol = 3 PMI - Datum Target = 4 PMI - Feature Control Frame = 5 PMI - Line Weld = 6 PMI - Spot Weld = 7 PMI - Measurement Point

	= 8 PMI - Surface Finish = 9 PMI - Locator Designator = 10 PMI - Reference Geometry = 11 PMI - Coordinate System = 12 PMI - Design Group = 13 PMI - User Attribute = 14 B-Rep - Vertex = 15 B-Rep - Edge = 16 B-Rep - Face = 17 PMI - Model View = 18 PMI - Generic = 19 Wireframe Rep - Edge = 20 PMI - Unspecified type = 21 Part Instance
Bit 31	Indirect Identifier Flag = 0 – Value in Bits 0-23 is not the actual CAD identifier, instead Bits 0-23 is an index into the source type‘s PMI array or index of the edge/face in B-Rep or Wireframe Rep for the source entity. = 1 – Value in Bits 0-23 is not the actual CAD identifier; instead Bits 0-23 is an index into the list of CAD Tags (as documented in **Error! Reference source not found. Error! Reference source not found.**) identifying the CAD Tag belonging to the particular source entity.

I32 : Source Owning Entity String ID

Source Owning Entity String ID specifies the string identifier for the string which contains the unique CAD identifier of the component (part or assembly) that owns the source PMI or B-Rep entity. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―1” indicates no string and implies that the entity is to be found on the current node s PMI/B-Rep/Wireframe-Rep segment. It is valid for the source owning entity to be the same as the destination owning entity (i.e. an association between two PMI or B-Rep entities in the same part/assembly).

I32 : Reason Code

Reason Code specifies the ―reason” for the association. Valid Reason Codes include the following:

Table 70 — PMI Associations Reason Code values

= 0	Association is to the primary entity being dimensioned
= 1	Association is to the secondary entity being dimensioned
= 2	Association is to the dimension plane
= 5	Association is to the entity used to specify the Z-Axis of a coordinate system
= 10	Association is to an entity "associated" to or "included in" a PMI symbol
= 11	Association is to an entity used to "attach" a PMI symbol.
= 12	Association is to first entity used to ―attach” a PMI symbol
= 13	Association is to second entity used to ―attach” a PMI symbol
= 14	Specifying PMI grouping, source is PMI/B-Rep entity and destination is design group.
= 15	Association is to a weld line entity
= 16	Association is to a ―hot spot”
= 17	Association is to a child in a PMI stack
= 72	Association is for PMI miscellaneous relation.
= 73	Association is for PMI related entity.
= 98	Association is to show the PMI when associated Model View is selected. Source is the PMI, and destination is Model View.
= 99	Association is to show/select PMI B, if showing/selecting PMI A. Source is PMI A, and destination is PMI B. This is different from an ―attached” PMI , where the convention is to show the PMI visibly linked to one another.

= 100	Association is to show all parts except the associated part instance. Source is the part instance, and destination is Model View

I32 : Destination Data

Destination Data is a collection of destination entity information encoded/packed within a single I32. The encoding schema and interpretation of this data is the same as that documented in Source Data.

I32 : Destination Owning Entity String ID

Destination Owning Entity String ID specifies the string identifier for the string which contains the unique CAD identifier of the component (part or assembly) that owns the destination PMI or B-Rep entity. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―1” indicates no string and implies that the entity is to be found on the current node‗s PMI/B-Rep/Wireframe-Rep segment. It is valid for the source owning entity to be the same as the destination owning entity (i.e. an association between two PMI or B-Rep entities in the same part/assembly).

11.2.4 PMI User Attributes

The PMI User Attributes collection defines data for a list of user attributes. PMI User Attributes are used to add attribute data to a part/assembly. Each user attribute is composed of key/value pair of strings.

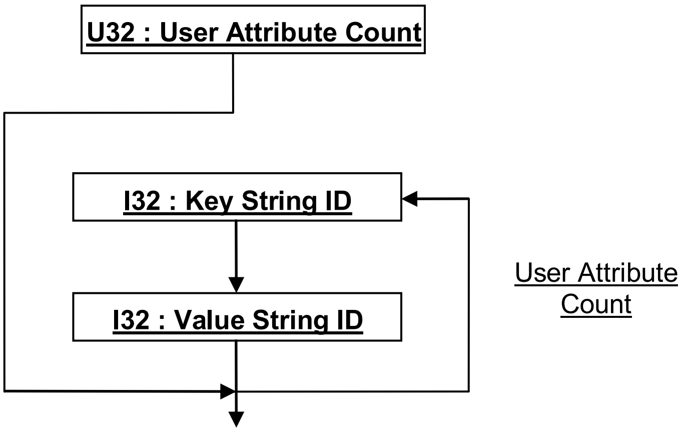


Figure 166 — PMI User Attributes data collection

U32 : User Attribute Count

User Attribute Count specifies the number of user attributes.

I32 : Key String ID

Key String ID specifies the string identifier for the user attribute key. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―1” indicates no string.

I32 : Value String ID

Value String ID specifies the string identifier for the user attribute value. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―1” indicates no string.

11.2.5 PMI String Table

The PMI String Table data collection defines data for a list of character strings and serves as a central repository for all character strings used by other PMI Entities within the same PMI Manager Meta Data Element. PMI Entities reference into this list/array of character strings to define usage of a particular character string using a simple list/array ―index” (i.e. String ID).

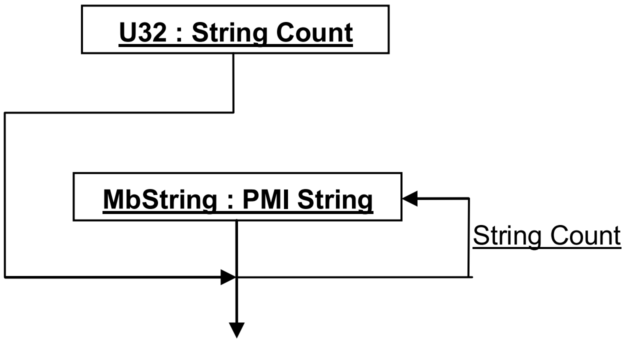


Figure 167 — PMI String Table data collection

U32 : String Count

String Count specifies the number of character strings in the string table.

MbString : PMI String

PMI String specifies the character string.

11.2.6 PMI Model Views

The PMI Model Views data collection defines data for a list of Model Views. A fully annotated part/assembly may contain so much PMI information, that it becomes very difficult to interpret the design intent when viewing a 3D Model (with PMI visible) of the part/assembly. Model Views provide a means to capture and organize PMI information about a 3D model so that the design intent can be clearly interpreted and communicated to others in later stages of the Product Lifecycle Management (PLM) process (e.g. manufacturing, inspection, assembly). This organization is achieved via PMI Associations (see PMI Associations), where specific PMI entities are associated as ―destinations" to a ―source" PMI Model View.

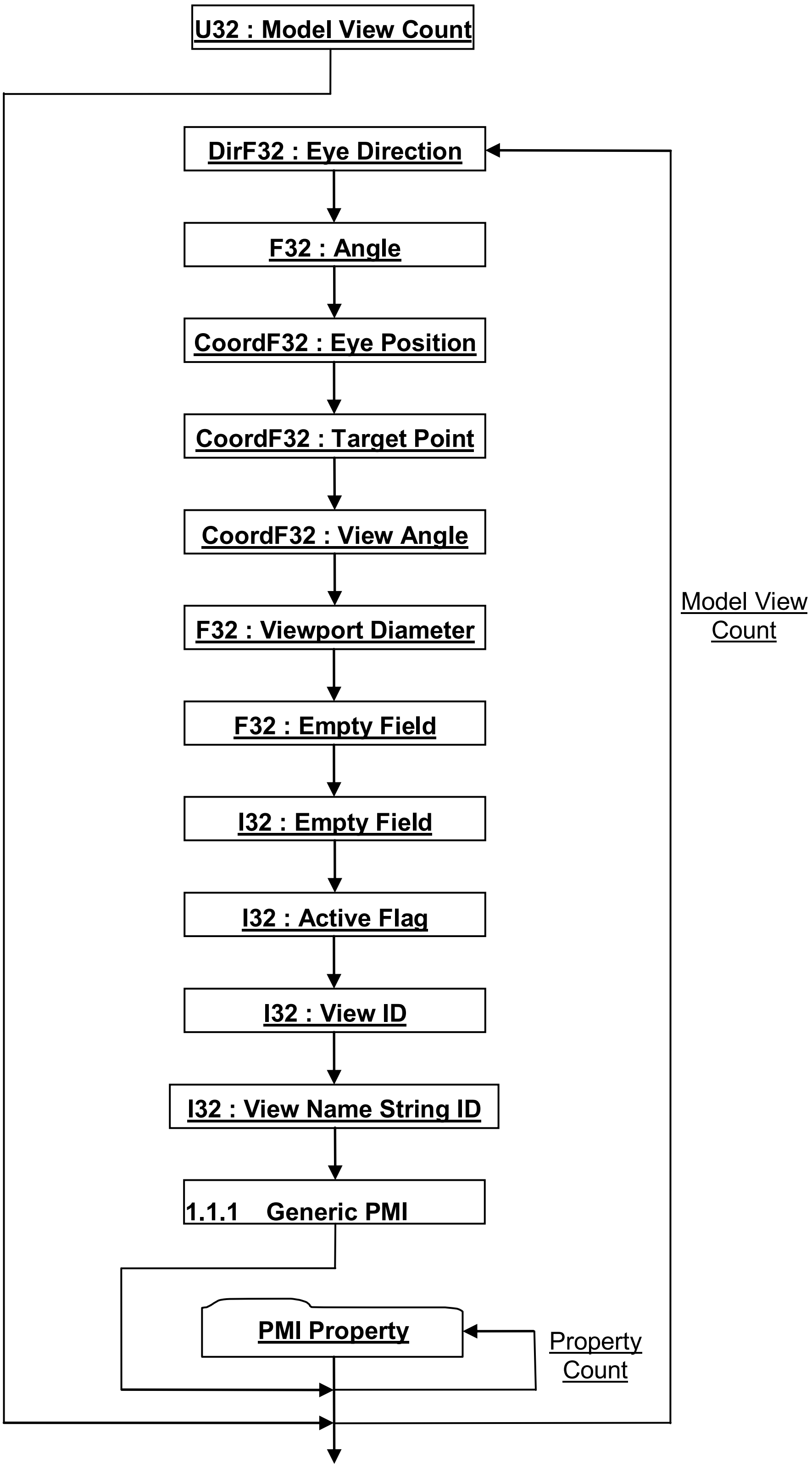


Figure 168 — PMI Model Views data collection

U32 : Model View Count

Model View Count specifies the number of Model Views.

DirF32 : Eye Direction

Eye Direction specifies the camera direction vector.

F32 : Angle

Angle specifies the camera rotation angle (in degrees where positive is counter-clockwise) about the Eye Direction. So this Angle in combination with the Eye Direction is equivalent to specifying a rotation using axis-angle representation.

CoordF32 : Eye Position

Eye Position specifies the WCS coordinates of the eye/camera —look from” position.

CoordF32 : Target Point

Target Point specifies the WCS coordinates of the eye/camera —look at” position.

CoordF32 : View Angle

View angle specifies the X, Y, Z rotation angles (in degrees) of the model‘s axis. The rotations are defined with respect to an initial orientation where the model‘s axis are aligned with the screen‘s axis (i.e. +X axis points to right, +Y axis points up, +Z axis points out at you).

F32 : Viewport Diameter

Viewport Diameter specifies the diameter in WCS coordinates of the largest possible circle that could be inscribed within viewport. If a large diameter value is specified, the model appears very small in relation to the viewport; whereas if a small diameter value is specified a close-up (—zoomed-in)” view of the model results.

F32 : Empty Field

Refer to best practice 14.4 Empty Field

I32 : Empty Field

Refer to best practice 14.4 Empty Field

I32 : Active Flag

Active Flag is a flag specifying whether this Model View is the —active” view. Valid values include the following:

Table 71 — PMI Model Views Active Flag values

= 0	Is not the active Model View.
= 1	Is the active Model View

I32 : View ID

View ID specifies the Model View unique identifier.

I32 : View Name String ID

View Name String ID specifies the string identifier for the Model View‘s name. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of —-1” indicates no string.

11.2.7 Generic PMI Entities

The Generic PMI Entities data collection provides a ‗generic” format for defining various PMI entity types, including user defined types. The generic format defines the data making up the PMI Entity through a combination of the PMI 2D Data collection and a list of PMI Property data collections.

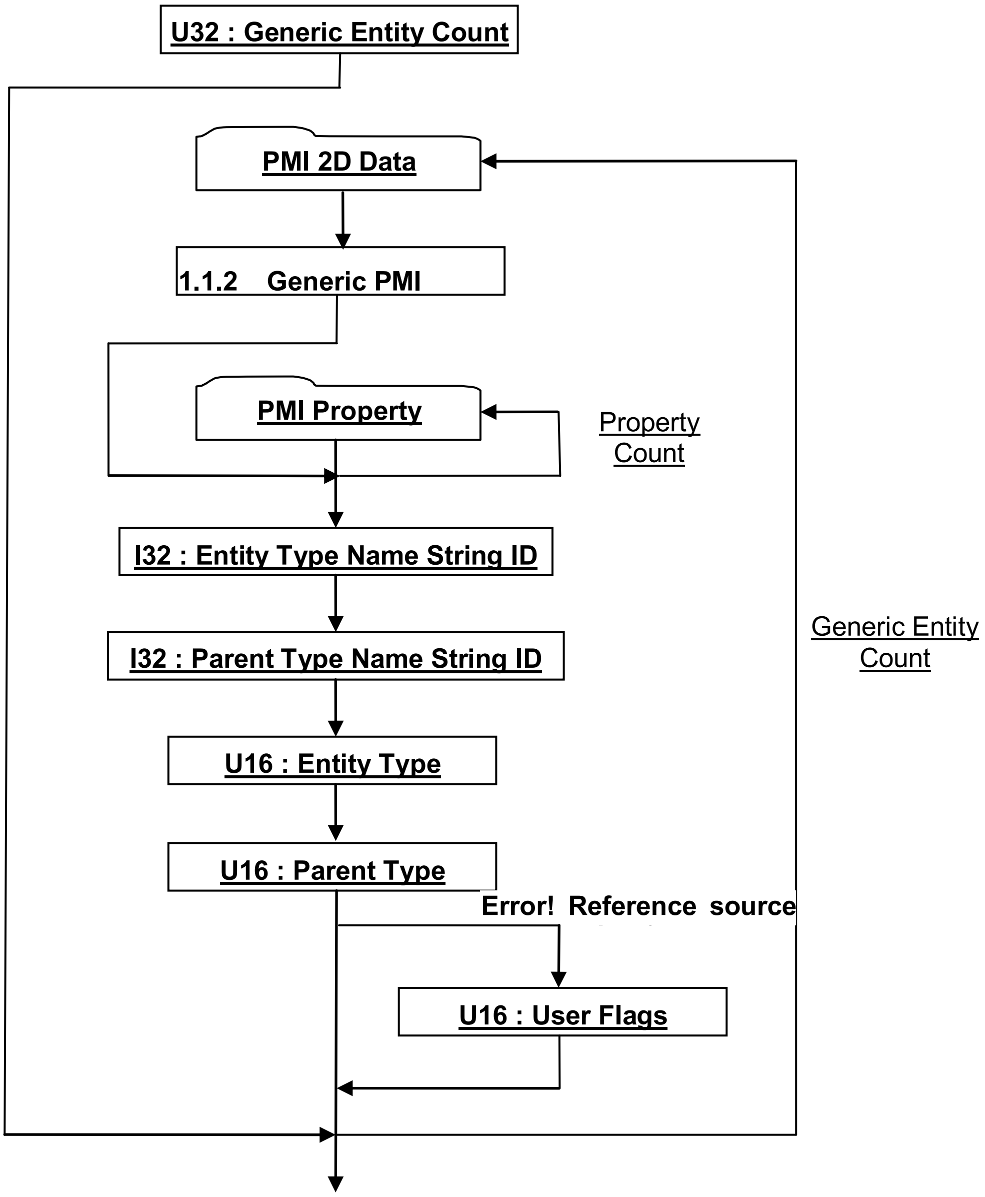


Figure 169 — Generic PMI Entities data collection

Complete description for PMI 2D Data can be found in 11.2.8.1 PMI 2D Data.

I32 : Generic Entity Count

Generic Entity Count specifies the number of Generic PMI Entities.

I32 : Property Count

Property Count specifies the number of PMI Properties.

11.2.7.1 PMI Property

A PMI Property data collection consists of a key/value pair and is used to describe attributes of Generic PMI Entity or other specific data.

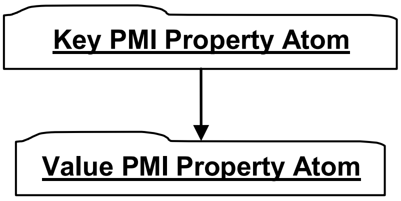


Figure 170 — PMI Property data collection

Both Key and Value have the same format as that documented in PMI Property Atom.

Although there is no reference compliant requirements for what the PMI Property key/value pairs shall be for each Generic PMI Entity type, there are some common PMI Property keys and corresponding value formats that appear in JT File. The below table documents these common PMI Property keys (i.e. the keys encoded string value) and what the format of the value data is in the values encoded string (see PMI Property Atom for an explanation of what is meant by —encoded string value” for the —key” and —value” data, see [3] for an explanation of the —%f” format).

Table 72 — Common Property Keys and Their Value Encoding formats

—Key” Property Atom Value String	—Value” Property Atom Value String Encoding Format	Decoding Notes
—PMI_PROP_ANCHOR_POINT"	—Px Py Pz”	Each Px, Py, Pz is a F32 value using —%f” format
—PMI_PROP_NOTE_HAS_URL”	—0” or —1”	0 = = False; 1 = = True
—PMI_PROP_NORMAL_DIR”	—Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using —%f” format
—PMI_PROP_APPROACH_DIR”	—Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using —%f” format
—PMI_PROP_CLAMPING_DIR”	—Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using —%f” format
—PMI_PROP_MEAS_DIR”	—Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using —%f” format
—PMI_PROP_COORD_DIR”	—Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using —%f” format
—PMI_PROP_MEAS_LEVEL”	—#”	Integer representing level number
—PMITextForegroundColour”	—#”	Hexadecimal integer representing RGB colour where value has —0x00bbggrr” form. The low-order byte contains a value for the relative intensity of red; the second byte contains a value for the relative intensity of green; and the third byte contains a value for the relative intensity of blue. The high-order byte shall be zero. The maximum value for a single byte is 0xFF (i.e. intensity value is in the range [0:255]).
—PMITextBackgroundColour”	—#”	Same as —PMITextForegroundColour”
—PMITextBackgroundOpacity”	—#”	Unsigned decimal integer representing opacity percentage. Actual opacity is: decoded# / 100.0
—PMITextShowBorder”	—#”	Unsigned decimal integer: 0 = = False; 1 = = True
—PMITextSize”	—#”	Unsigned decimal integer representing text size in units of pixels.
—PMITextInPlane”	—#”	Unsigned decimal integer: 0 = = False; 1 = = True where —1” indicates that text should be

—Key” Property Atom Value String	—Value” Property Atom Value String Encoding Format	Decoding Notes
		displayed in the plane of the entity so that it rotates with view.
—PMIGeometryColour”	—#”	Same as —PMITextForegroundColour”
—PMIGeometryWidth”	—#”	Unsigned decimal integer representing line width in units of pixels.
CLIP_NORMAL	—#,#,#”	Used for Entity Type = —0x0114” and Entity Type Name String = —Section” to specify the normal to the clipping plane. The clipping normal points toward the piece of the model that will be clipped away. Each # is a F64 value using —%lf” format.
CLIP_POSITION	—#,#,#”	Used for Entity Type = —0x0114” and Entity Type Name String = —Section” to specify one point on the clipping plane. Each # is a F64 value using —%lf” format.
TRANSFORMATION_MATRIX	—#,#,#,#,#,#,#,#, #,#,#,#,#,#,#,#”	Used for Entity Type = —0x0114” and Entity Type Name String = —Part Transform” to specify a transformation matrix. Each # is a F32 value using —%f” format.

11.2.7.1.1 PMI Property Atom

PMI Property Atom data collection represents the data format for both the key and value data of a PMI Property key/value pair.

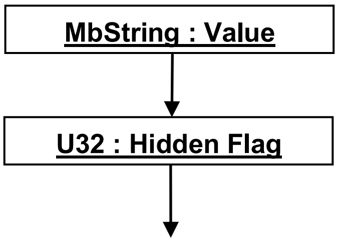


Figure 171 — PMI Property Atom data collection

MbString : Value

Value specifies the property atom value encoded into a String. See Table 72 — Common Property Keys and Their Value Encoding formats above for encoding formats of the Value string.

U32 : Hidden Flag

Hidden Flag specifies if the property is —hidden” or not. A JT file reader could use this flag to control whether read properties should be exposed to the end user of the application reading the JT file. Valid values include the following:

Table 73 — PMI Property Atom Hidden Flag values

= 0	Property is not hidden.
= 1	Property is hidden.

11.2.8 Generic PMI Entities

The Generic PMI Entities data collection provides a ―generic" format for defining various PMI entity types, including user defined types. The generic format defines the data making up the PMI Entity through a combination of the PMI 2D Data collection and a list of PMI Property data collections.

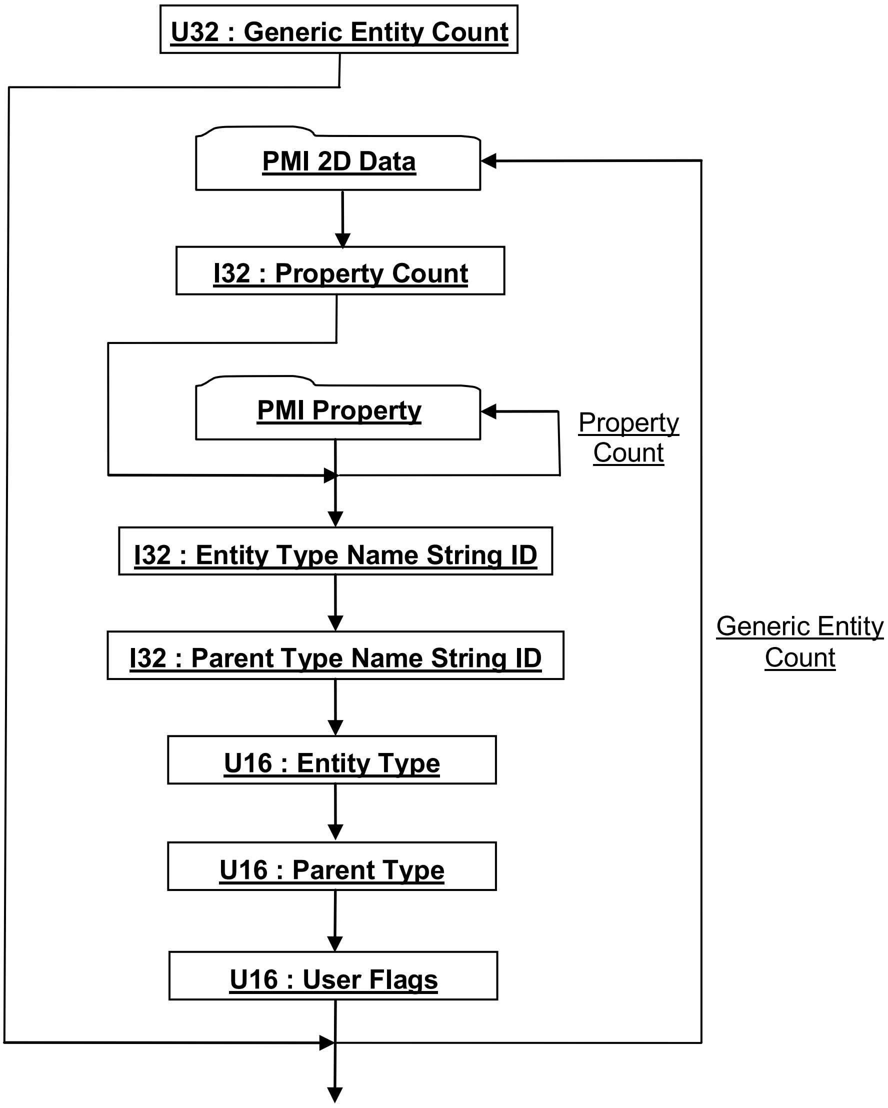


Figure 172 — Generic PMI Entities data collection

Complete description for PMI Property can be found in 11.2.7.1 PMI Property.

U32 : Generic Entity Count

Generic Entity Count specifies the number of Generic PMI Entities.

I32 : Property Count

Property Count specifies the number of PMI Properties.

I32 : Entity Type Name String ID

Entity Type Name String ID specifies the string identifier for the name of the Generic PMI Entity Type. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―-1" indicates no string.

I32 : Parent Type Name String ID

Parent Type Name String ID specifies the string identifier for the name of the parent Generic PMI Entity Type. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of —1” indicates no string.

U16 : Entity Type

Entity Type specifies the Generic PMI Entity Type. The valid Entity Type values (in hexadecimal format) are documented in the following table. Note that for —user defined” Generic PMI Entities a hexadecimal value of —0x0114” (as documented in table below) should be used.

Table 74 — Generic PMI Entity Type values

0x0001	PMI (generally only used as a Parent Type)
0x0002	Weld
0x0004	Spot Weld
0x0008	Line Weld
0x0010	Groove Weld
0x0011	Fillet Weld
0x0012	Slot Weld
0x0014	Edge Weld
0x0018	Arc Spot Weld
0x0020	Resistance Spot Weld
0x0021	Resistance Seam Weld
0x0022	Structural Adhesive Bead Shaped
0x0024	Structural Adhesive Tape Shaped
0x0028	Structural Adhesive Dollop Shaped
0x0040	Mechanical Clinch Connector
0x0041	Surface Finish
0x0042	Measurement Point
0x0044	Datum Locator
0x0048	Certification Point
0x0080	Geometric Dimensioning and Tolerancing
0x0081	Feature Control Frame
0x0082	Dimension
0x0084	Datum Feature Symbol
0x0088	Datum Target
0x0100	Note
0x0101	Face Attribute Note
0x0102	Model View Label Note
0x0104	Coordinate System
0x0108	Reference Geometry
0x0110	Reference Point
0x0111	Reference Axis
0x0112	Reference Plane
0x0114	User Defined
0x0118	Measurement Locator
0x0120	Datum Point
0x0121	Surface Vector Measurement Point
0x0122	Hole Vector Measurement Point
0x0124	Trimmed Sheet Vector Measurement Point
0x0128	Hem Vector Measurement Point
0x0230	Fastener PMI
0x0231	Material specification
0x0232	Process specification
0x0233	Part specification
0x0235	Balloon Note
0x0238	Circle Centre
0x0239	Coordinate Note

0x0240	AttributeNote
0x0241	Bundle or Dressing Note
0x0242	Cutting Plane Symbol
0x0243	Crosshatch
0x0244	E Marking (Note)
0x0245	Organization
0x0246	Region
0x0305	Section
0x0306	Centreline
0x0307	Fit Designation
0x0308	Composite Feature Control Frame

U16 : Parent Type

Parent Type specifies the parent Generic PMI Entity Type. The valid Parent Type values are the same as that documented above for Entity Type. The Parent Type is used to create a class hierarchy of PMI when presenting the PMI contents from a JT file.

U16 : User Flags

User Flags is a collection of flags. The flags are combined using the binary OR operator and store various state information for the Generic PMI Entity. All bits fields that are not defined as in use should be set to —'0

Table 75 — Generic PMI User Flag values

0x0001	Show PMI Entity —flat to screen only” flag = 0 – Allow PMI display plane to rotate with model. = 1 – Display PMI entity in the plane of the screen, so that it does not rotate with model.

11.2.8.1 PMI 2D Data

The PMI 2D Data collection defines a data format common to all 2D based PMI entities.

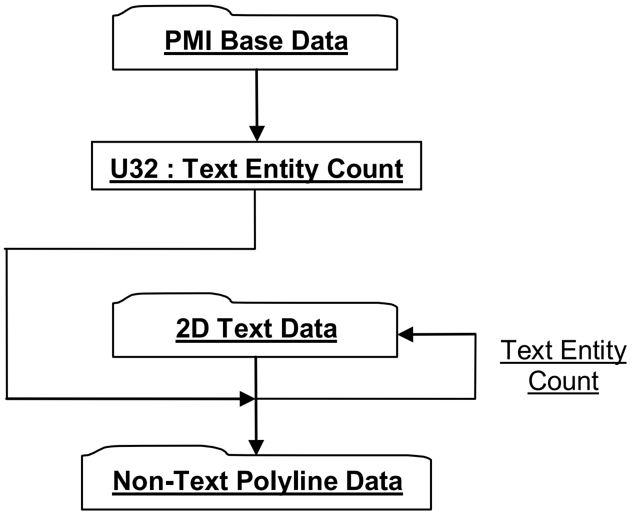


Figure 173 — PMI 2D Data collection

U32 : Text Entity Count

Text Entity Count specifies the number of Text entities in the particular PMI entity.

11.2.8.1.1 PMI Base Data

The PMI Base Data collection defines the basic/common data that every 2D and 3D PMI entity contains

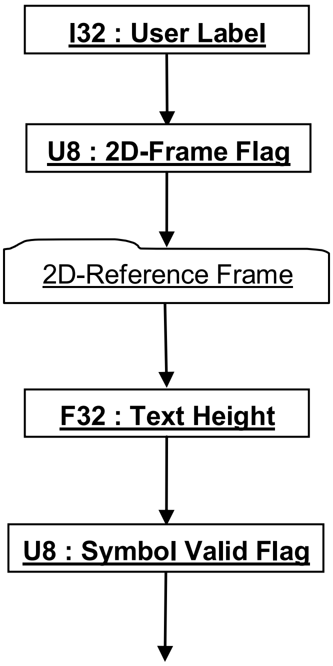


Figure 174 — PMI Base Data collection

I32 : User Label

User Label specifies the particular PMI entity identifier.

U8 : 2D-Frame Flag

2D-Frame Flag is a flag specifying whether 2D-Reference Frame data is stored. If 2D-Frame Flag has a non-zero value then 2D-Reference Frame data is included. If 2D-Frame Flag has a value of —2‖ then dummy (i.e. all zeros) 2D-Reference Frame data is written. The —2D-Frame Flag = = 2" case is used by 11.2.8 Generic PMI Entities because for Generic PMI Entities all the Non-Text Polyline Data is already in 3D form (i.e. XYZ coordinate data).

F32 : Text Height

Text Height specifies the PMI text height in WCS.

U8 : Symbol Valid Flag

Symbol Valid Flag is a flag specifying whether the particular PMI entity is valid. If Symbol Valid Flag has a non-zero value then PMI entity is valid.

2D-Reference Frame

The 2D-Reference Frame data collection defines a reference frame (2D coordinate system) where the PMI entity is displayed in 3D space. All the PMI entity's 2D and 3D polyline data is assumed to lie on the defined plane.

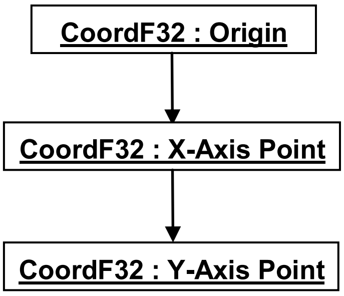


Figure 175 — 2D-Reference Frame data collection

CoordF32 : Origin

Origin defines the origin (min-corner) of the 2D coordinate system

CoordF32 : X-Axis Point

X-Axis Point defines a point along the X-Axis of the 2D coordinate system.

CoordF32 : Y-Axis Point

Y-Axis Point defines a point along the Y-Axis of the 2D coordinate system.

11.2.8.1.2 2D Text Data

The 2D Text Data collection defines a 2D text entity/primitive.

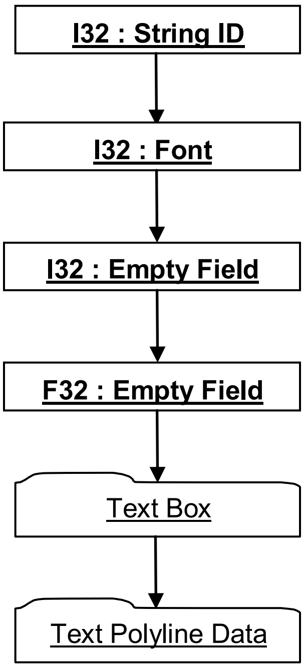


Figure 176 — 2D Text Data collection

I32 : String ID

String ID specifies the identifier for the character string. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―1” indicates no string.

I32 : Font

Font identifies the font to be used for this text. Valid values include the following:

Table 76 — PMI 2D Base Data Font values

= 1	Simplex
= 2	Din
= 3	Military
= 4	ISO
= 5	Lightline
= 6	IGES 1001
= 7	Century
= 8	IGES 1002
= 9	IGES 1003
= 101	Japanese JISX 0208 coded character set

= 102	Japanese Extended Unix Codes JISX 0208 coded character set
= 103	Chinese GB 2312.1980 Simplified coded character set
= 104	Korean KSC 5601 coded character set
= 105	Chinese Big5 Traditional coded character set

I32 : Empty Field

Refer to best practice 14.4 Empty Field

F32 : Empty Field

Refer to best practice 14.4 Empty Field

Text Box

The Text Box data collection specifies a 2D box that particular text fits within. All values are with respect to 2D-Reference Frame documented in 2D-Reference Frame.

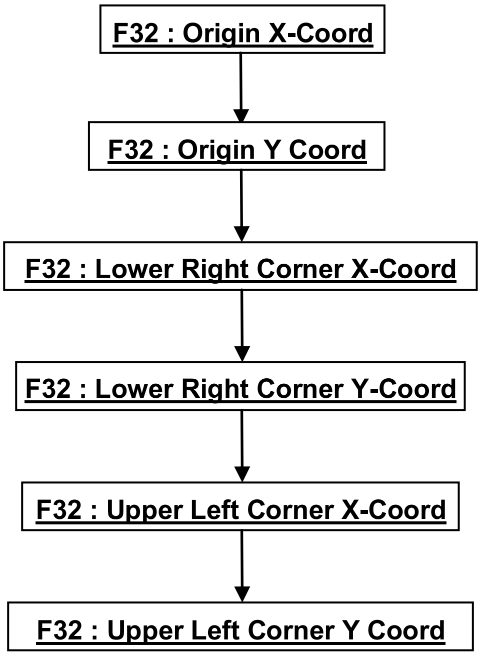

F32 : Origin X-Coord
F32 : Origin Y Coord
F32 : Lower Right Corner X-Coord
F32 : Lower Right Corner Y-Coord
F32 : Upper Left Corner X-Coord
F32 : Upper Left Corner Y Coord

Figure 177 — Text Box data collection

F32 : Origin X-Coord

Origin X-Coord defines the 2D X-coordinate of the text origin with respect to 2D-Reference Frame.

F32 : Origin Y Coord

Origin Y-Coord defines the 2D Y-coordinate of the text origin with respect to 2D-Reference Frame.

F32 : Lower Right Corner X-Coord

Lower Right Corner X-Coord defines the 2D X-coordinate of the lower right corner of the text with respect to 2D-Reference Frame.

F32 : Lower Right Corner Y-Coord

Lower Right Corner Y-Coord defines the 2D Y-coordinate of the lower right corner of the text with respect to 2D-Reference Frame.

F32 : Upper Left Corner X-Coord

Upper Left Corner X-Coord defines the 2D X-coordinate of the upper left corner of the text with respect to 2D-Reference Frame.

F32 : Upper Left Corner Y Coord

Upper Left Corner Y-Coord defines the 2D Y-coordinate of the upper left corner of the text with respect to 2D-Reference Frame.

Text Polyline Data

The Text Polyline Data collection defines any polyline segments which are part of the text representation. This existence of this polyline data is conditional (i.e. not all text has it) and is made up of an array of indices into an array of polyline segments packed as 2D vertex coordinates, specifying where each polyline segment begins and ends. Polylines are constructed from these arrays of data as follows:

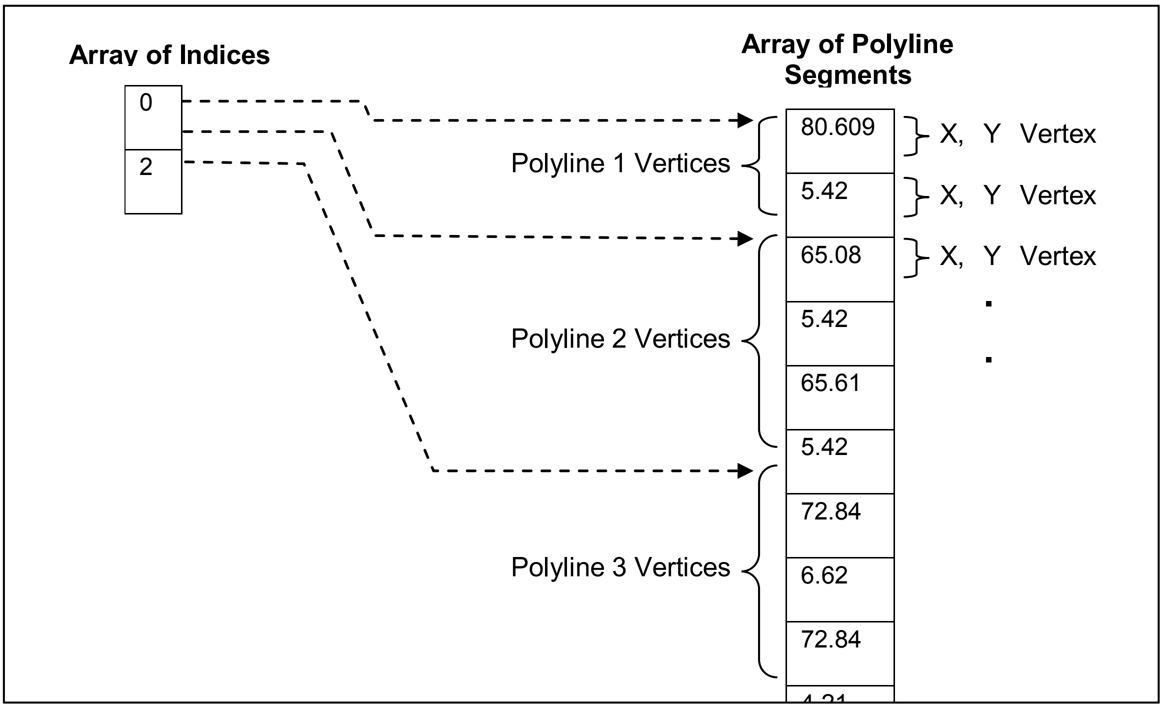
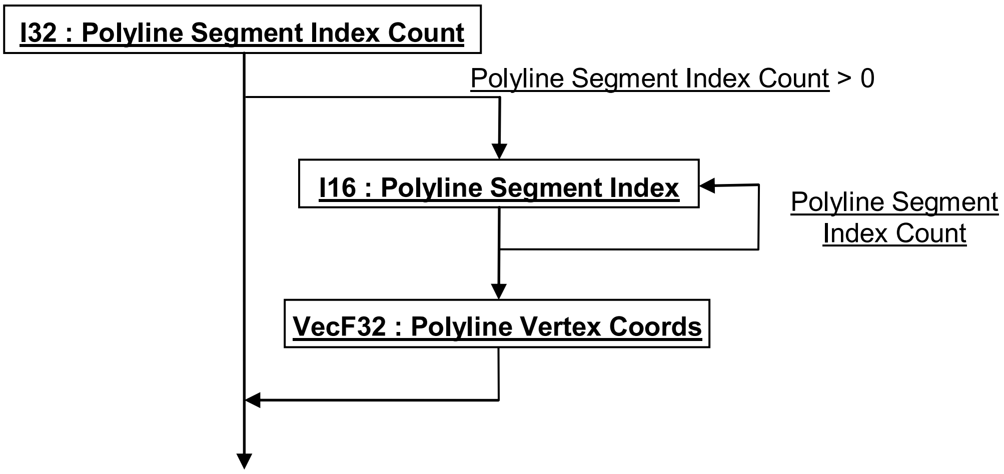


Figure 178 — Constructing Text Polylines from data arrays

This data is represented in JT file in the following forma



I32 : Polyline Segment Index Count
Polyline Segment Index Count > 0
I16 : Polyline Segment Index
Polyline Segment Index Count
VecF32 : Polyline Vertex Coords

Figure 179 — Text Polyline Data collection

I32 : Polyline Segment Index Count

Polyline Segment Index Count specifies the number of polyline segment indices.

I16 : Polyline Segment Index

Polyline Segment Index is an index into the Polyline Vertex Coords array specifying where polyline segment begins or ends. This index is a vertex coordinate index so the absolute index into the Polyline Vertex Coords array is computed by multiplying the index value by ―2" (i.e. for 2D coordinates).

VecF32 : Polyline Vertex Coords

Polyline Vertex Coords is an array of polyline segments packed as 2D point coordinates. These 2D point coordinates are with respect to the 2D-Reference Frame documented in 2D-Reference Frame.

11.2.8.1.3 Non-Text Polyline Data

The Non-Text Polyline Data collection contains all the non-text polylines making up the particular PMI entity. Examples of non-text polylines include line attachments, text boxes, symbol box dividers, etc. The Non-Text Polyline Data collection is made up of an array of indices into an array of polyline segments packed as either 2D or 3D vertex coordinates, specifying where each polyline segment begins and ends. Whether vertex coordinates are 2D or 3D is dependent upon the PMI entity type using this data collection. If it is a 11.2.8 Generic PMI Entities type then the packed coordinate data is 3D; for all other PMI entity types the packed coordinate data is 2D. Two arrays of values that sequentially specify the polyline type and width in the polyline segments array are included.

Figure 180 below shows how Polylines are constructed from these arrays of data for the packed 2D coordinates case. The packed 3D coordinates case is interpreted the same except that the coordinates array includes a Z component and is thus packed as ―X[YZ][XYZ][XYZ]…"

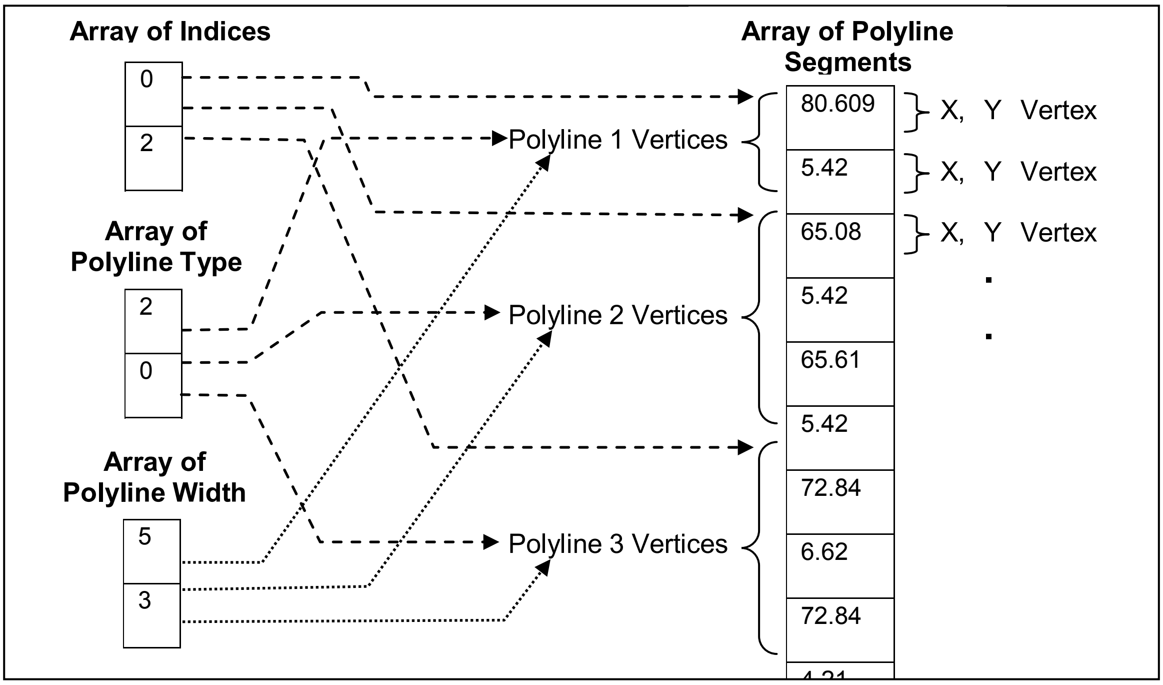


Figure 180 — Constructing Non-Text Polylines from packed 2D data arrays

This data is represented in the JT format as follows:

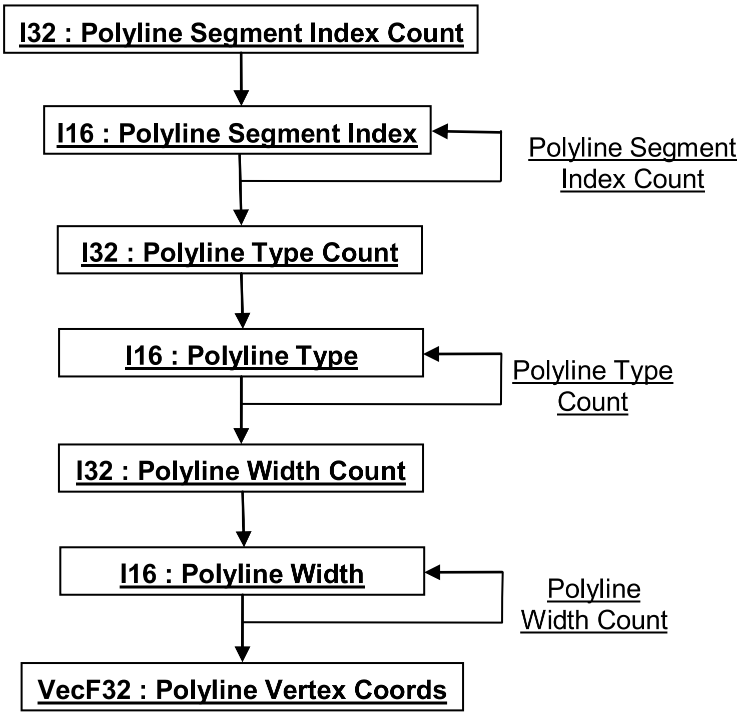

I32 : Polyline Segment Index Count
I16 : Polyline Segment Index
Polyline Segment Index Count
I32 : Polyline Type Count
I16 : Polyline Type
Polyline Type Count
I32 : Polyline Width Count
I16 : Polyline Width
Polyline Width Count
VecF32 : Polyline Vertex Coords

Figure 181 — Non-Text Polyline Data collection

I32 : Polyline Segment Index Count

Polyline Segment Index Count specifies the number of polyline segment indices.

I16 : Polyline Segment Index

Polyline Segment Index is an index into the Polyline Vertex Coords array specifying where polyline segment begins or ends. This index is a vertex/coordinate index so the absolute index into the Polyline Vertex Coords array is computed by multiplying the index value by —”2 (i.e. for 2D coordinates).

I32 : Polyline Type Count

Polyline Type Count specifies the number of polyline type values.

I16 : Polyline Type

Polyline Type specifies the type of polyline segment in Polyline Vertex Coords array. See Figure 180 — Constructing Non-Text Polylines from packed 2D data arrays for interpretation of this array of type values relative to the defined polylines. Valid values include the following:

Table 77 — PMI 2D Non-Text Polyline Type values

= 0	General line
= 1	General arrow
= 2	General circle
= 3	General arc
= 4	Extended line 1
= 5	Extended line 2
= 6	Extended arc
= 7	Extended circle
= 8	Text line (used in text boxes and symbol box dividers)
= 9	Text string

I32 : Polyline Width Count

Polyline Width Count specifies the number of polyline width values.

I16 : Polyline Width

Polyline Width specifies the width of polyline segment in Polyline Vertex Coords array. See Figure 180 — Constructing Non-Text Polylines from packed 2D data arrays for interpretation of this array of width values relative to the defined polylines.

VecF32 : Polyline Vertex Coords

Polyline Vertex Coords is an array of polyline segments packed as 2D point coordinates. These 2D point coordinates are with respect to the 2D-Reference Frame documented in 2D-Reference Frame.

11.2.9 PMI Polygon Data

The PMI Polygon Data collection contains a list of vertices classified as polygonal primitives. Its composition is shown in the figure 162. Each block of PMI PolygonData contains a list of 0 or more PolygonData elements. Empty PolygonData elements are written with 0 vertices and no additional fields.

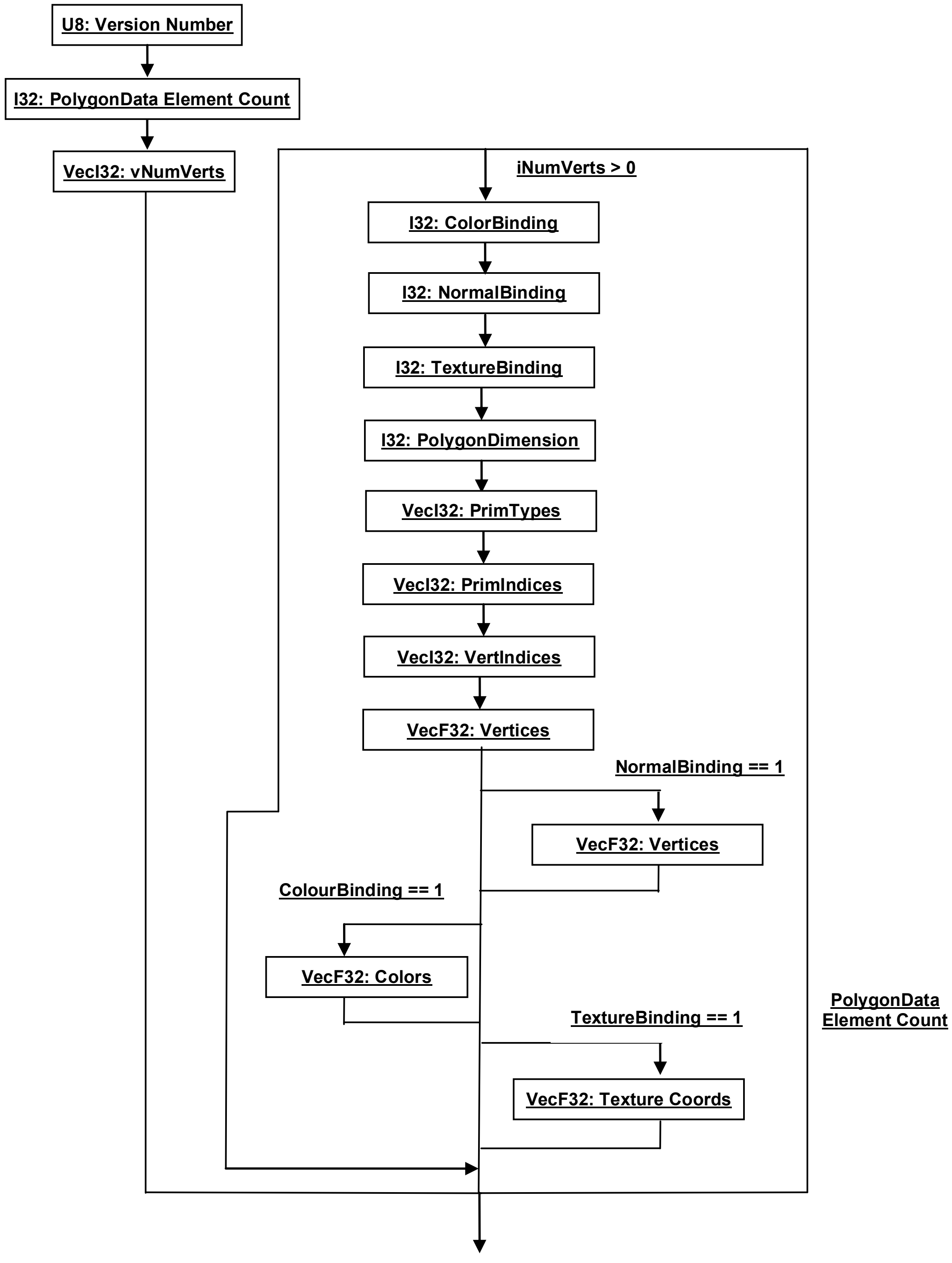


Figure 182 — PMI Polygon Data

U8: Version Number

Version number is the version identifier for this PMI Polygon Data. For information on local version numbers see best practice 14.5 Local version numbers

I32: PolygonData Element Count

PolygonData Element Count specifies the number of PolygonData elements.

VecI32: vNumVerts

An integer vector is used to record the number of vertices in each polygon data element. The length of this vector is equal to PolygonData Element Count written in this block of PMI PolygonData. The presence of additional data fields in each PolygonData element is hinged upon that element having more than 0 vertices recorded in this vector.

Retrieve next vertCount from vNumVerts

If the next element in the vNumVerts vector is non-zero, proceed to read other fields that make up a single PMI PolygonData element. Otherwise, skip reading more data for this element and loop back to seek the next element in the vector.

iNumVerts

Number of vertices for the i^{th} PolygonData element.

I32: ColorBinding

A Boolean value that indicates if there are colors present along with the list of coordinates at each vertex.

I32: NormalBinding

A Boolean value that indicates if there are normals present along with the list of coordinates at each vertex.

I32: TextureBinding

A Boolean value that indicates if there are Texture Coordinates present along with the list of coordinates at each vertex.

I32: PolygonDimension

Indicates the dimension of vertex coordinates.

VecI32: PrimTypes

An array indicating the type of each of the primitive stored in the PrimIndices array. Adjacent numbers in the array form tuples of the form [PrimIndex, PrimType]. All primitives to the left of the PrimIndex are of type PrimType unless they are already to the left of an earlier PrimIndex in this array.

VecI32: PrimIndices

Indices of vertices that form a single primitive. The difference between two adjacent values in this array determines the length of the primitive. An extra element is stored at the end of this array to identify the length of the last primitive. Values in this array are indices into the VertIndices array.

VecI32: VertIndices

An array of indices into the Vertices array. This index array eliminates the need to duplicate floating point vertices that are shared by multiple primitives.

VecF32: Vertices

The list of vertex coordinates. Each vertex is made of PolygonDimension coordinates. The length of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Normals

An optional list of Normals for each vertex. Presence of this list is indicated by the NormalBinding flag. Each normal consists of PolygonDimension components. The size of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Colors

An optional list of Colours for each vertex. Presence of this list is indicated by the ColorBinding flag. Each color consists of PolygonDimension components. The size of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Texture Coords

An optional list of Texture coordinates for each vertex. Presence of this list is indicated by the TexCoordBinding flag. Each TexCoord consists of 2 components. The size of this list is equal to number of vertices multiplied by 2.

11.2.10 PMI Model View Sort Orders

The PMI Model View Sort Orders collection defines data for a list of model view sort orders. Each model view sort order is composed of key/value pair of strings.

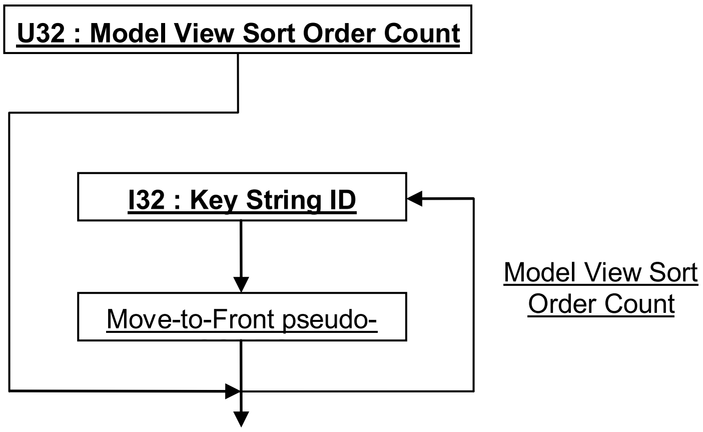

U32 : Model View Sort Order Count
I32 : Key String ID
Move-to-Front pseudo-
Model View Sort Order Count

Figure 183 — PMI Model View Sort Orders data collection

U32 : Model View Sort Order Count

Model View Sort Order Count specifies the number of model view sort orders.

I32 : Key String ID

Key String ID specifies the string identifier for the key of model view sort order. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―1” indicates no string.

I32 : Value String ID

Value String ID specifies the string identifier for the value of model view sort order. This identifier is an index to a particular character string in the PMI String Table as defined in 11.2.5 PMI String Table. An identifier value of ―1” indicates no string.

11.2.11 PMI CAD Tag Data

The **Error! Reference source not found.** collection contains the list of persistent IDs, as defined in the CAD System, to uniquely identify individual PMI entities. The existence of this **Error! Reference source not found.**

collection is dependent upon the value of previously read data field CAD Tags Flag as documented in 11.2 PMI Manager Meta Data Element.

If **Error! Reference source not found.** collection is present, there will be a CAD Tag for each PMI entity as specified by the below documented CAD Tag Index Count formula.

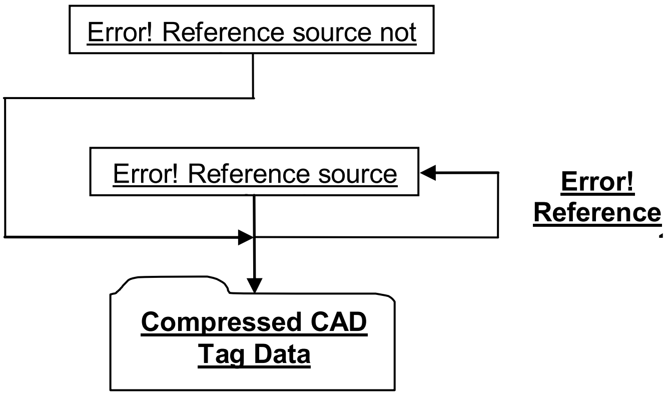


Figure 184 — Error! Reference source not found. **collection**

Complete description for Compressed CAD Tag Data can be found in 13.1.16 Compressed CAD Tag Data.

I32 : CAD Tag Index Count

CAD Tag Index Count specifies the total number of CAD Tag indices. This value shall be equal to the summation of the previously read count values for all the PMI entities supporting CAD Tags. The formula is the sum of the following:

- Line Weld Count
- Spot Weld Count
- SF Count
- MP Count
- Reference Geometry Count
- Datum Target Count
- FCF Count
- Locator Count
- Dimension Count
- DFS Count
- Note Count
- Model View Count
- Design Group Count
- Coord Sys Count
- Generic Entity Count

I32 : CAD Tag Index

CAD Tag Index specifies an index into a list of CAD Tags, identifying the CAD Tag belonging to a particular PMI entity. There will be a total of CAD Tag Index Count number of CAD Tag Indices and the order of the indices will be as defined by the above documented CAD Tag Index Count formula (i.e. Line Weld CAD Tag Indices are first, followed by the Spot Weld CAD Tag Indices, followed by the Surface Finish CAD Tag Indices, etc.)

11.2.12 PMI Polygon Data

The PMI Polygon Data collection contains a list of vertices classified as polygonal primitives. Its composition is shown in the figure 177. Each block of PMI PolygonData contains a list of 0 or more PolygonData elements. Empty PolygonData elements are written with 0 vertices and no additional fields.

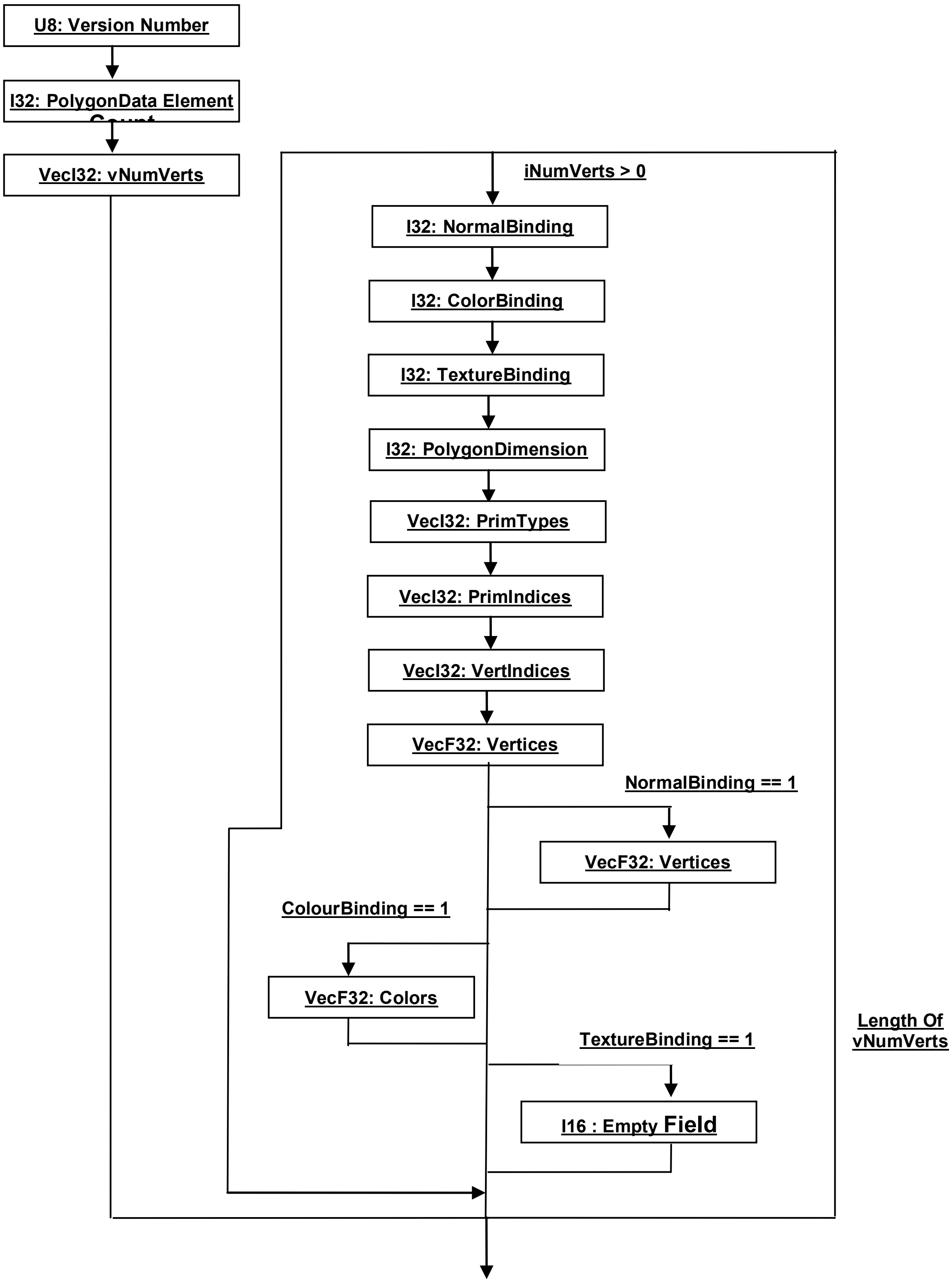


Figure 185 — PMI Polygon Data

I16: Version Number

Version number is the version identifier for this PMI Polygon Data Element. For informaiton on local version numbers see best practice 14.5 Local version numbers

I32: Empty Field

Refer to best practice 14.4 Empty Field

VecI32: vNumVerts

An integer vector used to record the number of vertices in each polygon data element. The length of this vector is equal to the number of PolygonData elements written in this block of PMI PolygonData. The presence of additional data fields in each PolygonData element is hinged upon that element having more than 0 vertices recorded in this vector.

Retrieve next vertCount from vNumVerts

If the next element in the vNumVerts vector is non-zero, proceed to read other fields that make up a single PMI PolygonData element. Otherwise, skip reading more data for this element and loop back to seek the next element in the vector.

iNumVerts

Number of vertices for the i^{th} PolygonData element.

Length Of vNumVerts

Number of Polygon Data elements.

I32: NormalBinding

A Boolean value that indicates if there are normals present along with the list of coordinates at each vertex.

I32: ColourBinding

A Boolean value that indicates if there are colours present along with the list of coordinates at each vertex.

I32: TextureBinding

A Boolean value that indicates if there are Texture Coordinates present along with the list of coordinates at each vertex.

I32: PolygonDimension

Indicates the dimension of vertex coordinates.

VecI32: PrimTypes

An array indicating the type of each of the primitive stored in the PrimIndices array. Adjacent numbers in the array form tuples of the form [PrimIndex, PrimType]. All primitives to the left of the PrimIndex are of type PrimType unless they are already to the left of an earlier PrimIndex in this array.

VecI32: PrimIndices

Indices of vertices that form a single primitive. The difference between two adjacent values in this array determines the length of the primitive. An extra element is stored at the end of this array to identify the length of the last primitive. Values in this array are indices into the VertIndices array.

VecI32: VertIndices

An array of indices into the Vertices array. This index array eliminates the need to duplicate floating point vertices that are shared by multiple primitives.

VecF32: Vertices

The list of vertex coordinates. Each vertex is made of PolygonDimension coordinates. The length of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Normals

An optional list of Normals for each vertex. Presence of this list is indicated by the NormalBinding flag. Each normal consists of PolygonDimension components. The size of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Colours

An optional list of Colours for each vertex. Presence of this list is indicated by the ColourBinding flag. Each colour consists of PolygonDimension components. The size of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Texture Coords

An optional list of Texture coordinates for each vertex. Presence of this list is indicated by the TexCoordBinding flag. Each TexCoord consists of 2 components. The size of this list is equal to number of vertices multiplied by 2.

12 JT ULP Segment

JT ULP Segment contains an Element that defines the semi-precise geometric Boundary Representation data for a particular Part in JT ULP format. Note that there is also two other Boundary Representation formats (i.e. JT B-Rep and XT B-Rep) supported by the JT file format within a different file Segment Type. Complete description for the JT B-Rep and the XT B-Rep can be found in 8 JT B-Rep Segment and 9 XT B-Rep Segment respectively.

JT ULP Segments are typically referenced by Part Node Elements (see 6.1.1.5 Part Node Element) using Late Loaded Property Atom Elements (see 6.2.7 Late Loaded Property Atom Element). The JT ULP Segment type supports compression on all element data, so all elements in JT ULP Segment

use the Logical Element Header Compressed form of element header data.

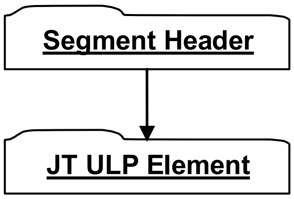


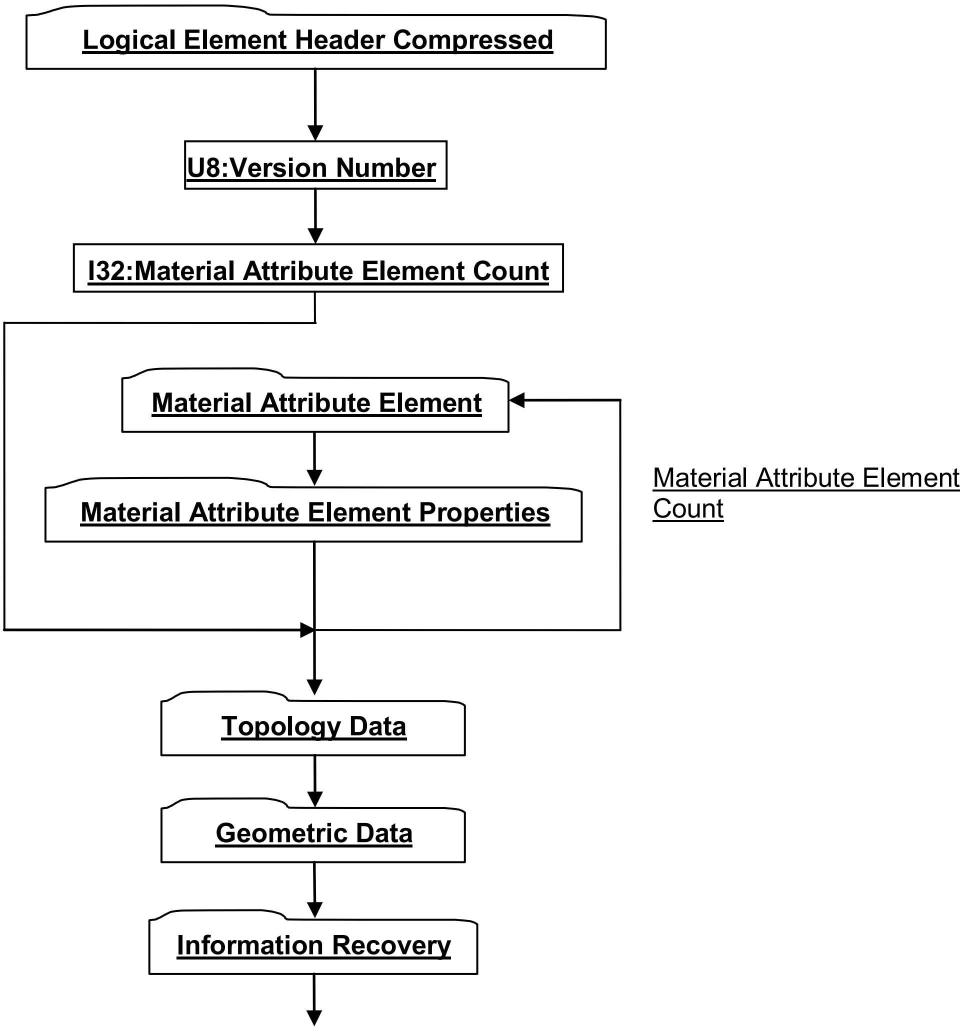
Figure 186 — JT ULP Segmentdata collection

Complete description for Segment Header can be found in 5.1.3.1 Segment Header.

12.1 JT ULP Element

Object Type ID: 0xf338a4af, 0xd7d2, 0x41c5, 0xbc, 0xf2, 0xc5, 0x5a, 0x88, 0xb2, 0x1e, 0x73

JT ULP Element represents a particular Part's ultra-lightweight semi-precise B-Rep data. Like JT B-Rep Element or XT B-Rep Element, JT ULP Element contains all the topological and geometric information that describes the shape of a part. The difference is that the size of JT ULP Element is typically around 10% of a typical JT file with B-Rep and LODs, and this is achieved by sophisticated compression techniques. In addition, JT ULP Element is semi-precise meaning that its geometric description is not as precise as either JT B-Rep Element or XT B-Rep Element. The precision loss of JT ULP Element, however, is carefully controlled to be equal to or better than 0.01% of the part size or 0.1mm, whichever is smaller.


Logical Element Header Compressed
U8:Version Number
I32:Material Attribute Element Count
Material Attribute Element
Material Attribute Element Properties
Material Attribute Element Count
Topology Data
Geometric Data
Information Recovery

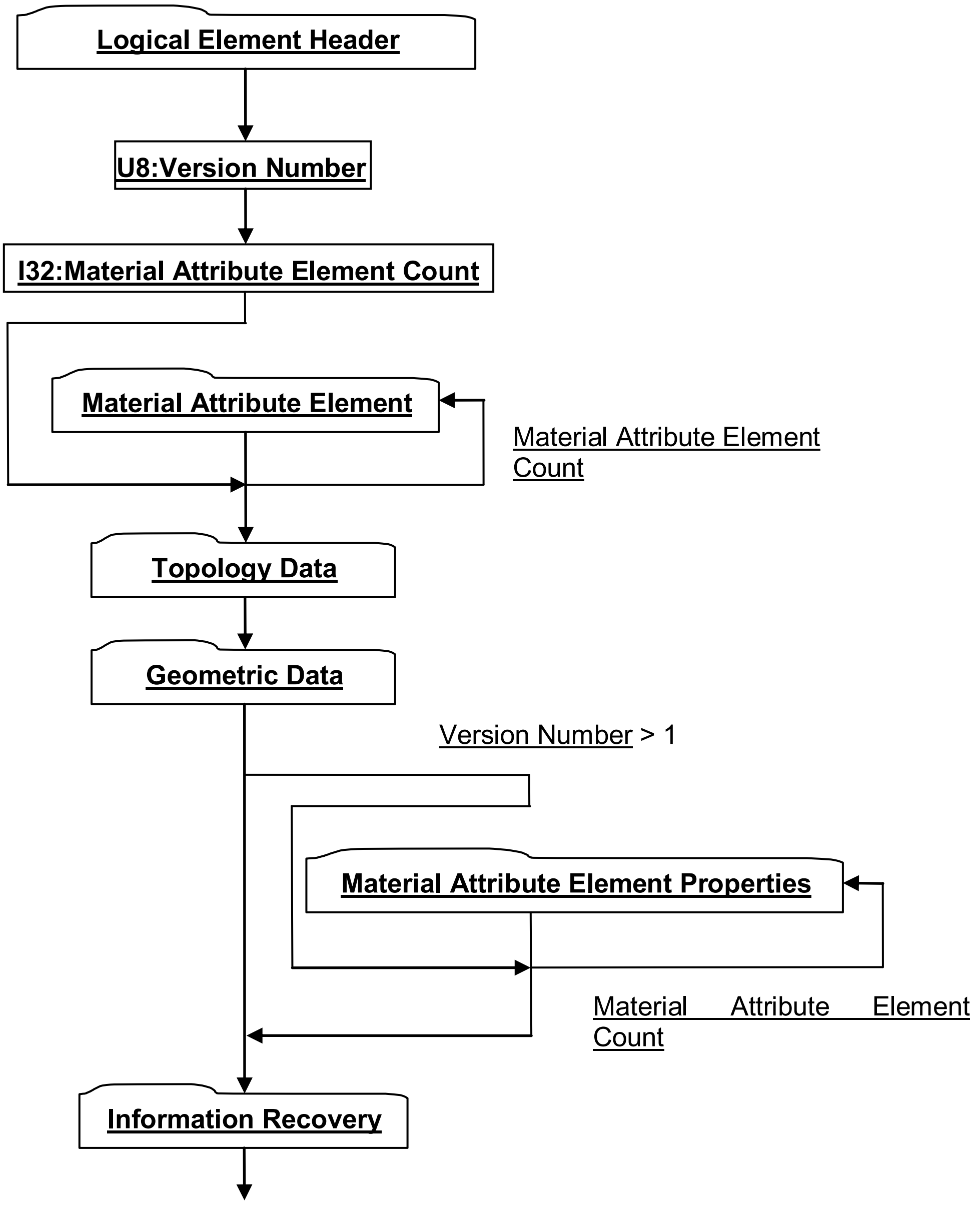

Logical Element Header
U8:Version Number
I32:Material Attribute Element Count
Material Attribute Element
Material Attribute Element Count
Topology Data
Geometric Data
Version Number > 1
Material Attribute Element Properties
Material Attribute Element Count
Information Recovery

Figure 187 — JT ULP Element data collection

Complete description for Logical Element Header Compressed can be found in Logical Element Header Compressed.

U8:Version Number

Version Number is the version identifier for this JT ULP Element. For informaiton on local version numbers see best practice 14.5 Local version numbers

I32:Material Attribute Element Count

Material Attribute Element Count is the number of material attribute elements.

Complete description for Material Attribute Element can be found in 6.1.2.2 Material Attribute Element.

12.1.1 Topology Data

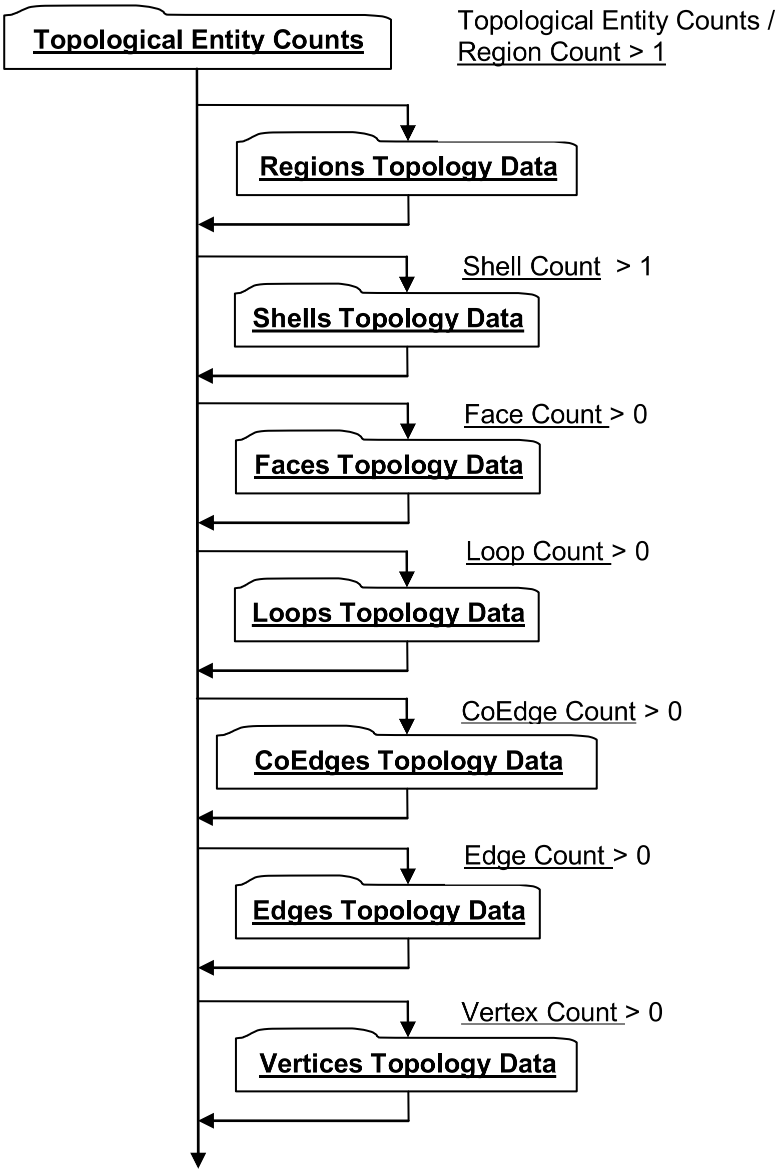


Figure 188 — Topology Data collection

Topological Entity Counts

Topological Entity Counts data collection defines the counts for each of the various topological entities within a ULP.

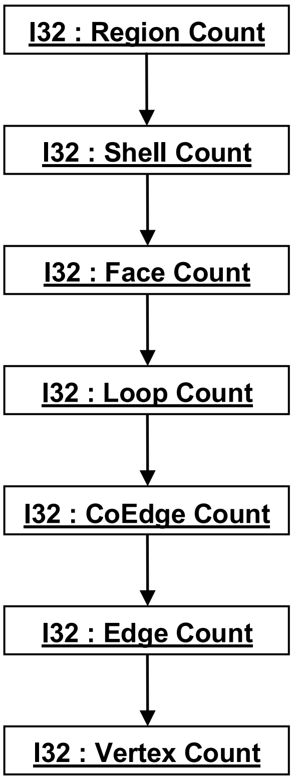


Figure 189 — Topological Entity Counts data collection

I32 : Region Count

Region Count indicates the number of topological region entities in the ULP.

I32 : Shell Count

Shell Count indicates the number of topological shell entities in the ULP.

I32 : Face Count

Face Count indicates the number of topological face entities in the ULP.

I32 : Loop Count

Loop Count indicates the number of topological loop entities in the ULP.

I32 : CoEdge Count

CoEdge Count indicates the number of topological coedge entities in the ULP.

I32 : Edge Count

Edge Count indicates the number of topological edge entities in the ULP.

I32 : Vertex Count

Vertex Count indicates the number of topological vertex entities in the ULP.

Combined Predictor Type

A predictor type may be combined with additional processing. When Combined Predictor Type is used, the additional processing step is encoded. For example, combined predictor type *Combined:NULL* means that the data collection follows the logical diagram in Figure 190 with ePredictorType set to be NULL.

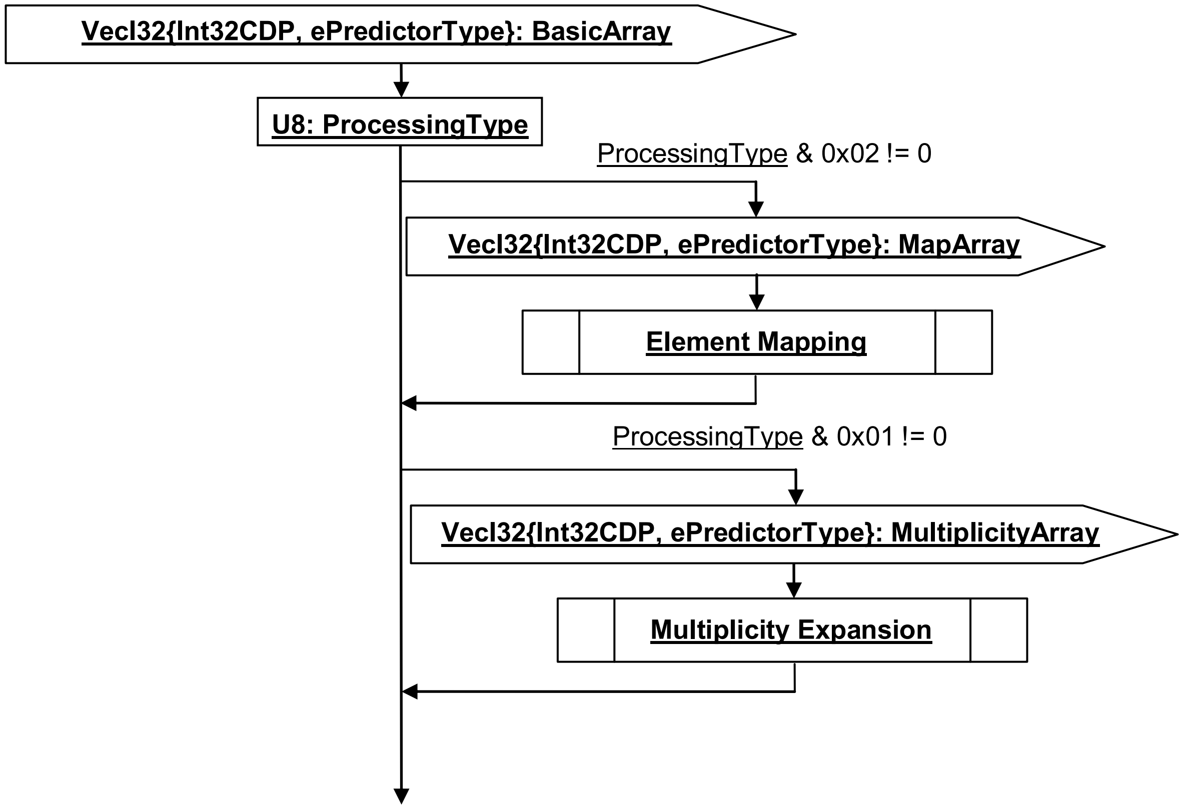

VecI32{Int32CDP, ePredictorType}: BasicArray
U8: ProcessingType
ProcessingType & 0x02 != 0
VecI32{Int32CDP, ePredictorType}: MapArray
Element Mapping
ProcessingType & 0x01 != 0
VecI32{Int32CDP, ePredictorType}: MultiplicityArray
Multiplicity Expansion

Figure 190 — Combined Predictor Type data collection

VecI32{Int32CDP, ePredictorType}: BasicArray

BasicArray is an integer array, compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

U8: ProcessingType

Two bits of this value are currently used. If bit 0x02 is set, then the integer array is a list of elements with unique values and Element Mapping step is needed to recover the original values. If bit 0x01 is set, then the some elements in the integer array may be repeated, and Multiplicity Expansion is used to recover the original values.

VecI32{Int32CDP, ePredictorType}: MapArray

MapArray is an integer array, where each element represents the index mapping information. MapArray is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet...

Element Mapping

Element Mapping recovers the original array from BasicArray and MapArray, using relationship $\mathrm{OriginalArray}[i] = \mathrm{BasicArray}[\mathrm{MapArray}[i]]$. After Element Mapping, the value of $\mathrm{BasicArray}$ is updated with $\mathrm{OriginalArray}$.

VecI32{Int32CDP, ePredictorType}: MultiplicityArray

MultiplicityArray is an integer array, where each element represents the multiplicity of each element in BasicArray. MultiplicityArray is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet..

Multiplicity Expansion

Multiplicity Expansion recovers the original array from BasicArray and MultiplicityArray. The original array is an expansion of the BasicArray. If the corresponding multiplicity value is greater than 1, the element in BasicArray is contiguously repeated in the original array according to multiplicity value.

Regions Topology Data

Regions Topology Data defines the disjoint set of non-overlapping Shells making up each Region. Each Region is defined by one or more non-overlapping Shells. The volume of a Region is that volume lying inside each ―anti-hole Shell” and outside each simply-contained ―hole Shell” belonging to the particular Region. A Region is analogous to a dimensionally elevated face where Region corresponds to Face and Shell corresponds to Trim Loop.

Each Region‘s defining Shells are identified in a list of Shells by an index for both the first Shell and the last Shell in each Region (i.e. all Shells inclusive between the specified first and last Shell list index define the particular Region). In addition, the indices of all the shells in a single Region are contiguous. The first shell index of the first region is 0, and the first shell index of other regions is one greater than the last shell index of the previous region. Therefore only the number of shells of each region is stored. In the special case when the number of regions is 1, no information needs be stored since its last Shell index is known to be Shell Count-1.

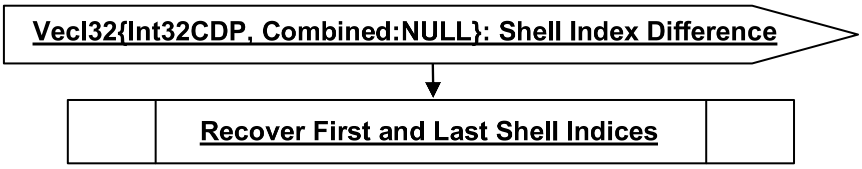

VecI32{Int32CDP, Combined:NULL}: Shell Index Difference
Recover First and Last Shell Indices

Figure 191 — Regions Topology Data collection

VecI32{Int32CDP, Combined:NULL}: Shell Index Difference

Shell Index Difference is a vector of indices representing the integer value by subtracting first shell index from last shell index in each region, encoded using Combined Predictor Type. Shell Index Difference is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet..

Recover First and Last Shell Indices

The first shell index of the first region is 0, and the last shell index of the first region is element 0 of Shell Index Difference. The first shell index of region $k, k \geq 1$ equals to the last shell index of region $k - 1$ plus 1. The last shell index of region $k, k \geq 1$ equals to the first shell index of region k plus element k of Shell Index Difference array.

Shells Topology Data

Shells Topology Data defines the set of topological adjacent Faces making up each Shell. A Shell‘s set of topological adjacent Faces define a single (usually closed) two manifold solid that in turn defines the boundary between the finite volume of space enclosed within the Shell and the infinite volume of space outside the Shell. In addition, each Shell has a flag that denotes whether the Shell refers to the finite interior volume (i.e. a ―hole Shell”) or the infinite exterior volume (i.e. an ―anti-hole Shell”).

Each Shell‘s defining Faces are identified in a list of Faces by an index for both the first Face and the last Face in each Shell (i.e. all Faces inclusive between the specified first and last Face list index define the particular Shell). In addition, the indices of all the faces in a single Shell are contiguous. The first face index of the first shell is 0, and the first face index of other shells is one greater than the last face index of the previous shell. Therefore only the number of faces of each shell is stored. In the special case when the number of shells is 1, no information needs be stored since its last face index is known to be Face Count-1.

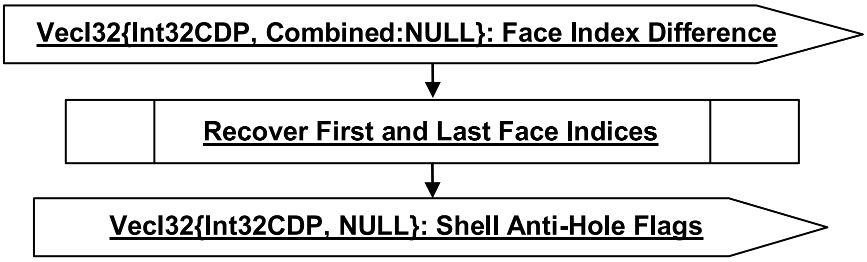

VecI32{Int32CDP, Combined:NULL}: Face Index Difference
Recover First and Last Face Indices
VecI32{Int32CDP, NULL}: Shell Anti-Hole Flags

Figure 192 — Shells Topology Data collection

VecI32{Int32CDP, Combined:NULL}: Face Index Difference

Face Index Difference is a vector of indices representing the integer value by subtracting first face index from last face index in each shell, encoded using Combined Predictor Type. Face Index Difference is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Recover First and Last Face Indices

The first face index of the first shell is 0, and the last face index of the first shell is element 0 of Face Index Difference. The first face index of shell $k, k \geq 1$ equals to the last face index of shell $k - 1$ plus 1. The last face index of shell $k, k \geq 1$ equals to the first face index of shell k plus element k of Face Index Difference array.

VecI32{Int32CDP, NULL}: Shell Anti-Hole Flags

Each Shell has a flag identifying whether the Shell is an anti-hole Shell. Shell Anti-Hole Flags is a vector of anti-hole flags for a set of Shells.

In an uncompressed/decoded form the flag values have the following meaning:

Table 78 — JT ULP Shell Anti-Hole Flag values

= 0	Shell is not an anti-hole Shell
= 1	Shell is an anti-hole Shell

Shell Anti-Hole Flags uses the Int32 version of the CODEC to compress and encode data.

Faces Topology Data

A Face shall be trimmed with at least one —anti-hole” Trim Loop and may be trimmed with one or more —hole” Trim Loops. The complete description of face and its relation to the trim loops can be found in Faces Topology Data.

Each Face‘s defining Trim Loops are identified in a list of trim Loops by an index for both the first Trim Loop and the last Trim Loop in each Face (i.e. all Trim Loops inclusive between the specified first and last Trim Loop list index define the particular Face). In addition, the indices of all the loops in a single Face are contiguous. The first loop index of the first face is 0, and the first loop index of other faces is one greater than the last loop index of the previous face. Therefore only the number of loops of each face is stored. In the special case when the number of faces is 1, no information needs be stored since its last loop index is known to be Loop Count-1.

Each Face‘s underlying Geometric Surface is identified by an index into a list of Geometric Surfaces. Each face‘s material is identified by an index into the list of Material Attribute Elements.

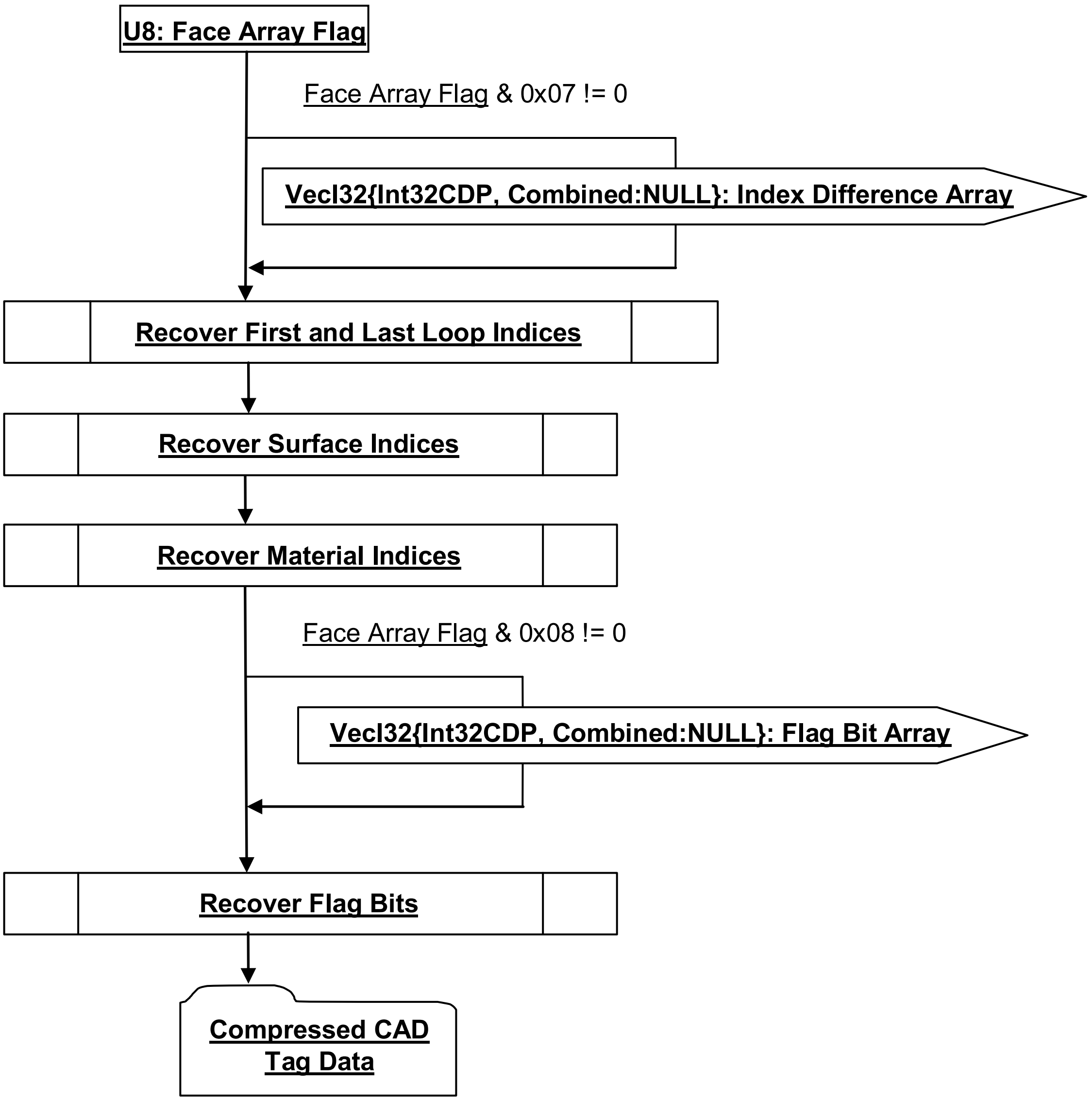

U8: Face Array Flag
Face Array Flag & 0x07 != 0
VecI32{Int32CDP, Combined:NULL}: Index Difference Array
Recover First and Last Loop Indices
Recover Surface Indices
Recover Material Indices
Face Array Flag & 0x08 != 0
VecI32{Int32CDP, Combined:NULL}: Flag Bit Array
Recover Flag Bits
Compressed CAD Tag Data

Figure 193 — Faces Topology Data collection

U8: Face Array Flag

Face Array Flag indicates which arrays of face topology data are not trivial and therefore encoded.

VecI32{Int32CDP, Combined:NULL}: Index Difference Array

Index Difference Array is a combined vector of indices encoded using Int32 version of CODEC and Combined Predictor Type, with its content decided by the value of Face Array Flag. If Face Array Flag has bit 0x01 set, then the vector of integer values obtained by subtracting first loop index from last loop index in each face is appended to the end of Index Difference Array. If Face Array Flag has bit 0x02 set, then the vector of integer values obtained by subtracting surface index from face index in each face is appended to the end of Index Difference Array. If Face Array Flag has bit 0x04 set, then the vector of integer values representing the material index of each face is appended to the end of Index Difference Array.

Recover First and Last Loop Indices

The first loop index of the first face is 0, and the last loop index of the first face is element 0 of Index Difference Array if the array is encoded, or 0 if bit 0x01 of Face Array Flag is not set. The first loop index of face $k, k \geq 1$ equals to the last loop index of face $k-1$ plus 1. The last loop index of face $k, k \geq 1$ equals to the first loop index of face k plus element k of Index Difference Array, or 0 if bit 0x01 of Face Array Flag is not set.

Recover Surface Indices

The surface index of each face equals to the face index if bit 0x02 of Face Array Flag is not set. Otherwise the surface index of face k is obtained by substracting element $k + \text{offset}$ of Index Difference Array from face index k, where offset is equal to Face Count if bit 0x01 of Face Array Flag is set and 0 if the bit is not set.

Recover Material Indices

The material index of each face equals to 0 if bit 0x04 of Face Array Flag is not set. Otherwise the material index of face k equals to the element $k + \text{offset}$ of Index Difference Array, where offset is equal to twice of Face Count if both bit 0x01 and bit 0x02 of Face Array Flag are set, is equal to Face Count if either bit 0x01 or bit 0x02 of Face Array Flag is set, and is equal to 0 if neither bit is set.

VecI32{Int32CDP, Combined:NULL}: Flag Bit Array

Only the lower 24 bits of the four integer indices, namely first loop index, last loop index, surface index, and material index, are used as integer identifiers. The other bits of these integers are documented as reserved for future use.

Table 79 — JT ULP Flag Bit Array Look Index values

	24	25	26	27	28	29	30	31
First Loop Index	Surface Type			U Knot Type		V Knot Type		isNormalReversed
Last Loop Index	isIsolated	Reserved						
Surface Index	Reserved							
Material Index	Reserved							

Each element of Flag Bit Array is a 32 bit integer obtained by combining all 32 flag bits from four different integers. More specifically:

— Bits 0~7 of Flag Bit Array are equal to bits 24~31 of First Loop Index.

— Bits 8~15 of Flag Bit Array are equal to bits 24~31 of Last Loop Index.

— Bits 16~23 of Flag Bit Array are equal to bits 24~31 of Surface Index.

— Bits 24~31 of Flag Bit Array are equal to bits 24~31 of Material Index.

Supported Surface Type

In an uncompressed/decoded form, the supported surface types are listed below.

Table 80 — JT ULP Supported Surface Type values

0	Nurbs
1	Plane
2	Cylinder
3	Cone
4	Sphere
5	Torus
6	Reserved
7	Reserved

Supported Knot Type

In an uncompressed/decoded form, the supported knot types are listed below. The knot type of the underlying surface along both U and V parameter directions are encoded.

Table 81 — JT ULP Supported Knot Type Values

0	No Pattern
1	No knot value in between the clamped end knots
2	All knot values in between the end knots increase with an even interval
3	All knot values in between the end knots repeat exactly once, and the distinct values increase with an even interval

In an uncompressed/decoded form, the Face Reverse Normal Flag has the following meaning:

Table 82 — JT ULP Face Reverse Normal Flag values

= 0	Face normal is not reversed
= 1	Face normal is reversed.

Recover Flag Bits

If Face Array Flag & 0x08 is equal to 0, then each element in Flag Bit Array is set to have value 0. The flag bits are recovered by assigning bits 0~7 of Flag Bit Array to bits 24~31 of First Loop Index, bits 8~15 of Flag Bit Array to bits 24~31 of Last Loop Index, bits 16~23 of Flag Bit Array to bits 24~31 of Surface Index, and bits 24~31 of Flag Bit Array to bits 24~31 of Material Index.

Loops Topology Data

A Loop (often called Trimming Loop) defines in parameter space a 1D boundary around which geometric surfaces are trimmed to form a Face. Loops Topology Data specifies the CoEdges making up each Loop along with an anti-hole flag and identifier tag for each Loop. The complete description of loop and its relation to the CoEdges can be found in Loops Topology Data.

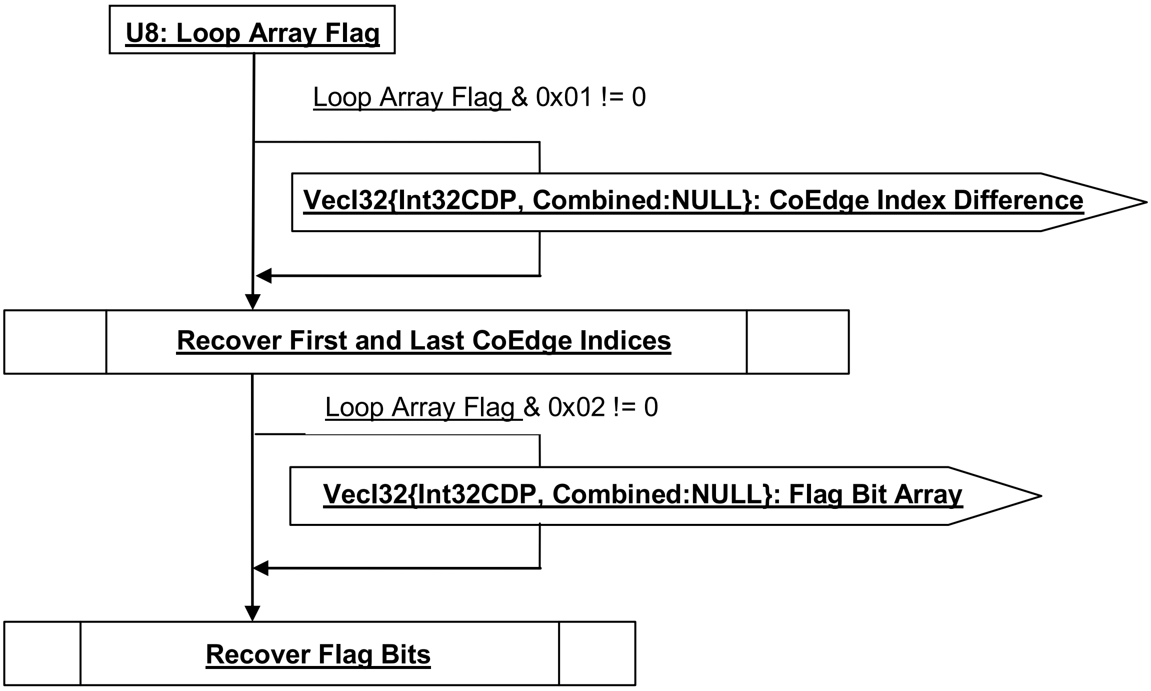

U8: Loop Array Flag
Loop Array Flag & 0x01 != 0
VecI32{Int32CDP, Combined:NULL}: CoEdge Index Difference
Recover First and Last CoEdge Indices
Loop Array Flag & 0x02 != 0
VecI32{Int32CDP, Combined:NULL}: Flag Bit Array
Recover Flag Bits

Figure 194 — Loops Topology Data collection

U8: Loop Array Flag

Loop Array Flag indicates which arrays of loop topology data are not trivial and therefore encoded.

VecI32{Int32CDP, Combined:NULL}: CoEdge Index Difference

CoEdge Index Difference is a vector of indices representing the integer value by subtracting first CoEdge index from last CoEdge index in each loop, encoded using Combined Predictor Type. CoEdge Index Difference is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Recover First and Last CoEdge Indices

The first CoEdge index of the first loop is 0, and the last CoEdge index of the first loop is element 0 of CoEdge Index Difference. The first CoEdge index of loop $k, k \geq 1$ equals to the last CoEdge index of loop $k - 1$ plus 1. The last CoEdge index of loop $k, k \geq 1$ equals to the first CoEdge index of loop k plus element k of CoEdge Index Difference array.

VecI32{Int32CDP, Combined:NULL}: Flag Bit Array

Only the lower 24 bits of the two integer indices, namely first CoEdge index and last CoEdge index are used as integer identifiers. The other bits of these integers documented as reserved for future use.

Table 83 — JT ULP Loops Topology Flag Bit Array values

	24	25	26	27	28	29	30	31
First CoEdge Index	Reserved							isAntiHoleLoop
Last CoEdge Index	Reserved							

Bits 0~7 of Flag Bit Array are equal to bits 24~31 of First CoEdge Index

Bits 8~15 of Flag Bit Array are equal to bits 24~31 of Last CoEdge Index

Bits 16~31 of Flag Bit Array are set to be 0

In an uncompressed/decoded form, the AntiHole Loop Flag has the following meaning:

Table 84 — JT ULP Loops Topology Reverse Normal Flag values

= 0	Loop is not an anti-hole Loop
= 1	Loop is an anti-hole Loop

Recover Flag Bits

The flag bits are recovered by assigning bits 0~7 of Flag Bit Array to bits 24~31 of First CoEdge Index, and bits 8~15 of Flag Bit Array to bits 24~31 of Last CoEdge Index.

CoEdges Topology Data

A CoEdge defines a parameter space edge trim Loop segment (i.e. the projection of an Edge into the parameter space of the Face). CoEdges Topology Data specifies the underlying Edge and PCS Curve making up each CoEdge along with a MCS curve reversed flag and tag for each CoEdge. The complete description of CoEdge and its relation to the Edge can be found in CoEdges Topology Data.

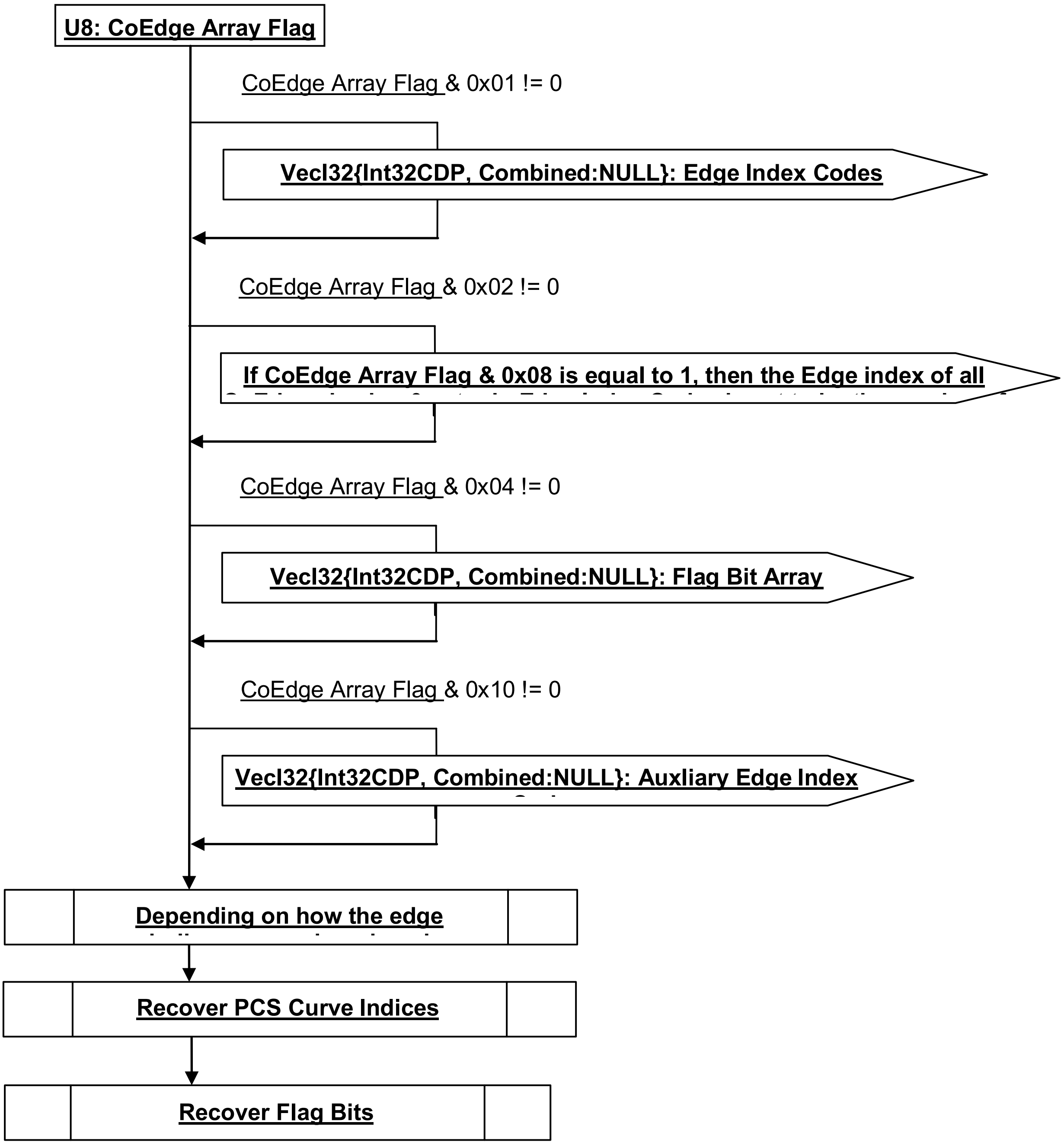

U8: CoEdge Array Flag
CoEdge Array Flag & 0x01 != 0
VecI32{Int32CDP, Combined:NULL}: Edge Index Codes
CoEdge Array Flag & 0x02 != 0
If CoEdge Array Flag & 0x08 is equal to 1, then the Edge index of all
CoEdge Array Flag & 0x04 != 0
VecI32{Int32CDP, Combined:NULL}: Flag Bit Array
CoEdge Array Flag & 0x10 != 0
VecI32{Int32CDP, Combined:NULL}: Auxliary Edge Index
Depending on how the edge
Recover PCS Curve Indices
Recover Flag Bits

Figure 195 — CoEdges Topology Data collection

U8: CoEdge Array Flag

CoEdge Array Flag indicates which arrays of coedge topology data are not trivial and therefore encoded.

VecI32{Int32CDP, Combined:NULL}: Edge Index Codes

Edge Index Codes is a vector of integer indices representing the Edge index for each CoEdge, encoded using Combined Predictor Type. Edge Index Codes is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Depending on how the edge indices are assigned, and indicated by bit 0x08 in CoEdge Array Flag, the Edge Index Codes may represent the Edge Index information in two different ways. If the edge indices are randomly assigned, for example as shown in Figure 196, then each element in Edge Index Codes represents the integer value by subtracting the Edge index from the CoEdge index for each CoEdge. For the example shown in Figure 196, the integer values in Edge Index Codes are -7, -4, -4, 3, 3, -1, -2, 5, 5, 1, 5, 7 and bit 0x08 is turned off in CoEdge Array Flag. If the edge indices are chosen to be based on the sequence of the reference from the parent coedges when the coedges are visited sequentially, as shown in Figure 197, then each element in Edge Index Codes has value 0 if the edge is visited the first time, and value 1 if the edge is visited the second time. For the example shown in Figure 197, the interger values in Edge Index Codes are 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, and bit 0x08 is turned on in CoEdge Array Flag. Note that the edge index for all CoEdges with 0 entry can be figured out by counting number of zeros in Edge Index Codes preceding (not including) this entry. Take CoEdge 6 that has entry 0 in Edge Index Codes for example. The number of zeros before this entry is 5, which is equal to the edge index of CoEdge 6. Therefore only the edge indices of those CoEdges with entry value 1 in Edge Index Codes need be stored. These edge indices are stored in Auxliary Edge Index Codes.

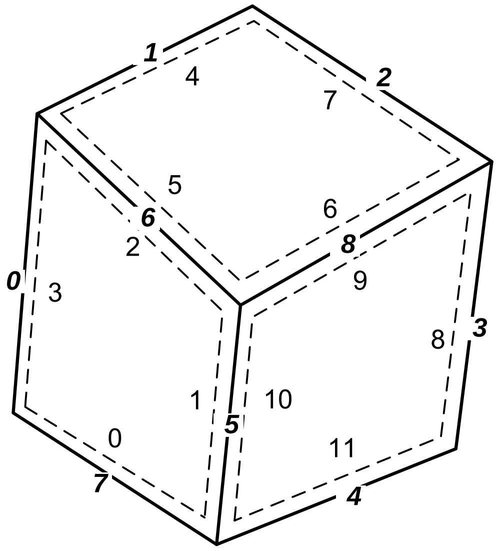


Figure 196 — Sample Model with Randomly Assigned Edge Indices

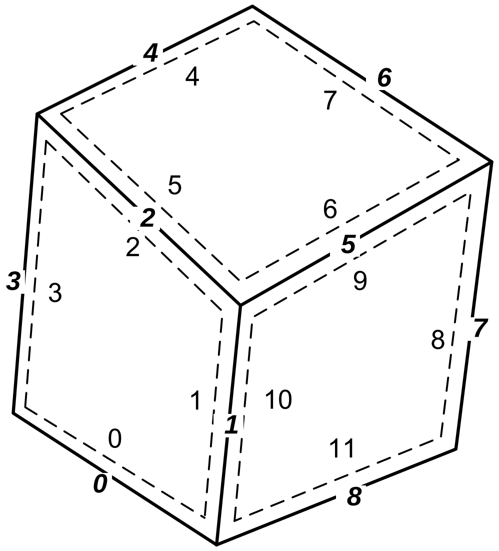


Figure 197 — Sequentially Assigned Edge Indices

VecI32{Int32CDP, Combined:NULL}: Auxliary Edge Index Codes

Auxliary Edge Index Codes is an optional field and only exists if CoEdge Array Flag & 0x08 is not equal to 0. It contains the Edge indices that appear again during sequential traversal of the CoEdges. For the example shown in Figure 197, the entries in Auxliary Edge Index Codes are 2, 5, 1. More detailed explanation of what these entires mean can be found in Edge Index Codes.

Recover Edge Indices

If CoEdge Array Flag & 0x08 is equal to 0, then the Edge index of each CoEdge is equal to the CoEdge index if CoEdge Array Flag & 0x01 is equal to 0.. Otherwise, the Edge index of CoEdge with index k can be computed by substracting element k of Edge Index Codes array from k, the CoEdge index.

If CoEdge Array Flag & 0x08 is equal to 1, then the Edge index of all CoEdges having 0 entry in Edge Index Codes is set to be the number of zeros in Edge Index Codes preceding (not including) this entry. For all CoEdges that have 1 in Edge Index Codes, their edge indices are sequentially assigned as the corresponding value in Auxliary Edge Index Codes.

VecI32{Int32CDP, Combined:NULL}: PCS Curve Index Difference

PCS Curve Index Difference is a vector of indices representing the integer value by subtracting the PCS Curve index from the CoEdge index for each CoEdge, encoded using Combined Predictor Type. PCS Curve Index Difference is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Recover PCS Curve Indices

If CoEdge Array Flag & 0x02 is equal to 0, then the PCS Curve index of each CoEdge is equal to the CoEdge index. Otherwise, the PCS Curve index of CoEdge with index k can be computed by substracting element k of PCS Curve Index Difference array from k, the CoEdge index.

VecI32{Int32CDP, Combined:NULL}: Flag Bit Array

Only the lower 24 bits of the two integer indices, namely Edge index and PCS Curve index, are used as integer identifiers. The other bits of these integers are documented as reserved for future use

Table 85 — JT ULP Recover Edge Indices Flag Bit Array values

	24	25	26	27	28	29	30	31
Edge Index	Knot Type		Domain Type			PCS Curve Type		isXYZReversed
PCS Curve Index	isUvInc	Reserved						

Bits 0~7 of Flag Bit Array are equal to bits 24~31 of Edge Index

Bits 8~15 of Flag Bit Array are equal to bits 24~31 of PCS Curve Index

Bits 16~31 of Flag Bit Array are set to be 0

The Knot Type, defined in Supported Knot Type, is an integer with its value between 0 and 3.

Domain Type

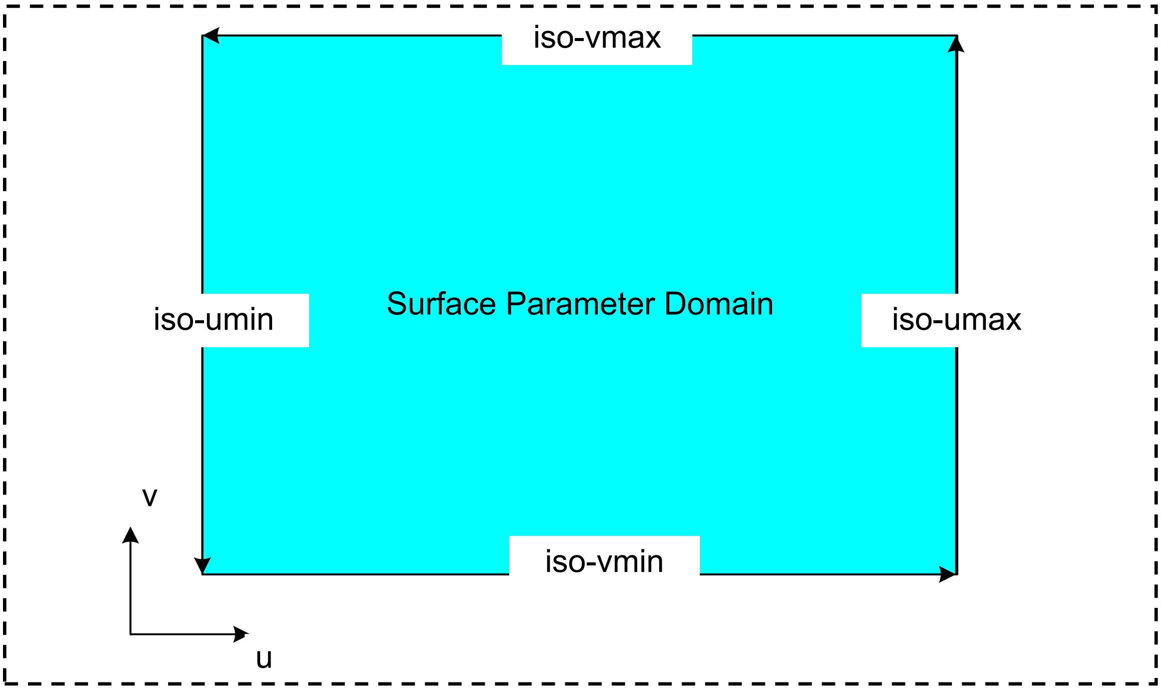


Figure 198 — Surface Domain Classification

In an uncompressed/decoded form, the supported PCS Curve types are listed below.

Table 86 — JT ULP Recover Edge Indices PCS curve type values

0	General
1	PCS curve is coincident with iso-umin curve in the surface parameter domain
2	PCS curve is coincident with iso-umax curve in the surface parameter domain
3	PCS curve is coincident with iso-vmin curve in the surface parameter domain
4	PCS curve is coincident with iso-vmax curve in the surface parameter domain
5	Reserved
6	Reserved
7	PCS curve is to be derived from MCS curve and surface geometry

PCS Curve Type

In an uncompressed/decoded form, the supported PCS Curve types are listed below.

Table 87 — JT ULP PCS Curve Type values

0	Nurbs
1	Line
2	Circle
3	Reserved

In an uncompressed/decoded form, the XYZReversed Flag has the following meaning:

Table 88 — JT ULP PCS Curve Type XYZ Reversed Flag values

= 0	Directional sense of associated edges MCS curve should not be interpreted as opposite the direction its parameterization implies.
= 1	Directional sense of associated edges MCS curve should be interpreted as opposite the direction its parameterization implies.

In an uncompressed/decoded form, the isUVInc Flag has the following meaning:

Table 89 — JT ULP PCS Curve Type isUVInc Flag values

= 0	PCS Curve is iso-parameteric in surface parameter domain in one direction and the parameter increases in the other direction
= 1	PCS Curve is iso-parameteric in surface parameter domain in one direction and the parameter decreases in the other direction

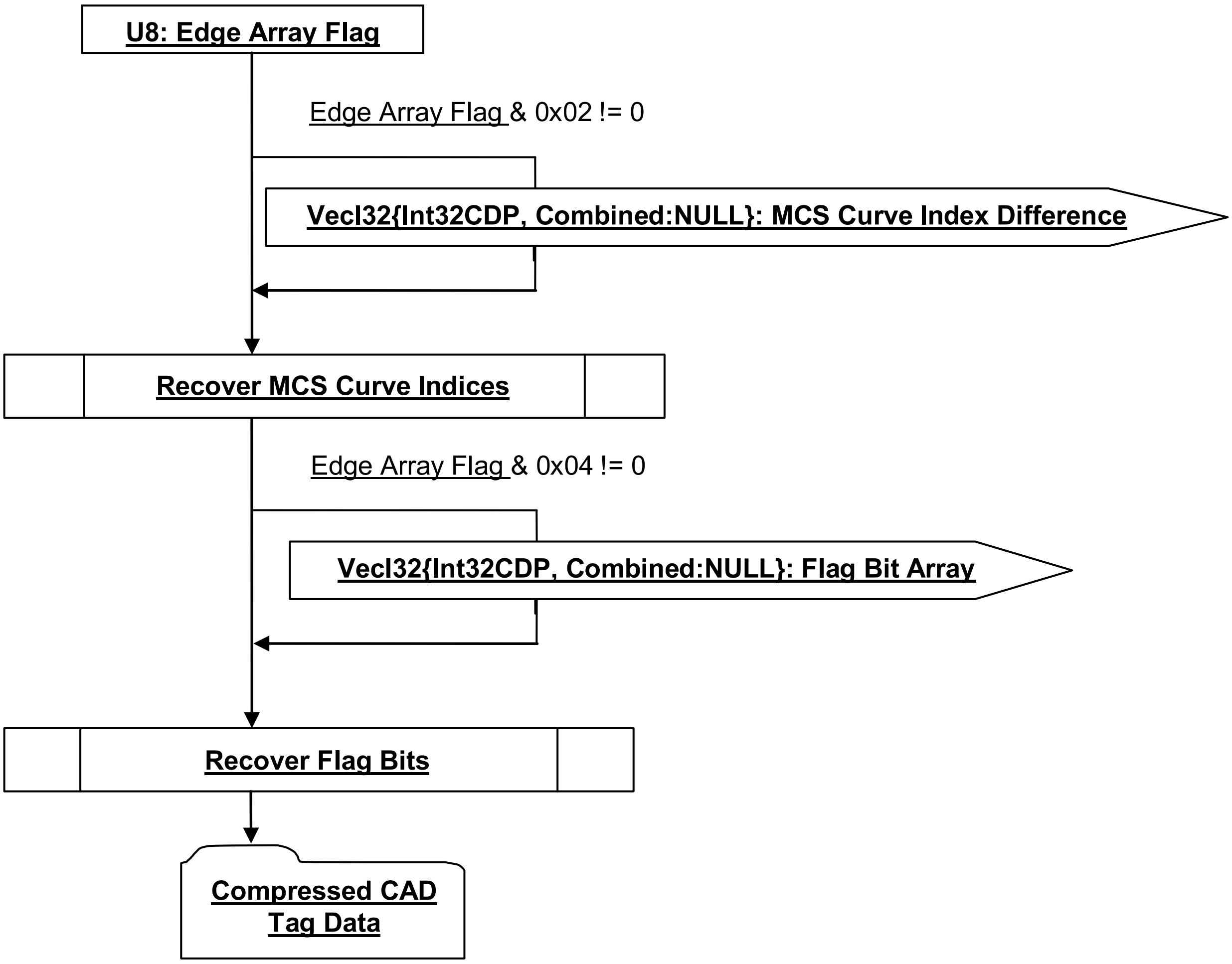
The isUVInc flag is set only if the Domain Type of this CoEdge has value between 1 and 4 inclusive.

Recover Flag Bits

If CoEdge Array Flag & 0x04 is equal to 0, then each element in Flag Bit Array is set to have value 0. The flag bits are recovered by assigning bits 0~7 of Flag Bit Array to bits 24~31 of Edge Index, and bits 8~15 of Flag Bit Array to bits 24~31 of PCS Curve Index.

Edges Topology Data

An Edge defines a model space trim Loop segment. Edges Topology Data specifies the underlying MCS Curve along with an identification tag for each Edge. The complete description of Edge can be found in Edges Topology Data. Note that the start and end vertex index information is not stored. Instead it is recovered (12.1.4 Information Recovery).


U8: Edge Array Flag
Edge Array Flag & 0x02 != 0
VecI32{Int32CDP, Combined:NULL}: MCS Curve Index Difference
Recover MCS Curve Indices
Edge Array Flag & 0x04 != 0
VecI32{Int32CDP, Combined:NULL}: Flag Bit Array
Recover Flag Bits
Compressed CAD Tag Data

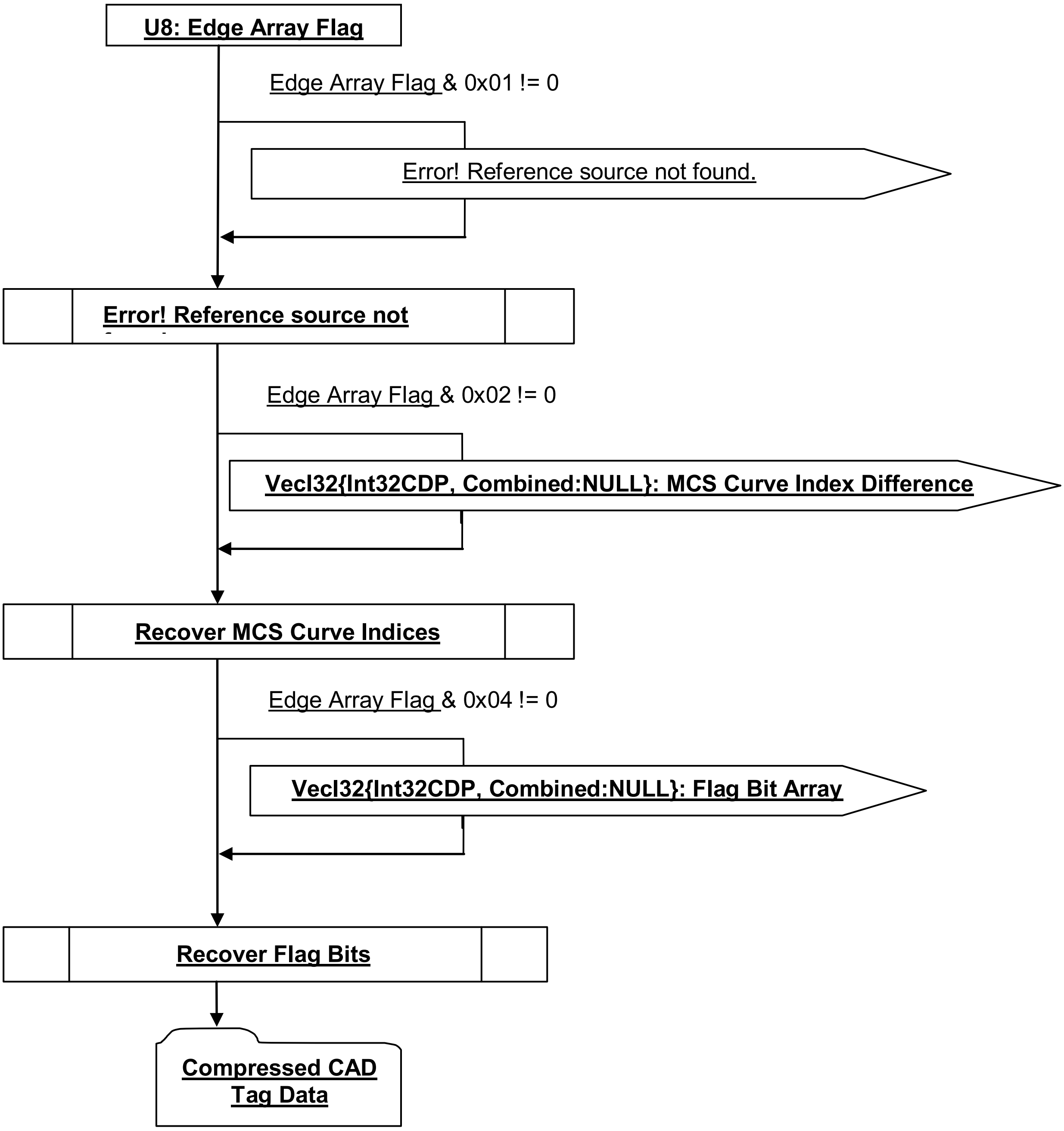

U8: Edge Array Flag
Edge Array Flag & 0x01 != 0
Error! Reference source not found.
Error! Reference source not
Edge Array Flag & 0x02 != 0
VecI32{Int32CDP, Combined:NULL}: MCS Curve Index Difference
Recover MCS Curve Indices
Edge Array Flag & 0x04 != 0
VecI32{Int32CDP, Combined:NULL}: Flag Bit Array
Recover Flag Bits
Compressed CAD Tag Data

Figure 199 — Edges Topology Data collection

U8: Edge Array Flag

Edge Array Flag indicates which arrays of edge topology data are not trivial and therefore encoded.

VecI32{Int32CDP, Combined:NULL}: MCS Curve Index Difference

MCS Curve Index Difference is a vector of indices representing the integer value by subtracting the MCS Curve index from the Edge index for each Edge, encoded using Combined Predictor Type. MCS Curve Index Difference is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Recover MCS Curve Indices

If Edge Array Flag & 0x02 is equal to 0, then the MCS Curve index of each Edge is equal to the Edge index. Otherwise, the MCS Curve index of Edge with index k can be computed by substracting element k of MCS Curve Index Difference array from k, the Edge index.

VecI32{Int32CDP, Combined:NULL}: Flag Bit Array

Only the lower 24 bits of the three integer indices, namely MCS Curve index, Start Vertex index, and End Vertex index, are used as integer identifiers. The other bits of these integers documented as reserved for future use..

Table 90 — JT ULP Edges Topology Recover MCS Curve Indices Flag Bit Array values

	24	25	26	27	28	29	30	31
MCS Curve Index	Knot Type		MCS Curve Type		Reserved			
Start Vertex Index	Reserved							
End Vertex Index	Reserved							

The Knot Type, defined in Supported Knot Type, is an integer with its value between 0 and 3.

MCS Curve Type

In an uncompressed/decoded form, the supported MCS Curve types are listed below.

Table 91 — JT ULP Edges Topology Recover MCS Curve Type values

0	Nurbs
1	Line
2	Circle
3	Projection: MCS curve geometry is to be computed from surface geometry and/or PCS curve geometry

Recover Flag Bits

If Edge Array Flag & 0x04 is equal to 0, then each element in Flag Bit Array is set to have value 0. The flag bits are recovered by assigning bits 0~7 of Flag Bit Array to bits 24~31 of MCS Curve Index.

Vertices Topology Data

Vertices Topology Data is not stored on disk. Instead it is constructed (12.1.4 Information Recovery).

12.1.2 Geometric Data

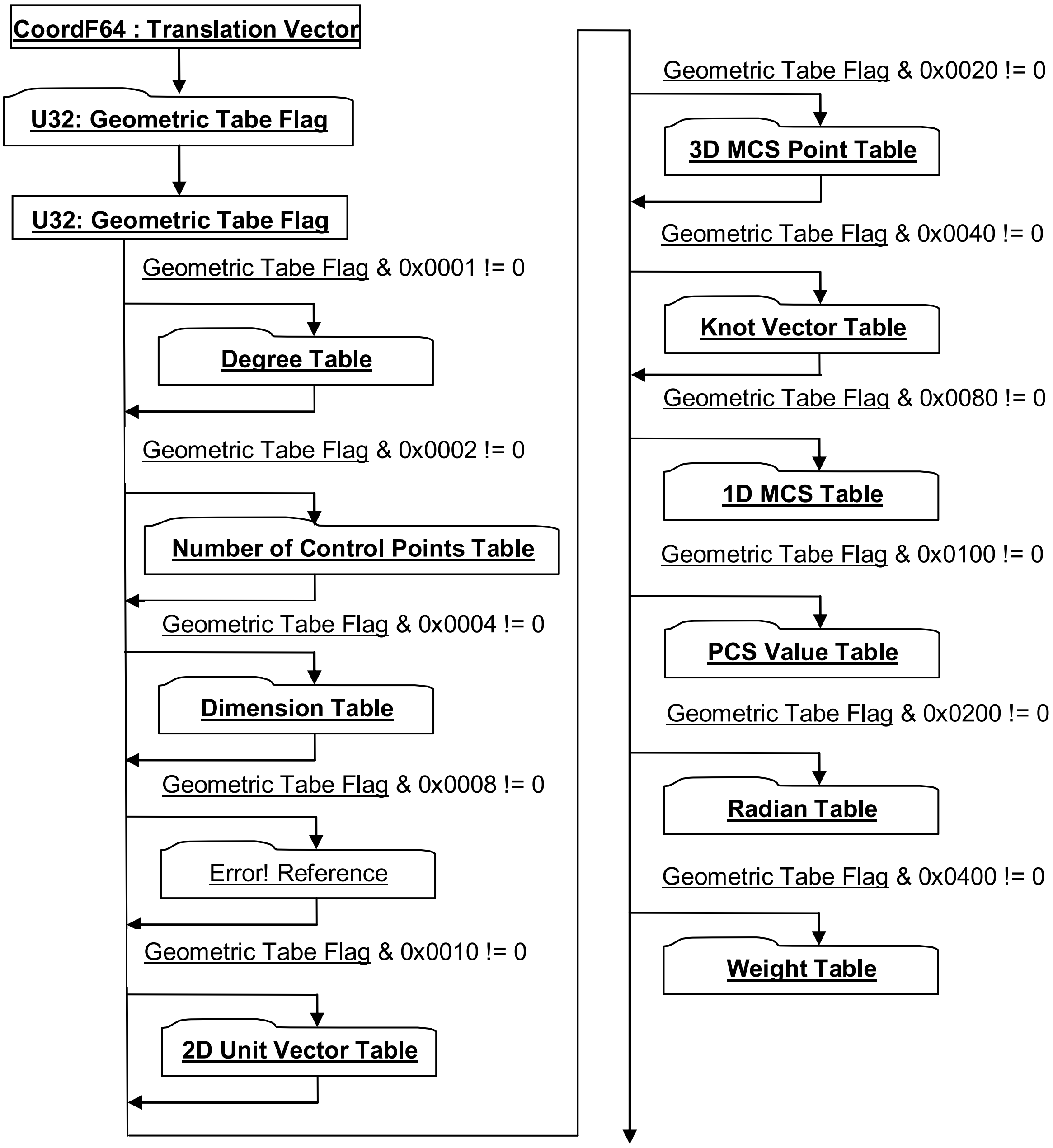

CoordF64 : Translation Vector
U32: Geometric Tabe Flag
U32: Geometric Tabe Flag
Geometric Tabe Flag & 0x0001 != 0
Degree Table
Geometric Tabe Flag & 0x0002 != 0
Number of Control Points Table
Geometric Tabe Flag & 0x0004 != 0
Dimension Table
Geometric Tabe Flag & 0x0008 != 0
Error! Reference
Geometric Tabe Flag & 0x0010 != 0
2D Unit Vector Table
Geometric Tabe Flag & 0x0020 != 0
3D MCS Point Table
Geometric Tabe Flag & 0x0040 != 0
Knot Vector Table
Geometric Tabe Flag & 0x0080 != 0
1D MCS Table
Geometric Tabe Flag & 0x0100 != 0
PCS Value Table
Geometric Tabe Flag & 0x0200 != 0
Radian Table
Geometric Tabe Flag & 0x0400 != 0
Weight Table

Figure 200 — Geometric Data collection

CoordF64 : Translation Vector

Translation Vector is a 3-dimensional vector that represents how the ULP geometry is defined w.r.t. the original B-Rep definition from which ULP geometry is derived. If the Translation Vector is not zero vector, then the ULP geometry read from disk is translated from original B-Rep definition by the amount of Translation Vector. This is usually done by the JT writer implementation to improve numerical accuracy of floating point numbers in the ULP geometry. It is important for all the JT readers to take this Translation Vector into consideration when consuming ULP geometry. For example if a LOD is generated from ULP geometry, e.g. by tessellation, then the LOD geometry shall be translated to undo the effect of Translation Vector for it be consistent with the original B-Rep definition. In other words, if we denote the Translation Vector as $\mathbf{v}$, then the LOD geometry from ULP shall be translated by $-\mathbf{v}$.

U32: Geometric Tabe Flag

Geometric Tabe Flag indicates which geometric tables are not trivial and therefore encoded.

Geometric Entity Counts

Geometric Entity Counts data collection defines the counts for each of the various geometric entities within a ULP.

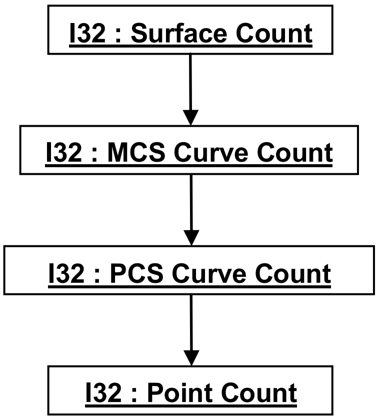


Figure 201 — Geometric Entity Counts

I32 : Surface Count

Surface Count indicates the number of distinct geometric surface entities in the ULP

I32 : MCS Curve Count

MCS Curve Count indicates the number of distinct geometric (Model Coordinate Space) curves (i.e. XYZ curve) entities in the ULP.

I32 : PCS Curve Count

PCS Curve Count indicates the number of distinct geometric Parameter Coordinate Space curves (i.e. UV curve) entities in the ULP

I32 : Point Count

Point Count indicates the number of distinct geometric point entities in the ULP.

Degree Table

Degree Table stores a vector of integers that represent the degree information of Nurbs surfaces and/or curves. If the ULP does not contain any Nurbs entity, then the table is empty and bit 0x0001 in Geometric Tabe Flag is set to be 0.

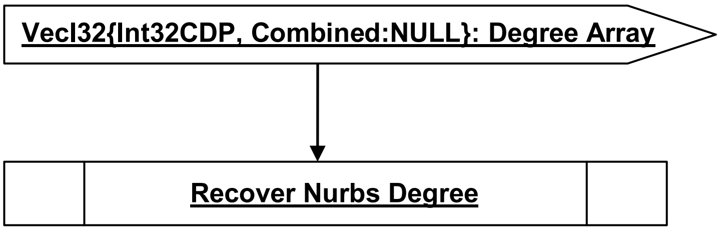


Figure 202 — Degree Table data collection

VecI32{Int32CDP, Combined:NULL}: Degree Array

Degree Array is a vector of integers that stores the degree information for all the Nurbs entities in the ULP, encoded using Combined Predictor Type. Degree Array is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Recover Nurbs Degree

The logic diagram to recover degree information for all the Nurbs entities in the ULP from the Degree Array is shown below.

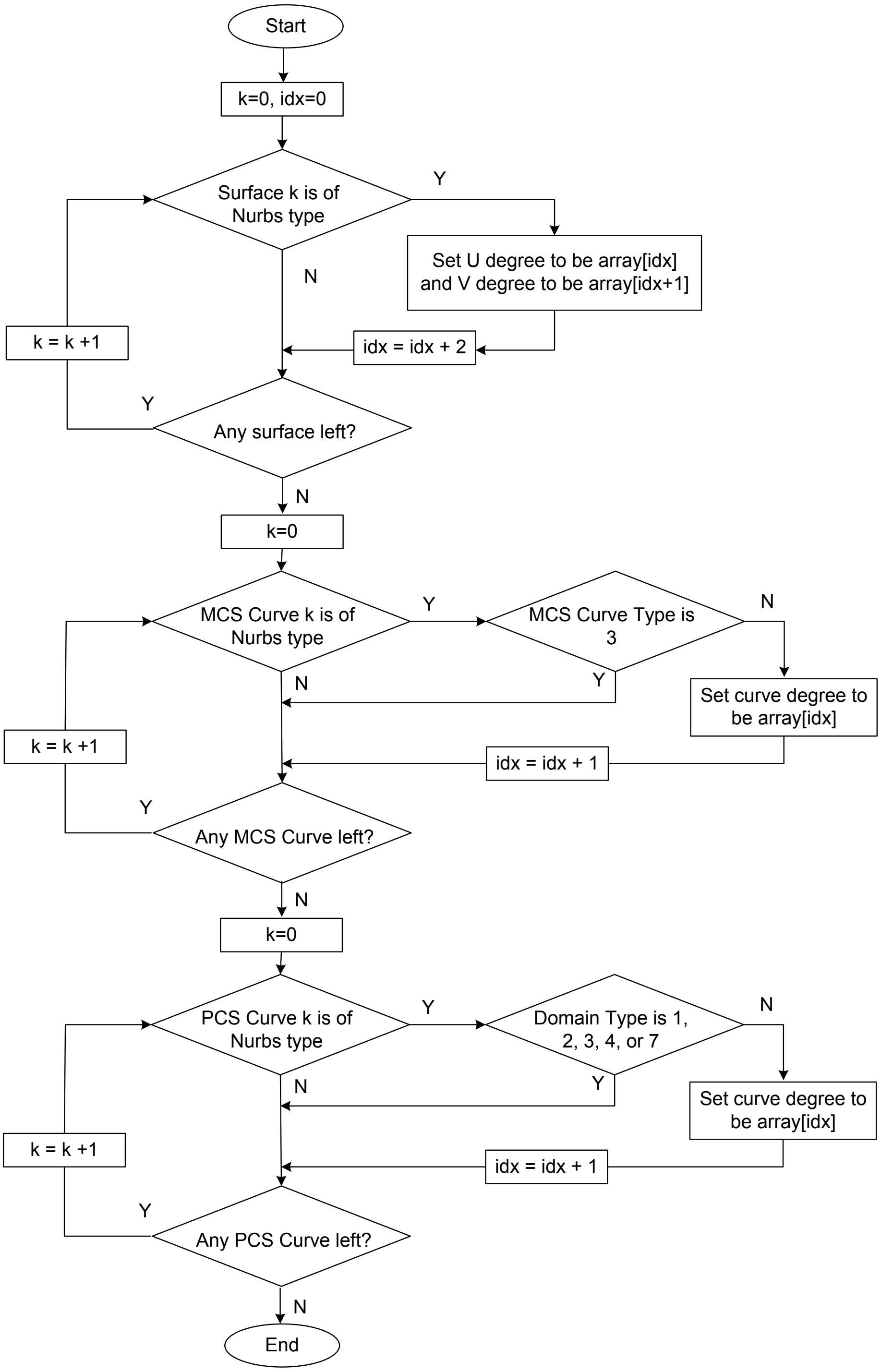

Start
k=0, idx=0
Surface k is of Nurbs type
Y
N
Set U degree to be array[idx] and V degree to be array[idx+1]
idx = idx + 2
k = k +1
Y
Any surface left?
N
k=0
MCS Curve k is of Nurbs type
Y
MCS Curve Type is 3
N
N
Y
Set curve degree to be array[idx]
k = k +1
idx = idx + 1
Y
Any MCS Curve left?
N
k=0
PCS Curve k is of Nurbs type
Y
Domain Type is 1, 2, 3, 4, or 7
N
N
Y
Set curve degree to be array[idx]
k = k +1
idx = idx + 1
Y
Any PCS Curve left?
N
End

Figure 203 — Recover Nurbs Degree

Number of Control Points Table

Number of Control Points Table stores a vector of integers that represent the number of control points information of Nurbs surfaces and/or curves. If the ULP does not contain any Nurbs entity, then the table is empty and bit 0x0002 in Geometric Tabe Flag is set to be 0.

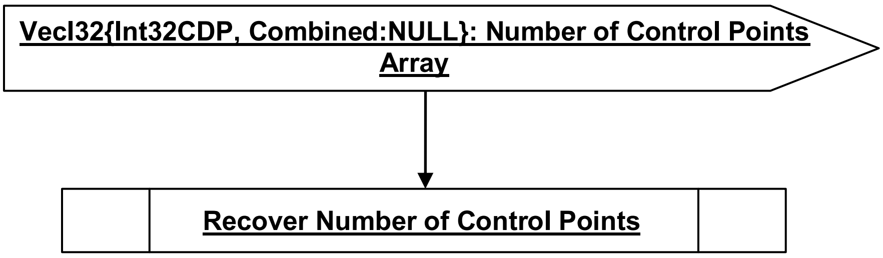

VecI32{Int32CDP, Combined:NULL}: Number of Control Points Array
Recover Number of Control Points

Figure 204 — Number of Control Points Table data collection

VecI32{Int32CDP, Combined:NULL}: Number of Control Points Array

Number of Control Points Array is a vector of integers that stores the number of control points information for all the Nurbs entities in the ULP, encoded using Combined Predictor Type. Number of Control Points Array is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Recover Number of Control Points

The logic diagram to recover number of control points information for all the Nurbs entities in the ULP from the Number of Control Points Array is shown below.

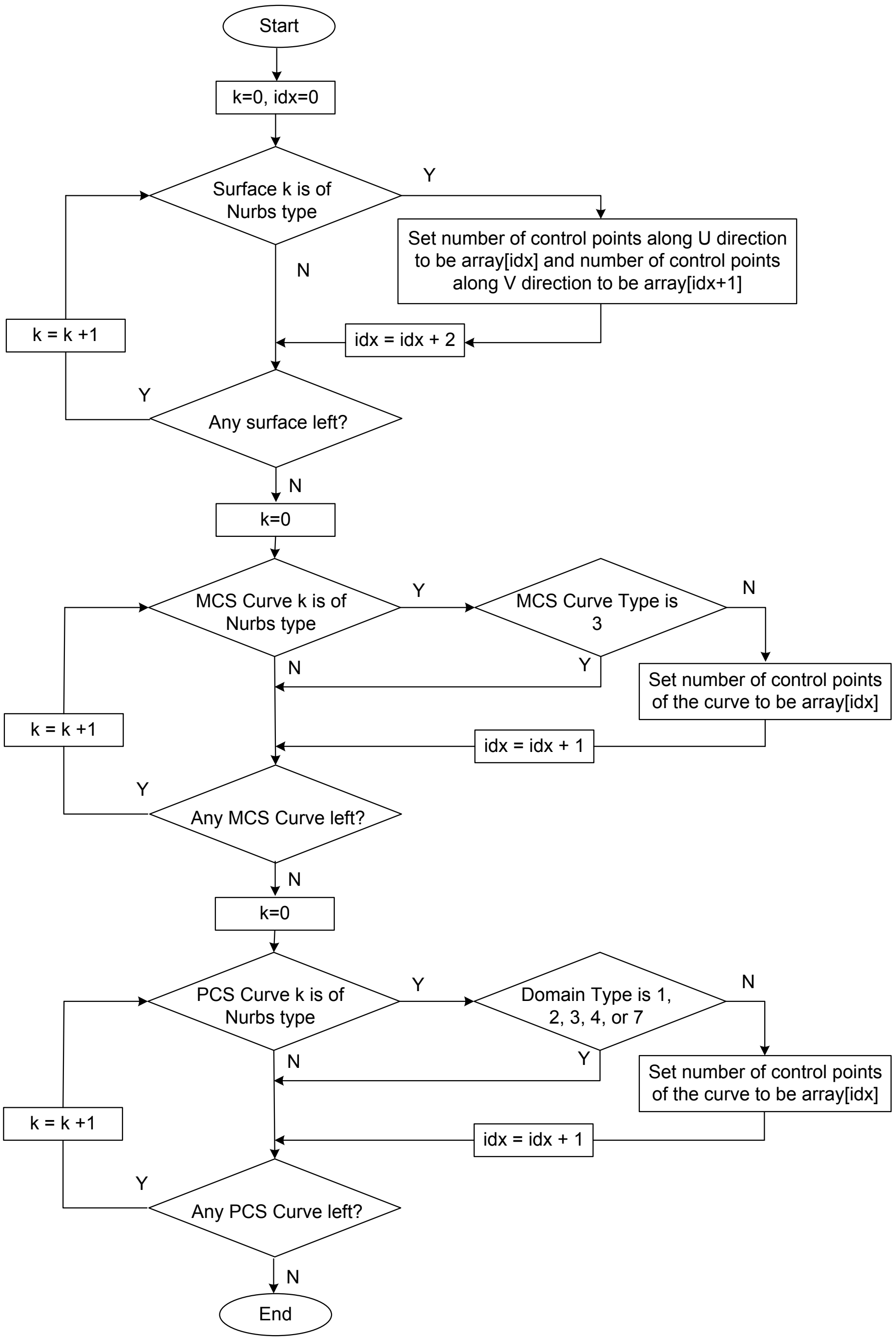

Start
k=0, idx=0
Surface k is of Nurbs type
Y
Set number of control points along U direction to be array[idx] and number of control points along V direction to be array[idx+1]
N
k = k +1
idx = idx + 2
Y
Any surface left?
N
k=0
MCS Curve k is of Nurbs type
Y
MCS Curve Type is 3
N
N
Y
Set number of control points of the curve to be array[idx]
k = k +1
idx = idx + 1
Y
Any MCS Curve left?
N
k=0
PCS Curve k is of Nurbs type
Y
Domain Type is 1, 2, 3, 4, or 7
N
N
Y
Set number of control points of the curve to be array[idx]
k = k +1
idx = idx + 1
Y
Any PCS Curve left?
N
End

Figure 205 — Recover Number of Control Points

Dimension Table

Dimension Table stores a vector of integers that represent the dimension information of Nurbs surfaces and/or curves. If the ULP does not contain any Nurbs entity, then the table is empty and bit 0x0004 in Geometric Tabe Flag is set to be 0.

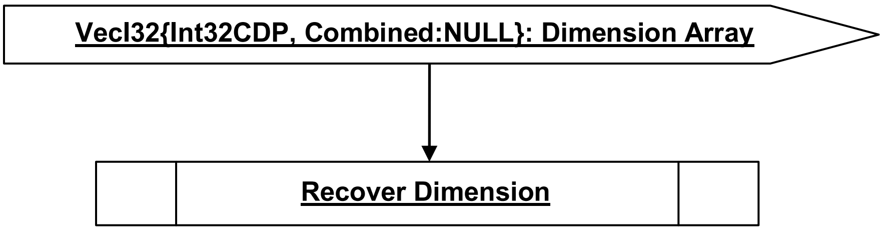


Figure 206 — Dimension Table data collection

VecI32{Int32CDP, Combined:NULL}: Dimension Array

Dimension Array is a vector of integers that stores the dimension information for all the Nurbs entities in the ULP, encoded using Combined Predictor Type. Dimension Array is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Recover Dimension

The logic diagram to recover dimension information for all the Nurbs entities in the ULP from the Dimension Array is shown below.

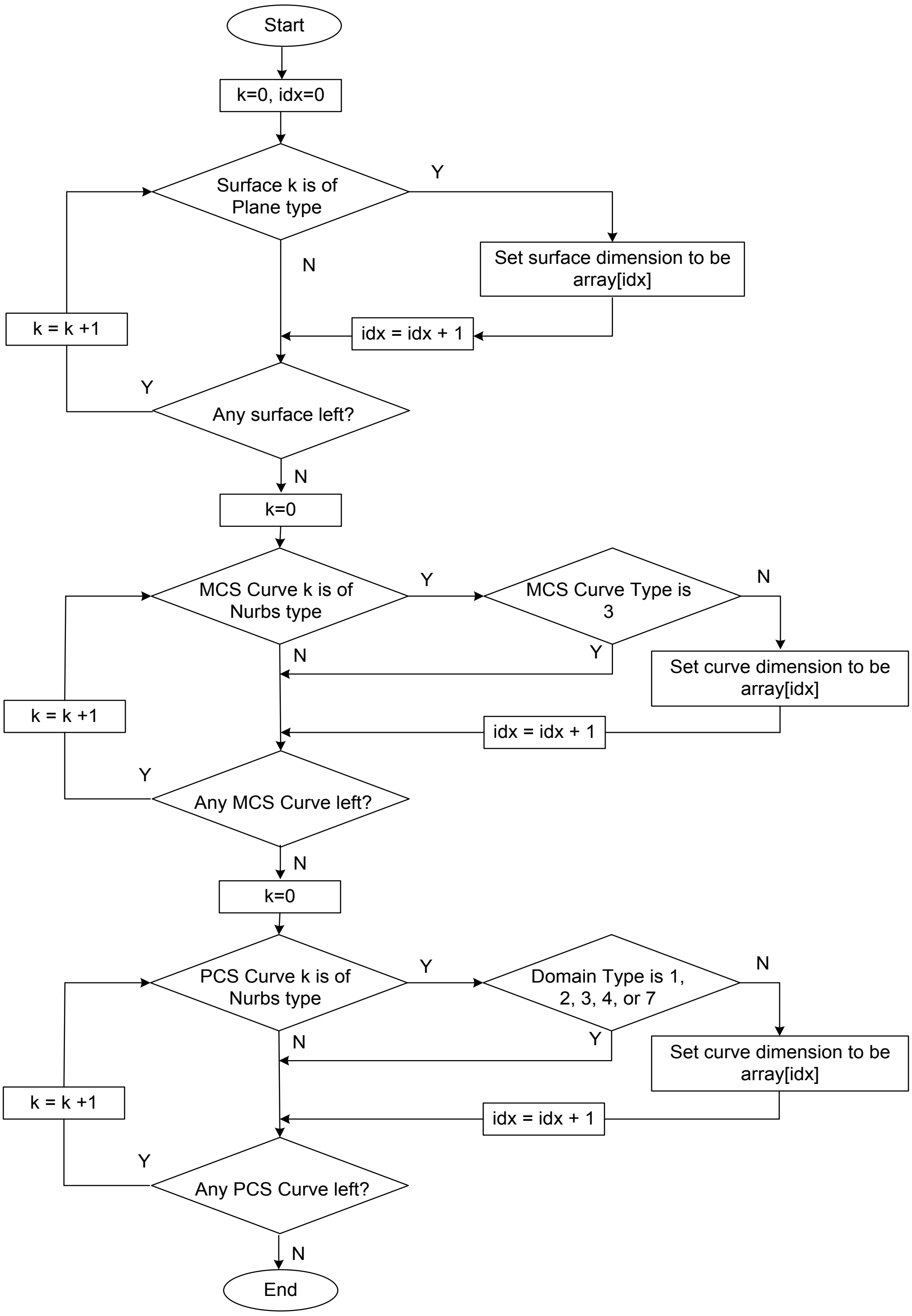

Start
k=0, idx=0
Surface k is of Plane type
Y
N
Set surface dimension to be array[idx]
k = k +1
idx = idx + 1
Y
Any surface left?
N
k=0
MCS Curve k is of Nurbs type
Y
MCS Curve Type is 3
N
N
Y
Set curve dimension to be array[idx]
k = k +1
idx = idx + 1
Y
Any MCS Curve left?
N
k=0
PCS Curve k is of Nurbs type
Y
Domain Type is 1, 2, 3, 4, or 7
N
N
Y
Set curve dimension to be array[idx]
k = k +1
idx = idx + 1
Y
Any PCS Curve left?
N
End

Figure 207 — Recover Dimension

Error! Reference source not found. stores an array of unit vectors in 3D that form part of the analytic surface or curve representation in ULP. If the ULP does not contain any analytic entity, then the table is empty and bit 0x0008 in Geometric Tabe Flag is set to be 0. The supported analytic surface types include plane, cylinder, cone, sphere, and torus, and the supported analytic curve types include line and circle for both parameter space and model space curves. The analytic representation of ULP follows XT convention as detailed in Annex E.

Similar to the coding of 13.1.4 Compressed Vertex Normal Array, each 3D unit vector is encoded as a single 32 bit integer using 13.2.4 Deering Normal CODEC.

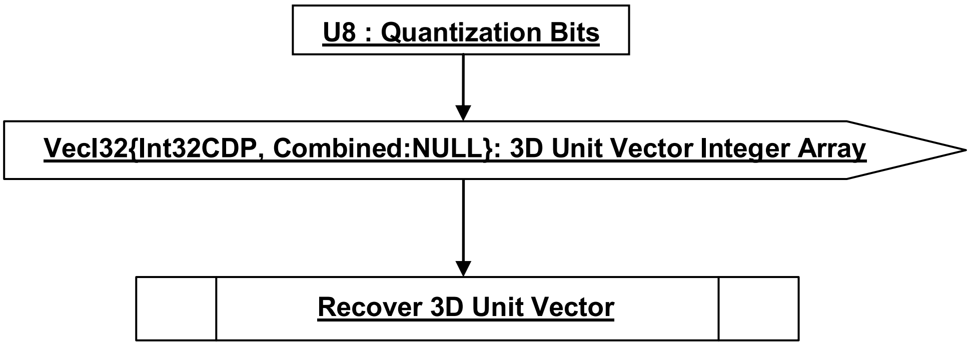

U8 : Quantization Bits
VecI32{Int32CDP, Combined:NULL}: 3D Unit Vector Integer Array
Recover 3D Unit Vector

Figure 208 — Error! Reference source not found. **data collection**

U8 : Quantization Bits

The number of bits used for the Deering Normal CODEC if quantization is enabled. A value of 0 denotes that quantization is disabled.

VecI32{Int32CDP, Combined:NULL}: 3D Unit Vector Integer Array

3D Unit Vector Integer Array is a vector of integers that stores the encoded 3D unit vector from all analytic entities in the ULP, encoded using Combined Predictor Type. 3D Unit Vector Integer Array is compressed and encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet.

Recover 3D Unit Vector

The logic diagram to recover 3D unit vector information for all the analytic entities in the ULP from the 3D Unit Vector Integer Array is shown below.

The recovery of a unit vector from an element in the 3D Unit Vector Integer Array is done as part of Deering Normal CODEC.

As described in Annex E, the representation of an analytic surface of types plane, cylinder, cone, sphere, or torus, includes two 3D unit vectors. One is called —xais" and the other is called -x_axis". These two unit vectors of each analytic surface are recovered for each analytic surface. In addition, the —onrmal" vector to the plane containing a 3D circle is also recovered.

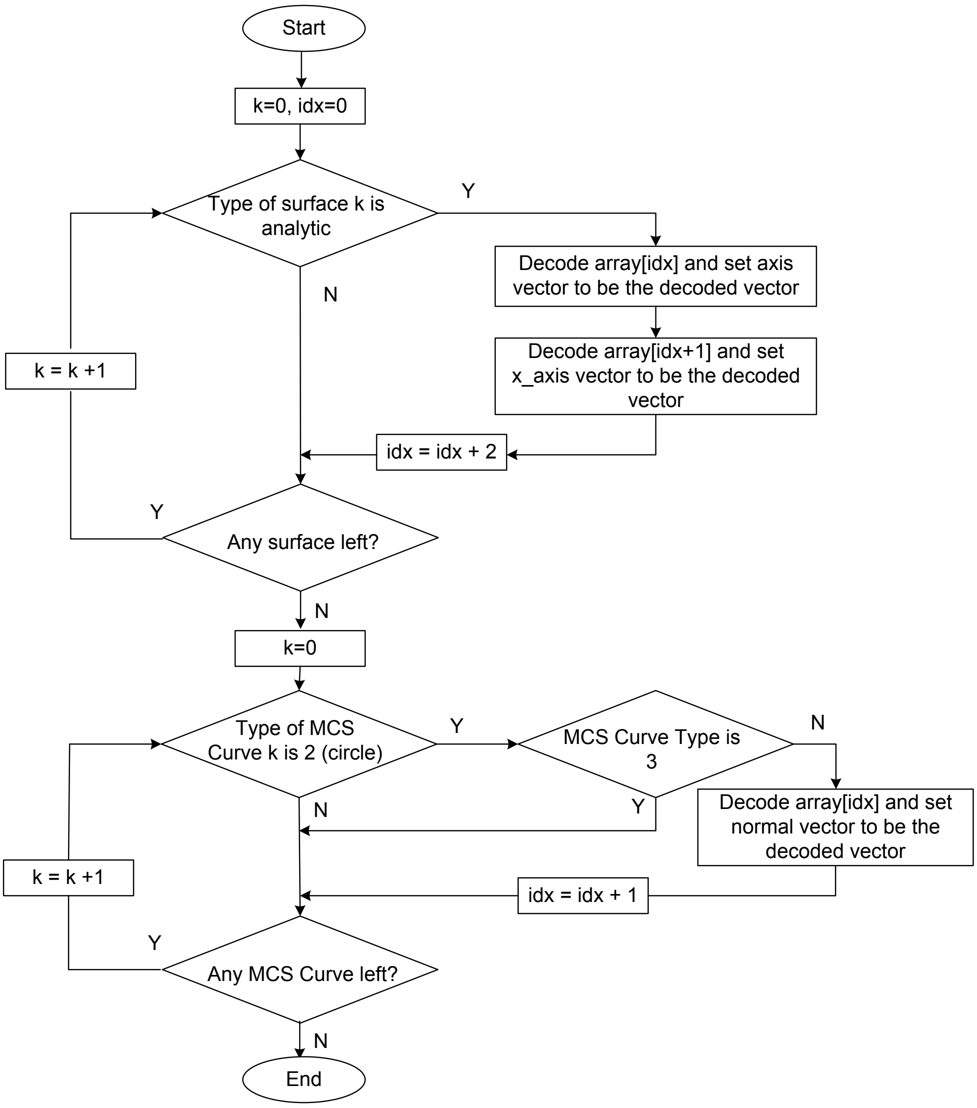

Start
k=0, idx=0
Type of surface k is analytic
Y
N
Decode array[idx] and set axis vector to be the decoded vector
Decode array[idx+1] and set x_axis vector to be the decoded vector
idx = idx + 2
k = k +1
Y
Any surface left?
N
k=0
Type of MCS Curve k is 2 (circle)
Y
MCS Curve Type is 3
N
Y
N
Decode array[idx] and set normal vector to be the decoded vector
idx = idx + 1
k = k +1
Y
Any MCS Curve left?
N
End

Figure 209 — Recover Dimension

2D Unit Vector Table

2D Unit Vector Table stores an array of unit vectors in 2D that form part of PCS analytic circle representation in ULP. If the ULP does not contain any analytic circle in the PCS, then the table is empty and bit 0x0010 in Geometric Tabe Flag is set to be 0. The analytic curve representation of ULP follows XTconvention as detailed in Annex E.

Similar to the coding of 13.1.4 Compressed Vertex Normal Array, each 2D unit vector is treated as a 3D unit vector with z component set to be 0.0, and encoded as a single 32 bit integer using 13.2.4 Deering Normal CODEC. In addition, the Quantization Bits information of Deering Normal CODEC used to encode 2D Unit Vector Table is always the same as the one used for **Error! Reference source not found.**.

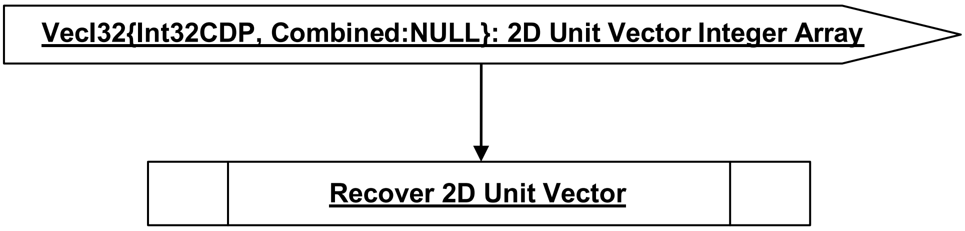

VecI32{Int32CDP, Combined:NULL}: 2D Unit Vector Integer Array
Recover 2D Unit Vector

Figure 210 — 2D Unit Vector Table data collection

VecI32{Int32CDP, Combined:NULL}: 2D Unit Vector Integer Array

2D Unit Vector Integer Array is a vector of integers that stores the encoded 2D unit vector from all analytic entities in the ULP.

Recover 2D Unit Vector

The logic diagram to recover 2D unit vector information for all the analytic entities in the ULP from the 2D Unit Vector Integer Array is shown below.

The recovery of a unit vector from an element in the 2D Unit Vector Integer Array is done as part of Deering Normal CODEC. The Quantization Bits read from **Error! Reference source not found.** should be used for Deering Normal CODEC to decode the vector information from each element in 2D Unit Vector Integer Array.

The ~~x~~_axis” vector to the circle in the PCS, as described in Annex E, is recovered.

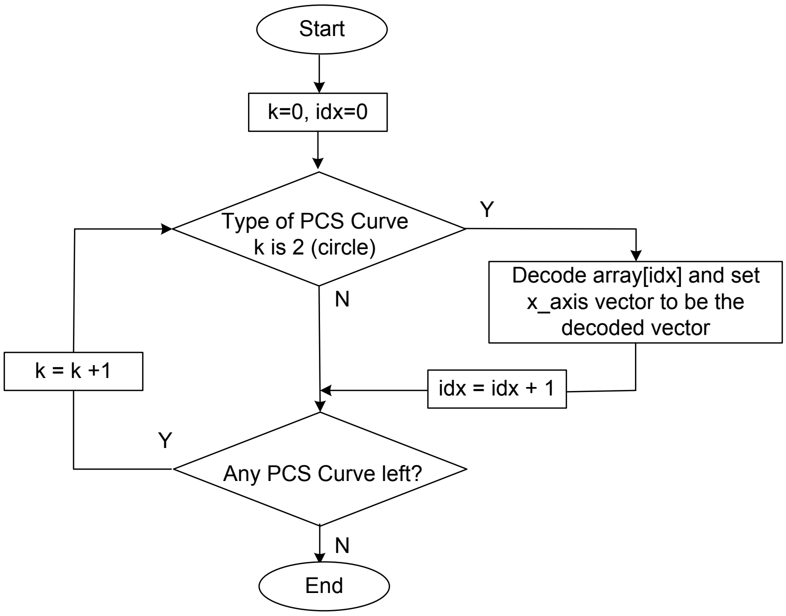

Start
k=0, idx=0
Type of PCS Curve k is 2 (circle)
Y
N
Decode array[idx] and set x_axis vector to be the decoded vector
idx = idx + 1
k = k +1
Y
Any PCS Curve left?
N
End

Figure 211 — Recover 2D Unit Vector

3D MCS Point Table

3D MCS Point Table stores the quantization representation of an array of 3D MCS points in ULP. If the ULP does not contain 3D MCS points, then the table is empty and bit 0x0020 in Geometric Tabe Flag is set to be 0.

Each point coordinate is first encoded into an integer with uniform quantizer (see 13.1.12 Uniform Quantizer Data) and then all the integers from each coordinate are grouped into an integer array, which is then encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type.

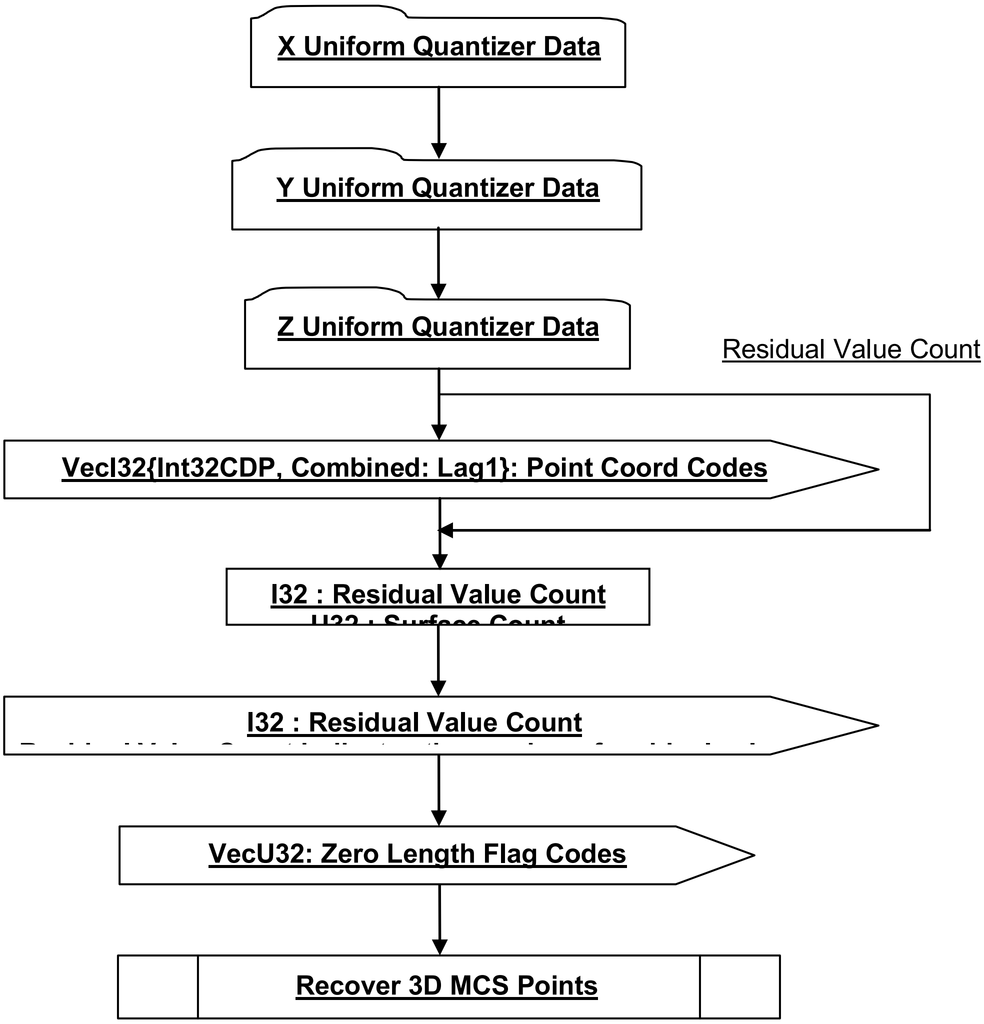

X Uniform Quantizer Data
Y Uniform Quantizer Data
Z Uniform Quantizer Data
Residual Value Count
VecI32{Int32CDP, Combined: Lag1}: Point Coord Codes
I32 : Residual Value Count
I32 : Residual Value Count
VecU32: Zero Length Flag Codes
Recover 3D MCS Points

Figure 212 — 3D MCS Point Table data collection

VecI32{Int32CDP, Combined: Lag1}: Point Coord Codes

Point Coord Codes is a vector of quantizer -codes" for all the coordinate components of an array of three dimensional points, arranged in the order of X-component of point 1, Y-component of point1, Z-component of point1, X-component of point2, etc.. Point Coord Codes uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet to compress and encode data.

I32 : Residual Value Count

Residual Value Count indicates the number of residual values.

VecI32{Int32CDP, Combined: NULL}: Residual Coord Codes

Residual Coord Codes is a vector of quantizer -codes" for all the coordinate components of an array of 3-dimensional residual points, arranged in the order of X-component of residual point 1, Y-component of residual point1, Z-component of residual point1, X-component of residual point2, etc.. The residual points are computed based on Paralleogram rule for the control points of NURBS surfaces. Denote $P_{i,i}, P_{i+1,i}, P_{i,i+1}, P_{i+1,i+1}$ as four control points of a NURBS surface, and $Q_{i,i}, Q_{i+1,i}, Q_{i,i+1}$ as quantized points of $P_{i,i}, P_{i+1,i}, P_{i,i+1}$ respectively, the residual point of $P_{i+1,i+1}$ is defined as $R_{i+1,i+1} = P_{i+1,i+1} + Q_{i,i} - Q_{i,i+1} - Q_{i+1,i}$. Residual Coord Codes uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet to compress and encode data.

VecU32: Zero Length Flag Codes

Zero Length Flag Codes is a vector of 32 bit unsigned integers, with each bit indicating whether or not a MCS curve with line geometry has zero length. The bits are arranged the same sequence as the MCS curve array. After decoding, the first N bits, where N is the total number of MCS line curves in the ULP, can be assigned to an integer array of length N with its element assigned with value 0 or 1. Each element in the decoded integer array describes whether or not the corresponding MCS line curve has zero length.

Recover 3D MCS Points

The logic diagram to recover 3D MCS points information in the ULP from the three decoded arrays, point coordinate array Pcc (with index ip) decoded from Zero Length Flag Codes, residual coordinate array Rcc (with index ir) decoded from Residual Coord Codes, and zero length flag array Zlf (with index iz) decoded from Zero Length Flag Codes, is shown below. Note that the point coordinates are decoded from the integer elements with Uniform Quantizer (see 13.1.12 Uniform Quantizer Data).

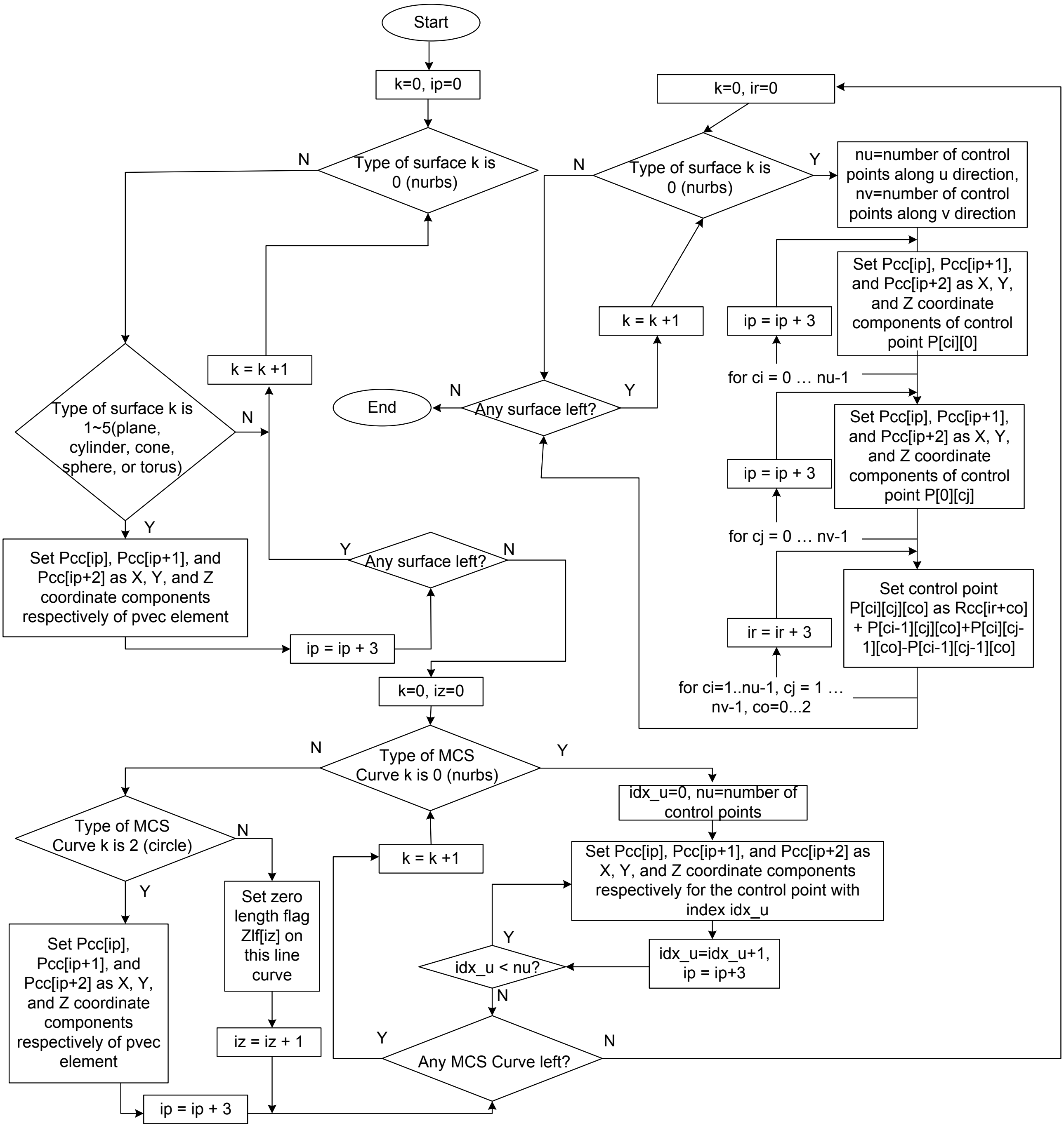

Start
k=0, ip=0
Type of surface k is 0 (nurbs)
N
k=0, ir=0
N
Type of surface k is 0 (nurbs)
Y
nu=number of control points along u direction, nv=number of control points along v direction
Set Pcc[ip], Pcc[ip+1], and Pcc[ip+2] as X, Y, and Z coordinate components of control point P[ci][0]
k = k +1
ip = ip + 3
for ci = 0 … nu-1
k = k +1
Type of surface k is 1~5(plane, cylinder, cone, sphere, or torus)
N
End
N
Any surface left?
Y
Set Pcc[ip], Pcc[ip+1], and Pcc[ip+2] as X, Y, and Z coordinate components of control point P[0][cj]
ip = ip + 3
for cj = 0 … nv-1
Y
Set Pcc[ip], Pcc[ip+1], and Pcc[ip+2] as X, Y, and Z coordinate components respectively of pvec element
Y
Any surface left?
N
Set control point P[ci][cj][co] as Rcc[ir+co] + P[ci-1][cj][co]+P[ci][cj-1][co]-P[ci-1][cj-1][co]
ir = ir + 3
ip = ip + 3
k=0, iz=0
for ci=1..nu-1, cj = 1 … nv-1, co=0...2
N
Type of MCS Curve k is 0 (nurbs)
Y
idx_u=0, nu=number of control points
Type of MCS Curve k is 2 (circle)
N
k = k +1
Set Pcc[ip], Pcc[ip+1], and Pcc[ip+2] as X, Y, and Z coordinate components respectively for the control point with index idx_u
Y
Set zero length flag Zlf[iz] on this line curve
Y
Set Pcc[ip], Pcc[ip+1], and Pcc[ip+2] as X, Y, and Z coordinate components respectively of pvec element
idx_u < nu?
idx_u=idx_u+1, ip = ip+3
N
iz = iz + 1
Y
Any MCS Curve left?
N
ip = ip + 3

Figure 213 — Recover 3D MCS Points

Knot Vector Table

Knot Vector Table stores the quantization representation of knot vectors in ULP. If the ULP does not contain any knot vector that needs be stored, then the table is empty and bit 0x0040 in Geometric Tabe Flag is set to be 0.

In ULP every knot vector starts with 0.0 and ends with 1.0 and is always clamped at both ends. The encoding of knot vector depends on its classified knot type. The knot values in the middle of a knot vector need be written only if the knot type is 0 (see Supported Knot Type). For all the knot values that need be written, each of them is encoded into an integer with uniform quantizer (see 13.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type.

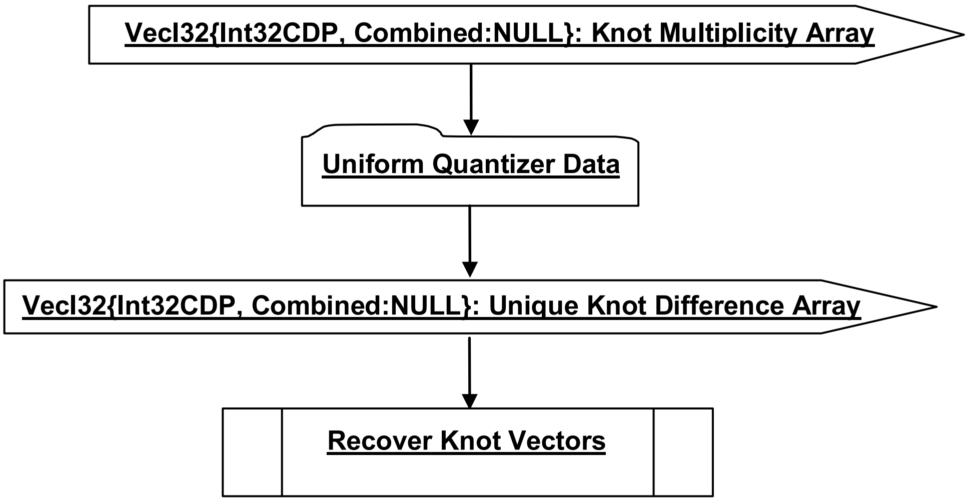

VecI32{Int32CDP, Combined:NULL}: Knot Multiplicity Array
Uniform Quantizer Data
VecI32{Int32CDP, Combined:NULL}: Unique Knot Difference Array
Recover Knot Vectors

Figure 214 — Knot Vector Table data collection

VecI32{Int32CDP, Combined:NULL}: Knot Multiplicity Array

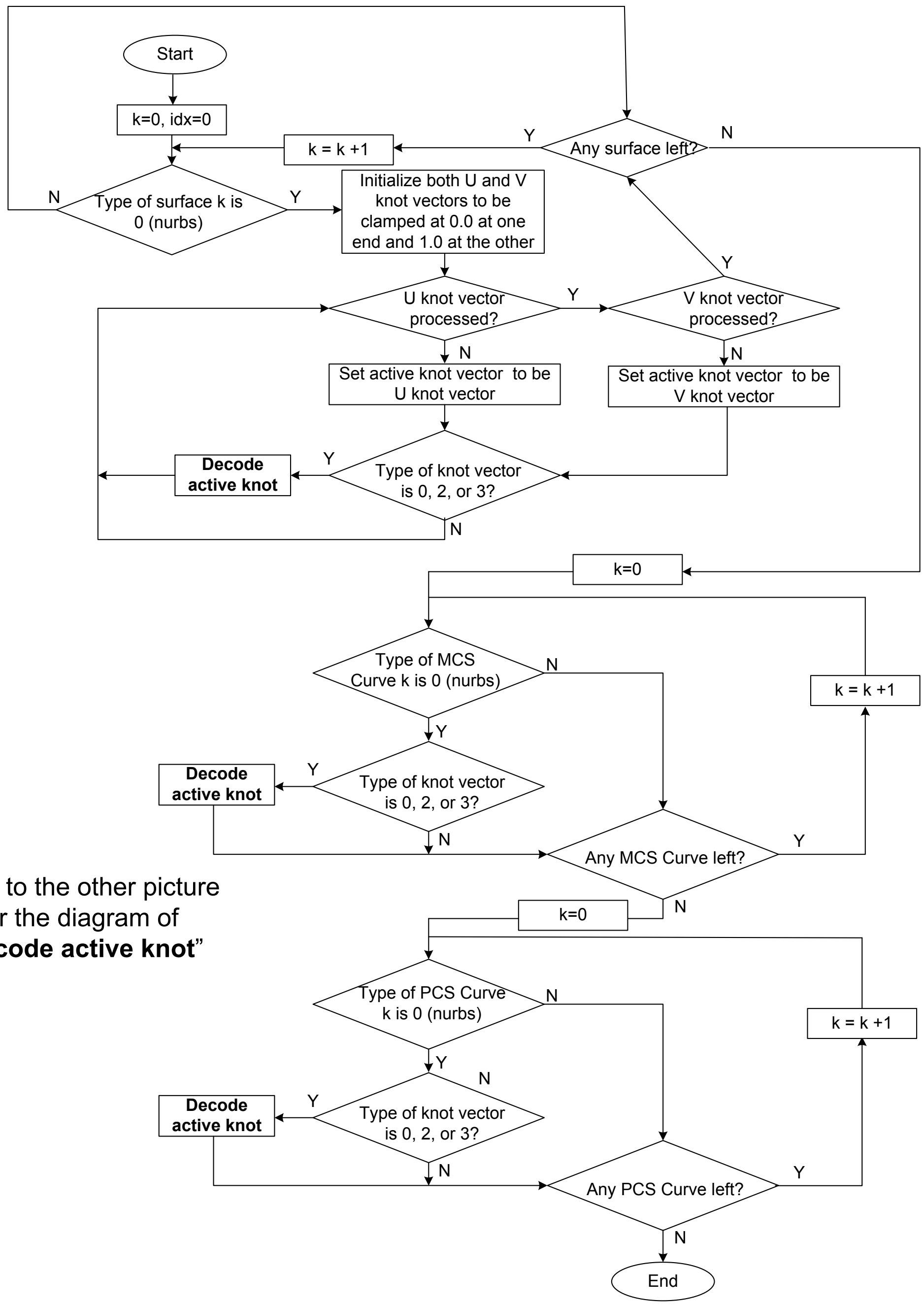
Knot Multiplicity Array is a vector of integers that describes knot multiplicity for all the knot vectors. The value of knot multiplity is set to be 0 if the knot value does not repeat itself. Knot Multiplicity Array is parallel to Knot Multiplicity Array with the same length, and uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type to compress and encode data.

VecI32{Int32CDP, Combined:NULL}: Unique Knot Difference Array

Unique Knot Difference Array is a vector that represents the unique knot values. The first element has the value of the first unique knot value. Each subsequent element k represents the value difference between unique knot value k and the quantized value of unique knot value $k-1$. Unique Knot Difference Array is first quantized (Uniform Data Quantization) and then the quantized values uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type to compress and encode data.

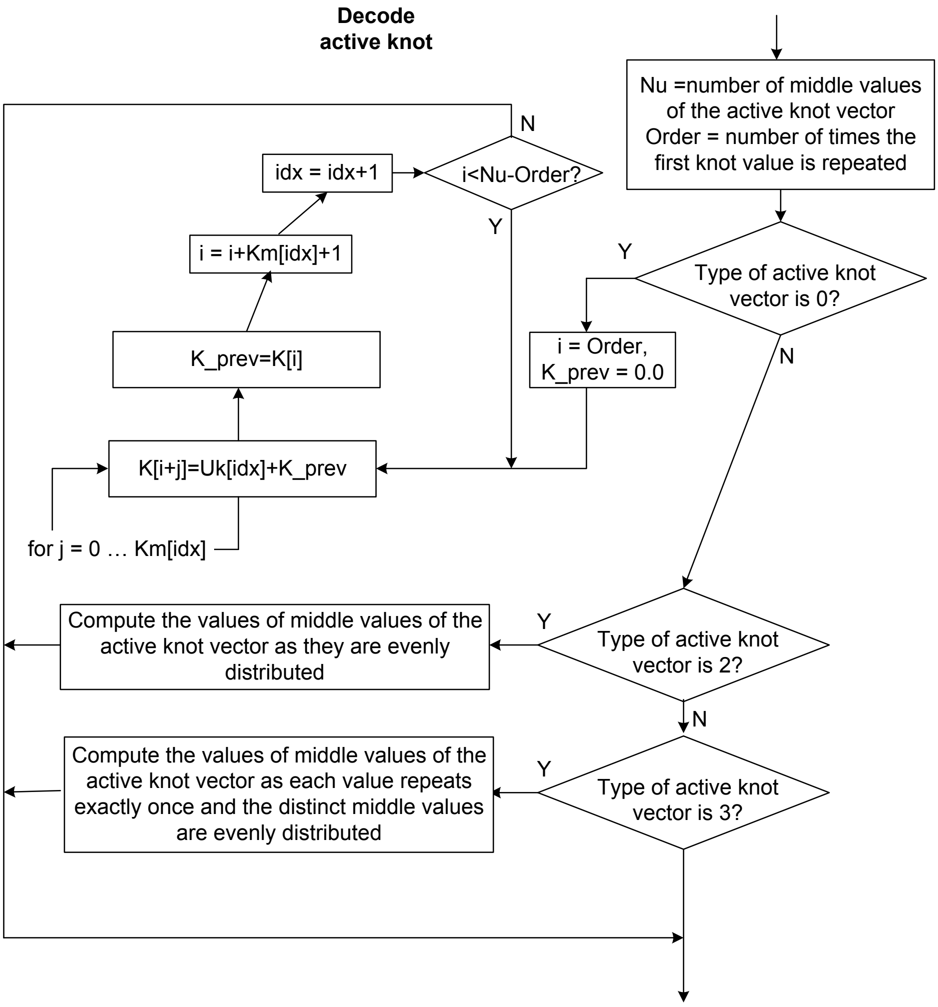
Recover Knot Vectors

The logic diagram to recover knot vector information in the ULP from the Knot Multiplicity Array (array Km) and Unique Knot Difference Array (array Uk) is shown below. Note that each integer element in the Unique Knot Difference Array is decoded with Uniform Quantizer.


Start
k=0, idx=0
k = k +1
Any surface left?
Y
N
N
Type of surface k is 0 (nurbs)
Y
Initialize both U and V knot vectors to be clamped at 0.0 at one end and 1.0 at the other
Y
U knot vector processed?
Y
V knot vector processed?
N
N
Set active knot vector to be U knot vector
Set active knot vector to be V knot vector
Decode active knot
Y
Type of knot vector is 0, 2, or 3?
N
k=0
Type of MCS Curve k is 0 (nurbs)
N
k = k +1
Y
Decode active knot
Y
Type of knot vector is 0, 2, or 3?
N
Any MCS Curve left?
Y
N
k=0
Type of PCS Curve k is 0 (nurbs)
N
k = k +1
Y
N
Decode active knot
Y
Type of knot vector is 0, 2, or 3?
N
Any PCS Curve left?
Y
N
End

Refer to the other picture for the diagram of –**Decode active knot**”

Figure 215 — Recover Knot Vectors


Decode
active knot
Nu =number of middle values of the active knot vector
Order = number of times the first knot value is repeated
Type of active knot vector is 0?
Y
N
i = Order,
K_prev = 0.0
i<Nu-Order?
N
Y
idx = idx+1
i = i+Km[idx]+1
K_prev=K[i]
K[i+j]=Uk[idx]+K_prev
for j = 0 … Km[idx]
Type of active knot vector is 2?
Y
N
Compute the values of middle values of the active knot vector as they are evenly distributed
Type of active knot vector is 3?
Y
Compute the values of middle values of the active knot vector as each value repeats exactly once and the distinct middle values are evenly distributed

Recover Knot Vectors (continued)

1D MCS Table

1D MCS Table stores the quantization representation of floating point values in MCS. If the ULP does not contain any such value, then the table is empty and bit 0x0080 in Geometric Tabe Flag is set to be 0. Each floating point value is encoded into an integer with uniform quantizer (see 13.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type.

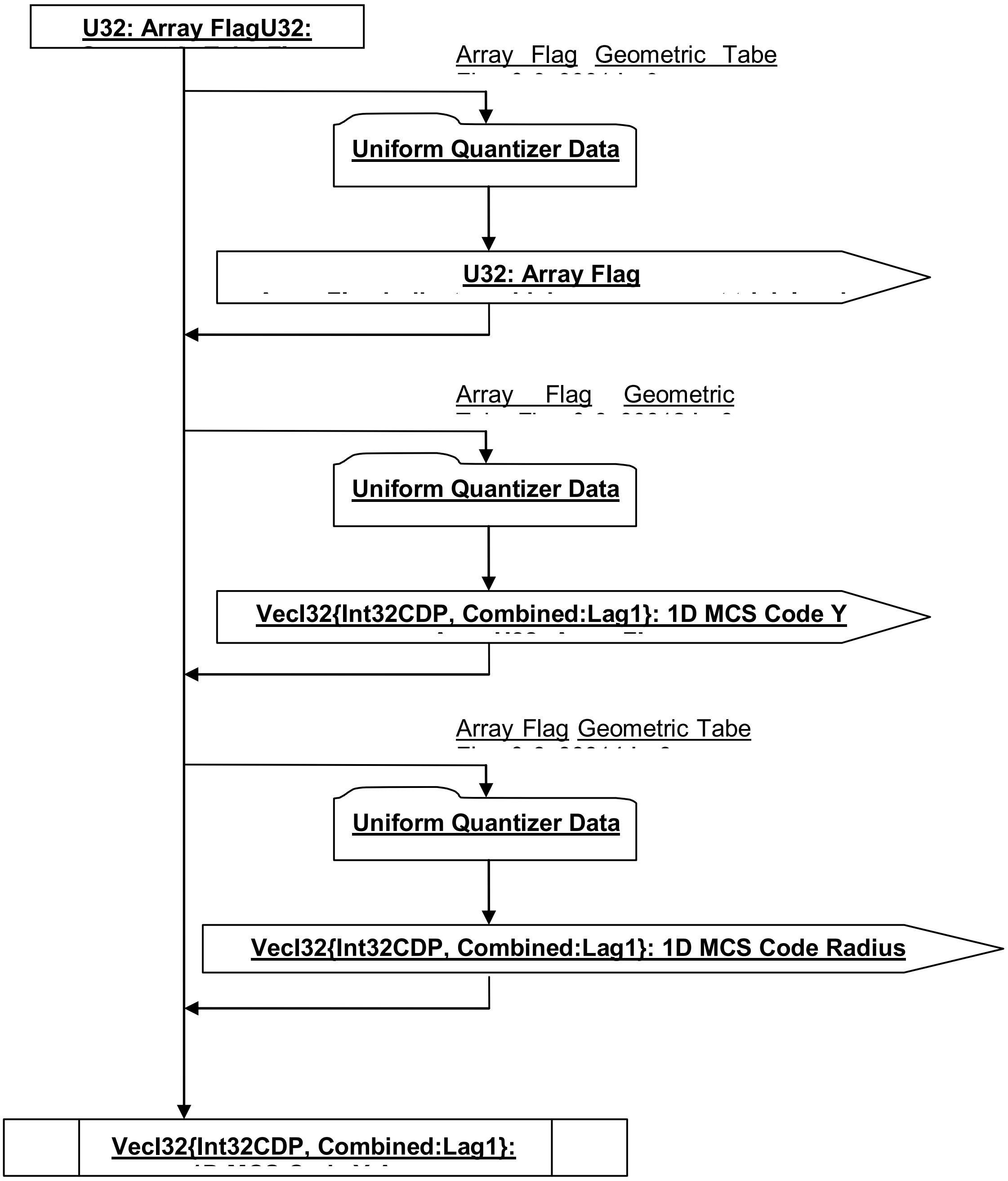

U32: Array FlagU32:
Array Flag Geometric Tabe
Uniform Quantizer Data
U32: Array Flag
Array Flag Geometric
Uniform Quantizer Data
VecI32{Int32CDP, Combined:Lag1}: 1D MCS Code Y
Array Flag Geometric Tabe
Uniform Quantizer Data
VecI32{Int32CDP, Combined:Lag1}: 1D MCS Code Radius
VecI32{Int32CDP, Combined:Lag1}:

Figure 216 — 1D MCS Table data collection

U32: Array Flag

Array Flag indicates which arrays are not trivial and therefore encoded.

VecI32{Int32CDP, Combined:Lag1}: 1D MCS Code X Array

1D MCS Code X Array is a vector of quantizer -codes” for one group of 1D floating point values in MCS that represent X coordinates. 1D MCS Code X Array uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type to compress and encode data.

VecI32{Int32CDP, Combined:Lag1}: 1D MCS Code Y Array

1D MCS Code Y Array is a vector of quantizer -codes” for one group of 1D floating point values in MCS that represent Y coordinates. 1D MCS Code Y Array uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type to compress and encode data.

VecI32{Int32CDP, Combined:Lag1}: 1D MCS Code Radius Array

1D MCS Code Radius Array is a vector of quantizer ―codes" for one group of 1D floating point values in MCS that represent radius or other MCS metric values. 1D MCS Code Radius Array uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type to compress and encode data.

Recover 1D MCS Table

The representation of each surface or curve in ULP includes information that describes the extent of the surface or curve in the parameter domain. For curves the extent information is represented by two numbers, umin and umax, while for surfaces it is represented by two additional numbers for the other parametric direction, vmin and vmax. For surfaces or curves of NURBS type such extent information is implied by the knot vector information. For surfaces or curves of other types the extent information needs be read from 1D MCS Table if the parameter value represents value in MCS, or Radian Table if the parameter value represents angle information. The detailed information about how the parameter domain information of different entities should be read is listed in Table 92.

Table 92 — Parameter Domain

Entity Type	umin	umax	vmin	vmax
NURBS Surface	n/a (from knot)	n/a (from knot))	n/a (from knot)	n/a (from knot)
Plane	n/a (always 0)	1D MCS Table	n/a (always 0)	1D MCS Table
Cylinder	n/a (always 0)	Radian Table	n/a (always 0)	1D MCS Table
Cone	n/a (always 0)	Radian Table	n/a (always 0)	1D MCS Table
Sphere	n/a (always 0)	Radian Table	Radian Table	Radian Table
Torus	n/a (always 0)	Radian Table	Radian Table	Radian Table
XYZ NURBS Curve	n/a (from knot)	n/a (from knot)	n/a	n/a
XYZ Line	n/a (always 0)	n/a (from vertex geometry)	n/a	n/a
XYZ Circle	n/a (always 0)	Radian Table	n/a	n/a
UV NURBS Curve	n/a (from knot)	n/a (from knot)	n/a	n/a
UV Line	n/a (always 0)	n/a (from next uv curve)	n/a	n/a
UV Circle	Radian Table	Radian Table	n/a	n/a

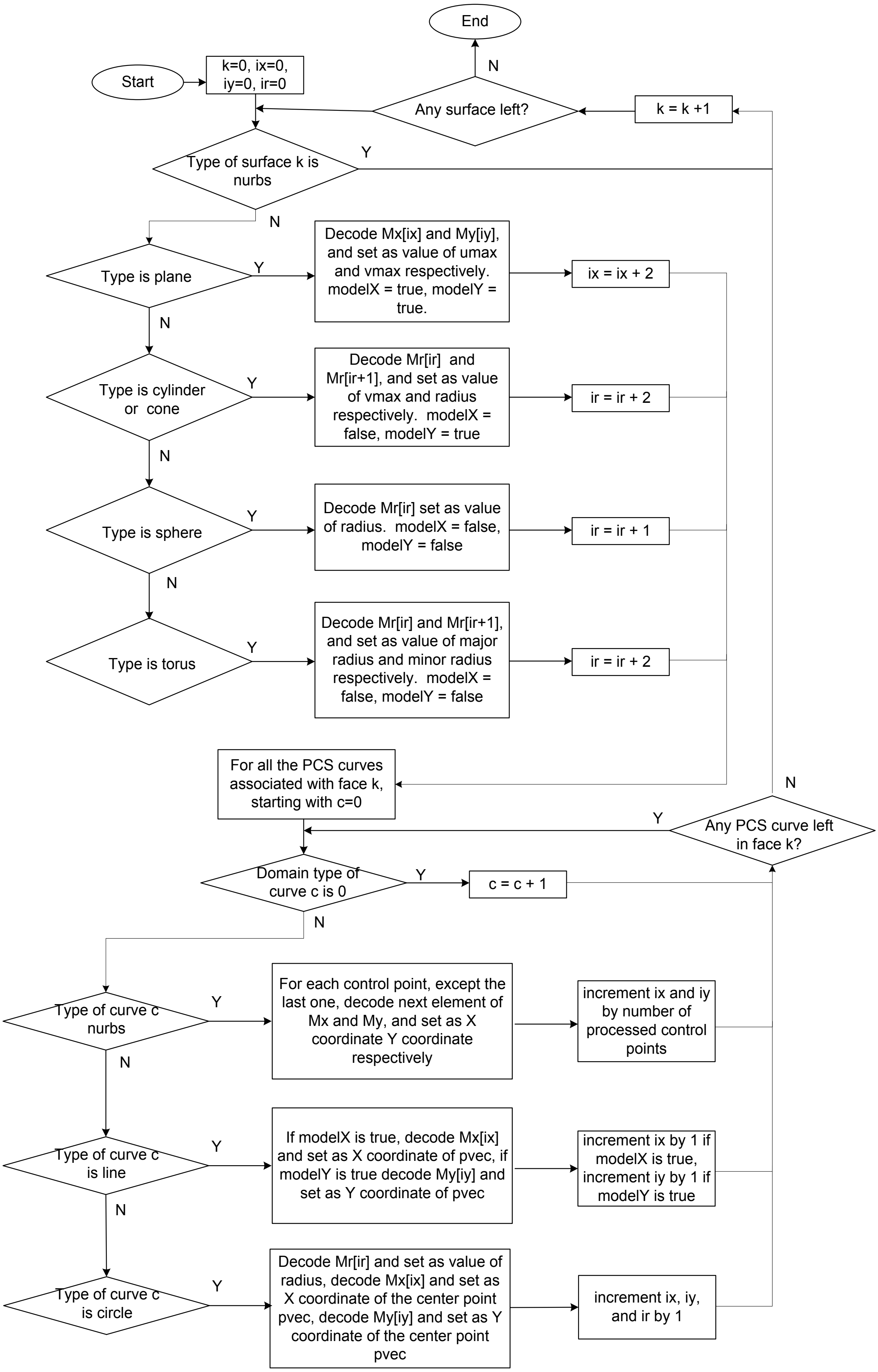

End
N
Start
k=0, ix=0, iy=0, ir=0
Any surface left?
k = k +1
Type of surface k is nurbs
Y
N
Type is plane
Y
Decode Mx[ix] and My[iy], and set as value of umax and vmax respectively. modelX = true, modelY = true.
ix = ix + 2
N
Type is cylinder or cone
Y
Decode Mr[ir] and Mr[ir+1], and set as value of vmax and radius respectively. modelX = false, modelY = true
ir = ir + 2
N
Type is sphere
Y
Decode Mr[ir] set as value of radius. modelX = false, modelY = false
ir = ir + 1
N
Type is torus
Y
Decode Mr[ir] and Mr[ir+1], and set as value of major radius and minor radius respectively. modelX = false, modelY = false
ir = ir + 2
For all the PCS curves associated with face k, starting with c=0
N
Y
Any PCS curve left in face k?
Domain type of curve c is 0
Y
c = c + 1
N
Type of curve c nurbs
Y
For each control point, except the last one, decode next element of Mx and My, and set as X coordinate Y coordinate respectively
increment ix and iy by number of processed control points
N
Type of curve c is line
Y
If modelX is true, decode Mx[ix] and set as X coordinate of pvec, if modelY is true decode My[iy] and set as Y coordinate of pvec
increment ix by 1 if modelX is true, increment iy by 1 if modelY is true
N
Type of curve c is circle
Y
Decode Mr[ir] and set as value of radius, decode Mx[ix] and set as X coordinate of the center point pvec, decode My[iy] and set as Y coordinate of the center point pvec
increment ix, iy, and ir by 1

Figure 217 — Recover 1D MCS Table

The logic diagram to recover 1D MCS table information in the ULP from the 1D MCS Code X Array is shown in Figure 217 — Recover 1D MCS Table. 1D MCS Code X Array is denoted as Mx (with index ix), 1D MCS Code Y Array is denoted as My (with index iy), and 1D MCS Code Radius Array is denoted as Mr (with index ir). Note that each integer element in the arrays is decoded with Uniform Quantizer.

PCS Value Table

PCS Value Table stores the quantization representation of floating point values in PCS. If the ULP does not contain any such value, then the table is empty and bit 0x0100 in Geometric Tabe Flag is set to be 0. Each floating point value is encoded into an integer with uniform quantizer (see 13.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type.

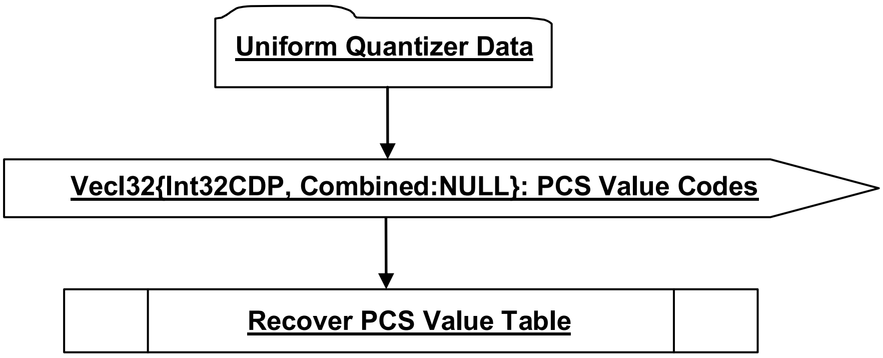


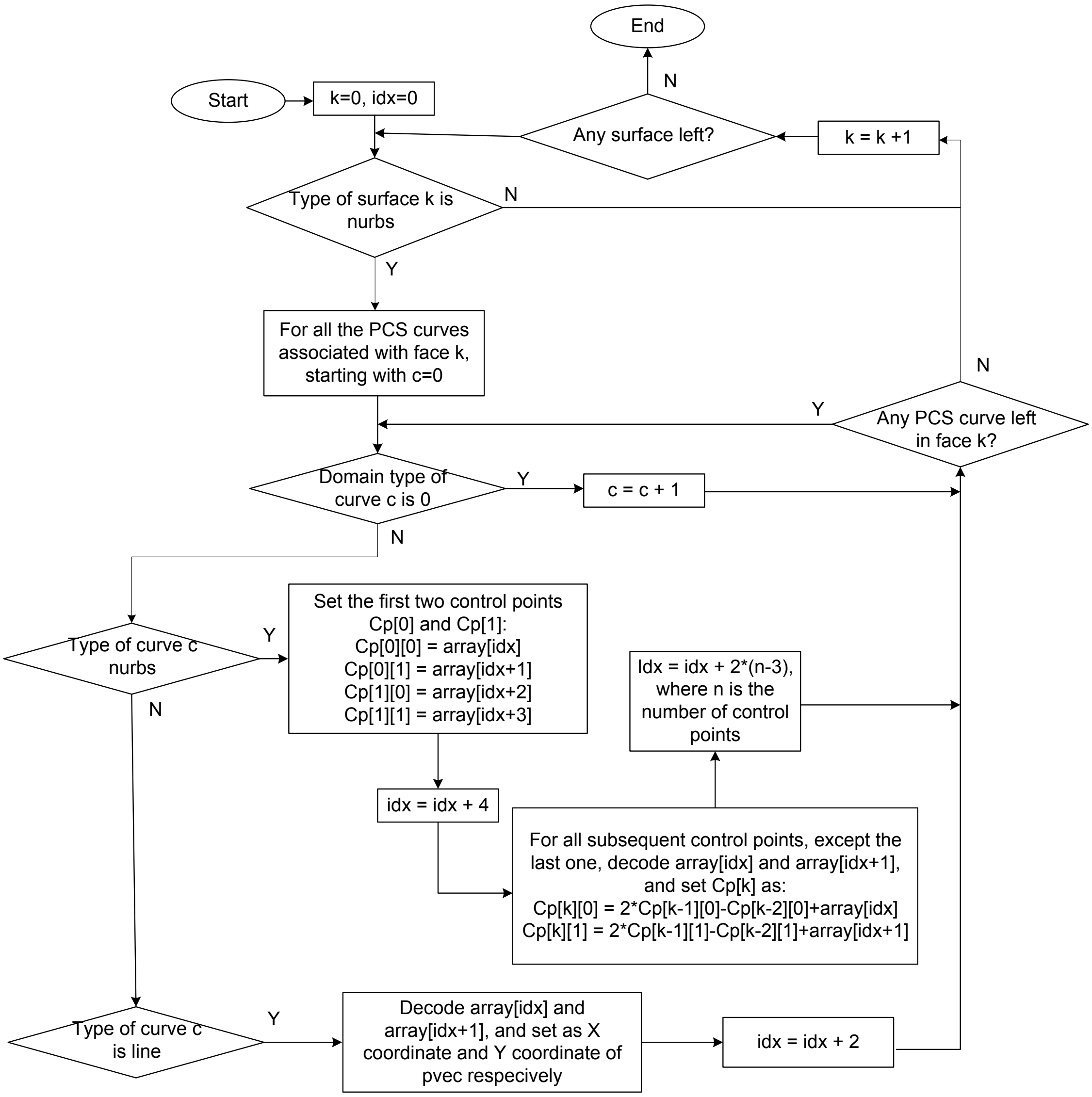
Figure 218 — PCS Value Table data collection

VecI32{Int32CDP, Combined:NULL}: PCS Value Codes

PCS Value Codes is a vector of quantizer ~~"c~~codes" for all the floating point values in PCS . PCS Value Codes uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type to compress and encode data.

Recover PCS Value Table

The logic diagram to recover PCS Value Table information in the ULP from the PCS Value Codes is shown in Figure 219. Note that each integer element in the PCS Value Codes array is decoded with Uniform Quantizer.



End
N
Start
k=0, idx=0
Any surface left?
k = k +1
Type of surface k is nurbs
N
Y
For all the PCS curves associated with face k, starting with c=0
N
Y
Any PCS curve left in face k?
Domain type of curve c is 0
Y
c = c + 1
N
Set the first two control points Cp[0] and Cp[1]: Cp[0][0] = array[idx] Cp[0][1] = array[idx+1] Cp[1][0] = array[idx+2] Cp[1][1] = array[idx+3]
Type of curve c nurbs
Y
N
Idx = idx + 2*(n-3), where n is the number of control points
idx = idx + 4
For all subsequent control points, except the last one, decode array[idx] and array[idx+1], and set Cp[k] as: Cp[k][0] = 2*Cp[k-1][0]-Cp[k-2][0]+array[idx] Cp[k][1] = 2*Cp[k-1][1]-Cp[k-2][1]+array[idx+1]
Type of curve c is line
Y
Decode array[idx] and array[idx+1], and set as X coordinate and Y coordinate of pvec respecively
idx = idx + 2

Figure 219 — Recover PCS Value Table

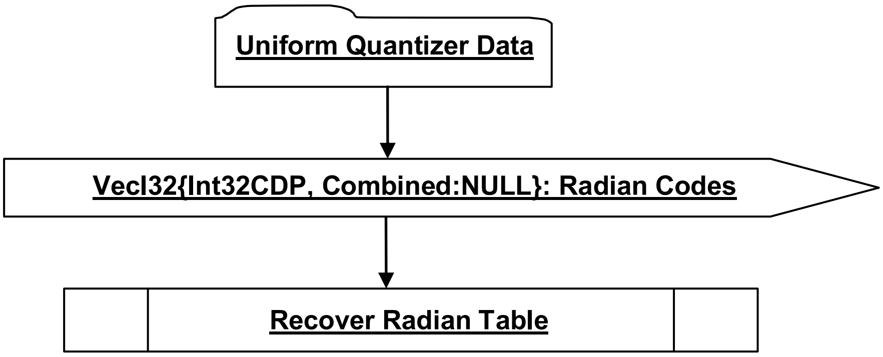


Figure 220 — Radian Table data collection

Radian Table

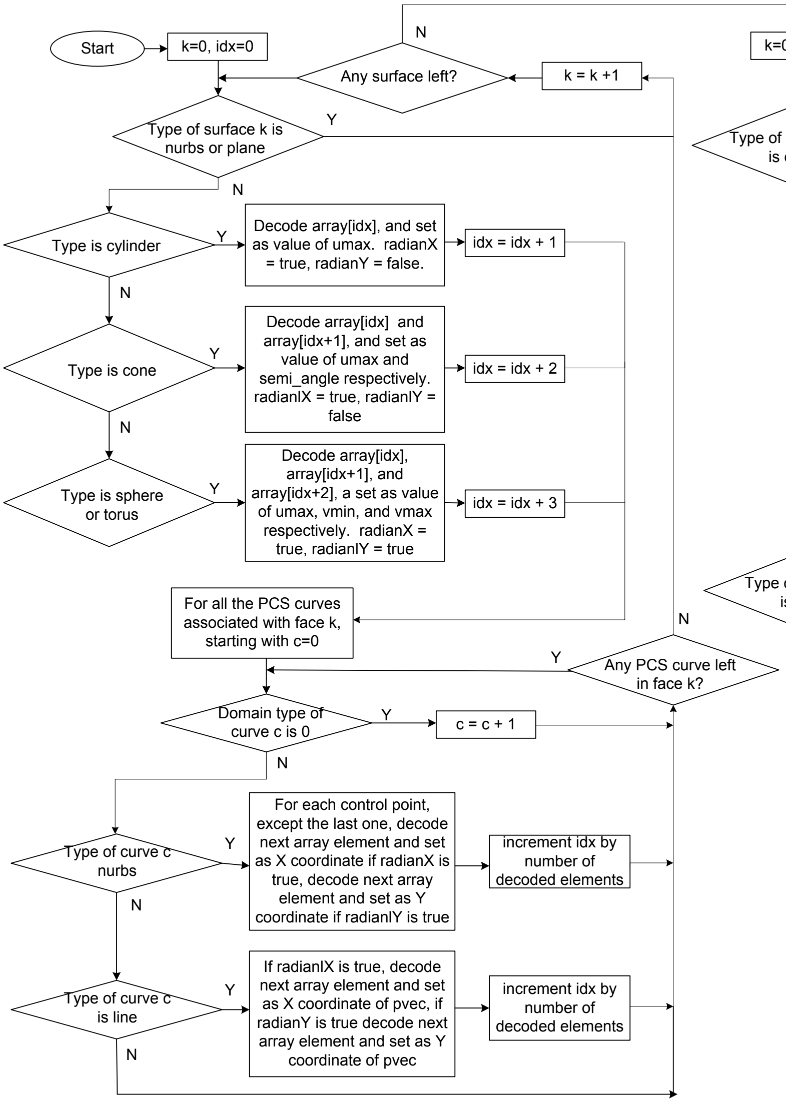
Radian Table stores the quantization representation of angular values. If the ULP does not contain any such angular value, then the table is empty and bit 0x0200 in Geometric Tabe Flag is set to be 0. Each angular value is encoded into an integer with uniform quantizer (see 13.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type.

VecI32{Int32CDP, Combined:NULL}: Radian Codes

Radian Codes is a vector of quantizer —codes” for all the angular values. Radian Codes uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type to compress and encode data.

Recover Radian Table

The logic diagram to recover Radian Table information in the ULP from the Radian Codes is shown in Figure 221. Note that each integer element in the Radian Codes array is decoded with Uniform Quantizer.



Start
k=0, idx=0
Any surface left?
N
k = k +1
Type of surface k is nurbs or plane
Y
N
Type is cylinder
Y
Decode array[idx], and set as value of umax. radianX = true, radianY = false.
idx = idx + 1
N
Type is cone
Y
Decode array[idx] and array[idx+1], and set as value of umax and semi_angle respectively. radianlX = true, radianlY = false
idx = idx + 2
N
Type is sphere or torus
Y
Decode array[idx], array[idx+1], and array[idx+2], a set as value of umax, vmin, and vmax respectively. radianX = true, radianlY = true
idx = idx + 3
For all the PCS curves associated with face k, starting with c=0
Any PCS curve left in face k?
Y
N
Domain type of curve c is 0
Y
c = c + 1
N
Type of curve c nurbs
Y
For each control point, except the last one, decode next array element and set as X coordinate if radianX is true, decode next array element and set as Y coordinate if radianlY is true
increment idx by number of decoded elements
N
Type of curve c is line
Y
If radianlX is true, decode next array element and set as X coordinate of pvec, if radianY is true decode next array element and set as Y coordinate of pvec
increment idx by number of decoded elements
N
k=0, idx=0
Type of xyzcurve k is circle
Y
Decode array[idx], and set as value of umax.
idx = idx + 1
N
k = k +1
Any xyzcurve left?
Y
N
k=0, idx=0
Type of uvcurve k is circle
Y
Decode array[idx] and array[idx+1], and set as value of umin and umax respectively.
idx = idx + 2
N
k = k +1
Any uvcurve left?
Y
N
End

Figure 221 — Recover Radian Table

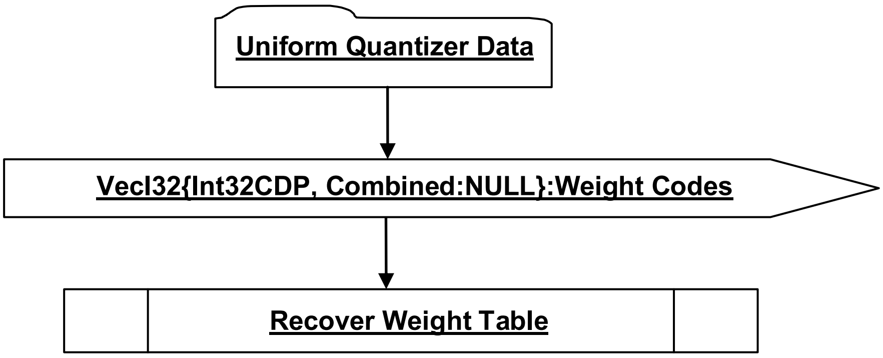


Figure 222 — Weight Table data collection

Weight Table

Weight Table stores the quantization representation of weight values. If the ULP does not contain any such weight value, then the table is empty and bit 0x0400 in Geometric Tabe Flag is set to be 0. Each weight value is encoded into an integer with uniform quantizer (see 13.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type.

VecI32{Int32CDP, Combined:NULL}:Weight Codes

Weight Codes is a vector of quantizer —codes" for all the weight values. Weight Codes uses the Int32CDP CODEC described in 13.1.1 Int32 Compressed Data Packet with Combined Predictor Type to compress and encode data.

Recover Weight Table

The logic diagram to recover Weight Table information in the ULP from the Weight Codes is shown in Figure 223. Note that each integer element in the Weight Codes array is decoded with Uniform Quantizer.

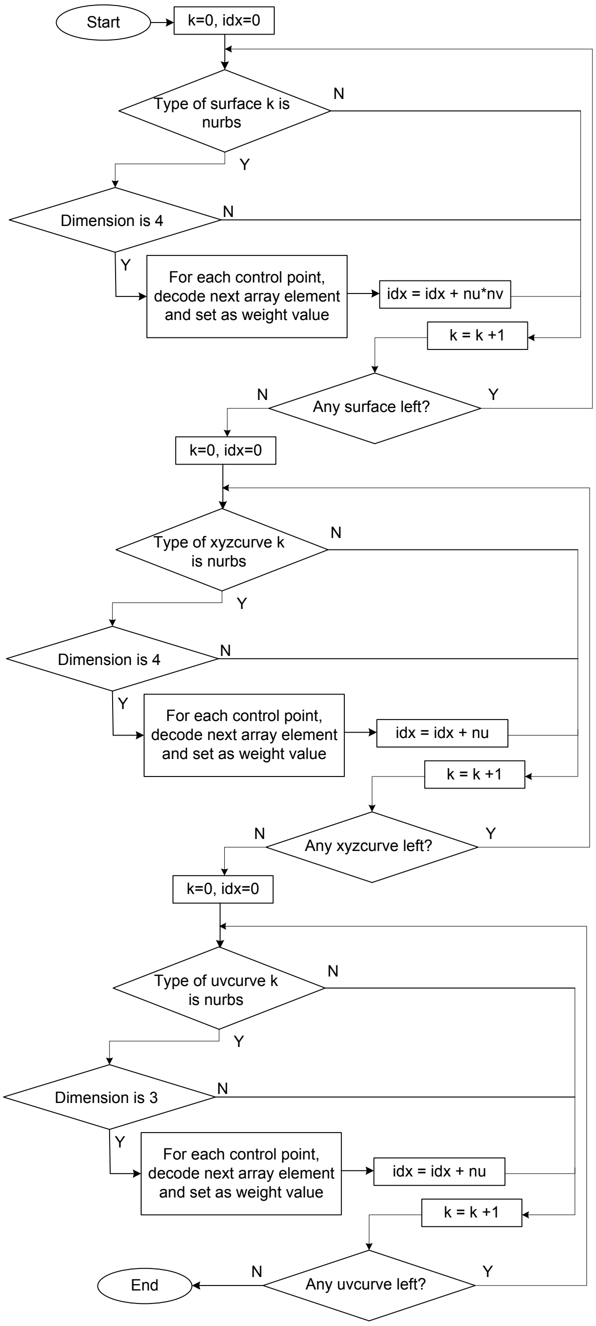

Start
k=0, idx=0
Type of surface k is nurbs
N
Y
Dimension is 4
N
Y
For each control point, decode next array element and set as weight value
idx = idx + nu*nv
k = k +1
N
Any surface left?
Y
k=0, idx=0
Type of xyzcurve k is nurbs
N
Y
Dimension is 4
N
Y
For each control point, decode next array element and set as weight value
idx = idx + nu
k = k +1
N
Any xyzcurve left?
Y
k=0, idx=0
Type of uvcurve k is nurbs
N
Y
Dimension is 3
N
Y
For each control point, decode next array element and set as weight value
idx = idx + nu
k = k +1
End
N
Any uvcurve left?
Y

Figure 223 — Recover Weight Table

12.1.3 Material Attribute Element Properties

The properties attached to material attribute are standard JT properties, and the logic diagram to read the properties attached a material attribute is shown in Figure 224.

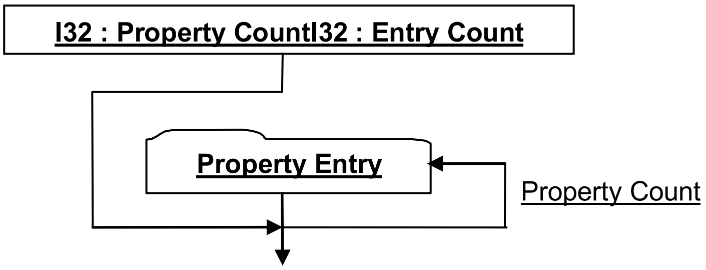


Figure 224 — Material Attribute Element Properties

I32 : Property Count

Property count is the number of properties attached.

Property Entry

Standard JT property entry, consisting of key and value pair.

12.1.4 Information Recovery

The information in ULP is classified as —essential information” that is explicitly written on disk, and —derivative information” that can be computed from the —essential information”. How —essential information” of ULP can be read from disk was covered in previous sections, and this section focuses on the logic to recover —derivative information” from —essential information”.

The derivative information consists of curve information either in the parameter or model space. For example, the PCS curves associated with an untrimmed face can be inferred from the parameter domain of the surface, or an MCS curve may be computed from vertex information and/or the combination of corresponding PCS curve geometry and surface geometry, etc.. Shown in Figure 227 is the high level diagram to recover —derivative information”. First, all the PCS line geometry are recovered from the associated surface domain information if the domain type of those PCS curves, stored in its associated coedge, are of value 1, 2, 3, 4 meaning that the PCS curve is identical to one of the parameter boundaries of the surface. The PCS curve geometry is then finalized by leveraging the knowledge that all the PCS curves in the same loop are joined in a head to tail fashion. The geometry of two end points of every MCS curve can then be computed from the corresponding PCS curve and surface geometry. Second, the MCS curve geometry is recovered depending on its type. If the MCS curve type is 0, 1, or 2, then the geometry of its two end points is used to compute the curve geometry. If the MCS curve type is 3, then its geometry is computed by projecting PCS curve onto the surface geometry. The logical steps that are displayed with dark colour indicate steps that will be elaborated in more detail later.

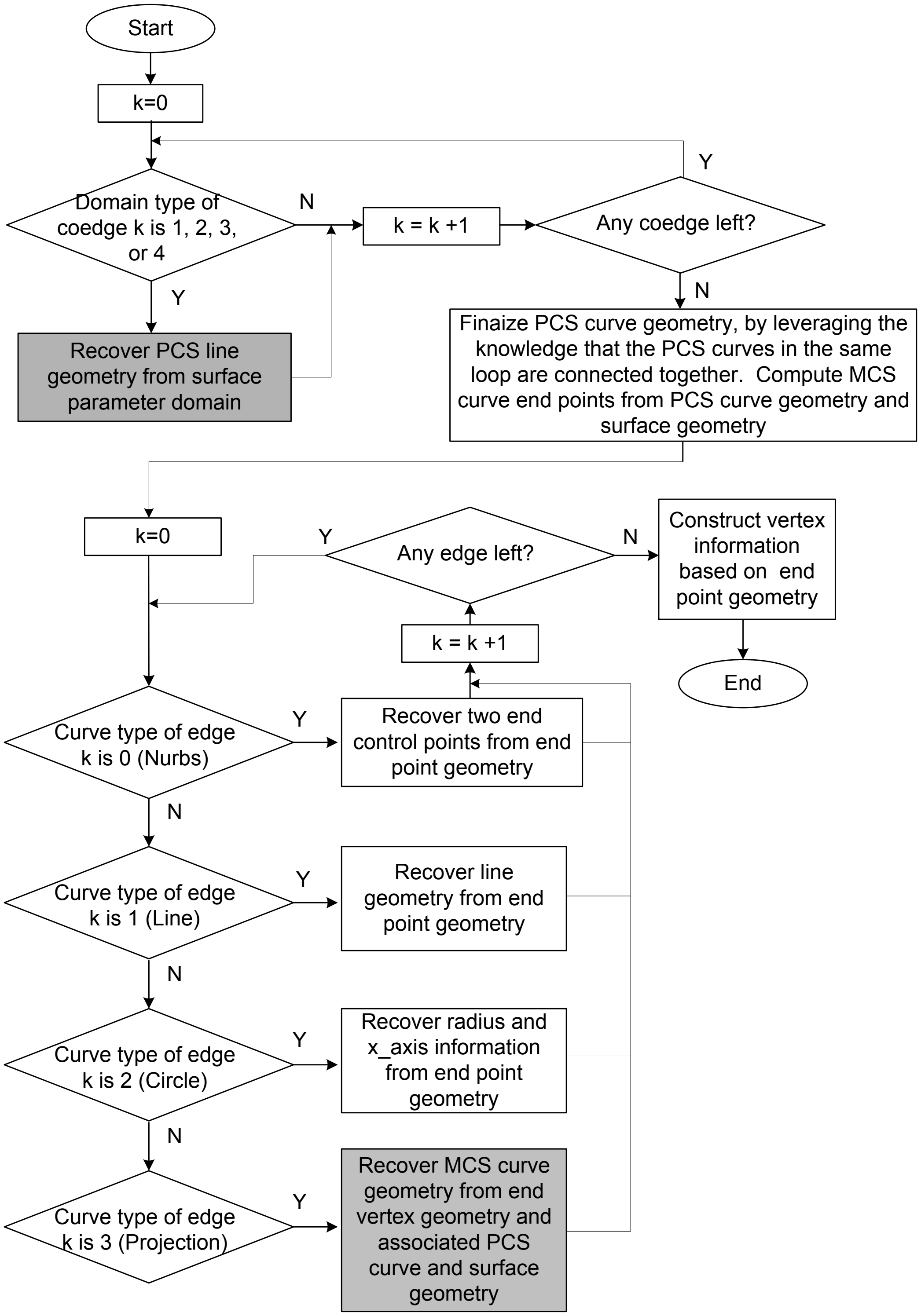

Start
k=0
Domain type of coedge k is 1, 2, 3, or 4
N
Y
k = k +1
Any coedge left?
Y
N
Recover PCS line geometry from surface parameter domain
Finaize PCS curve geometry, by leveraging the knowledge that the PCS curves in the same loop are connected together. Compute MCS curve end points from PCS curve geometry and surface geometry
k=0
Y
Any edge left?
N
Construct vertex information based on end point geometry
k = k +1
End
Curve type of edge k is 0 (Nurbs)
Y
Recover two end control points from end point geometry
N
Curve type of edge k is 1 (Line)
Y
Recover line geometry from end point geometry
N
Curve type of edge k is 2 (Circle)
Y
Recover radius and x_axis information from end point geometry
N
Curve type of edge k is 3 (Projection)
Y
Recover MCS curve geometry from end vertex geometry and associated PCS curve and surface geometry

Figure 225 — Information Recovery

PCS Curve Recovery from Surface Domain

Shown in Figure 226 is the diagram illustrating how the PCS curve geometry is recovered from surface parameter domain information.

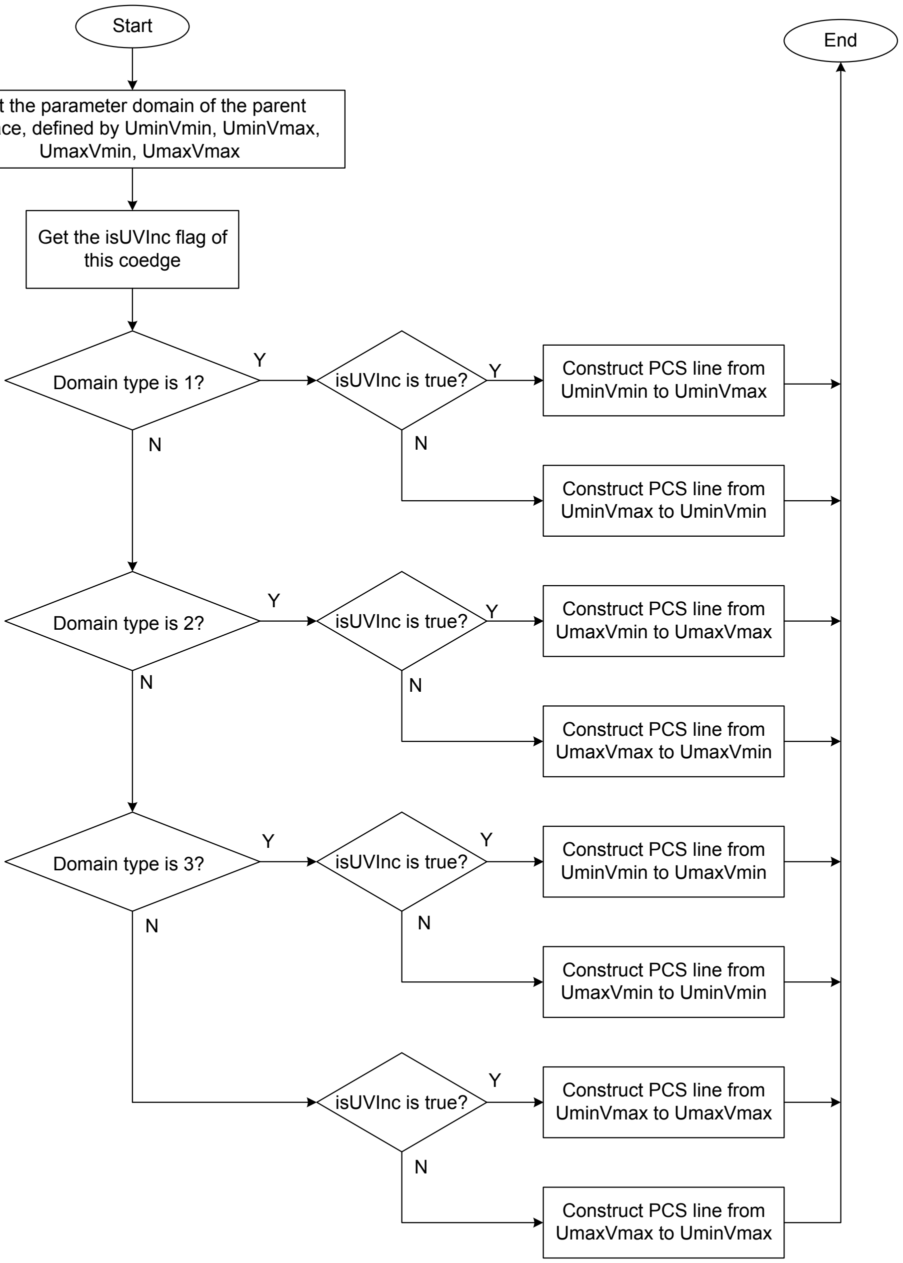

Start
End
Get the parameter domain of the parent surface, defined by UminVmin, UminVmax, UmaxVmin, UmaxVmax
Get the isUVInc flag of this coedge
Domain type is 1?
Y
N
isUVInc is true?
Y
N
Construct PCS line from UminVmin to UminVmax
Construct PCS line from UminVmax to UminVmin
Domain type is 2?
Y
N
isUVInc is true?
Y
N
Construct PCS line from UmaxVmin to UmaxVmax
Construct PCS line from UmaxVmax to UmaxVmin
Domain type is 3?
Y
N
isUVInc is true?
Y
N
Construct PCS line from UminVmin to UmaxVmin
Construct PCS line from UmaxVmin to UminVmin
isUVInc is true?
Y
N
Construct PCS line from UminVmax to UmaxVmax
Construct PCS line from UmaxVmax to UminVmax

Figure 226 — PCS Curve Recovery from Surface Domain

MCS Curve Recovery

Shown in Figure 227 is the diagram illustrating how MCS curve geometry is recovered from its end point geometry, and/or its associated PCS curve geometry and surface geometry. If the associated PCS curve is coincident with one of the parameter boundaries of the parent surface, then the MCS curve can be recovered from parent surface geometry. Otherwise, if the surface type is planar and PCS curve is of type NURBS or circle, then the MCS curve geometry can be recovered by projecting the PCS curve from parameter domain to model space onto the planar surface.

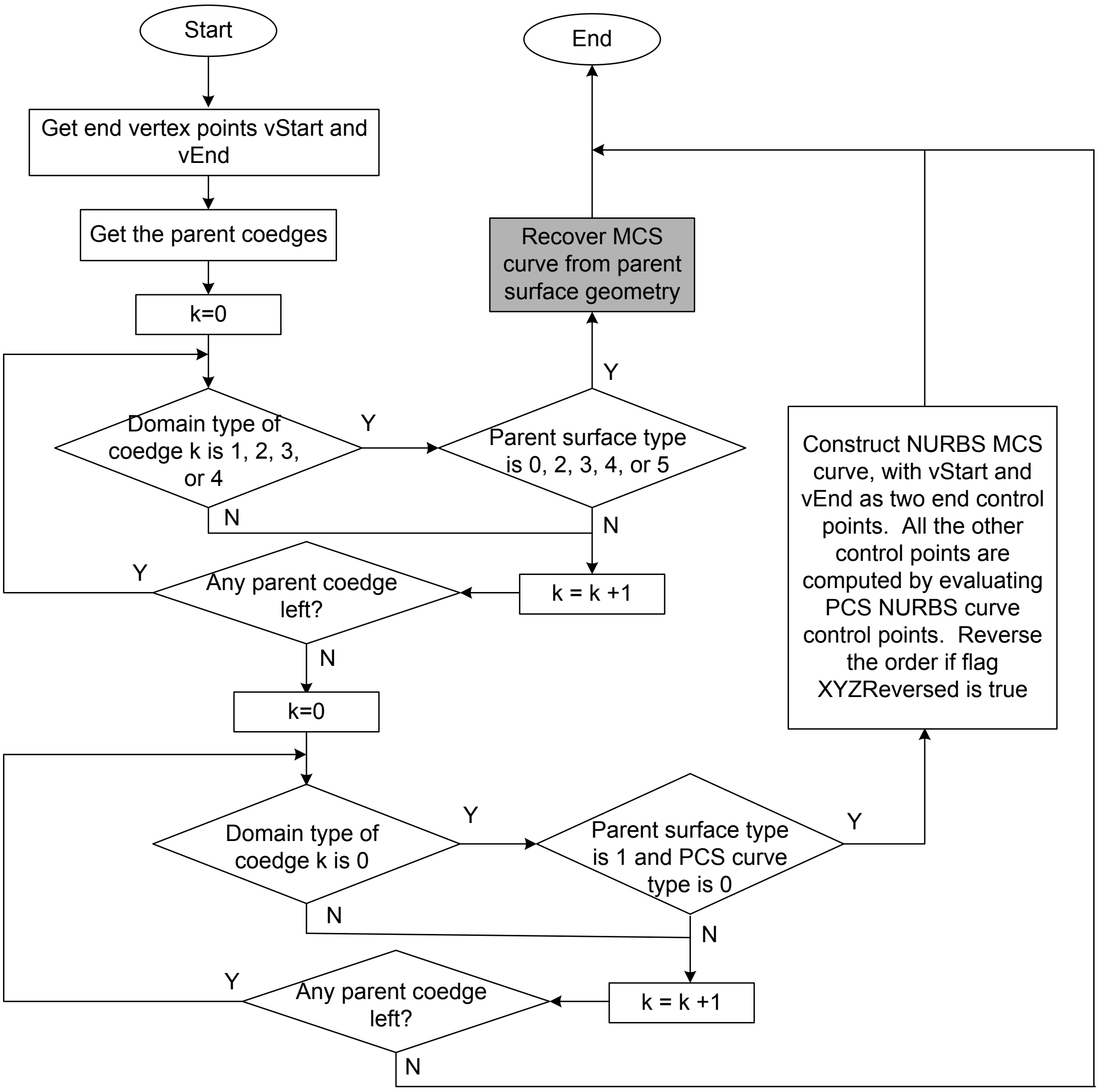

Start
Get end vertex points vStart and vEnd
Get the parent coedges
k=0
Domain type of coedge k is 1, 2, 3, or 4
Y
N
Parent surface type is 0, 2, 3, 4, or 5
Y
N
Recover MCS curve from parent surface geometry
End
Any parent coedge left?
Y
N
k = k +1
k=0
Domain type of coedge k is 0
Y
N
Parent surface type is 1 and PCS curve type is 0
Y
N
Construct NURBS MCS curve, with vStart and vEnd as two end control points. All the other control points are computed by evaluating PCS NURBS curve control points. Reverse the order if flag XYZReversed is true
Any parent coedge left?
Y
N
k = k +1

Figure 227 — MCS Curve Recovery

Shown in Figure 228 is the detailed description of how MCS curve can be recovered from surface geometry.

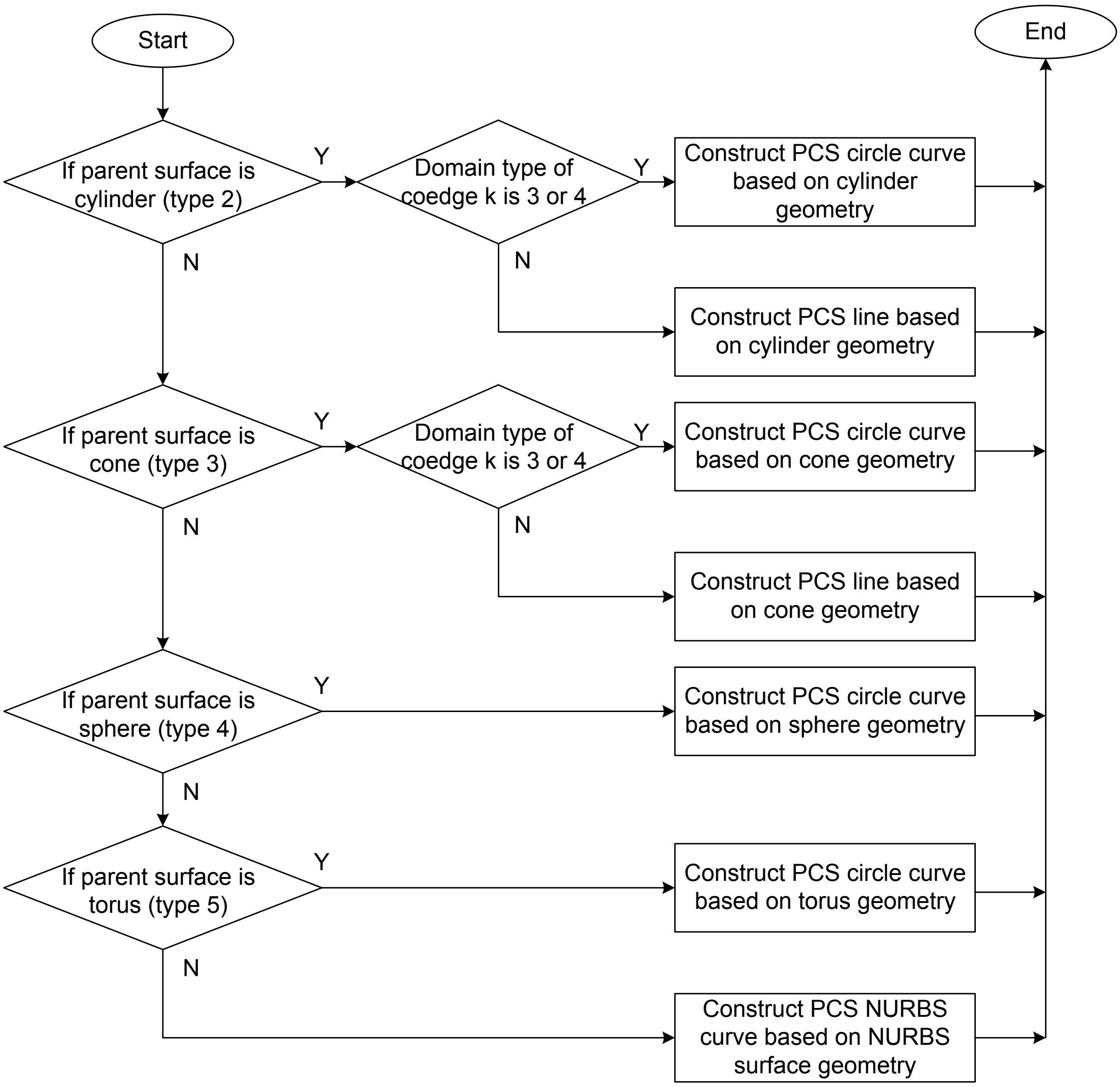

Start
End
If parent surface is cylinder (type 2)
Y
Domain type of coedge k is 3 or 4
Y
Construct PCS circle curve based on cylinder geometry
N
N
Construct PCS line based on cylinder geometry
If parent surface is cone (type 3)
Y
Domain type of coedge k is 3 or 4
Y
Construct PCS circle curve based on cone geometry
N
N
Construct PCS line based on cone geometry
If parent surface is sphere (type 4)
Y
Construct PCS circle curve based on sphere geometry
N
If parent surface is torus (type 5)
Y
Construct PCS circle curve based on torus geometry
N
Construct PCS NURBS curve based on NURBS surface geometry

Figure 228 — MCS Curve Recovery from Surface Geometry

12.2 JT LWPA Segment

JT LWPA Segment contains an Element that defines light weight precise analytic data for a particular part. More specifically LWPA contains the collection of analytic surfaces in the B-Rep definition of the part.

JT LWPA Segments are typically referenced by Part Node Elements (see 6.1.1.5 Part Node Element) using Late Loaded Property Atom Elements (see 6.2.7 Late Loaded Property Atom Element). The JT LWPA Segment type supports LZMA compression on all element data, so all elements in JT LWPA Segment use the Logical Element Header Compressed form of element header data.

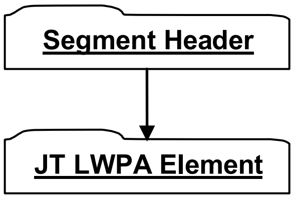


Figure 229 — JT LWPA Segment data collection

Complete description for Segment Header can be found in 5.1.3.1 Segment Header.

12.3 JT LWPA Element

Object Type ID: 0xd67f8ea8, 0xf524, 0x4879, 0x92, 0x8c, 0x4c, 0x3a, 0x56, 0x1f, 0xb9, 0x3a

JT LWPA Segment represents a particular Part's precise analytic surfaces. It can be viewed as a subset of B-Rep representation where the subset refers to the complete collection of all the surfaces that are of one of the analytic types shown in the Supported Surface Type table, i.e., plane, cylinder, cone, sphere, or torus. Unlike JT B-Rep Element or XT B-Rep Element, JT LWPA Element does not contain any B-Rep topology information, nor does it contain geometric curve or point information. LWPA is designed to represent most essential part geometry information with much lighter weight on disk and much faster to load than B-Rep. Typically LWPA is less than 2 percent of B-Rep size on disk, and takes less than 5 percent time to load into memory. The analytic representation of LWPA follows XT convention as detailed in Annex E.

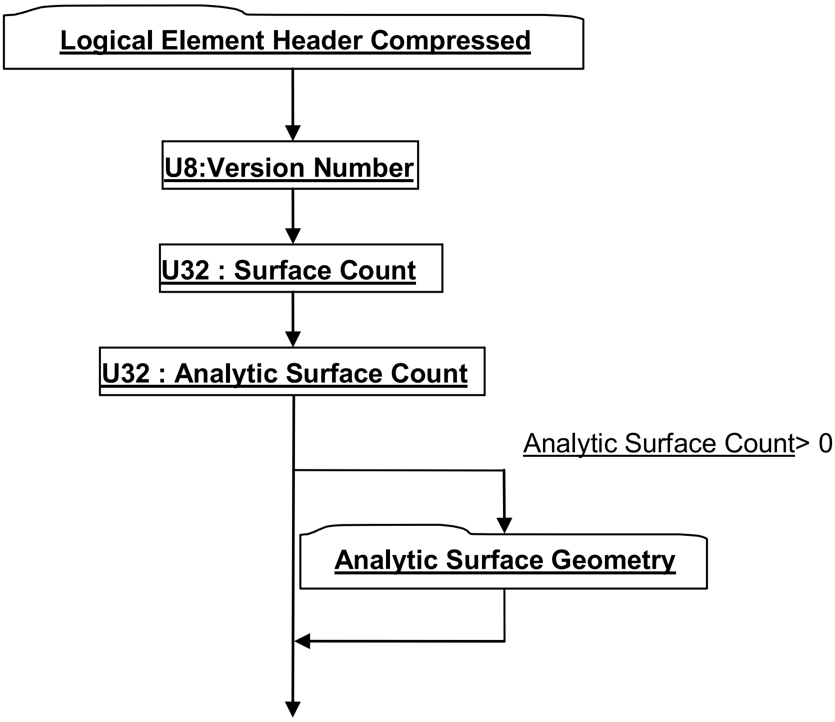

Logical Element Header Compressed
U8:Version Number
U32 : Surface Count
U32 : Analytic Surface Count
Analytic Surface Count> 0
Analytic Surface Geometry

Figure 230 — JT LWPA Element data collection

U8:Version Number

Version Number is the version identifier for this JT LWPA Element. For informaiton on local version numbers see best practice 14.5 Local version numbers

U32 : Surface Count

Surface Count indicates the number of surface entries in LWPA. The number of surface entries is equal to the number of surfaces in the B-Rep representation. The surface entry does not contain any information if the corresponding B-Rep surface is not of analytic type.

U32 : Analytic Surface Count

Analytic Surface Count indicates the number of analytic surface entries in LWPA.

12.3.1 Analytic Surface Geometry

Analytic Surface Geometry defines a collection of analytic surfaces and their mapping to the original B-Rep surfaces.

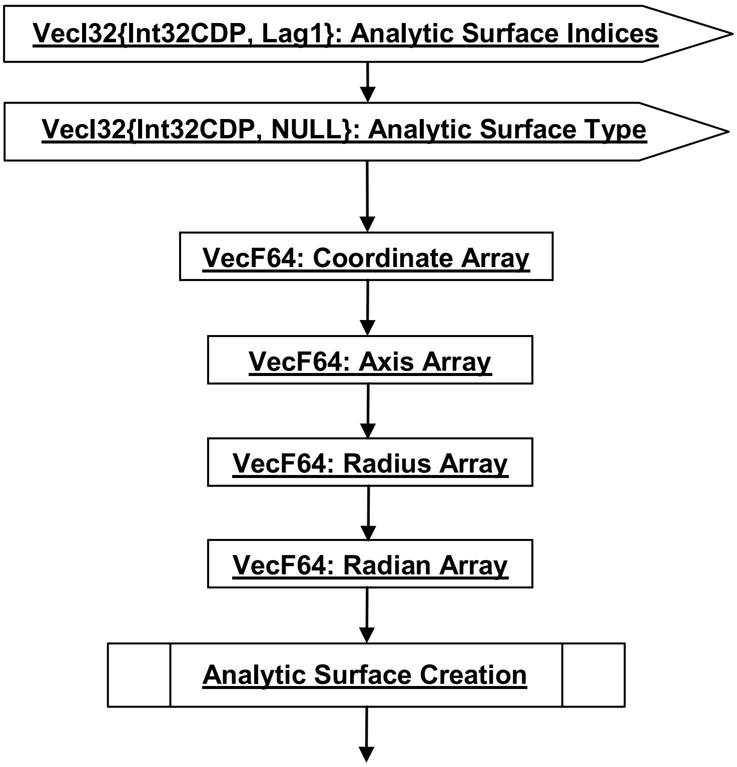

VecI32{Int32CDP, Lag1}: Analytic Surface Indices
VecI32{Int32CDP, NULL}: Analytic Surface Type
VecF64: Coordinate Array
VecF64: Axis Array
VecF64: Radius Array
VecF64: Radian Array
Analytic Surface Creation

Figure 231 — Analytic Surface Geometry data collection

VecI32{Int32CDP, Lag1}: Analytic Surface Indices

Analytic Surface Indices is an integer array with its length equal to the number of analytic surfaces in the LWPA. The value of each element in this array represents the index of this analytic surface in the original B-Rep representation.

VecI32{Int32CDP, NULL}: Analytic Surface Type

Analytic Surface Type is an integer array with its length equal to the number of analytic surfaces in the LWPA. The value of each element in this array represents the type of each analytic surface, as defined in table Supported Surface Type

VecF64: Coordinate Array

Coordinate Array contains an array of double precision floating point numbers that represent the collection of point coordinate information in the definition of the analytic surface entities. The composite type VecF64 is defined in Table 6. Each floating point number in the array is written in binary form.

VecF64: Axis Array

Axis Array contains an array of double precision floating point numbers that represent the collection of unit vector information in the definition of the analytic surface entities. The composite type VecF64 is defined in Table 6. Each floating point number in the array is written in binary form.

VecF64: Radius Array

Radius Array contains an array of double precision floating point numbers that represent the collection of radius information in the definition of the analytic surface entities. The composite type VecF64 is defined in Table 6. Each floating point number in the array is written in binary form.

VecF64: Radian Array

Radian Array contains an array of double precision floating point numbers that represent the collection of radian information in the definition of the analytic surface entities. The composite type VecF64 is defined in Table 6. Each floating point number in the array is written in binary form.

Analytic Surface Creation

Analytic surfaces in LWPA is constructed based on the information of the above arrays, as illustrated by logical diagram in Figure 232.

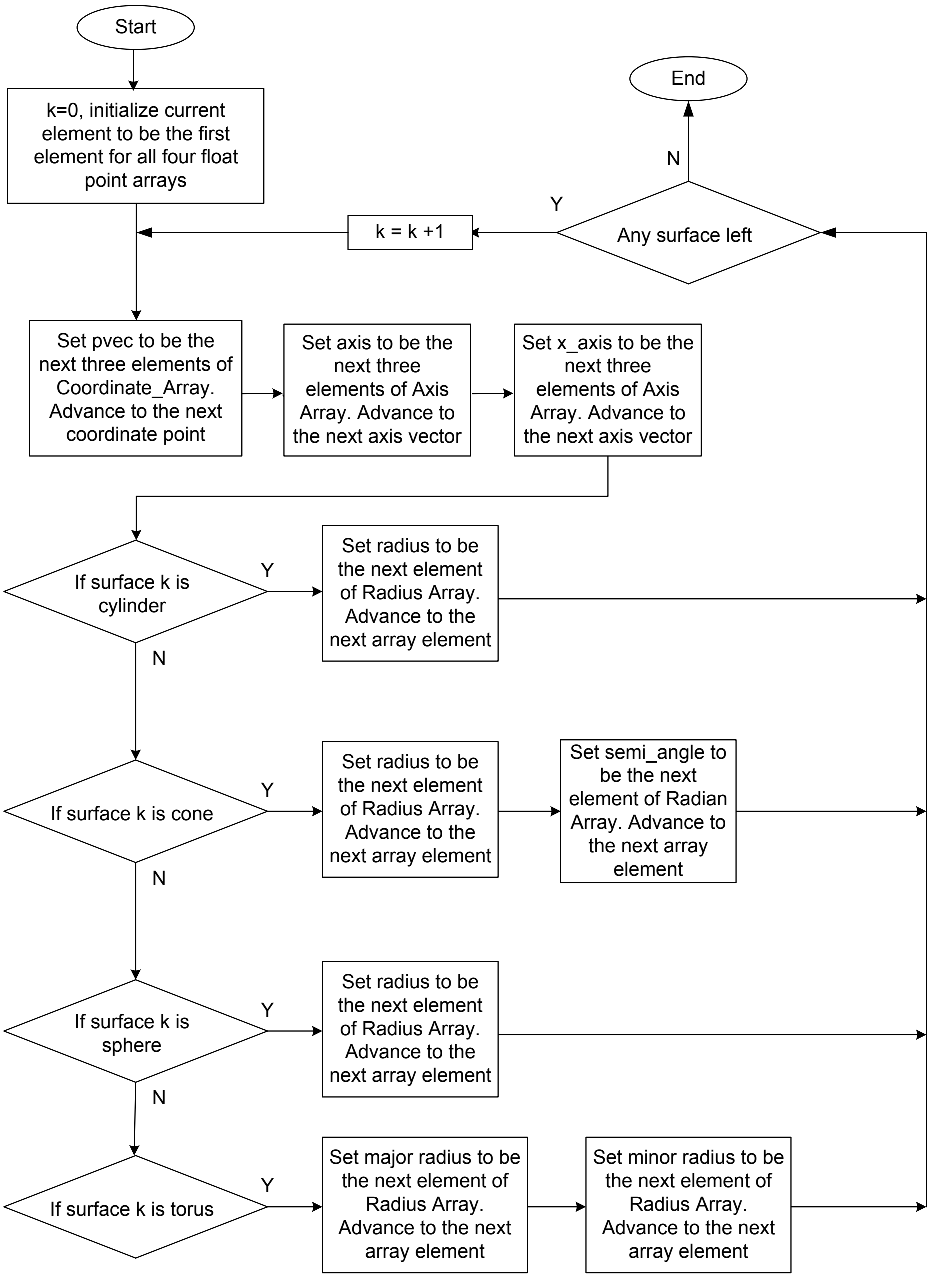

Start
End
k=0, initialize current element to be the first element for all four float point arrays
N
Y
k = k +1
Any surface left
Set pvec to be the next three elements of Coordinate_Array. Advance to the next coordinate point
Set axis to be the next three elements of Axis Array. Advance to the next axis vector
Set x_axis to be the next three elements of Axis Array. Advance to the next axis vector
If surface k is cylinder
Y
Set radius to be the next element of Radius Array. Advance to the next array element
N
If surface k is cone
Y
Set radius to be the next element of Radius Array. Advance to the next array element
Set semi_angle to be the next element of Radian Array. Advance to the next array element
N
If surface k is sphere
Y
Set radius to be the next element of Radius Array. Advance to the next array element
N
If surface k is torus
Y
Set major radius to be the next element of Radius Array. Advance to the next array element
Set minor radius to be the next element of Radius Array. Advance to the next array element

Figure 232 — Analytic Surface Creation

13 Data Compression and Encoding

The JT File format utilizes best-in-class compression and encoding algorithms to produce compact and efficient representations of data. The types of compression algorithms supported by the JT format vary from standard data type agnostic LZMA dictionary compreesion to entropy coding algorithms that exploit knowledge of the characteristics of the data types they are compressing. Some of the JT format data collections are always stored in a compressed format, whereas other data collections support multiple compression storage formats that qualitatively vary from ―lossless" compression to more aggressive strategies that employ ―lossy" compression. This support by the JT format of varying qualitative levels of compression allows producers of JT data to fine tune the tradeoff between compression ratio and fidelity of the data.

In some instances, data may be encoded/compressed using multiple techniques applied on top of one another in a serial fashion (i.e. encoding applied to the output of another encoder). One common example of this multiple encoding is when an array/vector of floating point data is first quantized into some integer codes and then these resulting integer codes are further compressed/encoded using an Arithmetic or BitLength CODEC (see 13.2 Encoding Algorithms).

Beyond the data collection specific compression/encoding, some JT format Data Segment types (see Data Segment) also support having LZMA compression conditionally applied to all the bytes of information persisted within the segment. So individual fields or collections of data may first have data type specific encoding/compression algorithms applied to them, and then if their Data Segment type supports it, the resulting data may be additionally compressed using LZMA.

Whether, and at what qualitative level, a particular Data Segment‘s data is compressed/encoded is indicated through compression related data values stored as part of the particular Data Segment storage format. In general, aggressive application of advanced compression/encoding techniques is reserved for the heavy-weight renderable geometric data (e.g. triangles and wireframe lines) which can exist in a JT File.

The following sections document the format of the data compression/encoding within the JT file. Along with documenting the format, a technical description of the various compression/encoding algorithms is included and an example implementation of the decoding portion of the algorithms can be found within Annex B.

13.1 Common Compression Data Collection Formats

For convenience and brevity in documenting the JT format, this section of the reference documents the format for several common —data compression/encoding” related data collections that can exist in the JT format. You will find references to these common compression data collections in the 5.2 Data Segments section of the document.

13.1.1 Int32 Compressed Data Packet

The Int32CDP (i.e. Int32 Compressed Data Packet) represents a third-generation format used to encode/compress a collection of data into a series of Int32 based symbols. This version of the Int32CDP supersedes the two similarly-named ones from the Version 9 JT Specification, and should not be confused with either of its predecessors. Note that the Int32 Compressed Data Packet collection can in itself contain another nested Int32 Compressed Data Packet collection in some cases.

Four distinct CODECs are available for use within the Int32 Compressed Data Packet, depending on the nature of the data to be compressed.

The Arithmetic CODEC is a so-called —entropy coder” because it can exploit the statistics present in the relative frequencies of the values being encoded. Basically, the more often a value is present, the fewer bits it takes to represent that value in the compressed codetext. Values that occur too infrequently to take advantage of this property are written *aside* into the —out-of-band data” array to be encoded separately. An —escape” symbol is encoded in their place as a placeholder in the primal CODEC (note, see —Symbol” data field definition in Int32 Compressed Data Packet for further details on the representation of —escape” symbol).

Essentially the —out-of-band data” is the high-entropy residue left over after the CODEC has squeezed all the advantage out of the original data stream that it can. However, this —out-of-band data” is sent back around for another pass because sometimes there are *new* or *different* statistics to be exploited.

The *Chopper* pseudo-CODEC‘s is used to identify fields of bits in a sequence of otherwise incompressible data that may be hiding low-entropy statistics that can be profitably exploited. In other words, it "chops" the input data up into bit fields, and then encodes them separately using the other CODECs, or in some cases, another round of chopping. The Chopper also removes *value bias* from the original input data array. Some input data arrays may contain values that are clustered around a certain central value. In these cases, it is profitable to first subtract out a *bias value* from the original input data. In some cases, this simple expedient may dramatically reduce the apparent field width necessary to code the variation in the original sequence.

In some cases, all values may be written as "out of band" when the Codec cannot perform *any* useful compression. In this case, the encoded CodeText Length field will be 0, and the I32 : Out-of-Band Value Count will be equal to I32 : Value Count. The implied action in this case is to merely copy the Out-Of-Band value data into the output Value Element array instead of invoking the Codec.

The **Move-to-Front pseudo-CODEC** is useful for data that exhibits spatial coherence (i.e. if a given value is likely to be used again in the near future). It decomposes the incoming data stream into two streams called —values” and —offsets”. Each time a new value is observed in the data stream, it is added to a small —cache” or

—window” of the most recently seen few values (16 in this case), the value emitted to the —values” array and an —escape” emitted into the —offsets” array. When a value is seen that is already in the cache, then only its offset into the window is emitted, and the value is moved to the front of the window (hence the name). Runs and clusters are thus more efficiently represented by the values/offsets arrays. These arrays in turn are subjected to a different CODEC to finish the job – most likely the Arithmetic or Bitlength.

When all other coding options have been exhausted, the Bitlength CODEC is invoked. The Bitlength CODEC directly encodes all values given to it, does not require a probability context, and hence never produces additional —out-of-band data”. The byte stops there, in other words.

Note that in the diagram below, encoding can loop back recursively for Out-Of-Band data and chopper fields. *For JT v10 files, the maximum recursion depth may not exceed eight*.

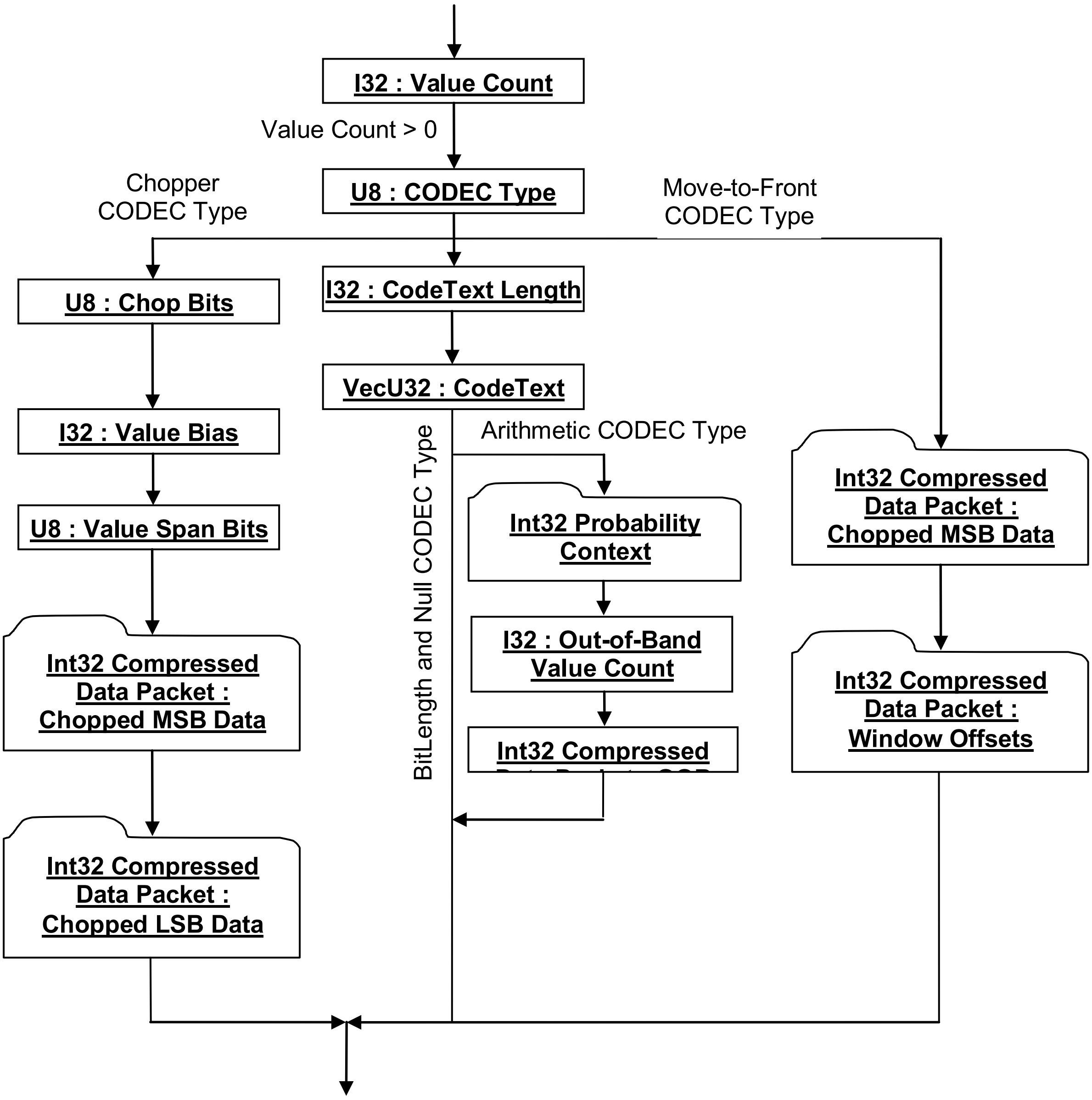

I32 : Value Count
Value Count > 0
U8 : CODEC Type
Chopper CODEC Type
Move-to-Front CODEC Type
U8 : Chop Bits
I32 : CodeText Length
VecU32 : CodeText
I32 : Value Bias
U8 : Value Span Bits
Int32 Compressed Data Packet : Chopped MSB Data
Int32 Compressed Data Packet : Chopped LSB Data
BitLength and Null CODEC Type
Arithmetic CODEC Type
Int32 Probability Context
I32 : Out-of-Band Value Count
Int32 Compressed Data Packet : OOB
Int32 Compressed Data Packet : Chopped MSB Data
Int32 Compressed Data Packet : Window Offsets

Figure 233 — Int32 Compressed Data Packet data collection

I32 : Value Count

Value Count specifies the number of values that the CODEC is expected to decode (i.e. it‘s like the —length” field written if you‘re just writing out a vector of integers). Upon completion of decoding the CodeText data field below, the number of decoded Values should be equal to Value Count. When only a single Probability Context Table is used, Value Count will also be equal to the number of Symbols decoded upon completion of decoding.

U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See 13.2 Encoding Algorithms for complete explanation of each of the encoding algorithms.

Table 93 — Int32 Probability Contexts CODEC Type values

= 0	Null CODEC
= 1	Bitlength CODEC
= 2	Illegal value
= 3	Arithmetic CODEC
= 4	Chopper CODEC
= 5	Move-to-front CODEC

I32 : CodeText Length

CodeText Length specifies the total number of bits of CodeText data .

VecU32 : CodeText

CodeText is the array/vector of encoded symbols. For CODEC Type not equal to —Null CODEC”, the total number of bits of encoded data in this array is indicated by the previously described CodeText Length data field.

U8 : Chop Bits

Chop Bits specifies the number of high-order bits "chopped off" from the *biased* input data array and coded separately from the low-order bits. Repeated applications of the Chopper pseudo-CODEC can expose low-entropy bit fields that would be inaccessible by directly coding the data array. Chop Bits is the number of bits coded into the Chopped MSB Data field. The number of Chop Bits is always greater than 0, and less than 32.

I32 : Value Bias

Value Bias is the (signed) number that is subtracted from the original input data array elements *before* computing Value Span Bits and Chop Bits. See Chopped LSB Data below for a full explanation of how to reconstitute the original data values using Value Bias and the two chopped fields.

U8 : Value Span Bits

Value Span Bits specifies the total bit width of the *biased* input data array. Note that Value Span Bits minus Chop Bits is the number of low-order bits present in the Chopped LSB Data field.

Int32 Compressed Data Packet : Chopped MSB Data

This field contains the separately compressed most significant bits of the *biased* input data array, whose elements contain Value Span Bits bits of significance. In other words, this field contains the bit field from the *biased* data array beginning at bit number ValueSpan-ChopBits and ending at bit number ValueSpan-1 inclusive. This field may contain negative numbers.

Int32 Compressed Data Packet : Chopped LSB Data

This field contains the separately compressed most significant bits of the original input data array, whose elements contain Value Span Bits bits of significance. In other words, this field contains the bit field from the original data array beginning at bit number 0 and ending at bit number ValueSpan-ChopBits-1 inclusive. This field may only contain positive numbers; all bits above this range shall encode to 0. A pseudo-code representation of the re-constituting the original data values is as follows:

```
OrigValue[i] = (LSBValue[i] | (MSBValue[i] << (ValSpanBits - ChopBits))) + ValueBias;
```

I32 : Out-of-Band Value Count

This field encodes the number of out-of-band values associated with the Arithmetic CODEC.

Int32 Compressed Data Packet : OOB Values

This field encodes the out-of-band Int32 values associated with the Arithmetic CODEC.

Int32 Compressed Data Packet : Window Values Used by the move to pseudo codec, reference Move-To-Front psuedo CODEC

Int32 Compressed Data Packet : Window Offsets

Used by the move to pseudo codec, reference Move-To-Front psuedo CODECInt32 Probability Context

Int32 Probability Context data collection encodes a Probability Context Table, and is present only for the Arithmetic CODEC Type. A Probability Context Table is a trimmed and scaled histogram of the input values. It tallies the frequencies of the several most frequently occurring values. It is central to the operation of the Arithmetic CODEC.

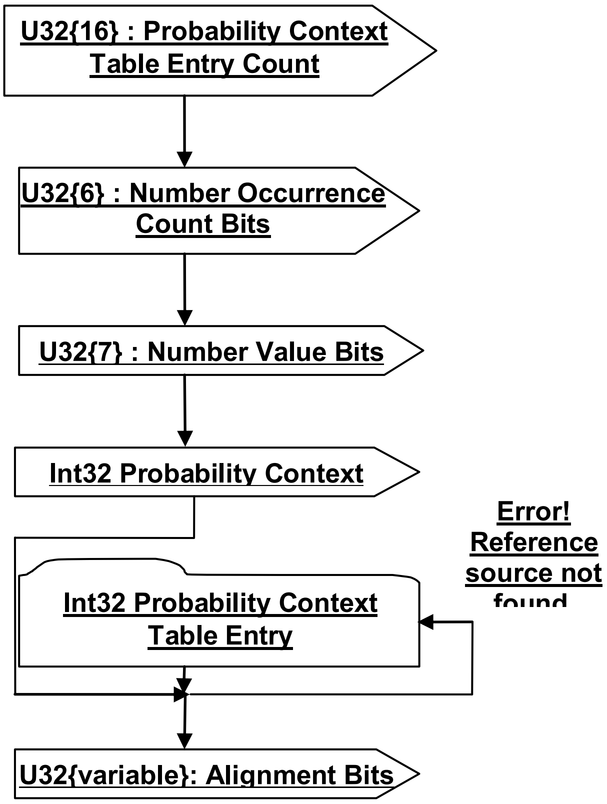

U32{16} : Probability Context Table Entry Count
U32{6} : Number Occurrence Count Bits
U32{7} : Number Value Bits
Int32 Probability Context
Error! Reference source not found
Int32 Probability Context Table Entry
U32{variable}: Alignment Bits

Figure 234 — Int32 Probability Context data collection

U32{16} : Probability Context Table Entry Count

Probability Context Table Entry Count specifies the number of entries in this Probability Context Table.

U32{6} : Number Occurrence Count Bits

Number Occurrence Count Bits specifies the number of bits used to encode the Occurrence Count range.

U32{7} : Number Value Bits

Number Value Bits specifies the number of bits used to encode the Associated Value range. Note that Number Value Bits is only specified in the JT file for the *first* Probability Context Table. If a second Probability Context Table is present, the Number Value Bits from the first should be used for the second as well.

I32{32} : Min Value

Min Value specifies the minimum of all Associated Values (i.e. one per table entry) stored in this Probability Context Table. This value is used to compute the real Associated Value for a Probability Context Table Entry. See Associated Value description in Int32 Probability Context Table Entry

U32{variable}: Alignment Bits

Alignment Bits represents the number of additional padding bits stored to arrive at the next even multiple of 8 bits. Values of —0‟ are stored in the alignment bits.

Note: Data written into a JT file is always aligned on bytes. Therefore after reading in a block of bit data such as the probability context tables it is necessary to discard any remaining bits on the last byte that is read in. This is represented by the —Alignment Bits” entry.

Int32 Probability Context Table Entry

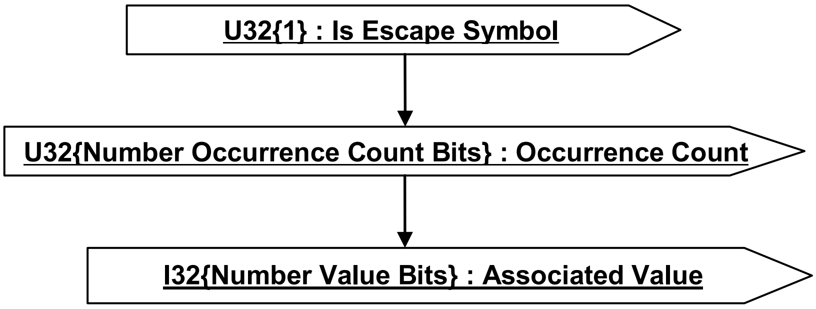

U32{1} : Is Escape Symbol
U32{Number Occurrence Count Bits} : Occurrence Count
I32{Number Value Bits} : Associated Value

Figure 235 — Int32 Probability Context Table Entry data collection

U32{1} : Is Escape Symbol

This Boolean flag denotes whether the context entry is the escape symbol entry. At most one entry will have this flag set to true in any context.

U32{Number Occurrence Count Bits} : Occurrence Count

Occurrence Count specifies the relative frequency of the value. Complete description for Number Occurrence Count Bits can be found in Int32 Probability Context.

Note: Occurrence Counts for all symbols are normalized (converted to a relative frequency) during the write process in order to ensure the minimum amount of bits possible is used to write them while closely approximating their actual frequency. This has several implications the reader should be aware of:

The sum of all Occurrence Counts is not guaranteed to equal the number of symbols to be decoded (see I32 : Value Count in section **Error! Reference source not found.** for number of symbols to be decoded).

During Arithmetic decoding as described in Annex B .

pDriver->numSymbolsToRead() – Refers to the total number of symbols to be decoded (i.e. I32 : Value Count in section **Error! Reference source not found.**).

pCurrContext->totalCount() – Refers to the sum of the ―Occurrence Count” values for all the symbols associated with a Probability Context.

I32{Number Value Bits} : Associated Value

Associated Value is the value (from the input data) that the symbol represents. The CODECs don‘t directly encode values, they encode symbols. Symbols, then, are associated with specific values, so when the CODEC decodes an array of symbols, you can reconstruct the array of values that was intended by looking up the symbols in the Probability Context Table. This value is stored with —Min Value” subtracted from the value. Complete descriptions for —Min Value” and Number Value Bits can be found in Int32 Probability Context.

Note: The associated value for an escape symbol is undefined and therefore can be any valid U32 number.

13.1.2 Int64 Compressed Data Packet

The Int64CDP(i.e. Int64 Compressed Data Packet) represents a format used to encode/compress a collection of data into a series of Int64 based symbols. Int64CDP shares the same encoding and compression logic as Int32CDP (i.e. Int32 Compressed Data Packet), except the data being compressed consists of an array of Int64 numbers intead of Int32 numbers.

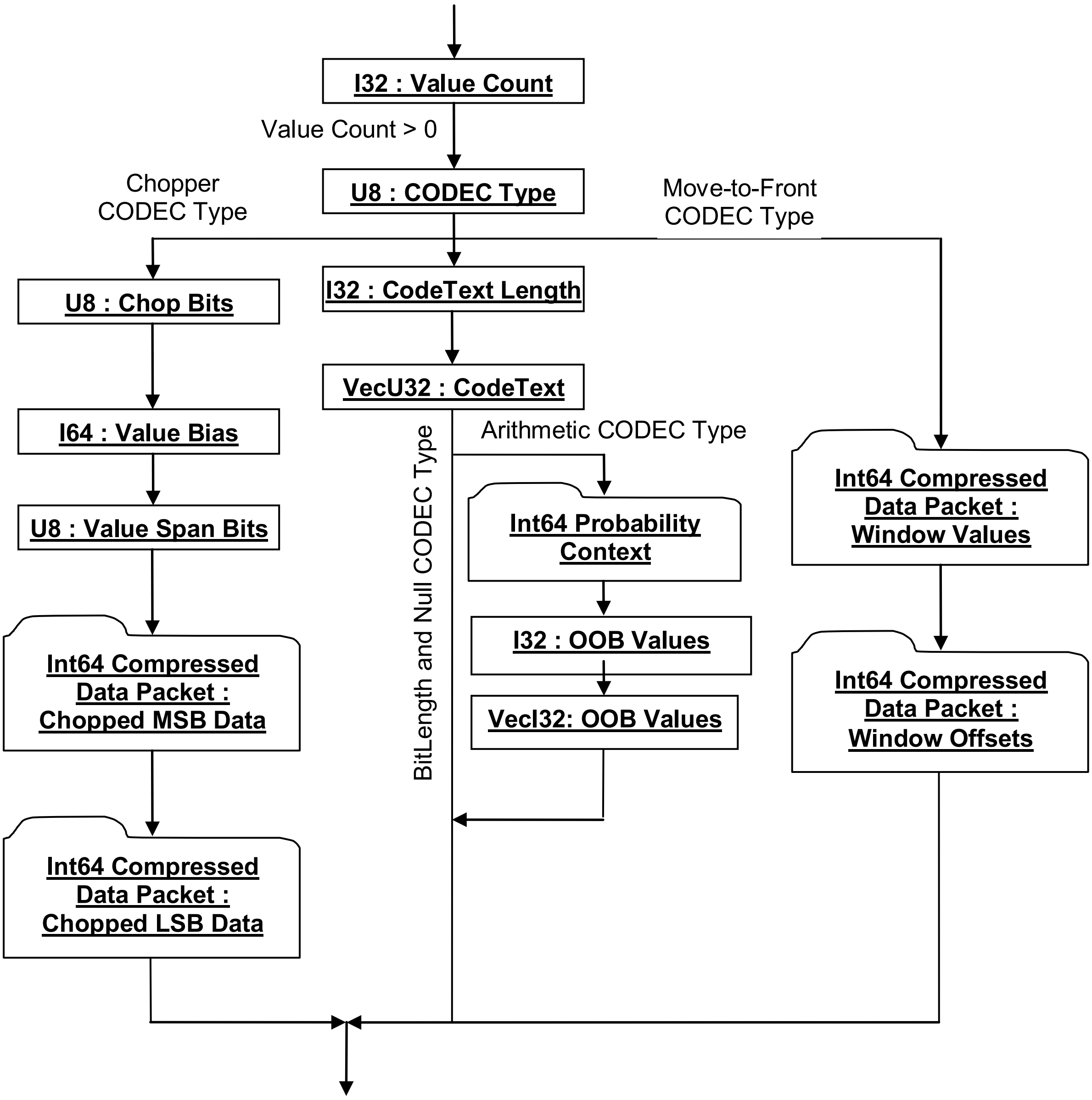

I32 : Value Count
Value Count > 0
U8 : CODEC Type
Chopper CODEC Type
Move-to-Front CODEC Type
U8 : Chop Bits
I32 : CodeText Length
VecU32 : CodeText
I64 : Value Bias
U8 : Value Span Bits
BitLength and Null CODEC Type
Arithmetic CODEC Type
Int64 Probability Context
I32 : OOB Values
VecI32: OOB Values
Int64 Compressed Data Packet : Chopped MSB Data
Int64 Compressed Data Packet : Chopped LSB Data
Int64 Compressed Data Packet : Window Values
Int64 Compressed Data Packet : Window Offsets

Figure 236 — Int64 Compressed Data Packet data collection

VecI32: OOB Values

This field encodes the out-of-band Int64 values associated with the Arithmetic CODEC.

I64 : Value Bias

The meaning of this field is the same as I32 : Value Bias except the data type is Int64 instead of Int32.

Int64 Compressed Data Packet : Chopped MSB Data

The meaning of this field is the same as Int32 Compressed Data Packet : Chopped MSB Data except the data type is Int64 instead of Int32.

Int64 Compressed Data Packet : Chopped LSB Data

The meaning of this field is the same as Int32 Compressed Data Packet : Chopped LSB Data except the data type is Int64 instead of Int32.

Int64 Compressed Data Packet : Window Values

The meaning of this field is the same as Int32 Compressed Data Packet : Chopped MSB Data except the data type is Int64 instead of Int32.

Int64 Compressed Data Packet : Window Offsets

The meaning of this field is the same as Int32 Compressed Data Packet : Window Offsets except the data type is Int64 instead of Int32.

Int64 Probability Context

Int64 Probability Context data collection encodes a Probability Context Table, and is present only for the Arithmetic CODEC Type. Int64 Probability Context is the same as Int32 Probability Context, except the data element is of type Int64 insteada of Int32.

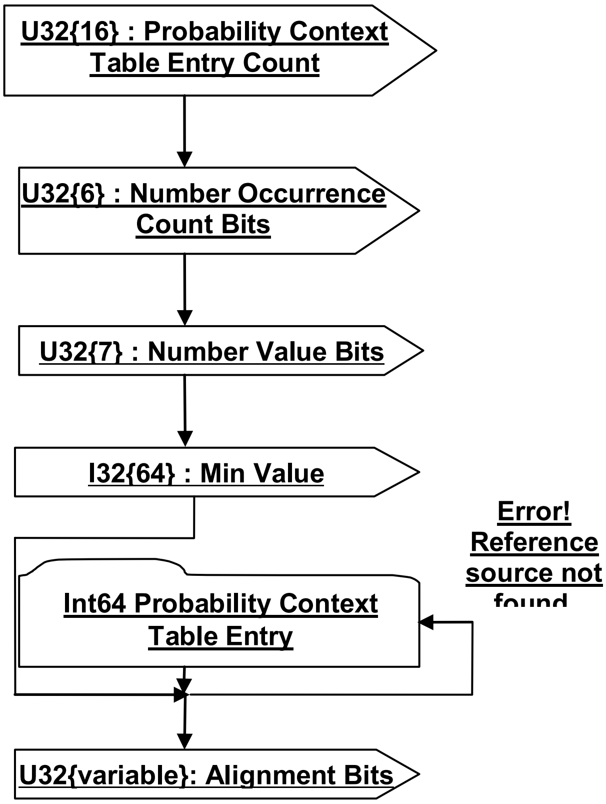

U32{16} : Probability Context Table Entry Count
U32{6} : Number Occurrence Count Bits
U32{7} : Number Value Bits
I32{64} : Min Value
Error! Reference source not found
Int64 Probability Context Table Entry
U32{variable}: Alignment Bits

Figure 237 — Int64 Probability Context data collection

I32{64} : Min Value

Min Value specifies the minimum of all Associated Values (i.e. one per table entry) stored in this Probability Context Table. This value is used to compute the real Associated Value for a Probability Context Table Entry. See Associated Value description in Int64 Probability Context Table Entry.

Int64 Probability Context Table Entry

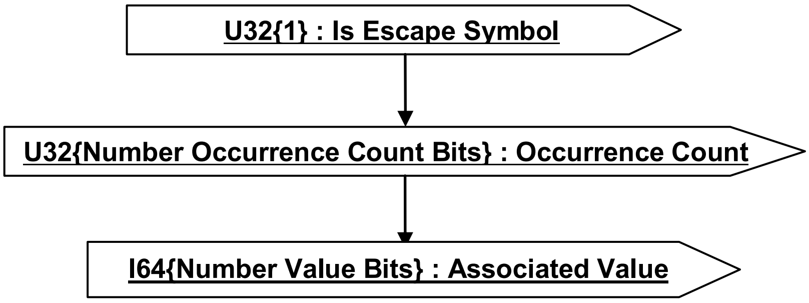

U32{1} : Is Escape Symbol
U32{Number Occurrence Count Bits} : Occurrence Count
I64{Number Value Bits} : Associated Value

Figure 238 — Int64 Probability Context Table Entry data collection

I64{Number Value Bits} : Associated Value

Similar to I32{Number Value Bits} : Associated Value, I64{Number Value Bits} : Associated Value is the value (from the input data) that the symbol represents. This value is stored with —Min Value” subtracted from the value. Complete descriptions for —Min Value” and Number Value Bits can be found in Int64 Probability Context.

13.1.3 Compressed Vertex Coordinate Array

The Compressed Vertex Coordinate Array data collection contains the quantization data/representation for a set of vertex coordinates.

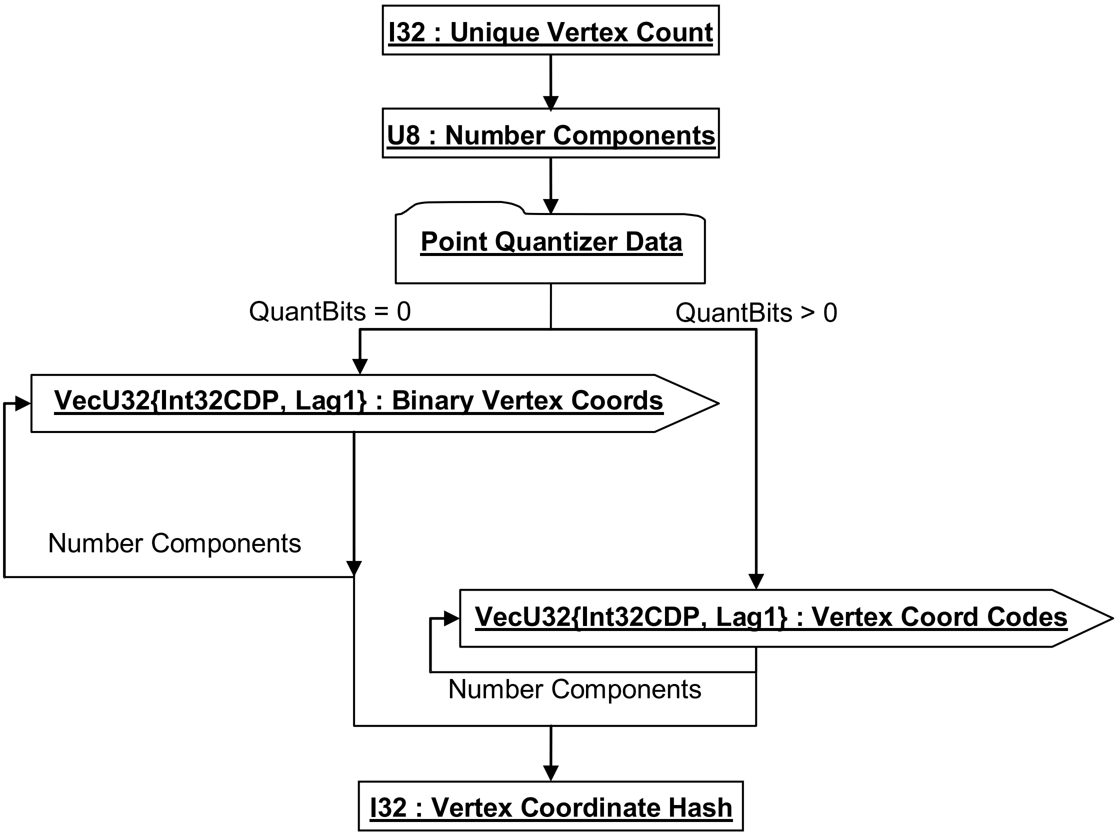

I32 : Unique Vertex Count
U8 : Number Components
Point Quantizer Data
QuantBits = 0
QuantBits > 0
VecU32{Int32CDP, Lag1} : Binary Vertex Coords
Number Components
VecU32{Int32CDP, Lag1} : Vertex Coord Codes
Number Components
I32 : Vertex Coordinate Hash

Figure 239 — Compressed Vertex Coordinate Array data collection

Complete description for Point Quantizer Data can be found in 13.1.3 Point Quantizer Data.

The above predicates —QuantBits = 0” and —QuantBits > 0” refer to the value of the field U8 : Number Of Bits stored in the three components of 13.1.9 Point Quantizer Data. All three of these fields are required to be equal.

I32 : Unique Vertex Count

Vertex Count specifies the count (number of unique) vertices in the Vertex Codes arrays. Identical values are only stored once therefore it may be necessary to smear out the vertices as described in TopoMesh Compressed Rep Data and TopoMesh Topologically Compressed LOD Data.

U8 : Number Components

Number Components specifies the number of vertex components present for each vertex record in the set of vertex records. The only legal value for this field is 3.

VecU32{Int32CDP, Lag1} : Binary Vertex Coords

Binary Vertex Coords is a vector of the ith component values of a set of vertex coordinates *interpreted* as integers. That is to say, the binary IEEE-754 floating point representation of the coordinates is fed *directly* into the Lag1 predictor as if they were integers.

VecU32{Int32CDP, Lag1} : Vertex Coord Codes

Vertex Coord Codes is a vector of quantizer ―codes" for all the ith component values of a set of vertex coordinates. Vertex Coord Codes uses the Int32 version of the CODEC to compress and encode data.

I32 : Vertex Coordinate Hash

The Vertex Coordinate Hash is the combined hash of the unique vertex coordinate records. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex coordinate values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the vertex coordinates codes for each of the component arrays. Refer to section 14.6 for a more detailed description on hashing.

```
UInt32 uHash    = 0;
uInt32 nUniqVtx = 0;
vecF32 vCoord[nUniqVtx][3];
vecU32 vCodes[3];
...
if ( uQuantBits == 0 ) {
  for ( int i=0 ; i<nComp ; i++ ) {
    for ( int j=0 ; j<nUniqVtx ; j++) {
      uHash = hash32( (const UInt32*)(&vCoord[j][i]), 1, uHash );
    }
  }
} else {
  for ( int i=0 ; i<nComp ; i++ ) {
   uHash = hash32( &vCodes[i], nUniqVtx, uHash );
  }
}
```

13.1.4 Compressed Vertex Normal Array

The Compressed Vertex Normal Array data collection contains the compressed data/representation for a set of vertex normals. Compressed Vertex Normal Array data collection is only present if previously read vertex bindings denote normals are present (See Vertex Shape LOD Data U64 : Vertex Bindings for complete explanation of the vertex bindings).

A variation of the CODEC developed by Michael Deering at Sun Microsystems is used to encode the normals when quantization is enabled. The variation being that the ―Sextants" are arranged differently than in Deering's scheme [4], for better delta encoding. See 13.2.4 Deering Normal CODEC for a complete explanation on the Deering CODEC used.

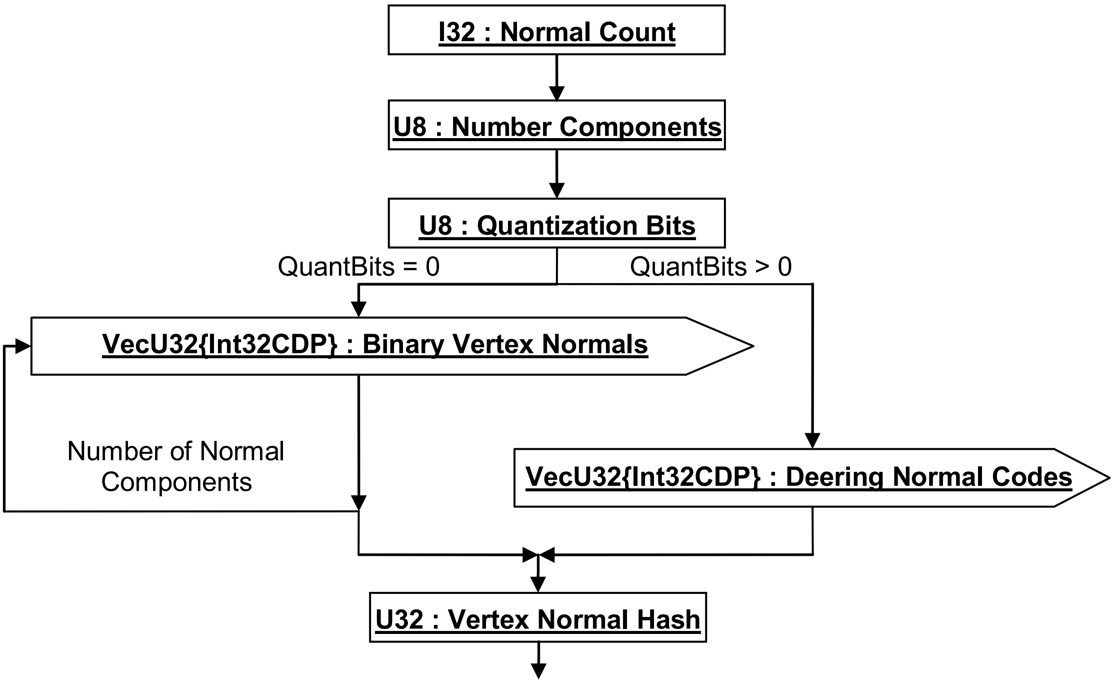


Figure 240 — Compressed Vertex Normal Array data collection

I32 : Normal Count

Normal count specifies the number of normals. This number should equal the total number of vertex records.

U8 : Number Components

Number Components specifies the number of normal components present for each vertex record in the set of vertex records.

U8 : Quantization Bits

The number of bits used when the Deering Normal CODEC if quantization is enabled. A value of 0 denotes that quantization is disabled. The maximum value for this field is 13 (so that the resulting Deering normal codes are of at most 32 bits).

VecU32{Int32CDP} : Binary Vertex Normals

Binary Vertex Normals is a vector of the ith component values of a set of vertex normals *interpreted* as integers. That is to say, the binary IEEE-754 floating point representation of the coordinates is fed *directly* into the Lag1 predictor as if they were integers.

VecU32{Int32CDP} : Deering Normal Codes

Deering Normal Codes is a vector of —codes” (one per normal) for a set of normals produced by the Deering Normal Codec (q.v.). Deering Normal Codes uses the Int32 version of the CODEC to compress and encode data.

U32 : Vertex Normal Hash

The Vertex Normal Hash is the combined hash of the vertex normals. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex normal values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the Sextant, Octant, Theta, and Psi Codes for all vertex records. Refer to section 14.6 for a more detailed description on hashing.

```
UInt32 uHash   = 0;
uInt32 nVtxRec = 0;
vecF32 vNorm[nVtxRec][3];
vecU32 vDeeringCodes;
...
if ( uQuantBits == 0 ) {
  for ( int i=0 ; i<nComp ; i++ ) {
    for ( int j=0 ; j<nVtxRec ; j++) {
      uHash = hash32( (UInt32*)(&vNorm[j][i]), 1, uHash );
    }
  }
} else {
   uHash = hash32( &vDeeringCodes, nVtxRec, uHash );
}
```

13.1.5 Compressed Vertex Texture Coordinate Array

The Compressed Vertex Texture Coordinate Array data collection contains the quantization data/representation for a set of vertex texture coordinates. Compressed Vertex Texture Coordinate Array data collection is only present if previously read vertex bindings denote texture coordinates are presents (See Vertex Shape LOD Data U64 : Vertex Bindings for complete explanation of the vertex bindings).

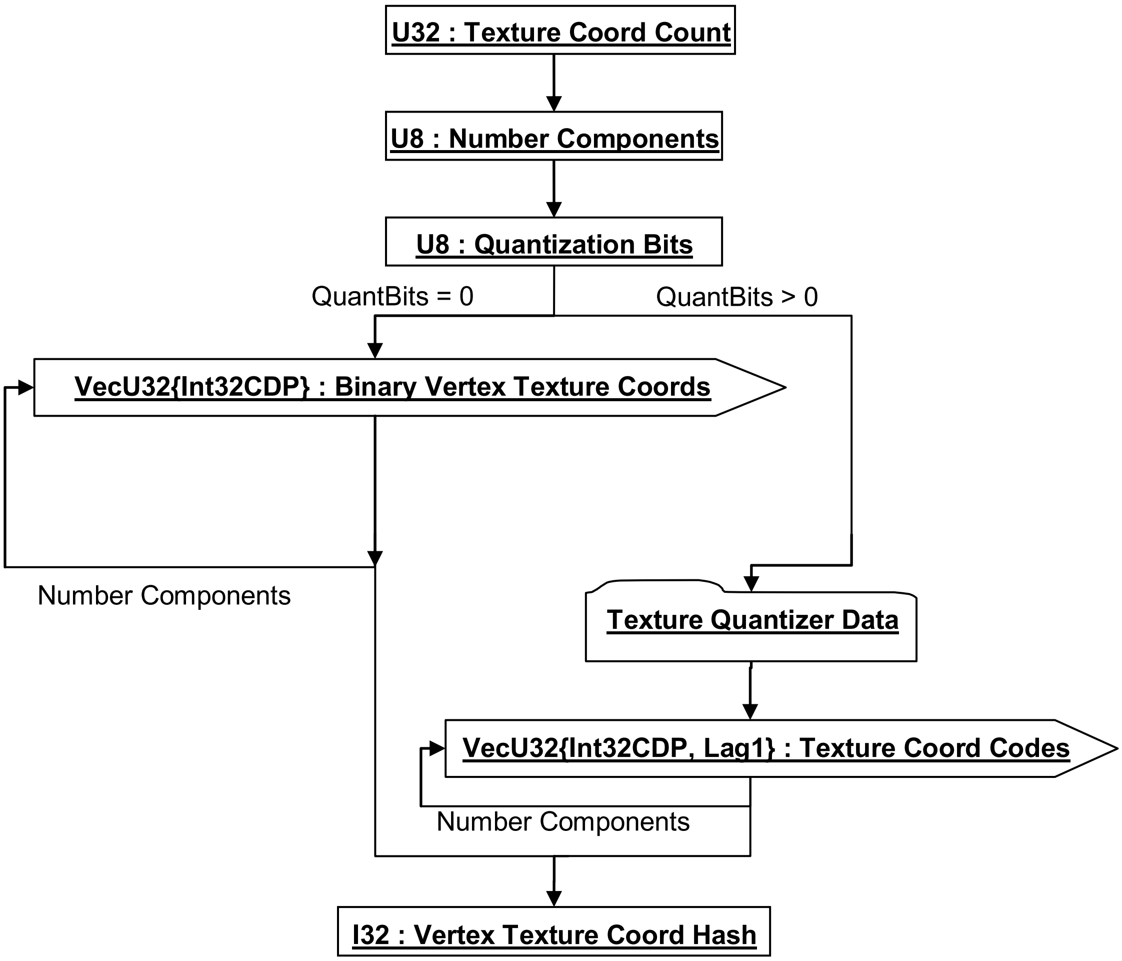

U32 : Texture Coord Count
U8 : Number Components
U8 : Quantization Bits
QuantBits = 0
QuantBits > 0
VecU32{Int32CDP} : Binary Vertex Texture Coords
Number Components
Texture Quantizer Data
VecU32{Int32CDP, Lag1} : Texture Coord Codes
Number Components
I32 : Vertex Texture Coord Hash

Figure 241 — Compressed Vertex Texture Coordinate Array data collection

Complete description for Texture Quantizer Data can be found in 13.1.10 Texture Quantizer Data.

U32 : Texture Coord Count

Colour count specifies the number of Texture Coordinates. This number should equal the total number of vertex records.

U8 : Number Components

Number Components specifies the number of Texture Coordinate components present for each vertex record in the set of vertex records.

U8 : Quantization Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision) for each of the components. The actual number of quantization bits used is specified within Texture Quantizer Data. Value shall be within range [0:24] inclusive.

VecU32{Int32CDP} : Binary Vertex Texture Coords

Binary Vertex Texture Coordinates is a vector of the ith component values of a set of texture coordinates *interpreted* as integers. That is to say, the binary IEEE-754 floating point representation of the coordinates is fed *directly* into the Lag1 predictor as if they were integers.

VecU32{Int32CDP, Lag1} : Texture Coord Codes

Texture Coord Codes is a vector of quantizer —codes” for all the nth-component of a set of vertex texture coordinates. Texture Coord Codes uses the Int32 version of the CODEC to compress and encode data.

I32 : Vertex Texture Coord Hash

The Vertex Texture Coord Hash is the combined hash of the Vertex Texture Coordinates. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex texture coordinate values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the vertex texture coordinates codes for each of the component arrays. Refer to section 14.6 for a more detailed description on hashing.

```
UInt32 uHash    = 0;
uInt32 nVtxRec = 0;
vecF32 vTexCoord[nVtxRec][4];
vecU32 vCodes[4];
...
if ( uQuantBits == 0 ) {
  for ( int i=0 ; i<nComp ; i++ ) {
    for ( int j=0 ; j<nVtxRec ; j++) {
      uHash = hash32( (UInt32*)(&vTexCoord[j][i]), 1, uHash );
    }
  }
} else {
  for ( int i=0 ; i<nComp ; i++ ) {
   uHash = hash32( &vCodes[i], nVtxRec, uHash );
  }
}
```

13.1.6 Compressed Vertex Colour Array

The Compressed Vertex Colour Array data collection contains the quantization data/representation for a set of vertex colours. Compressed Vertex Colour Array data collection is only present if previously read Colour Binding value is not equal to zero (See Vertex Shape LOD Data for complete explanation of Colour Binding data field).

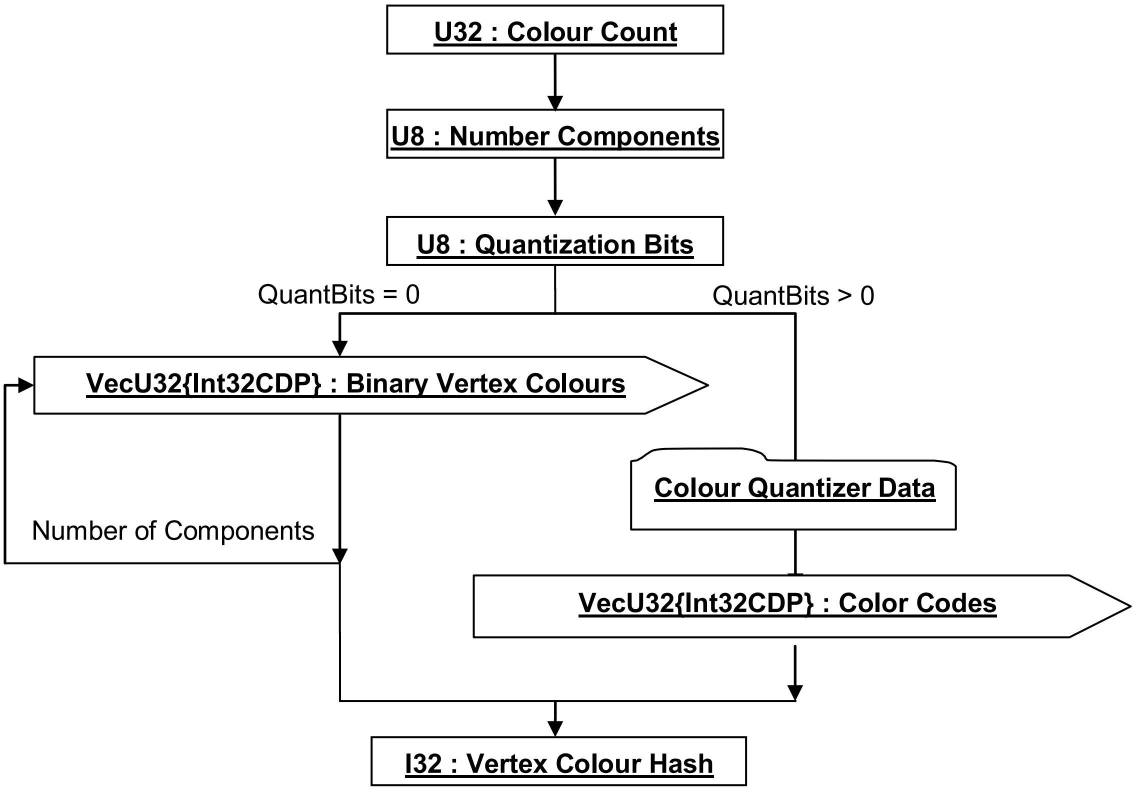


Figure 242 — Compressed Vertex Colour Array data collection

Complete description for Colour Quantizer Data can be found in 13.1.11 Colour Quantizer Data.

U32 : Colour Count

Colour count specifies the number of colour records. This number should equal the total number of vertex records.

U8 : Number Components

Number Components specifies the number of Colour components present for each vertex record in the set of vertex records.

U8 : Quantization Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision) for each of the 3 or 4 colour components. This value shall satisfy the following condition: —0<= Number Of Bits <= 8”.

VecU32{Int32CDP} : Binary Vertex Colours

Binary Vertex Texture Coordinates is a vector of the ith component values of a set of texture coordinates *interpreted* as integers. That is to say, the binary IEEE-754 floating point representation of the coordinates is fed *directly* into the Lag1 predictor as if they were integers.

VecU32{Int32CDP} : Color Codes

Color Codes is a vector of quantizer —codes” for all the vertex colours. Each Color Code contains up to four bit fields representing the RGBA or HSVA encoded color. The width of each field is set by the corresponding data in 13.1.11 Colour Quantizer Data. The alpha field lies in the least significant bits, the B/V field lies immediately to the left of the alpha field, the G/S field lies immediately to the left of B/V field, and so on toward the more significant bits.

I32 : Vertex Colour Hash

The Vertex Colour Hash is the combined hash of the vertex colours. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex colour values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the hash of the Color Codes vector. Refer to section 14.6 for a more detailed description on hashing.

```
UInt32 uHash    = 0;
uInt32 nVtxRec = 0;
vecF32 vCol[nVtxRec][3];
vecU32 vColorCodes;
...
if ( uQuantBits == 0 ) {
  for ( int i=0 ; i<nComp ; i++ ) {
    for ( int j=0 ; j<nVtxRec ; j++) {
      uHash = hash32( (UInt32*)(&vCol[j][i]), nVtxRec, uHash );
    }
  }
} else {
   uHash = hash32( &vColorCodes, nVtxRec, uHash );
}
```

13.1.7 Compressed Vertex Flag Array

The Compressed Vertex Flag Array data collection contains the quantization data/representation for per vertex flags. Compressed Vertex Flag Array data collection is only present if previously read Vertex Flag Binding value is not equal to zero.

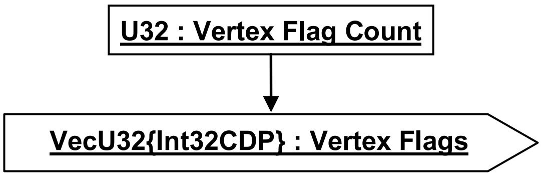


Figure 243 — Compressed Vertex Flag Array data collection

U32 : Vertex Flag Count

Vertex flag count specifies the number of vertex flags. This number should be equal to the total number of vertex records.

VecU32{Int32CDP} : Vertex Flags

Vertex Flags is a vector of per vertex bit flags encoded as integers with valid values of either 0 (false) or 1 (true). Vertex Flags uses the Int32 version of the CODEC to compress and encode data.

13.1.8 Compressed Auxiliary Fields Array

Compressed Auxiliary Fields Array data contains additional geometric shape data (auxiliary vertex fields) that may be associated with each vertex record defined in TopoMesh Compressed LOD Data. Each Auxiliary field contains data that is parallel to the existing vertex record fields in order capture additional information about each vertex (e.g. Vertex Identifiers, Weights, or other information). Importantly, each datum in the Auxiliary field may have a single value (of the specified type), or *may have many values* (called „steps—).Again, each data collection may have multiple Auxiliary fields, each of which contains one datum per vertex record in the TopoMesh Compressed LOD Data, each datum containing 1 or more values (steps).

Each Auxiliary field has a GUID tag (unique to the fields in the current TopoMesh Compressed LOD Data record), and a field data type that allows the user to store a variety of different data types. It is not intended that Auxiliary fields be required to directly participate in rendering – that is the province of the vertex attributes defined in TopoMesh Compressed LOD Data.

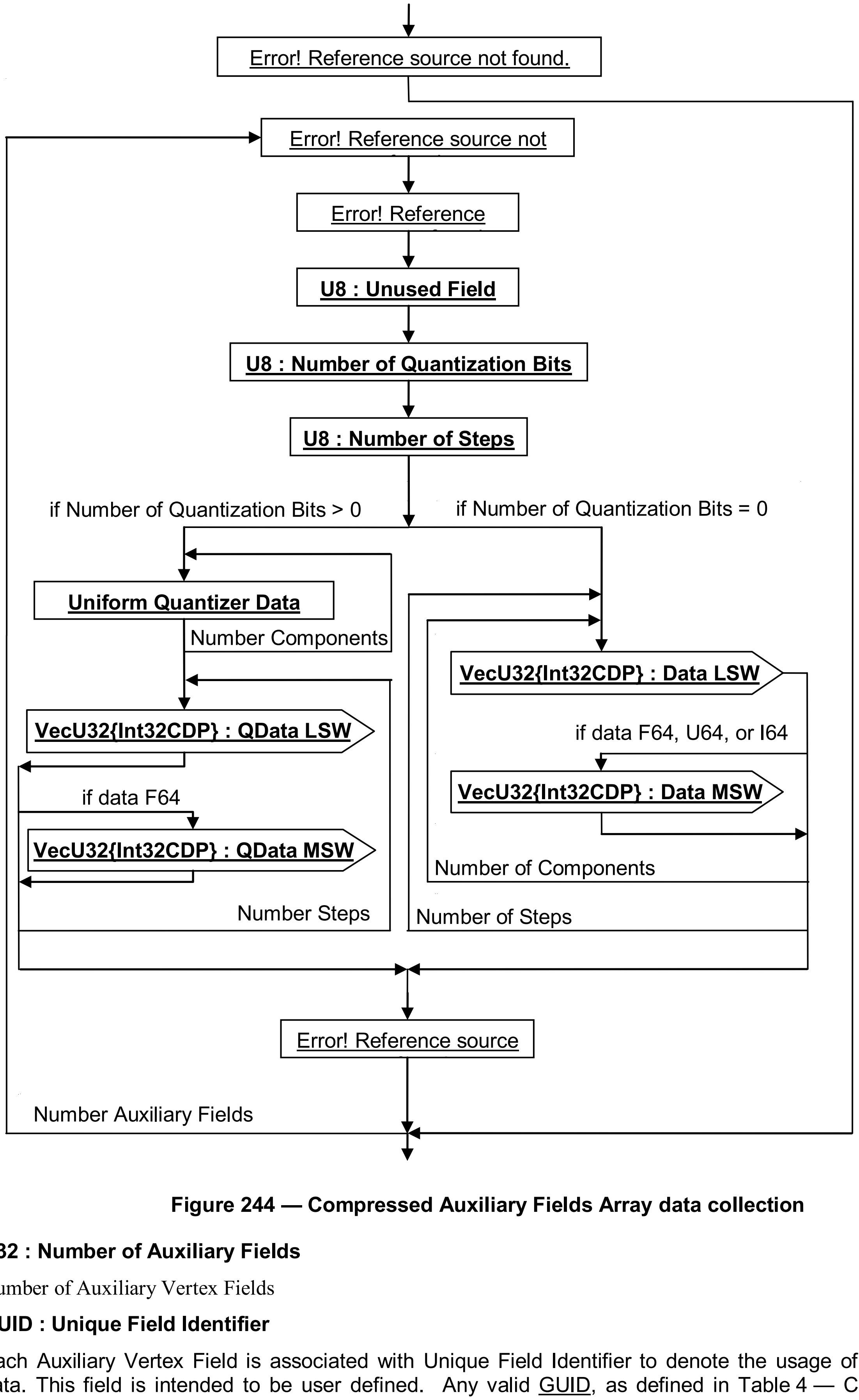

Error! Reference source not found.
Error! Reference source not
Error! Reference
U8 : Unused Field
U8 : Number of Quantization Bits
U8 : Number of Steps
if Number of Quantization Bits > 0
if Number of Quantization Bits = 0
Uniform Quantizer Data
Number Components
VecU32{Int32CDP} : QData LSW
if data F64
VecU32{Int32CDP} : QData MSW
Number Steps
VecU32{Int32CDP} : Data LSW
if data F64, U64, or I64
VecU32{Int32CDP} : Data MSW
Number of Components
Number of Steps
Error! Reference source
Number Auxiliary Fields

Figure 244 — Compressed Auxiliary Fields Array data collection

U32 : Number of Auxiliary Fields

Number of Auxiliary Vertex Fields

GUID : Unique Field Identifier

Each Auxiliary Vertex Field is associated with Unique Field Identifier to denote the usage of the contained data. This field is intended to be user defined. Any valid GUID, as defined in Table 4 — Composite Data Types, can be used as a Unique Field Identifier.

U8 : Field Type

Defines the number of components and type of data contained within the auxiliary field based upon the below table.

Table 94 — TopoMesh Compressed Rep Data V2 Field Type values

Type	Data	Components
1	U8	1
2	U8	2
3	U8	3
4	U8	4
5	I8	1
6	I8	2
7	I8	3
8	I8	4
9	U16	1
10	U16	2
11	U16	3
12	U16	4
13	I16	1
14	I16	2
15	I16	3
16	I16	4
17	U32	1
18	U32	2
19	U32	3
20	U32	4
21	I32	1
22	I32	2
23	I32	3
24	I32	4
25	U64	1
26	U64	2
27	U64	3
28	U64	4
29	I64	1
30	I64	2
31	I64	3
32	I64	4
33	F32	1
34	F32	2
35	F32	3
36	F32	4
37	F32	2x2
38	F32	3x3
39	F32	4x4
40	F64	1
41	F64	2
42	F64	3
43	F64	4
44	F64	2x2
45	F64	3x3
46	F64	4x4

U8 : Unused Field

This field is unused.

U8 : Number of Quantization Bits

Each Auxiliary field can be lossily or losslessly compressed. A value of 0 in this field means that that field data are to be losslessly compressed. This field must be 0 for all integer field types (i.e. lossy compression is not defined for integer field types). Only floating-point field types may be lossily compressed. A value between 1 and 32, or between 1 and 64 for double precision floating point typed fields, indicates that the data are quantized to the indicated number of bits of significance.

U8 : Number of Steps

This field represents the number of —steps” present with each vertex record for the given Auxiliary field. The field must be a number greater than 0. All vertex records have the same number of —steps.” One or more steps, called —suppressed steps”, may contain no auxiliary data. At least one step must be not suppressed for any auxiliary field.

VecU32{Int32CDP} : Data LSW

Data LSW is an array of the low order 32 bits of the vector formed from all steps, components, and vertex records of a single Auxiliary Data field. The data is laid out by cycling through vertex records first, then components, and steps last so that adjacent vertex records for the same step and component are contiguous (i.e. in —stepwise-major, component semi-major” order) For U8, I8, U16, I16, U32, I32 and lossless F32 data types this contains all bits. For U64, I64, and lossless F64 data types it contains bits 0 through 31. Data LSW uses the Int32 version of the CODEC to compress and encode data. In the case when the data is a zero length vector, it means that no auxiliary data exists on any vertex record for this component in this step. In addition, if no auxiliary data exists for one component in a particular step then it is guaranteed that no auxiliary data exists for all the other components in the same step.

VecU32{Int32CDP} : Data MSW

Data MSW is an array of the low order 32 bits of the vector formed from all steps, components, and vertex records of a single Auxiliary Data field. The data is laid out by cycling through vertex records first, then components, and steps last so that adjacent vertex records for the same step and component are contiguous (i.e. in ―stepwise-major, component semi-major” order). For U64, I64, and lossless F64 data types it contains bits 32 through 63. Data MSW uses the Int32 version of the CODEC to compress and encode data. In the case when the data is a zero length vector, it means that no auxiliary data exists on any vertex record for this component in this step. In addition, if no auxiliary data exists for one component in a particular step then it is guaranteed that no auxiliary data exists for all the other components in the same step.

VecU32{Int32CDP} : QData LSW

QData LSW is an array of the low order 32 bits of the vector formed from all components and vertex records of a single Auxiliary Data field. The data is laid out by cycling through vertex records first, then components so that adjacent vertex records for the same step are contiguous (i.e. in ―stepwise-major” order) For the F32 data type this field contains all bits. For F64 the data type it contains bits 0 through 31. QData LSW uses the Int32 version of the CODEC to compress and encode data. Note that there is one QData LSW packet for each step in the Auxfield rather than a single unified packet as with Data LSW. In the case when the data is a zero length vector, it means that no auxiliary data exists on any vertex record for this component in this step. In addition, if no auxiliary data exists for one component in a particular step then it is guaranteed that no auxiliary data exists for all the other components in the same step.

VecU32{Int32CDP} : QData MSW

QData MSW is an array of the low order 32 bits of the vector formed from all steps, components, and vertex records of a single Auxiliary Data field. The data is laid out by cycling through vertex records first, then components, and steps last so that adjacent vertex records for the same step and component are contiguous (i.e. in ―stepwise-major, component semi-major” order). For U64, I64, and lossless F64 data types it contains bits 32 through 63. QData MSW uses the Int32 version of the CODEC to compress and encode data. Note that there is one QData MSW packet for each step in the Auxfield rather than a single unified packet as with Data MSW. In the case when the data is a zero length vector, it means that no auxiliary data exists on any vertex record for this component in this step. In addition, if no auxiliary data exists for one component in a particular step then it is guaranteed that no auxiliary data exists for all the other components in the same step.

I32 : Auxiliary Data Hash

The Auxiliary Data Hash is the combined hash of auxiliary field data arrays. Refer to section 14.6 for a more detailed description on hashing.

```
UInt32 uHash      = 0;
UInt32 nVtxRec    = 0,  // Number of vertex records
       nComp      = 0,  // Number of components in current Auxfield
       nSteps     = 0;  // Number of steps in current Auxfield
vecU32 vDataLSW [nSteps][nComp],
       vDataMSW [nSteps][nComp],
       vQDataLSW[nSteps],
       vQDataMSW[nSteps];
...
if (nQuantBits == 0) {
  if ( bU8 || bI8 || bU16 || bI16 || bU32 || bI32 || bF32) {
    for ( int i=0 ; i<nSteps ; i++ )
      for ( int j=0 ; j<nComp ; j++ )
        uHash = hash32( vDataLSW[i][j], nVtxRec, uHash );
  }
  else {   // bU64 || bI64
    for ( int i=0 ; i<nSteps ; i++ ) {
      for ( int j=0 ; j<nComp ; j++ ) {
        uHash = hash32( vDataLSW[i][j], nVtxRec, uHash );
        uHash = hash32( vDataMSW[i][j], nVtxRec, uHash );
      }
    }
  }
} else {
  if (bF32) {
    for ( int i=0 ; i<nSteps ; i++ )
      uHash = hash32( vQDataLSW[i], nVtxRec * nComp, uHash );
  }
  else {   // bF64
```

```
    for ( int i=0 ; i<nSteps ; i++ ) {
      uHash = hash32( vQDataLSW[i], nVtxRec * nComp, uHash );
      uHash = hash32( vQDataMSW[i], nVtxRec * nComp, uHash );
    }
  }
}
```

13.1.9 Point Quantizer Data

A Point Quantizer Data collection is made up of three Uniform Quantizer Data collections; there is a separate Uniform Quantizer Data collection for the X, Y, and Z values of point coordinates.

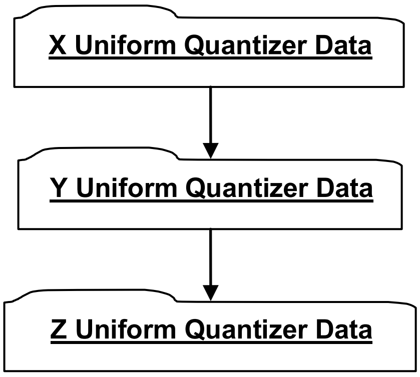


Figure 245 — Point Quantizer Data collection

Complete description for X Uniform Quantizer Data, Y Uniform Quantizer Data and Z Uniform Quantizer Data can be found in 13.1.12 Uniform Quantizer Data.

13.1.10 Texture Quantizer Data

A Texture Quantizer Data collection is made up of n Uniform Quantizer Data collections; there is a separate Uniform Quantizer Data collection for each component of the texture coordinates. The number of components is not specified within the quantizer, but rather is determined by the number of texture components present in the underlying data (See Compressed Vertex Texture Coordinate Arrays U8 : Number Components).

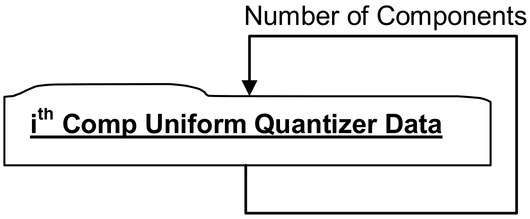


Figure 246 — Texture Quantizer Data collection

Complete description for U Uniform Quantizer Data, and V Uniform Quantizer Data can be found in 13.1.12 Uniform Quantizer Data.

13.1.11 Colour Quantizer Data

A Colour Quantizer Data collection contains the quantizer information for each of the colour components. The Colour Quantizer utilizes a separate Uniform Quantizer Data collection for each of the 4 colour components, but if the HSV colour model is being used, then it is not necessary to store a complete Uniform Quantizer Data Collection.

For the HSV model, since the range values for each colour component are constant, only the Number of Bits of precision for each colour component's Uniform Quantizer is stored. The Uniform Quantizer range values for the HSV colour components should always be assumed to be the following:

Table 95 — Colour Quantizer values

Component	Quantizer Range	
	Min	Max
Hue	0.0	6.0
Saturation	0.0	1.0
Value	0.0	1.0
Alpha	0.0	1.0

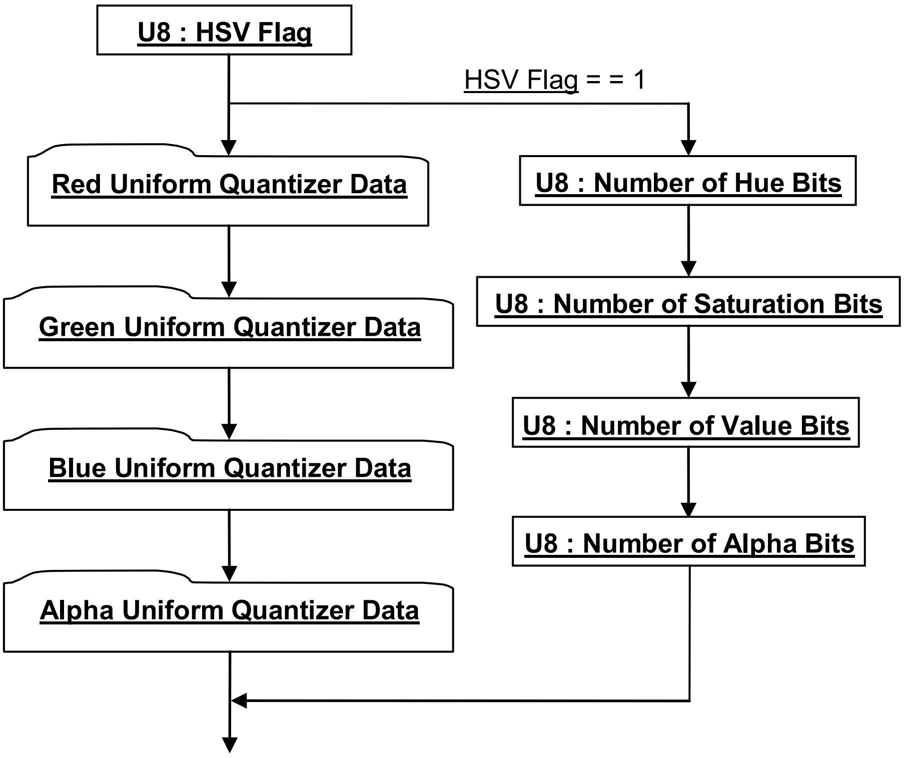

U8 : HSV Flag
HSV Flag = = 1
Red Uniform Quantizer Data
U8 : Number of Hue Bits
U8 : Number of Saturation Bits
Green Uniform Quantizer Data
U8 : Number of Value Bits
Blue Uniform Quantizer Data
U8 : Number of Alpha Bits
Alpha Uniform Quantizer Data

Figure 247 — Colour Quantizer Data collection

Complete descriptions for Red Uniform Quantizer Data, Green Uniform Quantizer Data, Blue Uniform Quantizer Data, and Alpha Uniform Quantizer Data can be found in 13.1.12 Uniform Quantizer Data. These four Uniform Quantizer Data collections are only present when data field HSV Flag = = 0.

U8 : HSV Flag

HSV Flag is a flag indicating whether colour component data is stored in HSV colour model form.

Table 96 — Colour Quantizer HSV Flag values

= 0	Colour component data stored in RGB colour model form.
= 1	Colour component data stored in HSV colour model form.

U8 : Number of Hue Bits

Number of Hue Bits specifies the quantized size (i.e. the number of bits of precision) for the Hue component of the colour. Number of Hue Bits data is only present when data field HSV Flag = = 1.

U8 : Number of Saturation Bits

Number of Saturation Bits specifies the quantized size (i.e. the number of bits of precision) for the Saturation component of the colour. Number of Saturation Bits data is only present when data field HSV Flag = = 1.

U8 : Number of Value Bits

Number of Value Bits specifies the quantized size (i.e. the number of bits of precision) for the Value component of the colour. Number of Value Bits data is only present when data field HSV Flag = = 1.

U8 : Number of Alpha Bits

Number of Alpha Bits specifies the quantized size (i.e. the number of bits of precision) for the Alpha component of the colour. Number of Alpha Bits data is only present when data field HSV Flag = = 1.

13.1.12 Uniform Quantizer Data

The Uniform Quantizer Data collection contains information that defines a scalar quantizer/dequantizer (encoder/decoder) whose range is divided into levels of equal spacing.

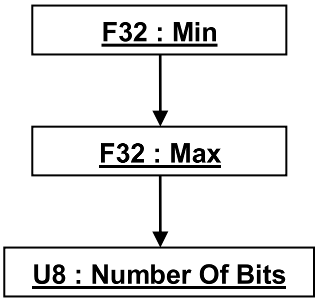


Figure 248 — Uniform Quantizer Data collection

F32 : Min

Min specifies the minimum of the quantized range.

F32 : Max

Max specifies the maximum of the quantized range.

U8 : Number Of Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision). In general, this value shall satisfy the following condition: —0<= Number Of Bits <= 32”.

13.1.13 Compressed Entity List for Non-Trivial Knot Vector

Compressed Entity List for Non-Trivial Knot Vector data collection specifies index identifiers (i.e. indices to particular entities within a list of entities) for a set of entities that contain Non-Trivial Knot Vectors. The entity types which can contain non-trivial knot vectors include:

JT B-Rep NURBS Surfaces

JT B-Rep PCS NURBS Curves

JT B-Rep MCS NURBS Curves

Wireframe MCS NURBS Curves

Note that any one occurrence of Compressed Entity List for Non-Trivial Knot Vector data collection will only contain index identifiers for one particular type of the above listed entities. The entity type is inferred based on the data collection which includes/references the Compressed Entity List for Non-Trivial Knot Vector.

A trivial knot vector is one which completely satisfies all conditions of at least one of the following cases:

Case-1 for trivial knot vector:

- Number of knots is an even number.
- Knot vector has a [0:1] knot range.
- There are no interior knots (i.e. NumberKnots = = 2 * (NurbsEntityDegree + 1).

Case-2 for trivial knot vector:

- Number of knots is an even number.
- Knot vector has a [0:1] knot range.
- NurbsEntityDegree < 3.
- Difference between successive non-repeating knots (i.e. KnotDelta) is:
  - KnotDelta = 2.0 / (NumberKnots – (2.0 * NurbsEntityDegree)).

Any knot vector which does not satisfy one of the above cases for ―trivial knot vector" is classified as a ―non-trivial knot vector."

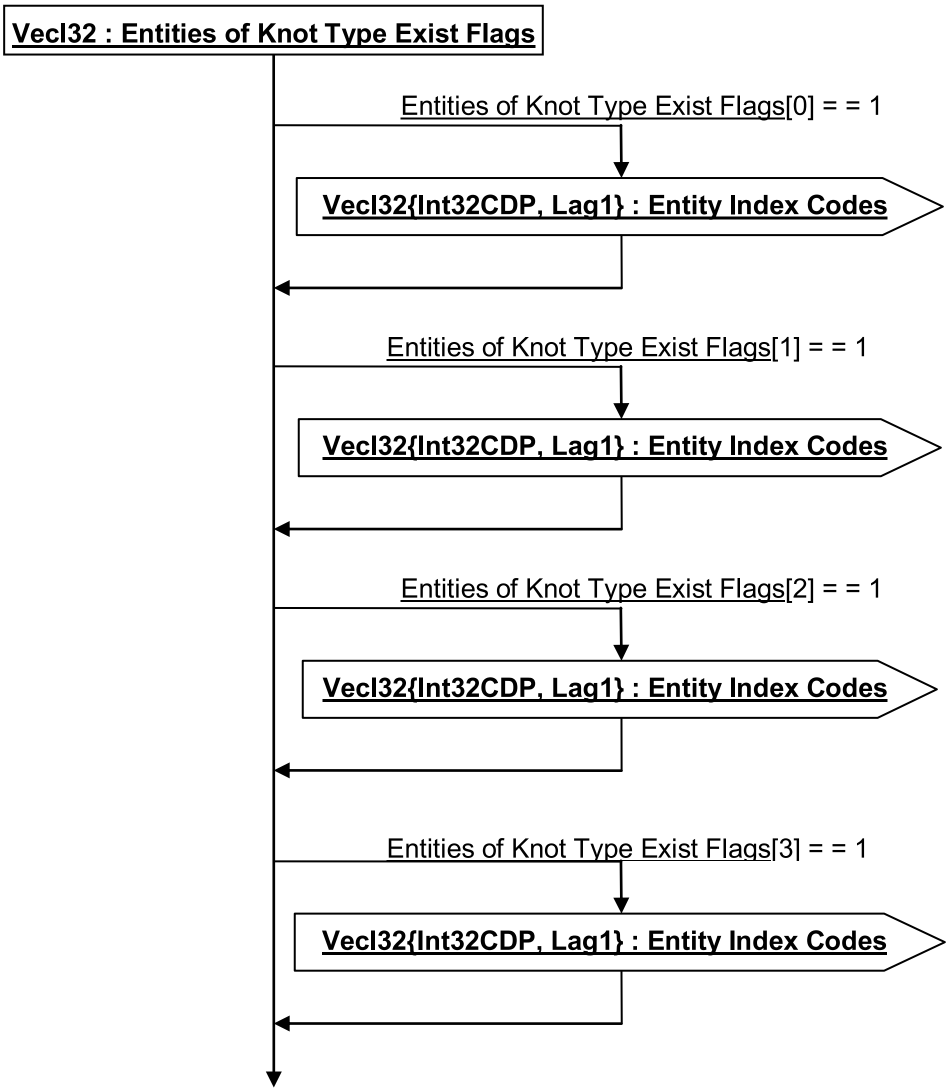

VecI32 : Entities of Knot Type Exist Flags
Entities of Knot Type Exist Flags[0] = = 1
VecI32{Int32CDP, Lag1} : Entity Index Codes
Entities of Knot Type Exist Flags[1] = = 1
VecI32{Int32CDP, Lag1} : Entity Index Codes
Entities of Knot Type Exist Flags[2] = = 1
VecI32{Int32CDP, Lag1} : Entity Index Codes
Entities of Knot Type Exist Flags[3] = = 1
VecI32{Int32CDP, Lag1} : Entity Index Codes

Figure 249 — Compressed Entity List for Non-Trivial Knot Vector data collection

VecI32 : Entities of Knot Type Exist Flags

Entities of Knot Type Exist Flags, is a vector of flags indicating for each knot vector type whether Entity Index ID data collections exist/follow for that knot vector type. Knot Vectors are categorized into types based on the following characteristics: whether internal knots occur in *adjacent pairs* and whether the knot range is [0:1] or some other [x_1:x_2] range.

Currently there are four knot vector types, so this Entities of Knot Type Exist Flags vector should be of length four. The four flags have the following meaning:

Table 97 — Knot Type Exist Flag values

[0]	Flag indicating whether Entity IDs data collection exists for -Even Count [0:1] Range" knot type. Knots in this category have their knot range on [0:1], internal knots occur in *adjacent pairs*, *except* when there are no internal knots, in which case Type = 1 instead. = 0 – No Entity IDs data collection exists. = 1 – Entity IDs data collection exists.
[1]	Flag indicating whether Entity IDs data collection exists for -Even Count $[x_1:x_2]$ Range" knot type. Knots in this category have their knot range on $[x_1:x_2]$, and internal knots occur in *adjacent pairs*. = 0 – No Entity IDs data collection exists. = 1 – Entity IDs data collection exists.
[2]	Flag indicating whether Entity IDs data collection exists for -Odd Count [0:1] Range" knot type. Knots of this type have their knot range on [0:1], and are not Type 0. = 0 – No Entity IDs data collection exists. = 1 – Entity IDs data collection exists.
[3]	Flag indicating whether Entity IDs data collection exists for -Odd Count $[x_1:x_2]$ Range" knot type. Knots of this type have their knot range on $[x_1:x_2]$, and are not Type 1. = 0 – No Entity IDs data collection exists. = 1 – Entity IDs data collection exists.

Examples of knot vectors of Type 0:

```
    0 0 X X 1 1
    0 0 X X Y Y 1 1
    0 0 X X Y Y Z Z 1 1
```

Examples of knot vectors of Type 1:

```
    0 0 1 1         (Note: This is the exception to Type 0)
    X X Y Y
    X X Y Y Z Z
    X X Y Y Z Z W W
```

Examples of knot vectors of Type 2:

```
    0 0 X 1 1
    0 0 X Y 1 1
    0 0 X Y Z 1 1
    0 0 X X X 1 1
    0 0 X X Y Z Z 1 1
```

Examples of knot vectors of Type 3:

```
    X X Y Z Z
    X X Y Z W W
```

With this information in hand, the reader is able to reconstruct complete knot vectors in the following manner. When reconstructing the knot vector, you only take just enough values from the decoded knot value array. This may be as few as one. All the other values are inferred. Here's a sketch of the reconstruction algorithm:

```
// Number of knots in the knot vector
cNumKnots = numCtlPts + degree + 1;
// Necessary knot multiplicity at both ends of the knot vector
cClamping = degree + 1;
switch (knotType) {
    // Clamping is 0..1, internal knots occur in ADJACENT PAIRS
    // *EXCEPT* when there are no internal knots, in which case
    // Type = 1 instead.
    case 0: numVals = (cNumKnots - 2 * cClamping)/2;
    // Clamping is X1..X2, internal knots occur in ADJACENT PAIRS
    case 1: numVals = (cNumKnots - 2 * cClamping)/2 + 2;
    // Clamping is 0..1, and not Type 0
    case 2: numVals = (cNumKnots - 2 * cClamping);
    // Clamping is X1..X2, and not Type 1
    case 3: numVals = (cNumKnots - 2 * cClamping) + 2;
}
// numVals is the number of non-inferrable knot values needed
// Let vVals be the knot vector value array
// vKnot will be the final output knot vector
if (knotType is either 0 or 2)
    Set vKnot[0 .. cClamping-1] to 0
    Set vKnot[cNumKnots-cClamping .. cNumKnots-1] to 1
else
    Set vKnot[0 .. cClamping-1] to vVals[0]
    Set vKnot[cNumKnots-cClamping .. cNumKnots-1] to vVals[numVals-1]
Set vKnot[cClamping .. cNumKnots-cClamping-1] from vVals[1 .. numVals-2]
```

VecI32{Int32CDP, Lag1} : Entity Index Codes

Entity Index Codes is a vector of quantizer —codes” representing entity index identifiers for a set of entities (i.e. indices to particular entities within a list of entities). Entity Index Codes uses the Int32 version of the CODEC to compress and encode data.

13.1.14 Compressed Control Point Weights Data

Compressed Control Point Weights Data collection is the compressed and/or encoded representation of weight data for some set of Control Points. All NURBS based geometry use this data collection to compress/encode Control Point Weight data.

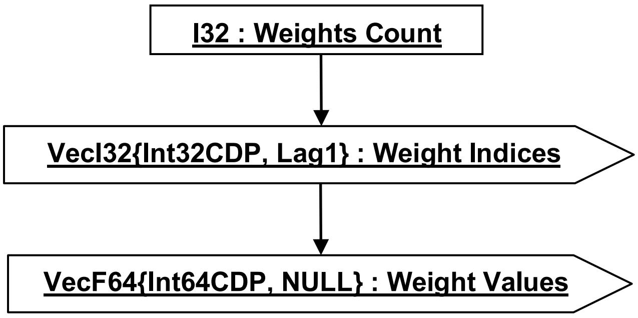

I32 : Weights Count
VecI32{Int32CDP, Lag1} : Weight Indices
VecF64{Int64CDP, NULL} : Weight Values

Figure 250 — Compressed Control Point Weights Data collection

I32 : Weights Count

Weights Count specifies the total number of Weights. This count can differ from the Control Point count (see 8.1.4.1.3 NURBS Surface Control Point Counts) because if the Control Point Dimensionality is non-rational (see data field NURBS Surface Control Point Dimensionality in Surfaces Geometric Data), then no Weight values are stored for the particular Control Point. Weights Count value also does not necessarily equate to the actual number of Weights stored, since if a particular Control Point‘s Weight values is —”1 then no actual Weight value is stored (i.e. JT file loaders/readers can infer that the Weight Value is —”1for Control Points that don‘t have a Weight value stored).

VecI32{Int32CDP, Lag1} : Weight Indices

Weight Indices is a vector of indices representing the index identifiers for the conditional set of weights for which an actual Weight Values is stored in Weight Values. Weight Indices uses the Int32 version of the CODEC to compress and encode data.

VecF64{Int64CDP, NULL} : Weight Values

Weight Values is a vector of weight values for the conditional set of weights. Weight Values uses the Int64 version of the CODEC to compress and encode data. Each deserialized 64 bit integer number should be converted to bit wise equivalent 64 bit floating number.

13.1.15 Compressed Curve Data

Compressed Curve Data collection contains JT B-Rep or Wireframe Rep compressed/encoded geometric Curve data. Currently only NURBS Curve types are supported as part of this data collection. Complete documentation for JT B-Rep and Wireframe Rep can be found in sections 8.1 JT B-Rep Element and 10.1 Wireframe Rep Element respectively.

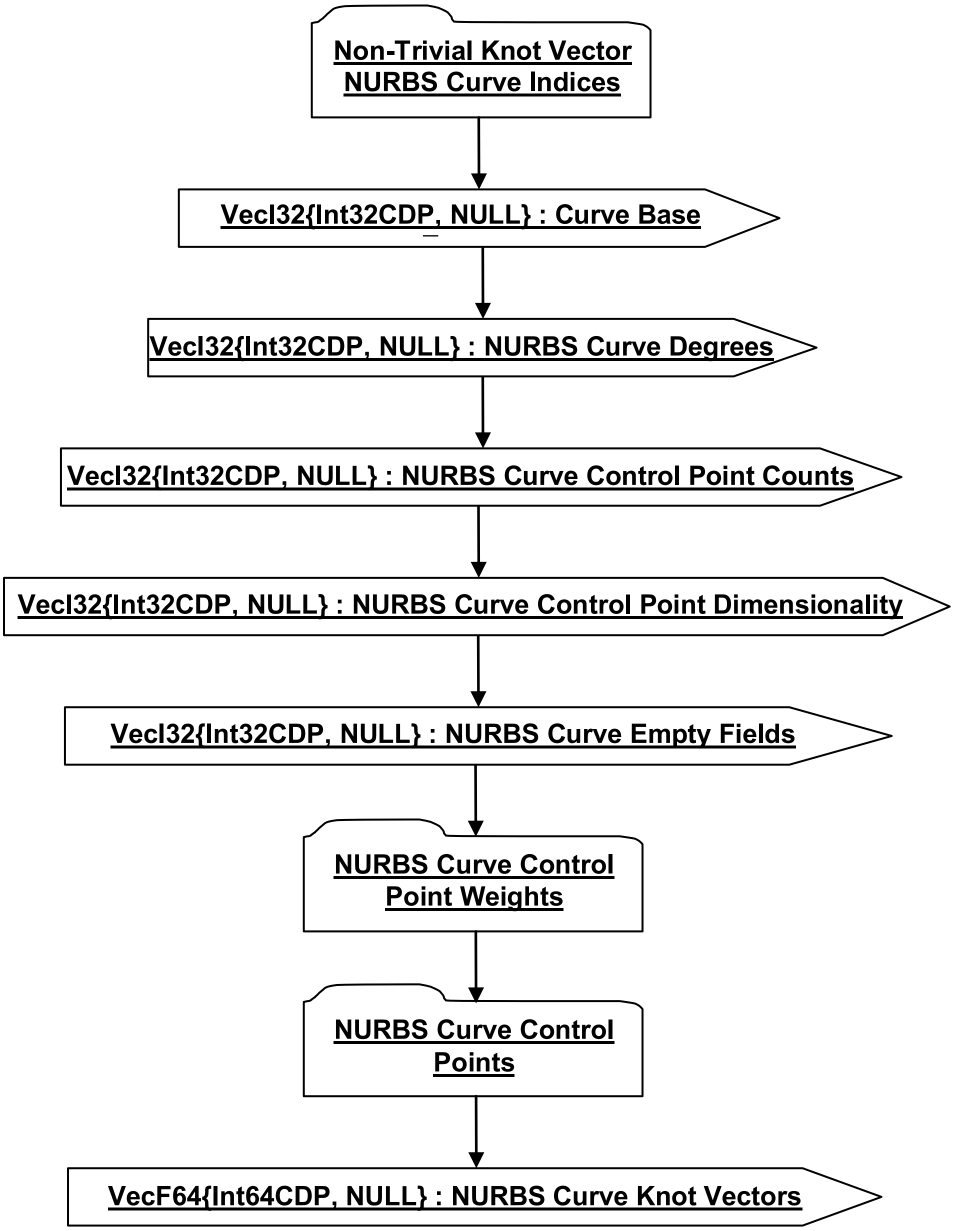

Non-Trivial Knot Vector NURBS Curve Indices
VecI32{Int32CDP, NULL} : Curve Base
VecI32{Int32CDP, NULL} : NURBS Curve Degrees
VecI32{Int32CDP, NULL} : NURBS Curve Control Point Counts
VecI32{Int32CDP, NULL} : NURBS Curve Control Point Dimensionality
VecI32{Int32CDP, NULL} : NURBS Curve Empty Fields
NURBS Curve Control Point Weights
NURBS Curve Control Points
VecF64{Int64CDP, NULL} : NURBS Curve Knot Vectors

Figure 251 — Compressed Curve Data collection

VecI32{Int32CDP, NULL} : Curve Base Types

Each Curve is assigned a base type identifier. Curve Base Types is a vector of base type identifiers for each Curve in a list of Curves. Currently only NURBS Curve Base Type is supported, but a type identifier is still included in the specification to allow for future expansion of the JT Format to support other curve types.

In an uncompressed/decoded form the Curves base type identifier values have the following meaning:

Table 98 — Compressed Curve Base Type values

= 1	Curve is a NURBS curve

Curve Base Types uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : NURBS Curve Degrees

NURBS Curve Degrees is a vector of Curve degree values for each NURBS Curve in a list of Curves (there is a stored value for each NURBS Curve in the list). NURBS Curve Degrees uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : NURBS Curve Control Point Counts

NURBS Curve Control Point Counts is a vector of control point counts for each NURBS Curve in a list of curves (there is a stored value for each NURBS Curve in the list). NURBS Curve Control Point Counts uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : NURBS Curve Control Point Dimensionality

NURBS Curve Control Point Dimensionality is a vector of control point dimensionality values for each NURBS Curve in a list of Curve s(i.e. there is a stored values for each NURBS Curve in the list).

In an uncompressed/decoded form the control point dimensionality values meaning is dependent upon the NURBS Entity type.

For NURBS UV Curve entities the dimensionality value has the following definition:

Table 99 — NURB UV Curve entity dimensionality values

= 2	Non-Rational (each control point has 2 coordinates)
= 3	Rational (each control point has 3 coordinates)

For NURBS XYZ Curve entities the dimensionality value has the following definition:

Table 100 — NURB XYZ Curve entity dimensionality values

= 3	Non-Rational (each control point has 3 coordinates)
= 4	Rational (each control point has 4 coordinates)

NURBS Curve Control Point Dimensionality uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, NULL} : NURBS Curve Empty Fields

NURBS Curve Empty Fields is a vector of data. Each NURBS Curve in a list of Curves has one reserved data field entry in this NURBS Curve Empty Fields vector. NURBS Curve Empty Fields uses the Int32 version of the CODEC to compress and encode data. Refer to best practice 14.4 Empty Field

VecF64{Int64CDP, NULL} : NURBS Curve Knot Vectors

NURBS Curve Knot Vectors is a list of knot vector values for each NURBS Curve having non-trivial knot vectors in a list of Curves (i.e. there are stored values for each non-trivial knot vector NURBS Curve in the list). All these NURBS Curve non-trivial knot vectors are accumulated into this single list in the same order as the Curve appears in the Curve list (i.e. Curve-N Non-Trivial Knot Vector, Curve-M Non-Trivial Knot Vector, etc.). The NURBS Curves for which knot vectors are stored (i.e. those containing non-trivial knot vectors) are identified in data collection Non-Trivial Knot Vector NURBS Curve Indices documented in Non-Trivial Knot Vector NURBS Curve Indices. NURBS Curve Knot Vectors uses the Int64 version of the CODEC to compress and encode data. Each deserialized 64 bit integer number should be converted to bit wise equivalent 64 bit floating number.

Non-Trivial Knot Vector NURBS Curve Indices

Non-Trivial Knot Vector NURBS Curve Indices data collection specifies the Curve index identifiers (i.e. indices to particular NURBS Curves within a list of Curves) for all NURBS Curves containing non-trivial knot vectors.

A description/definition for —non-trivial knot vector” can be found in 13.1.13 Compressed Entity List for Non-Trivial Knot Vector.

This Curve index data is stored in a compressed format.

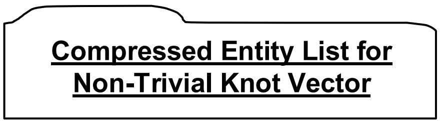


Figure 252 — Non-Trivial Knot Vector NURBS Curve Indices data collection

Complete description for Compressed Entity List for Non-Trivial Knot Vector can be found in 13.1.13 Compressed Entity List for Non-Trivial Knot Vector.

NURBS Curve Control Point Weights

NURBS Curve Control Point Weights data collection defines the Weight values for a conditional set of Control Points for a list of NURBS Curves. The storing of the Weight value for a particular Control Point is conditional, because if NURBS Curve Control Point Dimension is -non-rational” or the actual Control Point‘s Weight value is —1” then no Weight value is stored for the Control Point (i.e. Weight value can be inferred to be —1”).

The NURBS Curve Control Point Weights data is stored in a compressed format.

Figure 253 — NURBS Curve Control Point Weights data collection

Complete description for Compressed Control Point Weights Data can be found in 13.1.14 Compressed Control Point Weights Data.

NURBS Curve Control Points

NURBS Curve Control Points is the compressed and/or encoded representation of the Control Point coordinates for each NURBS Curve in a list of Curves (i.e. there are stored values for each NURBS Curve in the list). Note that these are non-homogeneous coordinates (i.e. Control Point coordinates have been divided by the corresponding Control Point Weight values).

Figure 254 — NURBS Curve Control Points data collection

VecF64{Int64CDP, NULL} : Control Points

Control Points is a vector of Control Point coordinates for all the NURBS Curves in a list of Curves. All the NURBS Curve Control Point coordinates are accumulated into this single vector in the same order as the Curve appears in the Curve list (i.e. Curve-1 Control Points, Curve-2 Control Points, etc.). Control Points uses the Int64 version of the CODEC to compress and encode data in a —lossless” manner. Each deserialized 64 bit integer number should be converted to bit wise equivalent 64 bit floating number.

13.1.16 Compressed CAD Tag Data

The Compressed CAD Tag Data collection contains the persistent IDs, as defined in the CAD System, to uniquely identify individual CAD entities (e.g. Faces and Edges of a JT B-Rep, PMI, etc.). Exactly what CAD entity types have CAD Tags and what order they are stored in Compressed CAD Tag Data is defined by users of this data collection (e.g. 8.1.6 B-Rep CAD Tag Data, **Error! Reference source not found. Error! Reference source not found.**)

What constitutes a CAD Tag is outside the scope of the JT File format and is indeed part of the CAD system. The JT File format simply provides a way to store any kind of CAD Tag as provided by the CAD system which produced the CAD entity.

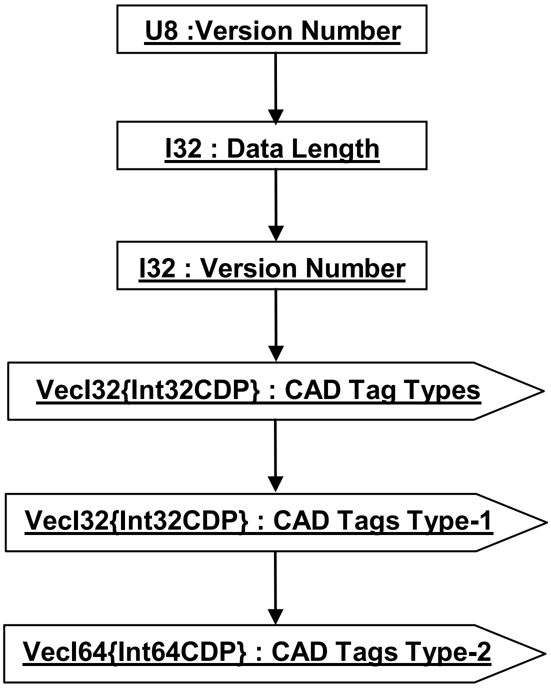

U8 :Version Number
I32 : Data Length
I32 : Version Number
VecI32{Int32CDP} : CAD Tag Types
VecI32{Int32CDP} : CAD Tags Type-1
VecI64{Int64CDP} : CAD Tags Type-2

Figure 255 — Compressed CAD Tag Data collection

U8 :Version Number

Version Number is the version identifier for the CADTag element. For informaiton on local version numbers see best practice 14.5 Local version numbers

I32 : Data Length

Data Length specifies the length in bytes of the Compressed CAD Tag Data collection. A JT file loader/reader may use this information to compute the end position of the Compressed CAD Tag Data within the JT file and thus skip reading the remaining Compressed CAD Tag Data.

I32 : Version Number

Version Number is the local version identifier for the Compressed CAD Tag Data. For informaiton on local version numbers see best practice 14.5 Local version numbers

VecI32{Int32CDP} : CAD Tag Types

CAD Tag Types is a vector of type identifiers for a list of CAD Tags (where each CAD Tag in the list has a type identifier value).

In an uncompressed/decoded form the CAD Tag type identifier values have the following meaning:

Table 101 — Compressed CAD Tag Type values

= 1	32 Bit Integer CAD Tag Type
= 2	64 Bit Integer CAD Tag Type

CAD Tag Types uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP} : CAD Tags Type-1

CAD Tags Type-1 is a vector of the Type-1 (i.e. 32 Bit Integer Type) CAD Tags for a list of CAD Tags. CAD Tags Type-1 uses the Int32 version of the CODEC to compress and encode data. CAD Tags Type-1 is only present if there are Type-1 CAD Tags in the CAD Tag Types vector.

VecI64{Int64CDP} : CAD Tags Type-2

CAD Tags Type-2 is a vector of the Type-2 (i.e. 64 Bit integer Type) CAD Tag data for a list of CAD Tags. CAD Tags Type-2 uses the Int64 version of the CODEC to compress and encode data. CAD Tags Type-2 is only present if there are Type-2 CAD Tags in the CAD Tag Types vector.

13.2 Encoding Algorithms

The following sections give a brief technical overview/descriptions of the various encoding algorithms used in the JT format. A sample implementation of the encoding and decoding portion of each algorithm can be found in Annex B.

13.2.1 Uniform Data Quantization

Uniform Data Quantization is a lossy encodingtechnique in which a continuous set of input values (floating point data) is approximated with integral multiples (i.e. integers) of a common factor. How close the quantization output approximates the original input data is dependent upon the quantization data range and the number of bits specified to hold the resulting integer value.

The quantization is considered ―uniform” because the algorithm divides the data input range into levels of equal spacing (i.e. a uniform scale). The form of Uniform Data Quantization used by the JT format is also considered scalar in nature, in that each input value is treated separately in producing the output integer value.

Given the following definitions:

```
inputVal:         Input floating point data to quantize
outputval:        Resulting quantized output integer value
minInputRange:    Specified minimum value of input data range
maxInputRange:    Specified maximum value of input data  range
nBits:        Specified number of bits of precision (quantized size)
```

The basic algorithm (using C++ style syntax) for Uniform Data Quantization is as follows:

```
UInt32 iMaxCode = (nBits < 32) ? (0x1 << nBits) - 1 : 0xffffffff;
Float64 encodeMultiplier = Float64(iMaxCode) / (maxInputRange – minInputRange);
UInt32 outputVal = UInt32( (inputVal - minInputRange) * encodeMultiplier + 0.5 );
```

Note: For reasons of robustness, ―outputVal” shall also be explicitly clamped to the range [0,iMaxCode]. This is because floating-point roundoff error in the calculation of ―encodeMultiplier” can otherwise cause ―outputVal” to sometimes come out equal to ―iMaxCode + 1”.

Note that all compression algorithms in the following sections operate on quantized integer data.

13.2.2 Bitlength CODEC

This is a very simple compression algorithm that runs either a fixed-width or adaptive-width bit field encoding for each value. It is used whenever none of the more sophisticated CODECs are able to extract any compression advantage. In essence, the Bitlength CODEC takes advantage of the fact that most of the values will require less than 32 bits to represent, and so can be written as bitfields narrower than 32 bits. In some cases, the best answer is to choose a fixed field width that can represent all values in the array. In other cases, a little more compression can be had by using an adjustable-width coding scheme.

When using the variable-width scheme, as each input value is encountered, the number of bits needed to represent it is calculated and compared to the current "field width". The current field width is then adjusted upwards or downwards by a constant ―step_size” number of bits (i.e. 2 bits for the JT format) to accommodate the input value storage. This increment or decrement of the current field width is indicated for each encoded value by a prefix code stored with each value.

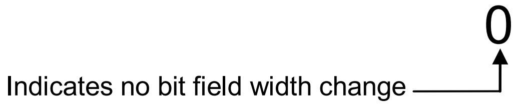
The prefix code will be one of the following two forms:

A single '0' bit to denote the same (i.e. current) field width is to be used for the next value.

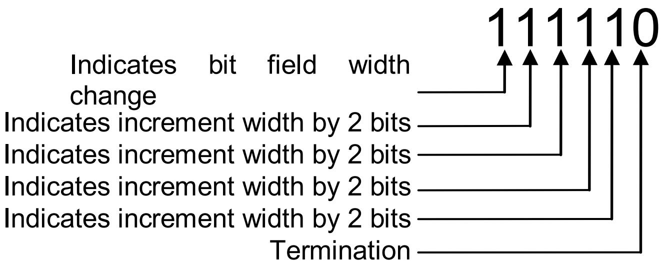
A '1' bit followed by a series of one or more bits where each bit indicates whether the field width is to be incremented (a '1' bit) or decremented (a '0' bit) by the field step_size, followed by a single terminator bit (which is complement of the previous increment/decrement bit). Note that there can only be increments or decrements in a given prefix code, never both, and that is why the prefix code terminator bit can be recognized as bits are read by simply looking for the complement of the previous increment/decrement bit.

Some examples of prefix codes and their interpretation are as follows:

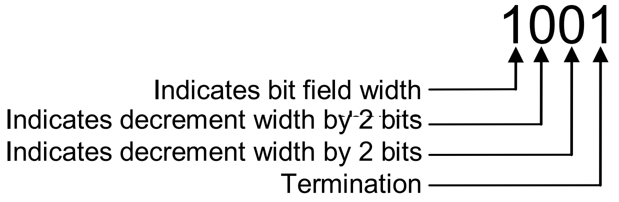
Example 1: Prefix code to maintain same (current) field width.



Example 2: Prefix code to increment field width four times (8 bits).


111110
Indicates bit field width change
Indicates increment width by 2 bits
Indicates increment width by 2 bits
Indicates increment width by 2 bits
Indicates increment width by 2 bits
Termination

Example 3: Prefix code to decrement field width two times.


1001
Indicates bit field width
Indicates decrement width by 2 bits
Indicates decrement width by 2 bits
Termination

A pseudo-code sample implementation of bit length decoding is available in Annex B.

13.2.3 Arithmetic CODEC

In 1948, Claude Shannon of Bell Laboratories published his seminal paper —Amathematical theory of communication” that launched the new field of Information Theory. In that same year, two Doctoral students at the Massachusetts Institute of Technology (MIT) made breakthroughs in the coding of information. The first to press was David Huffman, whose coding scheme we now know as Huffman Coding. In that same class with Huffman was Peter Elias who reportedly developed the first articulation of arithmetic coding, but it lay unpublished until 1976, when Jorma Rissanen and Richard Pasco, of IBM, refined it into a practically useful algorithm.

Arithmetic encoding is a so-called —entropy coding” algorithm that replaces an input stream of symbols or bytes with a single fixed point output number (i.e. only the mantissa bits to the right of the binary point are output from MSB to LSB). The total number of bits needed in the output number is dependent upon the length and statistical properties of the input message (i.e. the longer the input message the more bits needed in the output number). This single fixed point number output from an arithmetic encoding process shall be uniquely decodable to create the exact stream of input symbols that were used to create it.

Initially all symbols being encoded have a probability value assigned to them based on the likelihood that the symbol will occur next in the input stream (i.e. the frequency of the symbol in the input stream). Given probability value assignments, each individual symbol is then assigned an interval range along a nominal 0 to 1 —probability line”, where the size of each range corresponds to the symbol‘s probability value. Note that a particular symbol owns all values within its assigned range up to, but not including, the range high value, and that it does not matter which symbols are assigned which segment of the range as long it is done in the same manner by both the encoder and the decoder.

Given the above described input stream probability and interval range assignments, a high level description of the arithmetic encoding process is as follows:

Begin with a —current interval” initialized to [0,1). Note, that in interval range notation (i.e. —[0,1)”), the —[symbol indicates inclusive of the interval low limit and “)” symbol indicates exclusive of the interval high limit.

Sequentially for each symbol of the input stream, perform two steps

Subdivide the current interval into subintervals based on the input stream symbol probability values as described above.

Select the subinterval corresponding to the current input stream symbol being sequentially processed and make it the new —current interval”.

After all input stream symbols have been sequentially processed; output enough bits to distinguish the final —current interval” from all other possible final intervals.

In pseudo code form, the algorithm to accomplish the above described arithmetic encoding for an input stream message of any length could look as follows:

```
Set low to 0.0
Set high to 1.0
While there are still input symbols do
    cur_symbol = get next input symbol
    range = high – low
    high = low + range * high_range(cur_symbol)
    low = low + range * low_range(cur_symbol)
End of While
Output low
```

So the arithmetic encoding process is simply one in which we narrow the range of possible numbers with every new sequentially processed input symbol; where the new narrowed range is proportional to the predefined probability values assigned to each symbol in the input stream.

The arithmetic decoding process is the inverse procedure; where the range is expanded in proportion to the probability of each symbol as it is extracted. For the arithmetic decoding process we find the first symbol in the message by seeing which symbol owns the interval range that our encoded message falls in. Then, since we know the low and high range limit values of the first symbol we can remove their effects by reversing the process that put them in.

In pseudo code form, the algorithm for decoding the incoming number could look as follows:

```
Get encoded_number
Do
    find symbol whose range straddles the encoded_number
    output the symbol
    range = symbol_high_value – symbol_low_value
    encoded_number = encoded_number - symbol_low_value
    encoded_number = encoded_number / range
until no more symbols
```

Example

Following is an example to demonstrate in practice the basic principles of arithmetic coding.

Suppose you want to compress, using arithmetic coding, the following sequence/array of integer data:

{2, 9, 12, 12, 0, 7, 1, 20, 5, 19}

For this input stream of data, the assigned probability values will be as follows:

Table 102 — Example assigned probability values

Number	Probability
0	1/10
1	1/10
2	1/10
5	1/10
7	1/10
9	1/10
12	2/10
19	1/10
20	1/10

Then based on each input numbers probability value, an interval range along a 0 to 1 —probability line” can be assigned to each input number as follows:

Table 103 — Example —probability line” values

Number	Probability	Range
0	1/10	[0.00, 0.10)
1	1/10	[0.10, 0.20)
2	1/10	[0.20, 0.30)
5	1/10	[0.30, 0.40)
7	1/10	[0.40, 0.50)
9	1/10	[0.50, 0.60)
12	2/10	[0.60, 0.80)
19	1/10	[0.80, 0.90)
20	1/10	[0.90, 1.00)

Now proceeding with encoding the example input integer sequence {2, 9, 12, 12, 0, 7, 1, 20, 5, 19}, the first number to be encoded is —2”, so the final encoded value will be a number that is greater than or equal to 0.20 and less than 0.30. Now as each subsequent number in the input stream is sequentially processed for encoding, the possible range of the output number is further restricted. In our example the next number to be encoded is —9” which owns the range [0.50, 0.60) within the new sub-range of [0.20, 0.30); which now further restricts our output number to the range [0.25, 0.26). If we continue this logic for the complete input integer sequence we end up with the following:

Table 104 — Example input integer sequence values

New integer number	Low value	High value
	0.0	1.0
2	0.2	0.3
9	0.25	0.26
12	0.256	0.258
12	0.2572	0.2576
0	0.25720	0.25724
7	0.257216	0.257220
1	0.2572164	0.2572168
20	0.25721676	0.2572168
5	0.257216772	0.257216776
19	**0.2572167752**	0.2572167756

From the above table, are final low values is —0.2572167752” which is the output number that uniquely encodes the integer number sequence {2, 9, 12, 12, 0, 7, 1, 20, 5, 19}.

Given this encoding scheme, the decoding would simply follow the process previously described. We find the first number in the sequence by looking up in the probability range for the value, whose range, our encoded number —0.2572167752” falls within. In our example this equates to the value —2” and so our first decoded value shall be —2”. Then we apply the previously described decoding subtraction and division steps to arrive at a new encoded value of —0.572167752”. Using this new —0.572167752” encoded value and the same logic of the first step, the second decoded value will be —9”. We continue this process until there are no more numbers to decode.

In practice, due to floating point size (i.e. number of bits) restrictions and possible differences in floating point formats on machines, arithmetic encoding is best implemented using 16 bit or 32 bit integer math. Using 16 bit or 32 bit integer math, an incremental transmission scheme can be implemented, where fixed size integer state variables receive new bits in at the low end and shift them out the high end, forming a single number that can be as many bits long as are available on the computer‘s storage medium.

Using our example as a guide, define the starting range [0.0, 1.0) to instead be 0 to 0.999 (which is .111 in binary). Then in order to use integer registers to store these numbers, justify the values so that the implied decimal point is at the left hand side of the word. Now load the initial range values based on the word size we are using. In the case of a 16 bit implementation the initial range values will be low equals 0x0000 and high equals 0xFFFF. Since we know these values will go on forever (e.g. 0.999… will continue with FFs) we can shift those extra bits in as needed with no detrimental effects.

Going back to our example and using a 5 digit register, we start with the range:

High: 99999

Low: 00000

Applying the previously described encoding algorithm we first calculate the range between the low and high values; which in this case is 100000 (not 9999 since we assume the high value has an infinite number of 9‘s). Next, we calculate the new high value which in this example will be 30000. But before we store the new high value we shall decrement it to account for the implied digits appended to it; so new high value will be 29999. Applying similar logic to computing the new low value results in a new range of:

High: 29999 (999…)

Low: 20000 (000…)

In looking at the newly computed high and low range values, it can be seen that the most significant digits of high and low match. A property of arithmetic coding is that as this encoding process continues, the high and low values will continue to get closer, but will never match exactly. Given this property, once the most significant digit of high and low match, it will never change, and thus we can output this most significant digit as the first number in the coded word and continue working with just 16 bit high and low values. This output process is accomplished by shifting both the high and low values left by one digit and shifting in a —9” in the least significant digit of the high value.

Applying the previously described encoding algorithm and continuing the above described process of shifting out most significant digit into the coded word as high and low continually grow closer together looks as follows for encoding our example integer number sequence {2, 9, 12, 12, 0, 7, 1, 20, 5, 19}:

Table 105 — Example integer number sequence values

	High	Low	Range	Cumulative output
Initial State	99999	00000	100000	
Encode —2” [0.2, 0.3)	29999	20000		
Shift out 2	99999	00000	100000	.2
Encode —9” [0.5, 0.6)	59999	50000		.2
Shift out 5	99999	00000	100000	.25

	High	Low	Range	Cumulative output
Encode —2” [0.6, 0.8)	79999	60000	20000	.25
Encode —2” [0.6, 0.8)	75999	72000		.25
Shift out 7	59999	20000	40000	.257
Encode —0” [0.0, 0.1)	23999	20000		.257
Shift out 2	39999	00000	40000	.2572
Encode —7” [0.4, 0.5)	19999	16000		.2572
Shift out 1	99999	60000	40000	.25721
Encode —1” [0.1, 0.2)	67999	64000		.25721
Shift out 6	79999	40000	40000	.257216
Encode —0” [0.9, 1.0)	79999	76000		.257216
Shift out 7	99999	60000	40000	.2572167
Encode —5” [0.3, 0.4)	75999	72000		.2572167
Shift out 7	59999	20000	40000	.25721677
Encode —9” [0.8, 0.9)	55999	52000		.25721677
Shift out 5	59999	20000	40000	.257216775
Shift out 2				.2572167752
Shift out 0				.25721677520

As can be seen in the above table, after all values in the input stream have been encoded and any final matching most significant digit has been output, the arithmetic coding algorithm requires that two extra digits be shifted out of either the high or low value to finish up the cumulative output word.

Although the above example incrementally encodes very nicely with the arithmetic coding algorithm, there are certain cases where the computed high and low values get closer, but never actually converge to one value in the most significant digit (e.g. High = 0.300001, Low = 0.29992). Thus after a few iterations the difference between high and low becomes so small that 16 bits is not sufficient to represent any difference between the values (i.e. all calculations return the same values). This conditions is known as —underflow” and special logic shall added to the arithmetic coding algorithm to recognize that —underflow” is occurring and thus head it off before the computations reach an impasse.

The additional logic for recognizing that —underflow” is occurring would be executed after each recalculation of High and Low value set, and in pseudo code form this logic would look as follows:

```
underflow = FALSE
if(  (High and Low value's significant digits don't match but are on adjacent numbers) &&
 (2nd MSDMSD of High is "0" and the 2nd MSDMSD of low is "9")  )
{
   underflow = TRUE
}
```

When/If it is identified that —underflow” is occurring, the encoding algorithm shall perform the following steps to stop the current —underflow”:

Delete the 2nd most significant digit from both the High and Low value.

Shift the other digits (those to the right of the deleted 2nd digit) to the left to fill up the space (note that the most significant digit stays in place).

Increment a counter to remember that we threw away a digit and don‘t know whether it was going to converge to —0”or —9”

A before and after example of performing the above steps to the High and Low values when ‗underflow” occurs is as follows:

	Before	After
High	40344	43449
Low	39810	38100
Underflow_counter	0	1

Now as the encoding algorithm continues and the most significant digit of High and Low values once again converge to a common value, then that value shall be output to the coded word along with —underflow_counter” number of —underflow” digits that were previously deleted. The underflow digits output to the coded word will either be all 9s or 0s, depending on whether the High and Low value converged to the higher or lower value.

A pseudo-code sample implementation of arithmetic decoding is available in Annex C.

13.2.4 Deering Normal CODEC

Michael Deering first published his work on geometry compression in 1995 [2] and later helped present a course on the subject at SIGGRAPH‘99 [3]. Although Deering‘s approach to geometric compression involves compression of vertices, colours and normals, the description detailed here will focus solely on compression of normals since this is the only component of Deering‘s approach used in the JT format.

Through both theoretical examination and empirical testing, Deering found that an angular density of 0.01 radians between normals (about 100,000 normalized normals distributed over unit sphere) gave results that were not visually distinguishable from results obtained from finer normal representations. This observation reduced the problem of having to —exactly” represent any general surface normal, to only having to represent about 100,000 specific normals (i.e. general surface normal replaced by the appropriate one of the 100,000 specific normals).

If there were no run-time memory concerns and no concerns for on disk footprint size, these specific 100,000 normals could be simply represented in a table that is indexed into, to reference a particular normal. Instead, Deering‘s approach leverages symmetrical properties of the unit sphere to reduce the size of the table and allow any normal to be represented by, at max, an 18 bit index as summarized below:

- All normals are normalized (i.e. can be represented as points on the surface of the unit sphere).

- Unit sphere is divided into eight symmetrical octants based on sign bits of normal‘s X,Y,Z rectilinear representation (see Figure 256). Using three bits to represent the three sign bits of the normals XYZ components reduces the problem space to one eighth of the unit sphere.

- Each octant of the unit sphere is divided into six identical sextants by folding about the planes of symmetry; x=y, x=z, and y=z (see Figure 256). The particular sextant can be encoded using another three bits. So now unit sphere is divided into 48 identically shaped triangle patches reducing the normal look-up table to about 2000 entries (i.e. 100000/48).

- Then, a local rectangular orthogonal two dimensional grid is created on the sextant and all normals within the sextant are represented as two n-bit angular addresses (i.e. a quantization of two angular values along the unit sphere) where —n” is in the range from 0 to 6 bits.

- Resulting in a max grand total of 18 bits (3 + 3 + 6 + 6) to represent any normal on the unit sphere.

In the figure below, the sphere is divided into eight octants and each octant is divided into six sextants. Each sextant is assigned an identifying three bit code.

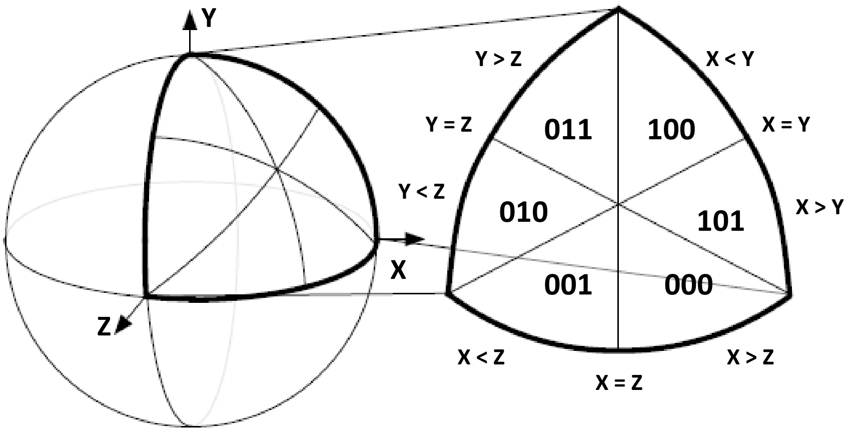


Figure 256 — Sextant Coding on the Sphere

Note that the sextant three bit code assignments used by the JT format (as seen in Figure 256) are slightly modified from the original assignments as specified by Deering.

The representation of all normals within a sextant by two n-bit angular addresses, as summarized above, is based on the following:

—In spherical coordinates, points on a unit sphere can be parameterized by two angles, θ and φ; where θ is the angle about the y axis and φ is the longitudinal angle from the y=0 plane.

—Mapping between rectangular and spherical coordinates is:

$$x = \cos\theta * \cos\varphi \qquad y = \sin\varphi \qquad z = \sin\theta * \cos\varphi.$$

—All encoding takes place in the positive octant.

—Angles θ and φ can be quantized into two n-bit integers θ'_n and φ'_n (where —”his in the range of 0 to 6) and the relationship between these n-bit integers and angles θ and φ for a given —”his:

$$\theta\,(\theta'_n) = \text{asin}\tan\,(\varphi_{max} * (n - \theta'_n) / 2n\,)$$
$$\varphi\,(\varphi'_n) = \varphi_{max} * \varphi'_n / 2n.$$

Thus to encode (i.e. quantize) a given normal **N** into θ'_n and φ'_n:

—N shall be first represented (see Figure 256) in the positive octant and appropriate sextant within that octant, resulting in N‘.

—Then N‘ shall be dotted with all quantized normals in the sextant.

—For a fixed —”n, the corresponding θ‘n and φ‘n values of the quantized sextant normal that result in the largest (nearest unity) dot product defines the proper θ‘n and φ‘n encoding of N.

With this encoding of normal **N** into θ'_n and φ'_n n-bit integers the complete bit representation of normal **N** can now be defined as follows:

—Uppermost three bits specify the octant.

—Next three bits specify the sextant code as defined in Figure 256.

—Next two n-bit fields specify θ‘n and φ‘n values respectively.

13.3 LZMA compression

LZMA is a (lossless) dictionary-based data compression algorithm and is essentially the same as that in 7-Zip.

14 Best Practices

The proceeding sections of this International Standard specify the mandatory clauses for creating a reference compliant Version 10.0 JT file. This ―Best Practices” section focusing on documenting format conventions that although not required to have a reference compliant JT file, have become commonplace within JT format translators to the point where these conventions are considered *best practices* for constructing JT files.

14.1 Late-Loading Data

The JT format was designed and structured to load entities from a JT file on a deferred or as-needed basis.. This concept is referred to within this International Standard as ―late-loading data”. The JT format has many structures in support of this and it is recommended as a best practice that writers/loaders of JT data leverage these capabilities.

Initial loading only requires the Table of Contents and the LSG. All Meta Data Node Elements, JT B-Rep Elements, XT B-Rep Elements, Wireframe Rep Elements, PMI Manager Meta Data Elements, JT ULP Elements, JT LWPA Elements, and Shape LOD Element

s may be ignored until they are actually needed. These Late-Loaded data containers are accessed via a Late Loaded Property Atom Element which appears in a LSG Node's Property list. Contained in this Property is the GUID associated with the segment to be loaded. This GUID can be looked up in the TOC Segment, which will give the location in the JT from which to load the actual Element via the Data Segment convention.

14.2 TOC Segment Location

Although there are no JT format compliance rules about where the TOC Segment shall be located within the file, it is best placed immediately following the file header.

14.3 Bit Fields

All bits fields that are not defined as in use should be set to ―0”

14.4 Empty Field

In the 5 File Format section of this reference some data fields may be named/documented ―Empty Field” (e.g. 6.1.1.7.1LOD Node Data ”Empty Field” field). Best practice suggests that these fields should be treated as follows:

If you are writing a JT file whose data did not originate from reading a previous JT file, then Empty Fields should be set to a value a ―0” when writing the field to a JT file.

If you are writing a JT file whose data originated from reading a previous JT file (i.e. rewriting a JT File), then ―Empty Fields” should be written with the same value that was read from the originating JT file.

14.5 Local version numbers

The version numbers seen throughout the data collections are version numbers local to those data types. They provide a simple means by which those data collections can be extended. All version information for 10.0 JT data is included within this document.

For each data collection, data for each local version should be written in sequence. When reading the data, the local version number allows readers to read up to the maximum local version they support and then use the segment length that was read in the Segment Header to skip over additional data. For example when version one and two data is present the user can choose to read only the version one data or additionally read the version two data as required.

Local version numbers are used for conditional branching as depicted in the element figures.

14.5.1 Version numbers

The only local version number value defined at the publication of this standard is —x001”. All other values for the local version number are reserved for future use.

14.6 Hash Value

Hashing is a means by which a large chunk of values can be represented by single value through the use of a mathematical function that provides a distinctive value for each unique set of ordered values. The hash function used within the v10.x format was published by Bob Jenkins in Dr Dobbs back in 1997 and its implementation is provided in Annex C. It is the same implementation that was used in JT v9.x.

The hash function takes a pointer to a set of values, the number of values, and a seed hash value. It returns the resulting hash value. Initially the seed value is set to 0, however when hashing multiple data fields together the hash of previous data field is used as the seed hash value of the next data field:

```
UInt32 uHash = 0;
uHash = hash32( pVal0, nVal0, uHash );
uHash = hash32( pVal1, nVal1, uHash );
```

The order that individual fields are hashed is extremely important since v10.x readers are strongly encouraged to assert that the stored hash value matches the calculated hash value of the corresponding fields after reading in all the corresponding data. To this end each hash value stored within the v10.x format carefully documents which fields it encompasses and the order in which they should be hashed.

14.7 Scene graph construction

The following guidlines apply for scenegraph construction;

1. use a 6.1.1.6 Meta Data Node Element to denote a CAD —Assembly”,
2. use a 6.1.1.5 Part Node Element to denote a CAD —part”.

Below is an example of a fully-fleshed out small assembly of a three-wheeled motorcycle:

```
Partition          ( 6.1.1.2 Partition Node Element )         Root node of JT file
 MetaDataNode   ( 6.1.1.6 Meta Data Node Element )         —Three-wheeler” Top-level assembly
  Instance          ( 6.1.1.4 Instance Node Element )          —Front wheel” assembly
   Partition         ( 6.1.1.2 Partition Node Element )        External reference to JT file for —Wheel” part
    PartNode     ( 6.1.1.5 Part Node Element )                 —Wheel” generic part
     RangeLOD   ( 6.1.1.8 Range LOD Node Element )        Level-of-detail node
            Group        ( 6.1.1.3 Group Node Element )               High-LOD group node
             TriStripSet  ( 7.1.3 Tri-Strip Set Shape LOD Element ) Rim geometry
             TriStripSet  ( 7.1.3 Tri-Strip Set Shape LOD Element ) Tire geometry
             TriStripSet  ( 7.1.3 Tri-Strip Set Shape LOD Element ) Low-LOD geometry
        MetaDataNode   ( 6.1.1.6 Meta Data Node Element )         —Rear Axle” assembly
         Instance          ( 6.1.1.4 Instance Node Element )          —Left rear wheel” assembly
          [—Wheel” PartNode above]       This instance node‘s child is same as the one above.
         Instance          ( 6.1.1.4 Instance Node Element )           —Right rear wheel” assembly
          [—Wheel” PartNode above]       This instance node‘s child is same as the one above.
```

6.1.1.4 Instance Node Element are used when referring to an *instanced* part but are not fundamentally different from a 6.1.1.3 Group Node Element having only one child.

14.8 Metadata Conventions

Although there are really no restrictions/limits/requirements on what metadata (i.e. properties) can/shall be attached to nodes in the LSG in order to have a reference compliant JT file, there are some conventions that have been generally followed in the industry when translating CAD data to the JT file format. See the Property Atom Elements section of this document for complete description of the file Elements used to attach this property information to nodes.

14.8.1 CAD Properties

The following table lists the conventions that CAD data translators typically (although not always) follow in placing CAD information in a JT file as properties on various LSG nodes. Some of these properties are considered required in order for the data in the file to be interpreted correctly while other properties are optional. See flowing sub-sections for additional information on required versus optional properties.

The convention is to place these Units properties on every Part and Assembly grouping node in the LSG. By following this convention, JT file format readers/writers are provided maximum flexibility in understanding/indicating the appropriate JT data unit processing for both, monolithic and shattered JT file Assembly structures.

Table 106 — CAD Property Conventions

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values	Required / Optional
JT_PROP_MEASUREMENT_UNITS	Model Units	MbString	MbString	millimeters centimeters meters inches feet yards micrometers decimeters kilometers mils miles	Required
CAD_MASS_UNITS	Units of mass	MbString	MbString	micrograms milligrams grams kilograms ounces pounds	Required
CAD_SURFACE_AREA	Surface area of solids within part.	MbString	F64	numeric	Optional
CAD_VOLUME	Volume of solids within part	MbString	F64	numeric	Optional
CAD_DENSITY	Density of solids within part (6)	MbString	F64	numeric	Optional
CAD_MASS	Mass or weight of solids within part	MbString	F64	numeric	Optional
CAD_CENTRE_OF_GRAVITY	Centre of gravity of solids within part	MbString	3 space separated F64	3 numeric values	Optional
CAD_PROP_MATERIAL_THICKNESS	Sheet thickness within part	MbString	F64	numeric	Optional
CAD_PART_NAME	Component name from translator	MbString	MbString	<string>	Optional
CAD_SOURCE	CAD program the Part originated from	MbString	MbString	<string>	Optional

14.8.1.1 Required Properties

The required unit properties are really necessary for viewers of JT file data to properly interpret numeric data for analysis operations (e.g. measure) and support the building of assemblies through the reading of multiple JT files in disparate units. There are two units of measure that are relevant, units of distance and units of weight.

The JT_PROP_MEASURMENT_UNITS property is used to specify units of distance. The CAD_MASS_UNITS property is used to specify units for weight. JT_PROP_MEASURMENT_UNITS property is strictly required, while CAD_MASS_UNITS property is "optionally required". By —optionally required", we mean, it is required if other optional metadata intends to specify properties that would depend on these units of measure (e.g. CAD_DENSITY and CAD_MASS). Notice that the Mass units are specified, instead of the Density units, since Density is a derived unit of Mass/Volume.

14.8.1.2 Optional Properties

Optional properties can be provided, but if the property is a units based value, then the value shall be in units that are consistent with the JT_PROP_MEASURMENT_UNITS and CAD_MASS_UNITS properties. Thus the units for the optional units based properties shall be as follows:

Table 107 — CAD Optional Property Units

Optional Property	Units
CAD_SURFACE_AREA	$(\text{JT\_PROP\_MEASUREMENT\_UNITS})^2$
CAD_VOLUME	$(\text{JT\_PROP\_MEASUREMENT\_UNITS})^3$
CAD_DENSITY	$\text{CAD\_MASS\_UNITS}/(\text{JT\_PROP\_MEASUREMENT\_UNITS})^3$
CAD_MASS	CAD_MASS_UNITS
CAD_CENTRE_OF_GRAVITY	JT_PROP_MEASUREMENT_UNITS
CAD_PROP_MATERIAL_THICKNESS	JT_PROP_MEASUREMENT_UNITS

Note of caution regarding the node placement for the CAD_DENSITY property. Following the recommended convention for the placing of CAD properties (see description in 14.8.1 CAD Properties) implies that all solids within a single JT part are of a uniform density, which may not be true in all cases.

14.8.2 PMI Properties

The table below lists properties that can be found in JT parts that contain Generic PMI Entities (see 11.2.8 Generic PMI Entities). The column titled Supported PMI_Type values defines the expected Generic PMI Entity Types the property will be found with (reference Enitity Type). The column Value Format / Expected Values defines the expected representation of the property data. The Property Key and Value Format / Expected Values data forms a key value pair for a standard JT property (see PMI Property Atom).

Table 123 – PMI Properties makes use of C programming string format specifiers in description of values supported by the Property Key. The following values are used
"%d" - Signed 32-bit integer
"%.16g" - 64-bit floating-point number with 16 digit precision
"%s" - Null-terminated array of 8-bit unsigned characters

Note: The recommended best practice for working with PMI in a JT file is to use the descriptions found in 11.2 PMI Manager Meta Data Element as opposed to working with PMI Data Segment.

Table 108 — PMI Properties

Property Key	Supported PMI_Type values	Value Format / Expected Values	Prerequisite Key	Additional Note
accountabilityId	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		

aftDistance	0x0242	"%.16g"	type	Expects type property to be closed
aftThroughAll	0x0242	"%d"	type	Expects type property to be closed
allAround	0x0041 0x0081 0x0008	"%d"		
allOver	0x0081	"%d"		
AltText[%d]	0x0100 0x0235 0x0244	NULL		
AltText[%d].bold	0x0100 0x0235 0x0244	"%d"	AltText[%d]	
AltText[%d].font	0x0100 0x0235 0x0244	"%s"	AltText[%d]	
AltText[%d].italic	0x0100 0x0235 0x0244	"%d"	AltText[%d]	
AltText[%d].italicAngle	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].Item[%d]	0x0100 0x0235 0x0244	"%s"	AltText[%d]	
AltText[%d].Item[%d].bold	0x0100 0x0235 0x0244	"%d"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].font	0x0100 0x0235 0x0244	"%s"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].format	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	AltText[%d].Item[%d]	
AltText[%d].Item[%d].italic	0x0100 0x0235 0x0244	"%d"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].italicAngle	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].justification	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	AltText[%d].Item[%d]	
AltText[%d].Item[%d].language	0x0100 0x0235 0x0244	"%s"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].lineFactor	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	

AltText[%d].Item[%d].outlineType	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	AltText[%d].Item[%d]	
AltText[%d].Item[%d].spaceFactor	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].string	0x0100 0x0235 0x0244	"%s"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].symbol	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism,	AltText[%d].Item[%d]	

		"29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface		
AltText[%d].Item[%d].textAspect	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].textColour	0x0100 0x0235 0x0244	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	AltText[%d].Item[%d]	
AltText[%d].Item[%d].textHeight	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].textThickness	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	AltText[%d].Item[%d]	
AltText[%d].Item[%d].underline	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=over, "1"=under	AltText[%d].Item[%d]	
AltText[%d].justification	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	AltText[%d]	
AltText[%d].language	0x0100 0x0235 0x0244	"%s"	AltText[%d]	
AltText[%d].lineFactor	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].name	0x0100 0x0235 0x0244	"%s"	AltText[%d]	
AltText[%d].outlineType	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle,	AltText[%d]	

		"7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram		
AltText[%d].spaceFactor	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].textAspect	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].textColour	0x0100 0x0235 0x0244	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	AltText[%d]	
AltText[%d].textHeight	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].textOrientation	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=horizontal, "1"=vertical	AltText[%d]	
AltText[%d].textThickness	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	AltText[%d]	
AltText[%d].underline	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=over, "1"=under	AltText[%d]	
angle	0x0306	"%.16g"		
angleFormat	0x0082 0x0081	String representing enumeration such that: "0"=degrees, "1"=seconds, "2"=minutes		
angleNumerator	0x0082	"%.16g"		
approachDirection	0x0004	"%.16g %.16g %.16g"		
Area[%d]	0x0246	NULL		
Area[%d].ConstraintPlane.origin	0x0246	"%.16g %.16g %.16g"	Area[%d].ConstraintPlane	
Area[%d].ConstraintPlane.xaxis	0x0246	"%.16g %.16g %.16g"	Area[%d].ConstraintPlane	
Area[%d].ConstraintPlane.zaxis	0x0246	"%.16g %.16g %.16g"	Area[%d].ConstraintPlane	

Area[%d].curveRef[%d]	0x0246	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formating is used to semantically tie a singlar semantic aspect with an entity.	Area[%d]	
Area[%d].diameter	0x0246	"%.16g"	Area[%d]	
Area[%d].height	0x0246	"%.16g"	Area[%d]	
Area[%d].innerDiameter	0x0246	"%.16g"	Area[%d]	
Area[%d].insidePoint	0x0246	"%.16g %.16g %.16g"	Area[%d]	
Area[%d].length	0x0246	"%.16g"	Area[%d]	
Area[%d].originAnchor	0x0246	String representing enumeration such that: "0"=topleft, "1"=topcentre, "2"=topright, "3"=middleleft, "4"=middlecentre, "5"=middleright, "6"=bottomleft, "7"=bottomcentre, "8"=bottomright	Area[%d]	
Area[%d].type	0x0246	String representing enumeration such that: "0"=rectangular, "1"=circular, "2"=annular, "3"=cylindrical, "4"=general	Area[%d]	
Area[%d].width	0x0246	"%.16g"	Area[%d]	
areaShape	0x0088	String representing enumeration such that: "0"=diameter, "1"=sphericaldiameter, "2"=square		
arrowSideFinishSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=unspecified, "2"=machining, "3"=grinding, "4"=chipping, "5"=none, "6"=hammering, "7"=peening, "8"=rolling		
arrowSideGrooveAngle	0x0008	"%s"		
arrowSideGrooveGap	0x0008	"%s"		
arrowSideLongitudinalSize	0x0008	"%s"		

arrowSideMainSize	0x0008	"%s"		
arrowSideStaggeredSize	0x0008	"%s"		
arrowSideSupplementalSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=convex, "2"=toesblended, "3"=concave, "4"=flush		
arrowSideSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=jisseam, "2"=singlebevelgroove, "3"=singleugroove, "4"=edgeflange, "5"=surface, "6"=steepflankedsinglebevel, "7"=flaresinglevgroove, "8"=surfacejoint, "9"=singlevgroove, "10"=inclinedjoint, "11"=plug, "12"=fillet, "13"=squaregroove, "14"=spot, "15"=singlej, "16"=square, "17"=overlay, "18"=singleu, "19"=singlejgroove, "20"=doubleflange, "21"=singleflange, "22"=spotprojected, "23"=plugandslot, "24"=singlebevelbroadrootface, "25"=edge, "26"=jisstaggeredfillet1, "27"=singlev, "28"=flaresinglebevelgroove, "29"=singlevbroadrootface, "30"=steepflankedsinglev, "31"=removablebackingstrip, "32"=permanentbackingstrip, "33"=jisfillet, "34"=singlebevel, "35"=backing, "36"=foldjoint, "37"=jisstaggeredfillet2, "38"=seam, "39"=bead		

attachmentType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=angulardimension, "1"=ongeometry, "2"=lineardimension, "3"=leader, "4"=stacked, "5"=ordinatedimension, "6"=radialdimension, "7"=diametraldimension, "8"=noleader		
AttributeName[%d]	0x0240 0x0241	"%s"		
axisDirection	0x0230 0x0238	"%.16g %.16g %.16g"		
balloonId	0x0235	"%s"		
basic	0x0082	"%d"		
blanked	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
bold	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
bottomExtensionLine	0x0041	"%d"		
category	0x0100 0x0235 0x0244	"%s"		
causality	0x0082	String representing enumeration such that: "0"=key, "1"=functional, "2"=reference, "3"=associated, "4"=pmi		
centrecrossSize	0x0306	"%.16g"		

centrelineType	0x0306	String representing enumeration such that: "0"=centreline2d, "1"=centreline3d, "2"=centremark, "3"=circularcentreline, "4"=boltcirclecentreline, "5"=symmetriccentreline, "6"=offsetcentreline		
changeLevel	0x0118	String representing enumeration such that: "0"=soft, "1"=medium, "2"=hard		
characteristic	0x0081	String representing enumeration such that: "0"=profileofaline, "1"=circularrunout, "2"=perpendicularity, "3"=position, "4"=totalrunout, "5"=profileofasurface, "6"=cylindricity, "7"=symmetry, "8"=angularity, "9"=parallelism, "10"=concentricity, "11"=flatness, "12"=circularity, "13"=straightness		
clampingDirection	0x0004	"%.16g %.16g %.16g"		
CLIP_CAPPING	0x0305	"%d"		
CLIP_CAPPING_COLOUR	0x0305	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string		
CLIP_CURVE	0x0305	"%d"		
CLIP_CURVE_COLOUR	0x0305	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string		
CLIP_NORMAL	0x0305	"%.16g,%.16g,%.16g"		
CLIP_POSITION	0x0305	"%.16g,%.16g,%.16g"		
CLIP_RENDER	0x0305	String representing enumeration such that: "0"=shaded, "1"=wireframe		

CLIP_STYLE	0x0305	Enumerated string supporting values "Both", "Near", "Far" and "None"		
commaAsDecimal	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
conformsToFaces	0x0246	"%d"		
ConstraintPlane.origin	0x0242	"%.16g %.16g %.16g"	ConstraintPlane	
ConstraintPlane.xaxis	0x0242	"%.16g %.16g %.16g"	ConstraintPlane	
ConstraintPlane.zaxis	0x0242	"%.16g %.16g %.16g"	ConstraintPlane	
CoordinatedEntity[%d]	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	NULL		
CoordinatedEntity[%d].bold	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%d"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].font	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].italic	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%d"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].italicAngle	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].Item[%d]	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].Item[%d].bold	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%d"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].font	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].format	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].italic	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%d"	CoordinatedEntity[%d].Item[%d]	

CoordinatedEntity[%d].Item[%d].italicAngle	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].justification	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].language	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].lineFactor	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].outlineType	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].spaceFactor	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].string	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d].Item[%d]	

CoordinatedEntity[%d].Item[%d].symbol	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].textAspect	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].textColour	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	CoordinatedEntity[%d].Item[%d]	

		encoded in the string		
CoordinatedEntity[%d].Item[%d].textHeight	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].textThickness	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].underline	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=over, "1"=under	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].justification	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	CoordinatedEntity[%d]	
CoordinatedEntity[%d].language	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].lineFactor	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].name	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].outlineType	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	CoordinatedEntity[%d]	
CoordinatedEntity[%d].spaceFactor	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].textAspect	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	

CoordinatedEntity[%d].textColour	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	CoordinatedEntity[%d]	
CoordinatedEntity[%d].textHeight	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].textOrientation	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=horizontal, "1"=vertical	CoordinatedEntity[%d]	
CoordinatedEntity[%d].textThickness	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	CoordinatedEntity[%d]	
CoordinatedEntity[%d].underline	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=over, "1"=under	CoordinatedEntity[%d]	
coordinatePlane	0x0118	String representing enumeration such that: "0"=y, "1"=xz, "2"=xy, "3"=yz, "4"=z, "5"=xyz, "6"=x		
coordinationDirection	0x0042	"%.16g %.16g %.16g"		
counterbored	0x0230	"%d"		
CpsLetters	0x0242	NULL		
CpsLetters.bold	0x0242	"%d"	CpsLetters	
CpsLetters.font	0x0242	"%s"	CpsLetters	
CpsLetters.italic	0x0242	"%d"	CpsLetters	
CpsLetters.italicAngle	0x0242	"%.16g"	CpsLetters	
CpsLetters.Item[%d]	0x0242	"%s"	CpsLetters	
CpsLetters.Item[%d].bold	0x0242	"%d"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].font	0x0242	"%s"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].format	0x0242	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	CpsLetters.Item[%d]	
CpsLetters.Item[%d].italic	0x0242	"%d"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].italicAngle	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].justification	0x0242	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	CpsLetters.Item[%d]	

CpsLetters.Item[%d].language	0x0242	"%s"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].lineFactor	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].outlineType	0x0242	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	CpsLetters.Item[%d]	
CpsLetters.Item[%d].spaceFactor	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].string	0x0242	"%s"	CpsLetters.Item[%d]	

CpsLetters.Item[%d].symbol	0x0242	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	CpsLetters.Item[%d]	
CpsLetters.Item[%d].textAspect	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].textColour	0x0242	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	CpsLetters.Item[%d]	

		encoded in the string		
CpsLetters.Item[%d].textHeight	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].textThickness	0x0242	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	CpsLetters.Item[%d]	
CpsLetters.Item[%d].underline	0x0242	String representing enumeration such that: "0"=over, "1"=under	CpsLetters.Item[%d]	
CpsLetters.justification	0x0242	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	CpsLetters	
CpsLetters.language	0x0242	"%s"	CpsLetters	
CpsLetters.lineFactor	0x0242	"%.16g"	CpsLetters	
CpsLetters.name	0x0242	"%s"	CpsLetters	
CpsLetters.outlineType	0x0242	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	CpsLetters	
CpsLetters.spaceFactor	0x0242	"%.16g"	CpsLetters	
CpsLetters.textAspect	0x0242	"%.16g"	CpsLetters	
CpsLetters.textColour	0x0242	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	CpsLetters	

CpsLetters.textHeight	0x0242	"%.16g"	CpsLetters	
CpsLetters.textOrientation	0x0242	String representing enumeration such that: "0"=horizontal, "1"=vertical	CpsLetters	
CpsLetters.textThickness	0x0242	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	CpsLetters	
CpsLetters.underline	0x0242	String representing enumeration such that: "0"=over, "1"=under	CpsLetters	
CrosshatchPattern.angle	0x0243 0x0246	"%.16g"	CrosshatchPattern	
CrosshatchPattern.pattern	0x0243 0x0246	String representing enumeration such that: "0"=unset, "1"=rubber, "2"=lead, "3"=user, "4"=steel, "5"=brass, "6"=solid, "7"=iron, "8"=aluminum, "9"=glass, "10"=electricalwinding, "11"=refractory, "12"=thermalinsulation	CrosshatchPattern	
CrosshatchPattern.rotate	0x0243 0x0246	"%d"	CrosshatchPattern	
CrosshatchPattern.spacing	0x0243 0x0246	"%.16g"	CrosshatchPattern	
Curve[%d]	0x0305	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formating is used to semantically tie a singlar semantic aspect with an entity.		
curveRef	0x0242	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formating is used to semantically tie a singlar semantic aspect with an entity.	type	Expects type property to not be plane
datumOnLeader	0x0081 0x0084 0x0100 0x0235 0x0244	String representing enumeration such that: "0"=none, "1"=solid, "2"=filled		
delta	0x0004 0x0008	"%d"		
depth[%d]	0x0230	"%.16g"		

Description	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
descriptiveModifier	0x0233	"%s"		
deviation	0x0082	"%s"		
diameter	0x0088 0x0004 0x0238	"%.16g"		
diameter[%d]	0x0230	"%.16g"		
diameterOverride	0x0238	"%d"		
diameterPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
dimensionLeadingZero	0x0082	"%d"		
DimensionText[%d]	0x0082	NULL		DimensionText[%d] index expected to range from 0-3 inclusive.Each instance of this is expected to have a different position such that DimensionText[x].position != DimensionText[y].position for any combination of x and y
DimensionText[%d].position	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after	DimensionText[%d]	
DimensionText[%d].Text	0x0082	NULL	DimensionText[%d]	
DimensionText[%d].Text.bold	0x0082	"%d"	DimensionText[%d].Text	
DimensionText[%d].Text.font	0x0082	"%s"	DimensionText[%d].Text	
DimensionText[%d].Text.italic	0x0082	"%d"	DimensionText[%d].Text	
DimensionText[%d].Text.italicAngle	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.Item[%d]	0x0082	"%s"	DimensionText[%d].Text	
DimensionText[%d].Text.Item[%d].bold	0x0082	"%d"	DimensionText[%d].Text.Item[%d]	

DimensionText[%d].Text.Item[%d].font	0x0082	"%s"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].format	0x0082	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].italic	0x0082	"%d"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].italicAngle	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].justification	0x0082	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].language	0x0082	"%s"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].lineFactor	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].outlineType	0x0082	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].spaceFactor	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].string	0x0082	"%s"	DimensionText[%d].Text.Item[%d]	

DimensionText[%d].Text.Item[%d].symbol	0x0082	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].textAspect	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].textColour	0x0082	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	DimensionText[%d].Text.Item[%d]	

		encoded in the string		
DimensionText[%d].Text.Item[%d].textHeight	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].textThickness	0x0082	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].underline	0x0082	String representing enumeration such that: "0"=over, "1"=under	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.justification	0x0082	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	DimensionText[%d].Text	
DimensionText[%d].Text.language	0x0082	"%s"	DimensionText[%d].Text	
DimensionText[%d].Text.lineFactor	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.name	0x0082	"%s"	DimensionText[%d].Text	
DimensionText[%d].Text.outlineType	0x0082	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	DimensionText[%d].Text	
DimensionText[%d].Text.spaceFactor	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.textAspect	0x0082	"%.16g"	DimensionText[%d].Text	

DimensionText[%d].Text.textColour	0x0082	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	DimensionText[%d].Text	
DimensionText[%d].Text.textHeight	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.textOrientation	0x0082	String representing enumeration such that: "0"=horizontal, "1"=vertical	DimensionText[%d].Text	
DimensionText[%d].Text.textThickness	0x0082	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	DimensionText[%d].Text	
DimensionText[%d].Text.underline	0x0082	String representing enumeration such that: "0"=over, "1"=under	DimensionText[%d].Text	
dimensionTrailingZero	0x0082	"%d"		
direction	0x0082 0x0041 0x0081	"%.16g %.16g %.16g"		
dislayClearance	0x0307	"%d"		
displayOption	0x0306	String representing enumeration such that: "0"=none, "1"=centreline, "2"=centrelinewithextension, "3"=centrepoint		
displayTolerance	0x0307	"%d"		
displayType	0x0230 0x0004 0x0100 0x0235 0x0244 0x0042 0x0238	String representing enumeration such that: "0"=urface, "1"=creen		
documentation	0x0082	"%d"		
DualDimensionText[%d]	0x0082	NULL		DualDimensionText[%d] index expected to range from 0-3 inclusive.Each instance of this is expected to have a different position such that DualDimensionText[x].position != DualDimensionText[y].position for any combination of x and y
DualDimensionText[%d].position	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after	DualDimensionText[%d]	
DualDimensionText[%d].Text	0x0082	NULL	DualDimensionText[%d]	

DualDimensionText[%d].Text.bold	0x0082	"%d"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.font	0x0082	"%s"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.italic	0x0082	"%d"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.italicAngle	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.Item[%d]	0x0082	"%s"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.Item[%d].bold	0x0082	"%d"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].font	0x0082	"%s"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].format	0x0082	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].italic	0x0082	"%d"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].italicAngle	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].justification	0x0082	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].language	0x0082	"%s"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].lineFactor	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	

DualDimensionText[%d].Text.Item[%d].outlineType	0x0082	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].spaceFactor	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].string	0x0082	"%s"	DualDimensionText[%d].Text.Item[%d]	

DualDimensionText[%d].Text.Item[%d].symbol	0x0082	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].textAspect	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].textColour	0x0082	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	DualDimensionText[%d].Text.Item[%d]	

		encoded in the string		
DualDimensionText[%d].Text.Item[%d].textHeight	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].textThickness	0x0082	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].underline	0x0082	String representing enumeration such that: "0"=over, "1"=under	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.justification	0x0082	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.language	0x0082	"%s"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.lineFactor	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.name	0x0082	"%s"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.outlineType	0x0082	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.spaceFactor	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.textAspect	0x0082	"%.16g"	DualDimensionText[%d].Text	

DualDimensionText[%d].Text.textColour	0x0082	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.textHeight	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.textOrientation	0x0082	String representing enumeration such that: "0"=horizontal, "1"=vertical	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.textThickness	0x0082	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.underline	0x0082	String representing enumeration such that: "0"=over, "1"=under	DualDimensionText[%d].Text	
dualLowerDeltaDenominator	0x0082	"%d"		
dualPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
dualPrecision	0x0082	"%d"		
dualTolerancePrecision	0x0082	"%d"		
dualType	0x0082	String representing enumeration such that: "0"=positional, "1"=bracket		
dualUnits	0x0082	"%s"		
dualUnitText	0x0082	"%d"		
dualUpperDeltaDenominator	0x0082	"%d"		
dualValueDenominator	0x0082	"%d"		
fastenerSubType	0x0230	"%s"		
FCF[%d].	0x0308	"%.16g %.16g %.16g"	FCF[%d]	
FCF[%d].Text.bold	0x0308	"%d"	FCF[%d].Text	
FCF[%d].Text.font	0x0308	"%s"	FCF[%d].Text	
FCF[%d].Text.italic	0x0308	"%d"	FCF[%d].Text	
FCF[%d].Text.italicAngle	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.Item[%d]	0x0308	"%s"	FCF[%d].Text	
FCF[%d].Text.Item[%d].bold	0x0308	"%d"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].font	0x0308	"%s"	FCF[%d].Text.Item[%d]	

FCF[%d].Text.Item[%d].format	0x0308	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].italic	0x0308	"%d"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].italicAngle	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].justification	0x0308	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].language	0x0308	"%s"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].lineFactor	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].outlineType	0x0308	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].spaceFactor	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].string	0x0308	"%s"	FCF[%d].Text.Item[%d]	

FCF[%d].Text.Item[%d].symbol	0x0308	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].textAspect	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].textColour	0x0308	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	FCF[%d].Text.Item[%d]	

		encoded in the string		
FCF[%d].Text.Item[%d].textHeight	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].textThickness	0x0308	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].underline	0x0308	String representing enumeration such that: "0"=over, "1"=under	FCF[%d].Text.Item[%d]	
FCF[%d].Text.justification	0x0308	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	FCF[%d].Text	
FCF[%d].Text.language	0x0308	"%s"	FCF[%d].Text	
FCF[%d].Text.lineFactor	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.name	0x0308	"%s"	FCF[%d].Text	
FCF[%d].Text.outlineType	0x0308	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	FCF[%d].Text	
FCF[%d].Text.spaceFactor	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.textAspect	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.textColour	0x0308	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	FCF[%d].Text	

FCF[%d].Text.textHeight	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.textOrientation	0x0308	String representing enumeration such that: "0"=horizontal, "1"=vertical	FCF[%d].Text	
FCF[%d].Text.textThickness	0x0308	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	FCF[%d].Text	
FCF[%d].Text.underline	0x0308	String representing enumeration such that: "0"=over, "1"=under	FCF[%d].Text	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.bold	0x0308	"%d"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.font	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.italic	0x0308	"%d"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.italicAngle	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].bold	0x0308	"%d"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].font	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].format	0x0308	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].italic	0x0308	"%d"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].italicAngle	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].justification	0x0308	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].language	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].lineFactor	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].outlineType	0x0308	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].spaceFactor	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].string	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].symbol	0x0308	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textAspect	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textColour	0x0308	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textHeight	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textThickness	0x0308	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].underline	0x0308	String representing enumeration such that: "0"=over, "1"=under	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.justification	0x0308	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.language	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.lineFactor	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.name	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.outlineType	0x0308	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.spaceFactor	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textAspect	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textColour	0x0308	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textHeight	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textOrientation	0x0308	String representing enumeration such that: "0"=horizontal, "1"=vertical	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textThickness	0x0308	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.underline	0x0308	String representing enumeration such that: "0"=over, "1"=under	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
fieldWeld	0x0008	"%d"		
filename	0x0114	"%s"		
fitGrade	0x0082	"%d"		
flag	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
font	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
foreDistance	0x0242	"%.16g"	type	Expects type property to be closed
foreThroughAll	0x0242	"%d"	type	Expects type property to be closed
fraction	0x0082	"%d"		
fractionSize	0x0082	String representing enumeration such that: "0"=full, "1"=twothirds, "2"=threequarters		
FreeState	0x0082	NULL		
fullCircle	0x0306	"%d"		
functionalSubscript	0x0118	"%s"		
GeneralText	0x0230	NULL		
GeneralText.bold	0x0230	"%d"	GeneralText	
GeneralText.font	0x0230	"%s"	GeneralText	
GeneralText.italic	0x0230	"%d"	GeneralText	
GeneralText.italicAngle	0x0230	"%.16g"	GeneralText	
GeneralText.Item[%d]	0x0230	"%s"	GeneralText	
GeneralText.Item[%d].bold	0x0230	"%d"	GeneralText.Item[%d]	

GeneralText.Item[%d].font	0x0230	"%s"	GeneralText.Item[%d]	
GeneralText.Item[%d].format	0x0230	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	GeneralText.Item[%d]	
GeneralText.Item[%d].italic	0x0230	"%d"	GeneralText.Item[%d]	
GeneralText.Item[%d].italicAngle	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].justification	0x0230	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	GeneralText.Item[%d]	
GeneralText.Item[%d].language	0x0230	"%s"	GeneralText.Item[%d]	
GeneralText.Item[%d].lineFactor	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].outlineType	0x0230	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	GeneralText.Item[%d]	
GeneralText.Item[%d].spaceFactor	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].string	0x0230	"%s"	GeneralText.Item[%d]	

GeneralText.Item[%d].symbol	0x0230	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	GeneralText.Item[%d]	
GeneralText.Item[%d].textAspect	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].textColour	0x0230	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	GeneralText.Item[%d]	

		encoded in the string		
GeneralText.Item[%d].textHeight	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].textThickness	0x0230	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	GeneralText.Item[%d]	
GeneralText.Item[%d].underline	0x0230	String representing enumeration such that: "0"=over, "1"=under	GeneralText.Item[%d]	
GeneralText.justification	0x0230	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	GeneralText	
GeneralText.language	0x0230	"%s"	GeneralText	
GeneralText.lineFactor	0x0230	"%.16g"	GeneralText	
GeneralText.name	0x0230	"%s"	GeneralText	
GeneralText.outlineType	0x0230	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	GeneralText	
GeneralText.spaceFactor	0x0230	"%.16g"	GeneralText	
GeneralText.textAspect	0x0230	"%.16g"	GeneralText	
GeneralText.textColour	0x0230	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	GeneralText	

GeneralText.textHeight	0x0230	"%.16g"	GeneralText	
GeneralText.textOrientation	0x0230	String representing enumeration such that: "0"=horizontal, "1"=vertical	GeneralText	
GeneralText.textThickness	0x0230	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	GeneralText	
GeneralText.underline	0x0230	String representing enumeration such that: "0"=over, "1"=under	GeneralText	
grade	0x0082	"%d"		
group	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
groupId	0x0004	"%s"		
holeDeviation	0x0307	"%s"		
holeGrade	0x0307	"%d"		
holeLowerDetla	0x0307	"%.16g"		
holeUpperDelta	0x0307	"%.16g"		
hotSpotPosition	0x0118	"%.16g %.16g %.16g"		
identifier	0x0231 0x0232 0x0233 0x0100 0x0235 0x0244	"%s"		
includeI	0x0239	"%d"		
includeJ	0x0239	"%d"		
includeK	0x0239	"%d"		
includeLabel	0x0239	"%d"		
includeLabels	0x0240 0x0241	"%d"		
includeLevel	0x0239	"%d"		
includeX	0x0239	"%d"		
includeY	0x0239	"%d"		
includeZ	0x0239	"%d"		
index	0x0088	"%d"		
indexDirection	0x0230	"%.16g %.16g %.16g"		
individualExtensions	0x0306	"%d"		
individuallyApplied	0x0084	"%d"		
innerDiameter	0x0088	"%.16g"		
invertText	0x0041	"%d"		
iPrefix	0x0239	"%s"		
isReference	0x0082	"%d"		

iSuffix	0x0239	"%s"		
italic	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
italicAngle	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
itemName	0x0233	"%s"		
jisArrowSidePartialPenetrationGrooveWeld	0x0008	"%d"		
jisOtherSidePartialPenetrationGrooveWeld	0x0008	"%d"		
jisStaggeredFilletWeld	0x0008	"%d"		
jointId	0x0004	"%s"		
jPrefix	0x0239	"%s"		
jSuffix	0x0239	"%s"		
justification	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=right, "1"=centre, "2"=left		

Keyword[%d]	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
kPrefix	0x0239	"%s"		
kSuffix	0x0239	"%s"		
label	0x0084 0x0088	"%s"		
labelPrefix	0x0239	"%s"		
labelSuffix	0x0239	"%s"		
language	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
lay	0x0041	String representing enumeration such that: "0"=parallel, "1"=multidirectional, "2"=perpendicular, "3"=circular, "4"=particulate, "5"=angularboth, "6"=radial		
layer	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		

Leader[%d]	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	NULL		
Leader[%d].arrow Angle	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].arrow Colour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Leader[%d]	
Leader[%d].arrow Length	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].arrow LineThickness	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Leader[%d]	

	0x0306 0x0307			
Leader[%d].arrow LineType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=unset, "1"=longdashed, "2"=dotted, "3"=dotteddashed, "4"=phantom, "5"=dashed, "6"=solid, "7"=centerline	Leader[%d]	
Leader[%d].arrow OutSideLength	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].arrow Placement	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=out, "1"=in	Leader[%d]	

Leader[%d].arrowType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=closed, "1"=open, "2"=filled, "3"=none, "4"=origin, "5"=cross, "6"=dot, "7"=filleddot, "8"=wedge, "9"=plus, "10"=x, "11"=closedsolid, "12"=closeddouble, "13"=closeddoublesolid, "14"=openddouble, "15"=filleddouble, "16"=integral, "17"=box, "18"=filledbox, "19"=datum, "20"=filleddatum, "21"=solidorigin, "22"=filledorigin, "23"=xorigin, "24"=solidbox	Leader[%d]	
Leader[%d].colour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Leader[%d]	
Leader[%d].dotDiameter	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].extensionLineColour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Leader[%d]	

Leader[%d].extensionLineExtension	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].extensionLineGap	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].extensionLineThickness	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Leader[%d]	
Leader[%d].extensionLineType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=unset, "1"=longdashed, "2"=dotted, "3"=dotteddashed, "4"=phantom, "5"=dashed, "6"=solid, "7"=centerline	Leader[%d]	
Leader[%d].lineTextGap	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246	"%.16g"	Leader[%d]	

	0x0306 0x0307			
Leader[%d].lineType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=unset, "1"=longdashed, "2"=dotted, "3"=dotteddashed, "4"=phantom, "5"=dashed, "6"=solid, "7"=centreline	Leader[%d]	
Leader[%d].radiusToCentre	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"	Leader[%d]	
Leader[%d].Reference	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formating is used to semantically tie a singlar semantic aspect with an entity.	Leader[%d]	
Leader[%d].stubDirection	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=left, "1"=right	Leader[%d]	

Leader[%d].stubLength	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].terminator	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g %.16g %.16g"	Leader[%d]	
Leader[%d].thickness	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Leader[%d]	
Leader[%d].tParm	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].uvParm.U	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246	"%.16g"		Expect to also have Leader[%d].uvParam.V.

	0x0306 0x0307			
Leader[%d].uvParm.V	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		Expect to also have Leader[%d].uvParam.U.
leadingZero	0x0081	"%d"		
length	0x0088	"%.16g"		
level	0x0042	"%s"		
levelPrefix	0x0239	"%s"		
levelSuffix	0x0239	"%s"		
limitFitOrder	0x0082 0x0307	String representing enumeration such that: "0"=valuelimitfittolerance, "1"=tolerancevaluelimitfit, "2"=valuetolerancelimitfit		
limitFitParenthesis	0x0082 0x0307	String representing enumeration such that: "0"=none, "1"=valuelimitfit, "2"=limitfit, "3"=tolerance		
linearPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
lineFactor	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		

lineStyle	0x0114	String representing enumeration such that: "0"=unset, "1"=longdashed, "2"=dotted, "3"=dotteddashed, "4"=phantom, "5"=dashed, "6"=solid, "7"=centerline		
locationOnCurve	0x0241	"%.16g"		
lowerDelta	0x0082 0x0041	"%.16g"		
lowerDeltaDenominator	0x0082	"%d"		
machiningAllowance	0x0041	"%s"		
mainEdgeType	0x0118	String representing enumeration such that: "0"=ce, "1"=e, "2"=e, "3"=ek, "4"=co		
mainHoleType	0x0118	String representing enumeration such that: "0"=dl, "1"=h, "2"=hk, "3"=ch, "4"=hk, "5"=ac, "6"=h, "7"=bl, "8"=ch		
mainRectSlotFreeformType	0x0118	String representing enumeration such that: "0"=dl, "1"=h, "2"=hk, "3"=bl, "4"=hk, "5"=ac, "6"=h, "7"=ch		
mainSurfaceType	0x0118	String representing enumeration such that: "0"=sk, "1"=co, "2"=j, "3"=s, "4"=sl, "5"=sk, "6"=cs, "7"=s, "8"=ck		
materialSide	0x0242	"%d"		
maxBonus	0x0081	"%d"		
maxBonusPrecision	0x0081	"%d"		
maxBonusValue	0x0081	"%.16g"		
maxRoughness	0x0041	"%s"		
maxRoughness2	0x0041	"%s"		
measurementDirection	0x0042	"%.16g %.16g %.16g"		
minRoughness	0x0041	"%s"		
minRoughness2	0x0041	"%s"		
modifier	0x0041	"%d"		
movable	0x0088	"%d"		
movableTargetStubDirection	0x0088	String representing enumeration such that: "0"=left, "1"=right		
movableTargetStubLength	0x0088	"%.16g"		
nameModifier	0x0233	"%s"		

Nomenclature	0x0231 0x0232	NULL		
Nomenclature.bold	0x0231 0x0232	"%d"	Nomenclature	
Nomenclature.font	0x0231 0x0232	"%s"	Nomenclature	
Nomenclature.italic	0x0231 0x0232	"%d"	Nomenclature	
Nomenclature.italicAngle	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.Item[%d]	0x0231 0x0232	"%s"	Nomenclature	
Nomenclature.Item[%d].bold	0x0231 0x0232	"%d"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].font	0x0231 0x0232	"%s"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].format	0x0231 0x0232	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	Nomenclature.Item[%d]	
Nomenclature.Item[%d].italic	0x0231 0x0232	"%d"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].italicAngle	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].justification	0x0231 0x0232	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Nomenclature.Item[%d]	
Nomenclature.Item[%d].language	0x0231 0x0232	"%s"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].lineFactor	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].outlineType	0x0231 0x0232	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangl	Nomenclature.Item[%d]	

		e, "21"=parallelogram		
Nomenclature.Item[%d].spaceFactor	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].string	0x0231 0x0232	"%s"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].symbol	0x0231 0x0232	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position,	Nomenclature.Item[%d]	

		"35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface		
Nomenclature.Item[%d].textAspect	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].textColour	0x0231 0x0232	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Nomenclature.Item[%d]	
Nomenclature.Item[%d].textHeight	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].textThickness	0x0231 0x0232	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Nomenclature.Item[%d]	
Nomenclature.Item[%d].underline	0x0231 0x0232	String representing enumeration such that: "0"=over, "1"=under	Nomenclature.Item[%d]	
Nomenclature.justification	0x0231 0x0232	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Nomenclature	
Nomenclature.language	0x0231 0x0232	"%s"	Nomenclature	
Nomenclature.lineFactor	0x0231 0x0232	"%.16g"	Nomenclature	

Nomenclature.name	0x0231 0x0232	"%s"	Nomenclature	
Nomenclature.outlineType	0x0231 0x0232	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Nomenclature	
Nomenclature.spaceFactor	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.textAspect	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.textColour	0x0231 0x0232	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Nomenclature	
Nomenclature.textHeight	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.textOrientation	0x0231 0x0232	String representing enumeration such that: "0"=horizontal, "1"=vertical	Nomenclature	
Nomenclature.textThickness	0x0231 0x0232	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Nomenclature	
Nomenclature.underline	0x0231 0x0232	String representing enumeration such that: "0"=over, "1"=under	Nomenclature	
normalDirection	0x0004 0x0042 0x0118 0x0238	"%.16g %.16g %.16g"		
Note	0x0118	NULL		
Note.bold	0x0118	"%d"	Note	
Note.font	0x0118	"%s"	Note	
Note.italic	0x0118	"%d"	Note	
Note.italicAngle	0x0118	"%.16g"	Note	

Note.Item[%d]	0x0118	"%s"	Note	
Note.Item[%d].bold	0x0118	"%d"	Note.Item[%d]	
Note.Item[%d].font	0x0118	"%s"	Note.Item[%d]	
Note.Item[%d].format	0x0118	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	Note.Item[%d]	
Note.Item[%d].italic	0x0118	"%d"	Note.Item[%d]	
Note.Item[%d].italicAngle	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].justification	0x0118	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Note.Item[%d]	
Note.Item[%d].language	0x0118	"%s"	Note.Item[%d]	
Note.Item[%d].lineFactor	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].outlineType	0x0118	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Note.Item[%d]	
Note.Item[%d].spaceFactor	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].string	0x0118	"%s"	Note.Item[%d]	

Note.Item[%d].symbol	0x0118	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	Note.Item[%d]	
Note.Item[%d].textAspect	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].textColour	0x0118	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the	Note.Item[%d]	

		string		
Note.Item[%d].textHeight	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].textThickness	0x0118	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Note.Item[%d]	
Note.Item[%d].underline	0x0118	String representing enumeration such that: "0"=over, "1"=under	Note.Item[%d]	
Note.justification	0x0118	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Note	
Note.language	0x0118	"%s"	Note	
Note.lineFactor	0x0118	"%.16g"	Note	
Note.name	0x0118	"%s"	Note	
Note.outlineType	0x0118	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Note	
Note.spaceFactor	0x0118	"%.16g"	Note	
Note.textAspect	0x0118	"%.16g"	Note	
Note.textColour	0x0118	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Note	
Note.textHeight	0x0118	"%.16g"	Note	

Note.textOrientation	0x0118	String representing enumeration such that: "0"=horizontal, "1"=vertical	Note	
Note.textThickness	0x0118	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Note	
Note.underline	0x0118	String representing enumeration such that: "0"=over, "1"=under	Note	
notToScale	0x0082	"%d"		
OpenField	0x0231 0x0232	NULL		
OpenField.bold	0x0231 0x0232	"%d"	OpenField	
OpenField.font	0x0231 0x0232	"%s"	OpenField	
OpenField.italic	0x0231 0x0232	"%d"	OpenField	
OpenField.italicAngle	0x0231 0x0232	"%.16g"	OpenField	
OpenField.Item[%d]	0x0231 0x0232	"%s"	OpenField	
OpenField.Item[%d].bold	0x0231 0x0232	"%d"	OpenField.Item[%d]	
OpenField.Item[%d].font	0x0231 0x0232	"%s"	OpenField.Item[%d]	
OpenField.Item[%d].format	0x0231 0x0232	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	OpenField.Item[%d]	
OpenField.Item[%d].italic	0x0231 0x0232	"%d"	OpenField.Item[%d]	
OpenField.Item[%d].italicAngle	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	
OpenField.Item[%d].justification	0x0231 0x0232	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	OpenField.Item[%d]	
OpenField.Item[%d].language	0x0231 0x0232	"%s"	OpenField.Item[%d]	
OpenField.Item[%d].lineFactor	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	

OpenField.Item[%d].outlineType	0x0231 0x0232	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	OpenField.Item[%d]	
OpenField.Item[%d].spaceFactor	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	
OpenField.Item[%d].string	0x0231 0x0232	"%s"	OpenField.Item[%d]	

OpenField.Item[%d].symbol	0x0231 0x0232	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	OpenField.Item[%d]	
OpenField.Item[%d].textAspect	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	
OpenField.Item[%d].textColour	0x0231 0x0232	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	OpenField.Item[%d]	

		encoded in the string		
OpenField.Item[%d].textHeight	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	
OpenField.Item[%d].textThickness	0x0231 0x0232	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	OpenField.Item[%d]	
OpenField.Item[%d].underline	0x0231 0x0232	String representing enumeration such that: "0"=over, "1"=under	OpenField.Item[%d]	
OpenField.justification	0x0231 0x0232	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	OpenField	
OpenField.language	0x0231 0x0232	"%s"	OpenField	
OpenField.lineFactor	0x0231 0x0232	"%.16g"	OpenField	
OpenField.name	0x0231 0x0232	"%s"	OpenField	
OpenField.outlineType	0x0231 0x0232	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	OpenField	
OpenField.spaceFactor	0x0231 0x0232	"%.16g"	OpenField	
OpenField.textAspect	0x0231 0x0232	"%.16g"	OpenField	
OpenField.textColour	0x0231 0x0232	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	OpenField	

OpenField.textHeight	0x0231 0x0232	"%.16g"	OpenField	
OpenField.textOrientation	0x0231 0x0232	String representing enumeration such that: "0"=horizontal, "1"=vertical	OpenField	
OpenField.textThickness	0x0231 0x0232	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	OpenField	
OpenField.underline	0x0231 0x0232	String representing enumeration such that: "0"=over, "1"=under	OpenField	
optDirection	0x0230	"%.16g %.16g %.16g"		
origin	0x0082	String representing enumeration such that: "0"=first, "1"=second		
otherSideFinishSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=unspecified, "2"=machining, "3"=grinding, "4"=chipping, "5"=none, "6"=hammering, "7"=peening, "8"=rolling		
otherSideGrooveAngle	0x0008	"%s"		
otherSideGrooveGap	0x0008	"%s"		
otherSideLongitudinalSize	0x0008	"%s"		
otherSideMainSize	0x0008	"%s"		
otherSideStaggeredSize	0x0008	"%s"		
otherSideSupplementalSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=convex, "2"=toesblended, "3"=concave, "4"=flush		

otherSideSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=jisseam, "2"=singlebevelgroove, "3"=singleugroove, "4"=edgeflange, "5"=surface, "6"=steepflankedsinglebevel, "7"=flaresinglevgroove, "8"=surfacejoint, "9"=singlevgroove, "10"=inclinedjoint, "11"=plug, "12"=fillet, "13"=squaregroove, "14"=spot, "15"=singlej, "16"=square, "17"=overlay, "18"=singleu, "19"=singlejgroove, "20"=doubleflange, "21"=singleflange, "22"=spotprojected, "23"=plugandslot, "24"=singlebevelbroadrootface, "25"=edge, "26"=jisstaggeredfillet1, "27"=singlev, "28"=flaresinglebevelgroove, "29"=singlevbroadrootface, "30"=steepflankedsinglev, "31"=removablebackingstrip, "32"=permanentbackingstrip, "33"=jisfillet, "34"=singlebevel, "35"=backing, "36"=foldjoint, "37"=jisstaggeredfillet2, "38"=seam, "39"=bead		

outlineType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram		
overrun	0x0306	"%.16g"		
parenthesis	0x0041	String representing enumeration such that: "0"=none, "1"=left, "2"=right, "3"=both		
PartNumber	0x0118	NULL		
PartNumber.bold	0x0118	"%d"	PartNumber	
PartNumber.font	0x0118	"%s"	PartNumber	
PartNumber.italic	0x0118	"%d"	PartNumber	
PartNumber.italicAngle	0x0118	"%.16g"	PartNumber	
PartNumber.Item[%d]	0x0118	"%s"	PartNumber	
PartNumber.Item[%d].bold	0x0118	"%d"	PartNumber.Item[%d]	
PartNumber.Item[%d].font	0x0118	"%s"	PartNumber.Item[%d]	
PartNumber.Item[%d].format	0x0118	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	PartNumber.Item[%d]	
PartNumber.Item[%d].italic	0x0118	"%d"	PartNumber.Item[%d]	
PartNumber.Item[%d].italicAngle	0x0118	"%.16g"	PartNumber.Item[%d]	

PartNumber.Item[%d].justification	0x0118	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	PartNumber.Item[%d]	
PartNumber.Item[%d].language	0x0118	"%s"	PartNumber.Item[%d]	
PartNumber.Item[%d].lineFactor	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].outlineType	0x0118	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	PartNumber.Item[%d]	
PartNumber.Item[%d].spaceFactor	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].string	0x0118	"%s"	PartNumber.Item[%d]	

PartNumber.Item[%d].symbol	0x0118	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	PartNumber.Item[%d]	
PartNumber.Item[%d].textAspect	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].textColour	0x0118	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	PartNumber.Item[%d]	

		encoded in the string		
PartNumber.Item[%d].textHeight	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].textThickness	0x0118	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	PartNumber.Item[%d]	
PartNumber.Item[%d].underline	0x0118	String representing enumeration such that: "0"=over, "1"=under	PartNumber.Item[%d]	
PartNumber.justification	0x0118	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	PartNumber	
PartNumber.language	0x0118	"%s"	PartNumber	
PartNumber.lineFactor	0x0118	"%.16g"	PartNumber	
PartNumber.name	0x0118	"%s"	PartNumber	
PartNumber.outlineType	0x0118	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	PartNumber	
PartNumber.spaceFactor	0x0118	"%.16g"	PartNumber	
PartNumber.textAspect	0x0118	"%.16g"	PartNumber	
PartNumber.textColour	0x0118	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	PartNumber	

PartNumber.textHeight	0x0118	"%.16g"	PartNumber	
PartNumber.textOrientation	0x0118	String representing enumeration such that: "0"=horizontal, "1"=vertical	PartNumber	
PartNumber.textThickness	0x0118	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	PartNumber	
PartNumber.underline	0x0118	String representing enumeration such that: "0"=over, "1"=under	PartNumber	
pinDirection	0x0118	"%.16g %.16g %.16g"		
pitchDiaDeviation	0x0082 0x0307	"%s"		
pitchDiaGrade	0x0082 0x0307	"%d"		
planar	0x0088	"%d"		
point1	0x0088	"%.16g %.16g %.16g"		
point2	0x0088	"%.16g %.16g %.16g"		
position	0x0230 0x0004 0x0042 0x0238	"%.16g %.16g %.16g"		
Position[%d]	0x0306	"%.16g %.16g %.16g"		
precision	0x0082 0x0230 0x0041 0x0088 0x0004 0x0008 0x0118 0x0238 0x0239 0x0241 0x0307	"%d"		
ProcessText	0x0041 0x0004 0x0008	NULL		
ProcessText.bold	0x0041 0x0004 0x0008	"%d"	ProcessText	
ProcessText.font	0x0041 0x0004 0x0008	"%s"	ProcessText	
ProcessText.italic	0x0041 0x0004 0x0008	"%d"	ProcessText	
ProcessText.italicAngle	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.Item[%d]	0x0041 0x0004 0x0008	"%s"	ProcessText	
ProcessText.Item[%d].bold	0x0041 0x0004 0x0008	"%d"	ProcessText.Item[%d]	
ProcessText.Item[%d].font	0x0041 0x0004 0x0008	"%s"	ProcessText.Item[%d]	
ProcessText.Item[%d].format	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	ProcessText.Item[%d]	
ProcessText.Item[	0x0041 0x0004	"%d"	ProcessText.Item[%d]	

%d].italic	0x0008			
ProcessText.Item[%d].italicAngle	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].justification	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	ProcessText.Item[%d]	
ProcessText.Item[%d].language	0x0041 0x0004 0x0008	"%s"	ProcessText.Item[%d]	
ProcessText.Item[%d].lineFactor	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].outlineType	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	ProcessText.Item[%d]	
ProcessText.Item[%d].spaceFactor	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].string	0x0041 0x0004 0x0008	"%s"	ProcessText.Item[%d]	

ProcessText.Item[%d].symbol	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	ProcessText.Item[%d]	
ProcessText.Item[%d].textAspect	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].textColour	0x0041 0x0004 0x0008	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	ProcessText.Item[%d]	

		encoded in the string		
ProcessText.Item[%d].textHeight	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].textThickness	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	ProcessText.Item[%d]	
ProcessText.Item[%d].underline	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=over, "1"=under	ProcessText.Item[%d]	
ProcessText.justification	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	ProcessText	
ProcessText.language	0x0041 0x0004 0x0008	"%s"	ProcessText	
ProcessText.lineFactor	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.name	0x0041 0x0004 0x0008	"%s"	ProcessText	
ProcessText.outlineType	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	ProcessText	
ProcessText.spaceFactor	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.textAspect	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.textColour	0x0041 0x0004 0x0008	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	ProcessText	

ProcessText.textHeight	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.textOrientation	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=horizontal, "1"=vertical	ProcessText	
ProcessText.textThickness	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	ProcessText	
ProcessText.underline	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=over, "1"=under	ProcessText	
profileType	0x0081	String representing enumeration such that: "0"=bilateral, "1"=unilateralinside, "2"=unilateraloutside, "3"=bilateralunequal		
profileValue	0x0081	"%.16g"		
profileValue2	0x0081	"%.16g"		
projected	0x0082	"%d"		
projectedPrecision	0x0081	"%d"		
punchDirection	0x0118	"%.16g %.16g %.16g"		
radialPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
radius	0x0235	"%.16g"		
referenceDisplay	0x0082	String representing enumeration such that: "0"=parenthesis, "1"=reference, "2"=matched		
referenceSign	0x0008	"%d"		
revision	0x0231 0x0232 0x0233 0x0100 0x0235 0x0244	"%s"		
roughness	0x0041	"%s"		
roughness2	0x0041	"%s"		
roughnessCutoff	0x0041	"%s"		
roughnessSpacing	0x0041	"%s"		
SafetyClassification	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		

scale	0x0114	"%.16g"		
shaftDeviation	0x0307	"%s"		
shaftGrade	0x0307	"%d"		
shaftLowerDelta	0x0307	"%.16g"		
shaftUpperDelta	0x0307	"%.16g"		
shapeTextOutsideSymbol	0x0088	"%d"		
shortDash	0x0306	"%.16g"		
singleLine	0x0041 0x0307	"%d"		
size	0x0088	"%s"		
spaceFactor	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
standard	0x0082 0x0081 0x0084 0x0088 0x0100 0x0235 0x0244 0x0239 0x0240 0x0241	String representing enumeration such that: "0"=asme_y145m_1994, "1"=jis, "2"=iso, "3"=bs, "4"=ansi_y145m_1982, "5"=asme_y1441m_2003, "6"=din, "7"=gm_addendum_1994, "8"=asme_y145_2009		
standard	0x0041	String representing enumeration such that: "0"=jis, "1"=ansi_y1436_1993, "2"=iso, "3"=asme_y1436m_1996, "4"=din, "5"=gb, "6"=iso_1302_2002, "7"=eskd, "8"=din_en_iso_1302_2002		
standard	0x0008	String representing enumeration such that: "0"=ansiaws_a24-98 "1"=iso_2556 "2"=jis_z_3021 "3"=din "4"=eskd "5"=gb		
Statistical	0x0082	NULL		

statisticalPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
style	0x0082	String representing enumeration such that: "0"=lineardiametral, "1"=radial, "2"=controlledradial, "3"=diametral, "4"=limits, "5"=ordinate, "6"=sphericalradial, "7"=sphericaldiametral		
subType	0x0118	String representing enumeration such that: "0"=mechanicallyfastened, "1"=manufacturingandassembly, "2"=diesmolds, "3"=fixing, "4"=temporarytransferred		
symbol	0x0118	String representing enumeration such that: "0"=rectangle, "1"=edge, "2"=hole, "3"=freeform, "4"=slot, "5"=surface		
symbolColour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string		
symbolType	0x0230	"%s"		
Text	0x0081	NULL		
Text.bold	0x0081	"%d"	Text	
Text.font	0x0081	"%s"	Text	
Text.italic	0x0081	"%d"	Text	
Text.italicAngle	0x0081	"%.16g"	Text	
Text.Item[%d]	0x0081	"%s"	Text	
Text.Item[%d].bold	0x0081	"%d"	Text.Item[%d]	
Text.Item[%d].font	0x0081	"%s"	Text.Item[%d]	

Text.Item[%d].format	0x0081	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	Text.Item[%d]	
Text.Item[%d].italic	0x0081	"%d"	Text.Item[%d]	
Text.Item[%d].italicAngle	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].justification	0x0081	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Text.Item[%d]	
Text.Item[%d].language	0x0081	"%s"	Text.Item[%d]	
Text.Item[%d].lineFactor	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].outlineType	0x0081	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Text.Item[%d]	
Text.Item[%d].spaceFactor	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].string	0x0081	"%s"	Text.Item[%d]	

Text.Item[%d].symbol	0x0081	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	Text.Item[%d]	
Text.Item[%d].textAspect	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].textColour	0x0081	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the	Text.Item[%d]	

		string		
Text.Item[%d].textHeight	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].textThickness	0x0081	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Text.Item[%d]	
Text.Item[%d].underline	0x0081	String representing enumeration such that: "0"=over, "1"=under	Text.Item[%d]	
Text.justification	0x0081	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Text	
Text.language	0x0081	"%s"	Text	
Text.lineFactor	0x0081	"%.16g"	Text	
Text.name	0x0081	"%s"	Text	
Text.outlineType	0x0081	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Text	
Text.spaceFactor	0x0081	"%.16g"	Text	
Text.textAspect	0x0081	"%.16g"	Text	
Text.textColour	0x0081	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Text	
Text.textHeight	0x0081	"%.16g"	Text	

Text.textOrientation	0x0081	String representing enumeration such that: "0"=horizontal, "1"=vertical	Text	
Text.textThickness	0x0081	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Text	
Text.underline	0x0081	String representing enumeration such that: "0"=over, "1"=under	Text	
Text[%d]	0x0100 0x0235 0x0244	NULL		
Text[%d].bold	0x0100 0x0235 0x0244	"%d"	Text[%d]	
Text[%d].font	0x0100 0x0235 0x0244	"%s"	Text[%d]	
Text[%d].italic	0x0100 0x0235 0x0244	"%d"	Text[%d]	
Text[%d].italicAngle	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].Item[%d]	0x0100 0x0235 0x0244	"%s"	Text[%d]	
Text[%d].Item[%d].bold	0x0100 0x0235 0x0244	"%d"	Text[%d].Item[%d]	
Text[%d].Item[%d].font	0x0100 0x0235 0x0244	"%s"	Text[%d].Item[%d]	
Text[%d].Item[%d].format	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	Text[%d].Item[%d]	
Text[%d].Item[%d].italic	0x0100 0x0235 0x0244	"%d"	Text[%d].Item[%d]	
Text[%d].Item[%d].italicAngle	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	
Text[%d].Item[%d].justification	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Text[%d].Item[%d]	
Text[%d].Item[%d].language	0x0100 0x0235 0x0244	"%s"	Text[%d].Item[%d]	
Text[%d].Item[%d].lineFactor	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	

Text[%d].Item[%d] .outlineType	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Text[%d].Item[%d]	
Text[%d].Item[%d] .spaceFactor	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	
Text[%d].Item[%d] .string	0x0100 0x0235 0x0244	"%s"	Text[%d].Item[%d]	

Text[%d].Item[%d].symbol	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	Text[%d].Item[%d]	
Text[%d].Item[%d].textAspect	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	
Text[%d].Item[%d].textColour	0x0100 0x0235 0x0244	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	Text[%d].Item[%d]	

		encoded in the string		
Text[%d].Item[%d].textHeight	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	
Text[%d].Item[%d].textThickness	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Text[%d].Item[%d]	
Text[%d].Item[%d].underline	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=over, "1"=under	Text[%d].Item[%d]	
Text[%d].justification	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Text[%d]	
Text[%d].language	0x0100 0x0235 0x0244	"%s"	Text[%d]	
Text[%d].lineFactor	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].name	0x0100 0x0235 0x0244	"%s"	Text[%d]	
Text[%d].outlineType	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Text[%d]	
Text[%d].spaceFactor	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].textAspect	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].textColour	0x0100 0x0235 0x0244	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Text[%d]	

Text[%d].textHeight	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].textOrientation	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=horizontal, "1"=vertical	Text[%d]	
Text[%d].textThickness	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Text[%d]	
Text[%d].underline	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=over, "1"=under	Text[%d]	
textAspect	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
textBox	0x0239	"%d"		
textColour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string		
textDirection	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g %.16g %.16g"		

textHeight	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
textOrigin	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g %.16g %.16g"		
textStubDirection	0x0088	String representing enumeration such that: "0"=left, "1"=right		
textStubLength	0x0088	"%.16g"		
textThickness	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin		
thickness	0x0004	"%.16g"		
thicknesses	0x0004	"%d"		
threadClass	0x0082	"%d"		
threaded	0x0230	"%d"		
ToleranceCompartment[%d]	0x0081	NULL		
ToleranceCompartment[%d].commonZone	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].FreeState	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].maxBonus	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].maxBonusValue	0x0081	"%.16g"	ToleranceCompartment[%d]	

ToleranceCompartment[%d].modifier	0x0081	String representing enumeration such that: "0"=lmc, "1"=mmc, "2"=rfs	ToleranceCompartment[%d]	
ToleranceCompartment[%d].precision	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	0x0081	NULL	ToleranceCompartment[%d].PrimaryDatum	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].datumRef	0x0081	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formating is used to semantically tie a singlar semantic aspect with an entity.	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].FreeState	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].label	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].modifier	0x0081	String representing enumeration such that: "0"=lmc, "1"=mmc, "2"=rfs	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	0x0081	NULL	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.bold	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.font	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.italic	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.italicAngle	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Ite	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.

m[%d]				no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].bold	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].font	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].format	0x0081	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].italic	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].italicAngle	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].justification	0x0081	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].language	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].lineFactor	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].outlineType	0x0081	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].spaceFactor	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].string	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].symbol	0x0081	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textAspect	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textColour	0x0081	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textHeight	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textThickness	0x0081	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].underline	0x0081	String representing enumeration such that: "0"=over, "1"=under	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.justification	0x0081	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.language	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.lineFactor	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.name	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.outlineType	0x0081	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota,	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

		"18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram		
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.spaceFactor	• 0x0081	• "%.16g"	• ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	• PrimaryDatum can be replaced with Secondary Datum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textAspect	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textColour	0x0081	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textHeight	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textOrientation	0x0081	String representing enumeration such that: "0"=horizontal, "1"=vertical	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textThickness	0x0081	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.underline	0x0081	String representing enumeration such that: "0"=over, "1"=under	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].projected	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].projectedValue	0x0081	"%s"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].projectionVector	0x0081	"%.16g %.16g %.16g"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].Statistical	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].tangentPlane	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].unitBasis	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].unitBasisValue	0x0081	"%s"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].value	0x0081	"%s"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].zoneShape	0x0081	String representing enumeration such that: "0"=diameter, "1"=sphericaldiameter, "2"=square	ToleranceCompartment[%d]	
toleranceDisplay	0x0082 0x0041 0x0307	String representing enumeration such that: "0"=minuslimit1, "1"=bilateral, "2"=pluslimit1, "3"=pluslimit2, "4"=equalbilateral, "5"=minuslimit2		
toleranceLeadingZero	0x0082 0x0041 0x0307	"%d"		
tolerancePrecision	0x0082 0x0307	"%d"		
toleranceTrailingZero	0x0082 0x0041 0x0307	"%d"		
trailingZero	0x0081	"%d"		
twoArrows	0x0242	"%d"		
type	0x0082	String representing enumeration such that: "0"=curvelength, "1"=linear, "2"=angular, "3"=radial		
type	0x0230	"%s"		
type	0x0041	String representing enumeration such that: "0"=basic, "1"=mrr, "2"=mrp		

type	0x0088	String representing enumeration such that: "0"=cylindrical, "1"=line, "2"=arbitrary, "3"=circular, "4"=point, "5"=annular, "6"=area, "7"=rectangular, "8"=spherical		
type	0x0004	String representing enumeration such that: "0"=unset, "1"=resistance, "2"=projection		
type	0x0241	String representing enumeration such that: "0"=bundlenote, "1"=dressingnote		
type	0x0242	String representing enumeration such that: "0"=plane, "1"=open, "2"=closed		
type	0x0244	String representing enumeration such that: "0"=exportcontrol, "1"=productproprietaryinfo, "2"=governmentsecurityinfo		
underline	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=over, "1"=under		
units	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=metres, "1"=millimetres, "2"=inches		
unitText	0x0082	"%d"		
upperDelta	0x0082 0x0041	"%.16g"		
upperDeltaDenomi	0x0082	"%d"		

nator				
uriRefs[%d]	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
usage	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
UserDefinedSymbolText[%d]	0x0114	NULL		
UserDefinedSymbolText[%d].Text	0x0114	NULL	UserDefinedSymbolText[%d]	
UserDefinedSymbolText[%d].Text.bold	0x0114	"%d"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.font	0x0114	"%s"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.italic	0x0114	"%d"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.italicAngle	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.Item[%d]	0x0114	"%s"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.Item[%d].bold	0x0114	"%d"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].font	0x0114	"%s"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].format	0x0114	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list,	UserDefinedSymbolText[%d].Text.Item[%d]	

		"10"=datetime		
UserDefinedSymbolText[%d].Text.Item[%d].italic	0x0114	"%d"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].italicAngle	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].justification	0x0114	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].language	0x0114	"%s"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].lineFactor	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].outlineType	0x0114	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].spaceFactor	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].string	0x0114	"%s"	UserDefinedSymbolText[%d].Text.Item[%d]	

UserDefinedSymbolText[%d].Text.Item[%d].symbol	0x0114	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].textAspect	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].textColour	0x0114	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are	UserDefinedSymbolText[%d].Text.Item[%d]	

		encoded in the string		
UserDefinedSymbolText[%d].Text.Item[%d].textHeight	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].textThickness	0x0114	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].underline	0x0114	String representing enumeration such that: "0"=over, "1"=under	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.justification	0x0114	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.language	0x0114	"%s"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.lineFactor	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.name	0x0114	"%s"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.outlineType	0x0114	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.spaceFactor	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.textAspect	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	

UserDefinedSymbolText[%d].Text.textColour	0x0114	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.textHeight	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.textOrientation	0x0114	String representing enumeration such that: "0"=horizontal, "1"=vertical	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.textThickness	0x0114	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.underline	0x0114	String representing enumeration such that: "0"=over, "1"=under	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].textDirection	0x0114	"%.16g %.16g %.16g"	UserDefinedSymbolText[%d]	
UserDefinedSymbolText[%d].textOrigin	0x0114	"%.16g %.16g %.16g"	UserDefinedSymbolText[%d]	
valid	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=true, "1"=false, "2"=unknown		
value	0x0082 0x0307	"%.16g"		
valueDenominator	0x0082	"%d"		
valueLeadingZero	0x0307	"%d"		
ValueToCustomer	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
valueTrailingZero	0x0307	"%d"		
wavinessHeight	0x0041	"%s"		
wavinessSpacing	0x0041	"%s"		
width	0x0088	"%.16g"		

xPrefix	0x0239	"%s"		
xSuffix	0x0239	"%s"		
yPrefix	0x0239	"%s"		
ySuffix	0x0239	"%s"		
zeroToleranceDisplay	0x0082 0x0041 0x0307	String representing enumeration such that: "0"=displayzero, "1"=blank, "2"=suppresstrailingzero, "3"=omit		
zPrefix	0x0239	"%s"		
zSuffix	0x0239	"%s"		

14.8.3 Tessellation Properties

When dealing with facetted graphical representations (i.e. LODs) of precise models (e.g. JT B-Rep), depending on the desired use it is often useful/necessary to know what tessellation tolerances were used to generate the facetted representation. To that end, two properties are typically stored on Part Node Elements (if part also has precise model) to indicate the tessellation tolerances used to generate each LOD. These two tessellation properties are as follows

Table 109 —Tessellatin Proptery values

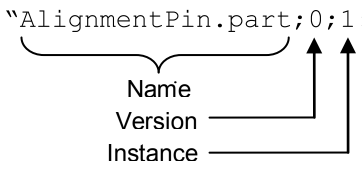
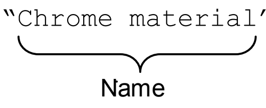
JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
Chordal::	Chordal deviation tessellation tolerance in MCS units for each LOD. Measure of maximum allowable distance a linear approximation for a curve/surface may deviate from the true curve/surface. Encoded value string would look as follows for the case of two LODs: —0.045603 0.069245”	MbString	space separated F32 values	Numeric
Angular::	Angular tessellation tolerance for each LOD in degrees. Two consecutive segments in a linear approximation of a curve/surface form an angle; this value specifies the maximum angle allowed. Encoded value string would look as follows for the case of two LODs: —3.000000 40.000000”	MbString	space separated F32 values	Numeric

14.8.4 Miscellaneous Properties

The below table documents some miscellaneous properties often placed on various nodes in the LSG to communicate specific information about the node or its contents.

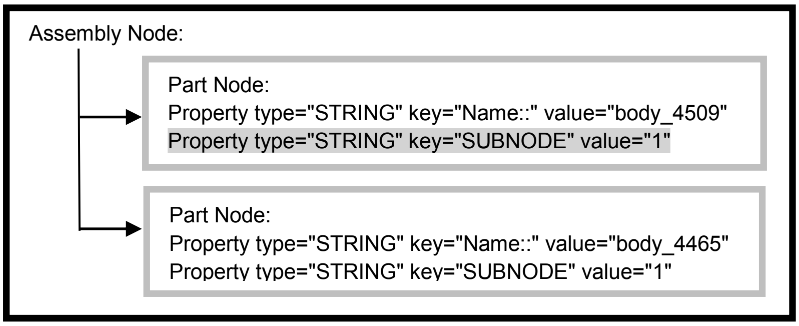
Table 110 — Miscellaneous Proptery values

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
PMI_TYPE_TABLE	May be attached to Part Node Element to indicate the list of PMI type values and associated names	MbString	<string>	

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
	for all PMI types (basically equivalent to the Entity Type field documented in Generic PMI Entities). The string is a ―.” and ―,” delimited string of the following form: ―10.Groove Weld,11.Fillet Weld,12.Plug/Slot Weld,14.Edge Weld”			
JT_PROP_SHAPE_DATA_TYPE	May be attached to Shape Node Elements to indicate what geometry type the shape data represents.	MbString	<string>	―Surface” ―Wire”
~~JT_PROP_TRISTRIP_DATA_LAYOUT~~	This property is deprecated, and is no longer used.			
JT_PROP_ORIGINATING_BREPTYPE	May be attached to Part Node Element to indicate the type of B-Rep associated with the Part.	MbString	<string>	―None” ―JtBrep” ―XtBrep”
JT_PROP_NAME	May be attached to any form of node or attribute with which one wants to associate a textual name (e.g. Part/Assembly/Instance name, Material name, Light Set name, etc.). For Part/Assembly/Instance names this string has the following encoded form where ―;” is a delimiter and ―:‘ is a terminator:  For attribute names this string has the following encoded form:  	MbString	<string>	

14.8.5 The SUBNODE property

A SUBNODE property can be defined on a part or assembly node in a JT file. By convention, the node which has the property defined is considered to be part of the parent as opposed to being a normal node entity in an assembly. SUBNODE properties can be used to represent a number of CAD constructs in JT files including reference sets.



Assembly Node:
Part Node:
Property type="STRING" key="Name::" value="body_4509"
Property type="STRING" key="SUBNODE" value="1"
Part Node:
Property type="STRING" key="Name::" value="body_4465"
Property type="STRING" key="SUBNODE" value="1"

Figure 257 — Assembly node with SUBNODE

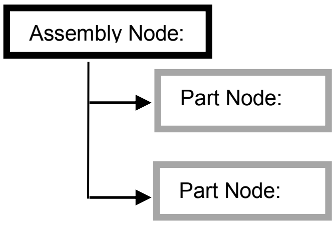
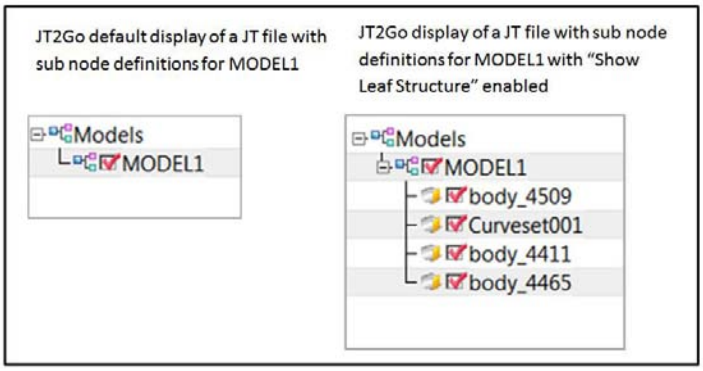


Figure 258 — Assembly node without SUBNODE

Best practice for JT viewing systems is by default to not have the trees structure for a node containing sub node definitions expanded.



JT2Go default display of a JT file with sub node definitions for MODEL1

JT2Go display of a JT file with sub node definitions for MODEL1 with "Show Leaf Structure" enabled

Figure 259 — Displaying Nodes that have SUBNODE properties

The node containing the SUBNODE property can be either a part node or an assembly node.

The value assigned to the SUBNODE property has no specific meaning. Convention is to set the value for the property to a string value of —"1 Parts that contain the SUBNODE property must contain a string property with a name for the part.   SUBNODE Property

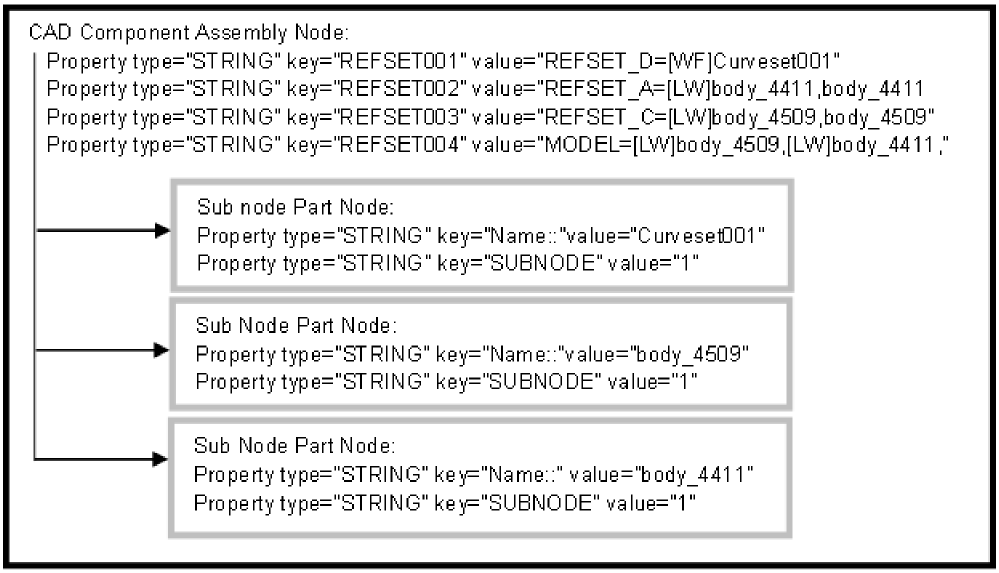
Table 111 — SUBNODE Property

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
SUBNODE	Specificies the node as being a sub node of the parent assembly	MbString	<string>	0 or 1

14.8.6 Reference Sets and the Reference Set Property

Many CAD systems are able to create a modeled construct whereby a single CAD part can contain a user defined substructure of geometry and PMI. A CAD part constructed this way is said to have reference sets or reference geometry. In JT a CAD part constructed this way, by convention, is referred to as a CAD component.

A JT CAD component is a unique assembly structure whereby the top level part in the assembly is the CAD Component and the part(s) that make up the assembly are sub nodes. When this arrangement exists in a JT file the CAD Component is said to contain Reference Sets. Reference Set definitions are a JT convention made up of assembly nodes, part nodes and properties. The part or assembly nodes defined in the CAD Component assembly are the actual reference set definitions.



```
CAD Component Assembly Node:
  Property type="STRING" key="REFSET001" value="REFSET_D=[WF]Curveset001"
  Property type="STRING" key="REFSET002" value="REFSET_A=[LW]body_4411,body_4411
  Property type="STRING" key="REFSET003" value="REFSET_C=[LW]body_4509,body_4509"
  Property type="STRING" key="REFSET004" value="MODEL=[LW]body_4509,[LW]body_4411,"

      Sub node Part Node:
      Property type="STRING" key="Name::"value="Curveset001"
      Property type="STRING" key="SUBNODE" value="1"

      Sub Node Part Node:
      Property type="STRING" key="Name::"value="body_4509"
      Property type="STRING" key="SUBNODE" value="1"

      Sub Node Part Node:
      Property type="STRING" key="Name::" value="body_4411"
      Property type="STRING" key="SUBNODE" value="1"
```

Figure 260 — CAD Component with Reference sets

This representation of the CAD Component structure is achieved through the use of the REFSET<XXX> property in the CAD Component part and the SUBNODE property in the parts that contain the reference set definitions.

The reference set property key has the form REFSETXXX with the XXX being a three digit incrementing number starting at 001. The numbers must be assigned concurrently. Up to 999 references set properties can be assigned for a CAD component. A reference set property is a comma delimited string of part names and hints. These parts can contain geometry such as solid bodies, wireframe or points that represent content relevant to the owning CAD component. To facilitate identification of the reference set content, hint strings can be combined with the part names. There are four reserved hint strings for reference sets; PMI, PT (point cloud), WF (wireframe) and LW (light weight, facet only).

Reference set encoding conventions

- Reference set names are not case sensitive
- The values = , and \ need to be escaped within the values of the property keys if they are in a part name or reference set name

Table 112 — Reference Set **Properties**, provides a description of the Reference set property values.

Table 112 — Reference Set Properties

Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
REFSETXXX	Specifies the string of part names that are included in the reference set. The property value defines the reference set name that is displayed in viewing systems	MbString	<string>	Comma delimited string of part names and hints

Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
	i.e.; type="STRING" key="REFSET<XXX>" value="REFSET_A_D=[WF]<ref set name>,[LW]<ref set name>,[PT]<ref set name>,<ref set name>,[PMI]" There are 3 reserved strings that can be added as prefixes to the JT part names included on the reference set string. They are; [PT] : precedes the referenced part name(s) that contain point cloud geometry [WF] : precedes the referenced part name(s) that contain wireframe geometry [LW] : precedes the referenced part name(s) that contain only facet geometry representations. B-Rep is not present. The reference hint string [PMI] is included without part names. When the [PMI] hint is included corresponding PMI entities with the JTTK_MULTICAD_REFSET property may exist. See table 2			

Table 113 — Properties related to the use of Reference **Sets**, provides a list of additional properties that assist applications with using and displaying reference sets.

Table 113 — Properties related to the use of Reference Sets

Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
REFSET_META	Contains a comma delimited list of aliased reference sets via the following convention; REFSET_META=”alias_1=name_1,alias_2= name_2 An aliased reference sets is a reference set definition used to record a group of reference sets that crosses multiple CAD Components.	MbString	<string>	Comma delimited string
JTTK_MULTICAD_REFSET	This property must be set on the PMI To have PMI visible within a reference it must have a JTTK_MULTICAD_REFSET property defined. The property is a string of Reference set names that the PMI will be visible in. i.e. type="STRING" key="	MbString	<string>	Comma delimited string

	JTTK_MULTICAD_REFSET" value=―<refset name>, <refset name>”			

The REFSET_CURRENT property is defined in instance nodes that are instances of a CAD Component. The property contains a string value that is the name of a reference set that exists in the CAD Component that has been instanced. Best practice is to use this property to determine the reference set that will become active if the model is set to the ―as saved” state.

Table 114 — REFSET_CURRENT property

Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
REFSET_CURRENT	Defines which reference set will be displayed in an instance node i.e. type=”STRING” key=”REFSET_CURRENT”value =<refset name>	MbString	<string>	Comma delimited string

14.9 LSG Attribute Accumulation Semantics

For applications producing or consuming JT format data, it is important that the JT format semantics of how attributes are meant to be applied and accumulated down the LSG are followed. If not followed, then consistency between the applications in terms of 3D positioning and rendering of LSG model data will not be achieved.

Although each attribute type defines its own application and accumulation LSG semantics (the details of which can be found in each attribute type sub-section under 6.1.2 Attribute Elements), there are some general rules which apply:

Attributes at lower level in the LSG take precedence and replace or accumulate with attributes set at higher levels. When multiple Attributes of the same type are present on a Node, they accumulate in the order they are specified (i.e. from the front of the Attribute list toward the back).

Nodes with no associated attributes inherit those of their parents.

Attributes are inherited only from a node's parents. Thus a given node‘s attributes do not affect those on the node's siblings.

The root of a partition inherits the attributes in effect at the referring partition node.

Attributes can be marked ―final”, which terminates accumulation of that attribute type at that marked attribute and propagates the accumulated value at that point to all descendants of the associated node. Descendants can override a "final” attribute using the ―force” flag. Note that ―force” does not turn OFF ―final” – it is simply a one-shot override of ―final” for the specific attribute marked as ―forcing.” Multiple attributes of the same type may be marked as "forcing" and in this case, the last one wins. Both of these flags are OFF by default. An analogy for this ―force” and ―final” interaction is that ―final” is a back-door in the attribute accumulation semantics, and that ―force” is the doggy-door in the back-door!

14.10 LSG Part Structure

The JT Format Reference does not mandate that a particular node hierarchy be used for modeling physical Parts within a LSG structure. In fact there are many node hierarchies for representing Parts in LSG that will function correctly in most JT enabled applications. Still, there is a convention that most JT translators follow (and some JT enabled applications may assume exists) for modeling Parts within a LSG. The convention is to model each Part within a LSG structure with the following node hierarchy:

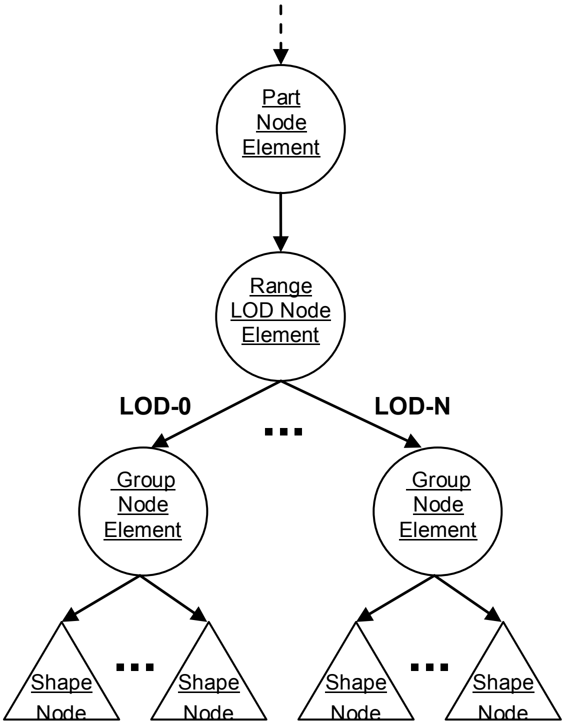


Figure 261 — JT Format Convention for Modeling each Part in LSG

14.11 Range LOD Node Alternative Rep Selection

Best practices suggest that LSG traversers apply the following strategy, at Range LOD Nodes (see 6.1.1.8 Range LOD Node Element), when making alternative representation selection decisions based on Range Limits: The first alternate representation is valid when the world coordinate distance between the centre and the eye point is less than or equal to the first range limit (and when no range limits are specified). The second alternate representation is valid when the distance is greater than the first limit and less than or equal to the second limit, and so on. The last alternate representation is valid for all distances greater than the last specified limit.

14.12 Brep Face Group Associations

The original purpose of the face group concept was to provide associativity between Brep faces and geometry. Exactly how a Brep face associates to a face group number is the topic of this section. An implicit scheme has been chosen for face group associativity, rather than storing some kind of explicit data on either the Vertex Shape LOD Data or the Brep. The primary motivation for this implicit scheme is to keep the JT files simple and small; additional association information would not only be redundant, but also wasteful. Tessellators shall exercise this policy when producing Vertex Shape LOD Data from Breps, grouping the triangles into face groups according to its rules. Tristrips may not cross face groups. Applications shall be able to count on this policy so that, for example, they can map a picking action back to its corresponding Brep face reliably.

JTBrep/ULP: In the case of JtBrep and ULP reps, the mapping is simple. These Reps have a consistent, sequential, index origin-0 numbering scheme for their regions, shells, and faces. So the Brep faces are simply assigned sequentially to face group by increasing region and shell. For example, suppose we have a JTBrep with 2 regions, each with 2 shells, each with 2 faces. The Face Group ⇔ Region/Shell/Face mapping will be as follows:

```
FG0 ⇔ R0 S0 F0
FG1 ⇔ R0 S0 F1
FG2 ⇔ R0 S1 F0
FG3 ⇔ R0 S1 F1
FG4 ⇔ R1 S0 F0
```

```
FG5 ⇔ R1 S0 F1
FG6 ⇔ R1 S1 F0
FG7 ⇔ R1 S1 F1
```

JtXTBrep: In the case of JtXTBrep, the mapping is based on an identifier of each XT face that is persisted on disk. The identifier is unique within each XT body, but it is not an index. XTBrep maintains a zero-based contiguous index of XT face based on increasing identifier value within the same XT body. In the case when multiple bodies are present in JtXTBrep, face index is assigned sequentially by increasing XT body index. For example, suppose we have a JtXTBrep with 2 bodies, each with 2 faces, then the Face Group to Body/Face mapping will be as follows:

```
FG0 ⇔ B0 F0
FG1 ⇔ B0 F1
FG2 ⇔ B1 F0
FG3 ⇔ B1 F1
```

Annex A

Object Type Identifiers

All objects stored in a JT file are classified by type and thus include an object type identifier as part of their persisted data. The data format for these Object Type identifiers is a GUID. These Object Type identifiers are consistent for all objects, of a particular type, in all Version 8.1 JT files.

Table A.1 — Object Type Identifiers lists the assigned identifier for each Object Type that can exist in a Version 10.0 JT file.

Table A.1 — Object Type Identifiers

GUID	Object Type
0xffffffff, 0xffff, 0xffff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff	Identifier to signal End-Of-Elements.
Types Stored Within LSG Segment (Segment Type = 1)	
0x10dd1035, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Node Element
0x10dd101b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Group Node Element
0x10dd102a, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Instance Node Element
0x10dd102c, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	LOD Node Element
0xce357245, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	Meta Data Node Element
0xd239e7b6, 0xdd77, 0x4289, 0xa0, 0x7d, 0xb0, 0xee, 0x79, 0xf7, 0x94, 0x94	NULL Shape Node Element
0xce357244, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	Part Node Element
0x10dd103e, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Partition Node Element
0x10dd104c, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Range LOD Node Element
0x10dd10f3, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Switch Node Element
0x10dd1059, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Shape Node Element
0x98134716, 0x0010, 0x0818, 0x19, 0x98, 0x08, 0x00, 0x09, 0x83, 0x5d, 0x5a	Point Set Shape Node Element
0x10dd1048, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Polygon Set Shape Node Element
0x10dd1046, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Polyline Set Shape Node Element
0xe40373c1, 0x1ad9, 0x11d3, 0x9d, 0xaf, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2	Primitive Set Shape Node Element

0x10dd1077, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Tri-Strip Set Shape Node Element
0x10dd107f, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Vertex Shape Node Element
0x10dd1001, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Attribute Data
0x10dd1014, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Draw Style Attribute Element
0x10dd1083, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Geometric Transform Attribute Element
0x10dd1028, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Infinite Light Attribute Element
0x10dd1096, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Light Set Attribute Element
0x10dd10c4, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Linestyle Attribute Element
0x10dd1030, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Material Attribute Element
0x10dd1045, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Point Light Attribute Element
0x8d57c010, 0xe5cb, 0x11d4, 0x84, 0xe, 0x00, 0xa0, 0xd2, 0x18, 0x2f, 0x9d	Pointstyle Attribute Element
0x10dd1073, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Texture Image Attribute Element
0xaa1b831d, 0x6e47, 0x4fee, 0xa8, 0x65, 0xcd, 0x7e, 0x1f, 0x2f, 0x39, 0xdc	Texture Coordinate Generator Attribute Element
0xa3cfb921, 0xbdeb, 0x48d7, 0xb3, 0x96, 0x8b, 0x8d, 0xe, 0xf4, 0x85, 0xa0	Mapping Plane Element
0x3e70739d, 0x8cb0, 0x41ef, 0x84, 0x5c, 0xa1, 0x98, 0xd4, 0x0, 0x3b, 0x3f	Mapping Cylinder Element
0x72475fd1, 0x2823, 0x4219, 0xa0, 0x6c, 0xd9, 0xe6, 0xe3, 0x9a, 0x45, 0xc1	Mapping Sphere Element
0x92f5b094, 0x6499, 0x4d2d, 0x92, 0xaa, 0x60, 0xd0, 0x5a, 0x44, 0x32, 0xcf	Mapping TriPlanar Element
0x10dd104b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Property Atom Element
0xce357246, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	Date Property Atom Element
0x10dd102b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Integer Property Atom Element
0x10dd1019, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Floating Point Property Atom Element
0xe0b05be5, 0xfbbd, 0x11d1, 0xa3, 0xa7, 0x00, 0xaa, 0x00, 0xd1, 0x09, 0x54	Late Loaded Property Atom Element
0x10dd1004, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	JT Object Reference Property Atom Element
0x10dd106e, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	String Property Atom Element
Types Stored Within JT B-Rep Segment (Segment Type = 2)	
0x873a70c0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	JT B-Rep Element

Types Stored Within Meta Data Segment (Segment Type = 4)	
0xce357249, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	PMI Manager Meta Data Element
0xce357247, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	Property Proxy Meta Data Element
Types Stored Within Shape LOD Segment (Segment Type = 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)	
0x3e637aed, 0x2a89, 0x41f8, 0xa9, 0xfd, 0x55, 0x37, 0x37, 0x3, 0x96, 0x82	Null Shape LOD Element
0x98134716, 0x0011, 0x0818, 0x19, 0x98, 0x08, 0x00, 0x09, 0x83, 0x5d, 0x5a	Point Set Shape LOD Element
0x10dd10a1, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Polyline Set Shape LOD Element
0xe40373c2, 0x1ad9, 0x11d3, 0x9d, 0xaf, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2	Primitive Set Shape Element
0x10dd10ab, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Tri-Strip Set Shape LOD Element
0x10dd109f, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Polygon Set LOD Element
0x10dd10b0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Vertex Shape LOD Element
Types Stored Within XT B-Rep Segment (Segment Type = 17)	
0x873a70e0, 0x2ac9, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	XT B-Rep Element
Types Stored Within Wireframe Segment (Segment Type = 18)	
0x873a70d0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Wireframe Rep Element
Types Stored Within JT ULP Segment (Segment Type = 20)	
0xf338a4af, 0xd7d2, 0x41c5, 0xbc, 0xf2, 0xc5, 0x5a, 0x88, 0xb2, 0x1e, 0x73	JT ULP Element
Types Stored Within JT LWPA Segment (Segment Type = 24)	
0xd67f8ea8, 0xf524, 0x4879, 0x92, 0x8c, 0x4c, 0x3a, 0x56, 0x1f, 0xb9, 0x3a	JT LWPA Element

Annex B

Coding Algorithms – An Implementation

This Appendix provides a sample C++ implementation for the encoding and decoding portion of the various compression CODECs (as detailed in 13.2 Encoding Algorithms) used in the JT format. This sample code is not intended to be fully functional encoder/decoder class implementations, but is instead intended to demonstrate the fundamentals of implementing the encoding/decoding portion of the CODEC algorithms used in the JT format.

B.1 Common classes

The following sub-sections define some general classes used by all the coding algorithms.

B.1.1 CntxEntryBase class

```
//
// Type used to build probability context tables.
// Used by ProbabilityContext class.
//
class CntxEntryBase
{
  public:

    // ---------- Housekeeping ----------
    CntxEntryBase() : _bIsEscape(false), _cCount(-1), _cCumCount(-1) {};
    CntxEntryBase(Bool bIsEsc, Int32 cCount)
        : _bIsEscape(bIsEsc), _cCount(cCount), _cCumCount(-1) {};
    CntxEntryBase(const CntxEntryBase &rhs) { *this = rhs; }
    ~CntxEntryBase() {};
    CntxEntryBase &operator=(const CntxEntryBase &rhs)
      { _bIsEscape = rhs._bIsEscape;
        _cCount    = rhs._cCount;
        _cCumCount = rhs._cCumCount;
        return *this;
      }

    // ---------- Operations Interface ----------
    Bool isEscape() const
      { return _bIsEscape; }
    Int32 operator==(const CntxEntryBase2 &rhs) const
      { return (_iSym == rhs._iSym); };

  public:

    // ---------- Member Variables ----------
    Int32         _cCount;    // Number of occurrences
    Int32         _cCumCount; // Cumulative number of occurrences
    Bool          _bIsEscape; // True if this symbol is the escape symbol
};

template <class ValueType>
class CntxEntry : public CntxEntryBase
{
  public:

    // ---------- Housekeeping ----------
    CntxEntry() : CntxEntryBase(), _val(ValueType()) {};

    CntxEntry( Bool bIsEsc, Int32 cCount, const ValueType &val ):
        CntxEntryBase(bIsEsc, cCount), _val(val) {};

    CntxEntry( const CntxEntry &rhs ) { *this = rhs; }
    ~CntxEntry() {};
    CntxEntry &operator=(const CntxEntry &rhs)
        { _val = rhs._val;
```

```
         CntxEntryBase::operator= (rhs);
         return *this;
        }
    Int32 operator==(const CntxEntry &rhs) const
        { return (_iSym == rhs._iSym); };

  public:

    // ---------- Member Variables ----------
    ValueType       _val;     // Associated value
};
```

B.1.2 ProbContext class

```
//
// Type used to build probability context tables.
// Used by CodecDriver class.
//
template <class ValueType>
class ProbContext
{
  public:
    typedef CntxEntry< ValueType > CntxEntryV;

    // ---------- Housekeeping ----------
    ProbContext();
    ProbContext(const ProbContext &rhs);
    ~ProbContext();
    ProbContext &operator=(const ProbContext &rhs);
    Bool operator==(const ProbContext &rhs) const;
    enum { cMaxCntxCount = 8192 };

    // ---------- Accessor Interface ----------
    Int32 totalCount() const
        { return _cTotalCount; }
    Int32 numEntries() const
        { return _vEntries.length(); }
    Bool getEntry(Int32 iEntry, const CntxEntryBase *&rpEntry) const
        { const CntxEntryV *aEntries = _vEntries.ptr();
         rpEntry = &aEntries[iEntry]; return True; }
    Bool getEntryV(Int32 iEntry, const CntxEntryV *&rpEntry) const
        { const CntxEntryV *aEntries =  _vEntries.ptr();
         rpEntry = &aEntries[iEntry]; return True; }
    Bool getEntryV(Int32 iEntry, CntxEntryV *&rpEntry)
        { CntxEntryV *aEntries = _vEntries.ptr();
         rpEntry = &aEntries[iEntry]; return True; }

    // ---------- Lookup Interface ----------
    Bool lookupValue(const ValueType &rValue, const CntxEntryV *&opCntxEntry) const;
    Bool lookupEntryByCumCount(Int32 iCount, const CntxEntryV *&opCntxEntry) const;

    // ---------- Reorganizing Interface ----------
    Bool accumulateCounts();
    Bool sortByValue();

  protected:
    Vec< CntxEntryV >  _vEntries;
    Int32              _cTotalCount;
    Int32              _iEscPosCache;

    static int _compareCounts(const void *pVal1,
                              const void *pVal2,
                              const void *uData);
};

template <class ValueType>
Bool ProbContext::lookupValue( const ValueType   &rValue,
                               const CntxEntryV *&opCntxEntry ) const
{
    // If we do not find the value, then return NULL for the entry
    opCntxEntry = NULL;
    // If the escape position is not cached, sort the context by value
    // and then set it.  Then, we can binary search for values.  We do
    // this because translateValuesToSymbols() will call this method in
    // a tight loop.  Anything is better than linear search!
    ProbContext *pThis = (ProbContext*) this;
    CntxEntryV *pEntries = pThis->_vEntries.ptr();
    Int32 nEntries = _vEntries.length();
    if (_iEscPosCache == -1) {
        // Search for the escape symbol
        Bool bFoundEsc = False;
        for (Int32 i = 0 ; i < nEntries ; i++) {
            if (pEntries[i].isEscape()) {
                // Move the escape symbol to context slot 0
                ::swap(pEntries[0], pEntries[i]);
                bFoundEsc = True;
                break;
            }
        }
        // Sort by value
        if (bFoundEsc) {
            // Sort by value _except_ leave the escape symbol in slot 0
            ::sort(&pEntries[1], nEntries-1, FtorCntxValue<ValueType>());
            pThis->accumulateCounts();
            pThis->_iEscPosCache = 0;
        }
        else {
            pThis->sortByValue();
            pThis->_iEscPosCache = -2;
        }
    }

    // Binary search for rValue!
    Int32 l = (_iEscPosCache == 0),
          h = nEntries - 1,
          m;
    while (l <= h) {
        m = (l + h) >> 1;
        if (pEntries[m]._val == rValue) {
            opCntxEntry = &pEntries[m];
            return True;
        }
        else if (pEntries[m]._val < rValue)
            l = m + 1;
        else
            h = m - 1;
    }

    // If we don't find the value, then we return the position of
    // the escape symbol.
    if (_iEscPosCache >= 0)
        opCntxEntry = &pEntries[_iEscPosCache];

    return True;
}

template <class ValueType>
Bool ProbContext2::lookupEntryByCumCount(Int32 iCount, const CntxEntryV *&opCntxEntry )
const
{
    const CntxEntryV *aEntries = _vEntries.ptr();
    const Int32       nEntries = _vEntries.length();

    const Int32       seqSearchLen = 4;
    Int32 ii=0;
    opCntxEntry = NULL;

    // For short lists, do sequential search
    if ( nEntries <= (seqSearchLen*2) ) {
        ii = 0;
        while ((iCount>=(aEntries[ii]._cCumCount + aEntries[ii]._cCount)) &&
               (ii<nEntries))
        {
            ii++;
        }

        if ( ii >= nEntries ) {
            Assert( 0 && "Bad probability table" );
        }
        opCntxEntry = &aEntries[ii];
    }

    // For long lists, do a short sequential searches through most likely
    // elements, then do a binary search through the rest.
    else {
        for (ii=0; ii<seqSearchLen; ii++) {
            if (iCount < (aEntries[ii]._cCumCount + aEntries[ii]._cCount)) {
                opCntxEntry = &aEntries[ii];
                return True;
            }
        }

        Int32 low=ii, high=nEntries-1, mid;
        while(1) {
            if ( high < low ) {
                break;
            }
            mid = low + ((high-low)>>1);

            if ( iCount < aEntries[mid]._cCumCount ) {
                high = mid-1;
                continue;
            }
            if ( iCount >= (aEntries[mid]._cCumCount + aEntries[mid]._cCount) ) {
                low = mid+1;
                continue;
            }

            opCntxEntry = &aEntries[mid];
            return True;
        }
        Assert( 0 && "Bad probability table" );
    }

    return True;
}

template <class ValueType>
Bool ProbContext2::accumulateCounts()
{
    // Check for zero length context
    CntxEntryV *aEntries = _vEntries.ptr();
    Int32       nEntries = _vEntries.length();
    if ( nEntries == 0 ) {
        _cTotalCount = 0;
        return True;
    }

    // Accumulate counts in _cCumCount for entries 1 and higher
    aEntries[0]._cCumCount = 0;
    Int32 ii;
    for ( ii=1 ; ii<nEntries ; ii++ ) {
        aEntries[ii]._cCumCount = aEntries[ii-1]._cCount + aEntries[ii-1]._cCumCount;
    }

    // Set the total count for the context
    _cTotalCount = aEntries[ii-1]._cCount + aEntries[ii-1]._cCumCount;

    return True;
}

template <class ValueType>
struct FtorCntxValue
{
   Bool operator () (const CntxEntry<ValueType>& l, const CntxEntry<ValueType>& r) const
   { return  (l._val < r._val); }
};

template <class ValueType>
Bool ProbContext2::sortByValue()
{
   // Sort the entries in order of values from smallest to largest
   ProbContextV *pThis = (ProbContextV*) this;
   sort( _vEntries.ptr(), (size_t)(_vEntries.length()), FtorCntxValue<ValueType>() );

   pThis->_iEscPosCache = -1;
   pThis->accumulateCounts();

   return True;
}
```

B.1.3 CodecDriver class

```
//
// A class that deals with the conversions from SYMBOL to VALUE and
// provides end-consumer APIs for using the codecs.
//
template <class ValueType>
class CodecDriver
{
  public:

    // ---------- Internal Types ----------
    typedef enum {
        CodecNull       = 0,    // Null Codec
        CodecBitLength  = 1,    // Bitlength Codec
        CodecArithmetic = 3,    // Arightmetic Codec
        CodecChopper    = 4,    // Chopper Pseudo-codec
        CodecMTF        = 5     // Move-to-front Pseudo-codec
    } CodecType;
    // Type of value predictor
    typedef enum {
            PredLag1       = 0, // Predicts as last values
            PredXor1       = 1, // Predict as last, but use xor instead of subtract
            PredNULL       = 2, // No prediction.
    } PredictorType;

    static Bool unpackResiduals(const Veci    &rvResidual,
                                Veci          &rvVals,
                                PredictorType  ePredType);
    static Bool unpackResiduals(const Vecu    &rvResidual,
                                Vecu          &rvVals,
                                PredictorType  ePredType);

    static Float64 log2( Float64 x ) { return (log(x) / 0.6931471805599453); }

    //////////////////////////////
    // Convenience Methods
    //////////////////////////////

  protected:
    static Int32 _predictValue( const Int32 *vVal, Int32 iIndex,
                                PredictorType ePredType );
};

Bool CodecDriver::unpackResiduals( const Veci      &rvResidual,
                                   Veci            &rvVals,
                                   PredictorType    ePredType )
{
    const Int32 len = rvResidual.length();
    Int32 iPredicted;
    rvVals.setLength(len);
    Int32 *aVals = rvVals.ptr();
    const Int32 *aResidual   = rvResidual.ptr();
    for ( Int32 i = 0 ; i < len ; i++ ) {
        if (i < 4) {
            // The first four values are just primers
            aVals[i] = aResidual[i];
        } else {
            // Get a predicted value
            iPredicted = _predictValue(rvVals.ptr(), i, ePredType);

            if (ePredType == PredXor1) {
                // Encode the residual as the current value XOR predicted
                aVals[i] = aResidual[i] ^ iPredicted;
            } else  {
                // Encode the residual as the current value plus predicted
                aVals[i] = aResidual[i] + iPredicted;
            }
        }
    }

    return True;
}

Bool
CodecDriver2::unpackResiduals( const Vecu     &rvResidual,
                               Vecu           &rvVals,
                               PredictorType  ePredType )
{
    return unpackResiduals(*((const Veci*)&rvResidual),
                          *((Veci*) &rvVals),
                          ePredType);
}

Int32
CodecDriver2::_predictValue( const Int32*  paVals,
                             Int32         iIndex,
                             PredictorType ePredType )
{
    Int32 iPredicted = 0;
    switch (ePredType) {

        default:
        case PredLag1:      // Predicts as last value
        case PredXor1:      // Predicts as last value
            iPredicted    = paVals[iIndex-1];
            break;
    }

    return iPredicted;
}
```

B.2 Bitlength decoding classe

The following sub-section contains a sample implementation of the decoding portion of the Bitlength CODEC algorithm. A summary technical explanation of the Bitlength CODEC can be found in 13.2.2 Bitlength CODEC.

B.2.1 BitLengthCodec class

```
template <class ValueType>
class BitLengthCodec : public Codec<ValueType>
{
  public:
    typedef Vec< ValueType >               VecValue;
    typedef ProbContext< ValueType >       ProbContextV;
    typedef CodecDataCntx< ValueType >     CodecDataCntxV;

    Bool encode( const VecValue &vValues,
                 VecValue       &ovOOBValues,
                 Vecu           &ovCodeText,
                 Int32          &onBitsCodeText,
                 ProbContextV   *pProbCntx       );
    Bool decode( Int32           nValues,
                 VecValue       &ovOOBValues,
                 const Vecu     &vCodeText,
                 Int32           nBitsCodeText,
                 VecValue       &ovValues,
                 ProbContextV   *pProbCntx       );

  protected:
    Int32 _nBitsInSymbol(Int32 iSymbol) const;
    Bool getNextCodeText (UInt32 &uCodeText, Int32 &nBits);

    Vecu     *_pvCodeText;
    Int32    *_pcCodeTextLen;
    Int32     _iCurCodeText;
    Vecus     _vnValBits;
};

#define GetSignedBits(iOut, n)                       \
    GetUnsignedBits(iOut,n)                          \
    iOut <<= (32 - n);                               \
    iOut >>= (32 - n);

#define GetUnsignedBits(uOut, n)                     \
    if (n == 0) uOut = 0;                            \
    else if (nValBits >= n) {                        \
        uOut = uVal >> (32 - n);                     \
        uVal   <<= n;                                \
        uVal &= (n==32)-1;                           \
        nValBits -= n;                               \
        nBits    += n;                               \
    }                                                \
    else {                                           \
        Int32 _nLBits = nValBits;                    \
        uOut = uVal >> (32 - n);                     \
        nBits    += _nLBits;                         \
        getNextCodeText (uVal, nValBits);            \
        Int32 _nRBits = (n - _nLBits);               \
        uOut |= uVal >> (32 - _nRBits);              \
        uVal   <<= _nRBits;                          \
        uVal &= (_nRBits==32)-1;                     \
        nValBits -= _nRBits;                         \
        nBits    += _nRBits;                         \
    }

Template <class ValueType>
Bool BitLengthCodec::encode(const VecValue   &vValues,
                        VecValue          &ovOOBValues,
                        Vecu              &ovCodeText,
                        Int32             &onBitsCodeText,
                        ProbContextV      *)
{
    Int32  i, j, k;
    Int32  iSymbol;               // Symbol value to encode
    Int32  cSymBits = 0,          // Number of bits in iSymbol
           nValues;               // Number of values to encode
    // Initialize output state
    ovOOBValues.setLength(0);
    ovCodeText.setLength(0);
    onBitsCodeText = 0;
    _pvCodeText = &ovCodeText;
    _pcCodeTextLen = &onBitsCodeText;

    // Short circuit for null array of values
    nValues = vValues.length();
    if (nValues <= 0)
        return True;

    _vnValBits.setLength(nValues);
    UInt16 *paiSymBits = _vnValBits.ptr();
    const ValueType *paiValues = vValues.ptr();
    // Find the minimum value and compute how many bits each value takes
    ValueType iMinSymbol = Limits<ValueType>::maxValue();
    ValueType iMaxSymbol = Limits<ValueType>::maxNegValue();
    Float64 fMean = 0.0;
    for (i = 0 ; i < nValues ; i++) {
        iMinSymbol = ::min(iMinSymbol, paiValues[i]);
        iMaxSymbol = ::max(iMaxSymbol, paiValues[i]);
        fMean += Float64(paiValues[i]);
    }
    fMean /= nValues;
    ValueType iMean = Int32(fMean);
    for (i = 0 ; i < nValues ; i++) {
        paiSymBits[i] = bitsize(paiValues[i] - iMean);
    }
    // A "block" is: 3 bits of number of bits (repeats while value is either 3 or -4 for
larger width changes)
    //               4 bits of block length
    Int32 cBlkLenBits = 4; // Number of bits used to express a block length. 0 means 0.
    Int32 cBlkValBits = 4; // *Delta* number of bits to express the current field width
    Int32 cBlkHdrBits = cBlkLenBits + cBlkValBits;
    Bool bMerged;
    // Block-forming: Merge Down/Up blocks
    do {
        bMerged = JtFalse;
        Int32 iPrevRunPos = 0,
              iCurRunPos  = 0,
              iNextRunPos = 0;
        while (iCurRunPos < nValues - 1) {
            // Advance the next run
            iNextRunPos++;
            while (iNextRunPos < nValues - 1 &&
                   paiSymBits[iNextRunPos] == paiSymBits[iNextRunPos-1])
            {
                iNextRunPos++;
            }
            if (iNextRunPos >= nValues)
                break;
            Int32  wab = (iCurRunPos - iPrevRunPos),
                   wbc = (iNextRunPos - iCurRunPos);
            if (wab == 0) {
                iCurRunPos = iNextRunPos;
                continue;
            }
            else if (wbc == 0) {
                continue;
            }
            // If we've bitten off more than one block's worth of data
            // we must start a new block.  Length 0 is allowed because
            // we may need to insert multiple consecutive block headers
            // in order to change the field width more bits than can be
            // represented in cBlkValBits.
            if (wab > (1 << cBlkLenBits)) {
                iPrevRunPos += (1 << cBlkLenBits);
            }
            else if (wab == (1 << cBlkLenBits)) {
                iPrevRunPos = iCurRunPos;
                iCurRunPos  = iNextRunPos;
                continue;
            }
            UInt16 &ua = paiSymBits[iPrevRunPos],
                   &ub = paiSymBits[iCurRunPos],
                   &uc = paiSymBits[iNextRunPos];
            // If the runs go "down-up"
            if (ua > ub && ub < uc) {
                // Test if we should increase ub to the lesser of ua and uc
                if (ua < uc) {
                    // Test if we should increase ub to ua
                    if (wbc * (ua - ub) <= cBlkHdrBits) {
                        for (j = iCurRunPos ; j < iNextRunPos ; j++)
                            paiSymBits[j] = ua;
                        iCurRunPos = iNextRunPos;
                        continue;
                    }
                }
                else if (ua > uc) {
                    // Test if we should increase ub to uc
                    if (wbc * (uc - ub) <= cBlkHdrBits) {
                        for (j = iCurRunPos ; j < iNextRunPos ; j++)
                            paiSymBits[j] = uc;
                        iNextRunPos = iCurRunPos;
                        bMerged = True;
                        continue;
                    }
                }
                else {  // ua == uc
                    // Test if we should increase ub to ua/uc
                    if (wbc * (ua - ub) <= 2 * cBlkHdrBits) {
                        for (j = iCurRunPos ; j < iNextRunPos ; j++)
                            paiSymBits[j] = ua;
                        iCurRunPos  = iPrevRunPos;
                        iNextRunPos = iPrevRunPos;
                        bMerged = True;
                        continue;
                    }
                }
            }

            // Shift down the runs
            iPrevRunPos = iCurRunPos;
            iCurRunPos  = iNextRunPos;
        }
    } while (bMerged);
    // Block forming: Merge down/down and up/up runs
    do {
        bMerged = JtFalse;
        Int32 iPrevRunPos = 0,
              iCurRunPos  = 0,
              iNextRunPos = 0;
        while (iCurRunPos < nValues - 1) {
            // Advance the next run
            iNextRunPos++;
            while (iNextRunPos < nValues - 1 &&
                   paiSymBits[iNextRunPos] == paiSymBits[iNextRunPos-1])
            {
                iNextRunPos++;
            }
            if (iNextRunPos >= nValues)
                break;
            Int32  wab = (iCurRunPos - iPrevRunPos),
                   wbc = (iNextRunPos - iCurRunPos);
            if (wab == 0) {
                iCurRunPos = iNextRunPos;
                continue;
            }
            else if (wbc == 0) {
                continue;
            }
            // If we've bitten off more than one block's worth of data
            // we must start a new block.  Length 0 is allowed because
            // we may need to insert multiple consecutive block headers
            // in order to change the field width more bits than can be
            // represented in cBlkValBits.
            if (wab > (1 << cBlkLenBits)) {
                iPrevRunPos += (1 << cBlkLenBits);
            }
            else if (wab == (1 << cBlkLenBits)) {
                iPrevRunPos = iCurRunPos;
                iCurRunPos  = iNextRunPos;
                continue;
            }
            UInt16 &ua = paiSymBits[iPrevRunPos],
                   &ub = paiSymBits[iCurRunPos],
                   &uc = paiSymBits[iNextRunPos];
            // If the runs go "up-up"
            if (ua < ub && ub < uc) {
                // Test if we should increase ua to ub
                if (wab * (ub - ua) < cBlkHdrBits) {
                    for (j = iPrevRunPos ; j < iCurRunPos ; j++)
```

```
                        paiSymBits[j] = ub;
                    iCurRunPos = iNextRunPos;
                    bMerged = True;
                    continue;
                }
                // Test if we should increase ub to uc
                else if (wbc * (uc - ub) < cBlkHdrBits) {
                    for (j = iCurRunPos ; j < iNextRunPos ; j++)
                        paiSymBits[j] = uc;
                    iNextRunPos = iCurRunPos;
                    bMerged = True;
                    continue;
                }
            }
            // If the runs go "down-down"
            else if (ua > ub && ub > uc) {
                // Test if we should increase ub to ua
                if (wbc * (ua - ub) < cBlkHdrBits) {
                    for (j = iCurRunPos ; j < iNextRunPos ; j++)
                        paiSymBits[j] = ua;
                    iCurRunPos = iNextRunPos;
                    continue;
                }
            }

            // Shift down the runs
            iPrevRunPos = iCurRunPos;
            iCurRunPos  = iNextRunPos;
        }
    } while (bMerged);

    // Compute the total bits
    UInt32 cMaxBlkLen = (1 << cBlkLenBits) – 1;
    UInt32 iLastRunPos = 0;
    Int32 cTotalBits = paiSymBits[0] + cBlkHdrBits;
    for (UInt32 iCurRunPos = 1 ; iCurRunPos < nValues ; iCurRunPos++) {
        cTotalBits += paiSymBits[iCurRunPos];
        if (paiSymBits[iCurRunPos] != paiSymBits[iCurRunPos -1]) {
            UInt32 cNumBlks = ((iCurRunPos - iLastRunPos) + (cMaxBlkLen-1)) / cMaxBlkLen;
            cTotalBits += cNumBlks * cBlkHdrBits;
            iLastRunPos = iCurRunPos;
        }
    }
    UInt32 cValSpanBits = bitsize(UInt32(iMaxSymbol – iMinSymbol));
    UInt32 cFixedWidBits = nValues * cValSpanBits + (13+2*(cValSpanBits+1));

    /////
    // If the fixed-width total bits are better, then write out the values
    // in a single fixed-width format.
    /////
    if (cFixedWidBits < cTotalBits) {
        // Write the fixed-width tag
        addCodeText(0, 1);

        // Write the min and max symbols into the stream
        nibblerEmit (iMinSymbol);
        nibblerEmit (iMaxSymbol);

        // Iterate over the remaining symbols
        UInt32 uCodeText = 0;
        for (Int32 i = 0; i < nValues; i++ ) {

            // Get the next symbol
            iSymbol = paiValues[i];

            // Write it
            uCodeText = iSymbol - iMinSymbol;
            addCodeText(uCodeText, cValSpanBits);
        }
    }
    /////
    // Otherwise, encode with variable-length fields
    /////
    else {
        // Write the variable-width tag
        addCodeText(1, 1);

        // Write out the mean value
        nibblerEmit(iMean);

        // Set the initial field-width
        Int32 cMaxFieldDecr = -(1 << (cBlkValBits - 1)),       // -ve number
              cMaxFieldIncr =  (1 << (cBlkValBits - 1)) - 1;   // +ve number
        Int32 cCurFieldWidth = 0;
        Int32 cTargFieldWidth;
        for (Int32 ii = 0 ; ii < nValues ;) {
            // Adjust the current field width to the target field width
            cTargFieldWidth = paiSymBits[ii];
            {
                if (cCurFieldWidth <= cTargFieldWidth) {
                    while (cTargFieldWidth - cCurFieldWidth >= cMaxFieldIncr) {
                        addCodeText(cMaxFieldIncr, cBlkValBits);
                        cCurFieldWidth += cMaxFieldIncr;
                    }
                }
                else {
                    while (cTargFieldWidth - cCurFieldWidth <= cMaxFieldDecr) {
                        addCodeText(cMaxFieldDecr, cBlkValBits);
                        cCurFieldWidth += cMaxFieldDecr;
                    }
                }
                addCodeText(cTargFieldWidth - cCurFieldWidth, cBlkValBits);
                cCurFieldWidth = cTargFieldWidth;
            }

            // Write out the run length
            for (j = ii+1 ; j < ii + (1 << cBlkLenBits) - 1 && j < nValues ; j++)
                if (paiSymBits[ii] != paiSymBits[j])
                    break;
            addCodeText(j - ii, cBlkLenBits);

            // Write out the data bits for the run
            for (k = ii ; k < j ; k++)
                addCodeText(paiValues[k] - iMean, cCurFieldWidth);

            // Advance to the end of the run
            ii = j;
        }
    }

    return True;
}

template <class ValueType>
Bool BitLengthCodec2::decode( Int32           nValues,
                              const VecValue &,
                              const Vecu     &vCodeText,
                              Int32           nBitsCodeText,
                              VecValue       &ovValues,
                              ProbContextV   *)
{
    Int32  nTotalBits = 0;    // Total number of codetext bits expected
    ValueType iSymbol;        // Decoded symbol value
    Int32  cNumCurBits = 0;   // Current field width in bits
    ValueType iMinSymbol = 0; // The minimum symbol value.  Used as bias.
    ValueType iMaxSymbol = 0; // The maximum symbol value.  Used as bias.
    Int32  nSyms = 0;         // Number of symbols read so far
    ValueType *paiValues;     // Pointer into ovValues where we write decoded values

    // Get codetext from the driver and loop over it until it's gone!
    ovValues.setLength(nValues);
    paiValues = ovValues.ptr();

    _iCurCodeText = 0;
    _pvCodeText = (Vecu*) &vCodeText;
    _pcCodeTextLen = &nBitsCodeText;

    /////
    // If the fixed-width total bits are better, then write out the values
    // in a single fixed-width format.
    /////
    // Read the variable-width tag
    Int32 iTmp;
    GetUnsignedBits(iTmp, 1); // 0 = Fixed-width, 1 = Variable width
    if (iTmp == 0) {
        // Read the min and max symbols from the stream
        nibblerGet(iMinSymbol);
        nibblerGet(iMaxSymbol);
        cNumCurBits = bitsize(UInt32(iMaxSymbol – iMinSymbol));

        // Read each fixed-width field and output the value
        while (nBits < nTotalBits || nSyms < nValues) {
            GetUnsignedBits(iSymbol, cNumCurBits);
            iSymbol += iMinSymbol;
            *paiValues++ = iSymbol;
            nSyms++;
        }
    }
    /////
    // Otherwise, encode with variable-length fields
    /////
    else {
        // Write out the mean value
        ValueType iMean;
        nibblerGet(iMean);

        // Set the initial field-width
        Int32 cMaxFieldDecr = -(1 << (cBlkValBits - 1)),       // -ve number
              cMaxFieldIncr =  (1 << (cBlkValBits - 1)) - 1;   // +ve number
        UInt32 cCurFieldWidth = 0, cRunLen, k;
        Int32 cDeltaFieldWidth;
        ValueType iTmp;
        for (Int32 ii = 0 ; ii < nValues ;) {
            // Adjust the current field width to the target field width
            do {
                GetSignedBits(cDeltaFieldWidth, cBlkValBits);
                cCurFieldWidth += cDeltaFieldWidth;
            } while (cDeltaFieldWidth == cMaxFieldDecr || cDeltaFieldWidth ==
cMaxFieldIncr);

            // Read in the run length
            GetUnsignedBits(cRunLen, cBlkLenBits);

            // Read in the data bits for the run
            for (k = ii ; k < ii + cRunLen ; k++) {
                GetSignedBits(iTmp, cCurFieldWidth);
                *paiValues++ = iTmp + iMean;
            }

            // Advance to the end of the run
            ii += cRunLen;
        }
    }

    // Assert that we have consumed exactly all of the bits
    Assert(nValBits == 0);
    Assert(uVal == 0);

    return True;
}

// Number of bits necessary to encode a SIGNED integer
UInt32 bitsize(Int32 x) const
{
    x = x ^ (x >> 31);
    return 33 – nlz(UInt32(x));
```

```
}

// Number of bits necessary to encode an UNsigned integer
UInt32 bitsize(UInt32 x) const
{
    return 32 – nlz(x);
}

// Number of Leading Zeros
UInt32 nlz(UInt32 x)
{
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >> 16);
    return popcnt(~x);
}

// Population count - # of 1 bits in x
UInt32 popcnt(UInt32 x)
{
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x + (x >> 4)) & 0x0f0f0f0f;
    x = x + (x >> 8);
    x = x + (x >> 16);
    return x & 0x3f;
}

Bool BitLengthCodec2::getNextCodeText (UInt32 &uCodeText, Int32 &nBits)
{
    uCodeText = _pvCodeText->value(_iCurCodeText);
    nBits = ::min(32, *_pcCodeTextLen - 32 * _iCurCodeText);
    _iCurCodeText++;
    return True;
}
```

B.3 Arithmetic decoding classes

The following sub-sections contain a sample implementation of the decoding portion of the Arithmetic CODEC algorithm. A summary technical explanation of the Arithmetic CODEC can be found in 13.2.3 Arithmetic CODEC.

B.3.1 ArithmeticCodec class

```
template <class ValueType>
class ArithmeticCodec: public Codec
{
  public:
    Bool encode( const VecValue &vValues,
                 VecValue       &ovOOBValues,
                 Vecu           &ovCodeText,
                 Int32          &onBitsCodeText,
                 ProbContextV   *pProbCntx       );
    Bool decode( Int32           nValues,
                 VecValue       &ovOOBValues,
                 const Vecu     &vCodeText,
                 Int32           nBitsCodeText,
                 VecValue       &ovValues,
                 ProbContextV   *pProbCntx       );

  protected:
    Bool _encodeSymbol(UInt16 uLowCt, UInt16 uHighCt, UInt16 uScale );
    Bool _flushEncoder();
    Bool _removeSymbolFromStream( UInt16 uLowCt, UInt16 uHighCt, UInt16 uScale );
    Bool _flushDecoder();

    Bool getNextCodeText (UInt32 &uCodeText, Int32 &nBits);
    UInt16  _code;              // Present input code value, for decoding only
    UInt16  _low;               // Start of the current code range
    UInt16  _high;              // End of the current code range
    Int32   _nUnderflowBits;    // Number of underflow bits pending

    Vecu   *_pvCodeText;
    Int32  *_pcCodeTextLen;
    Int32   _iCurCodeText;

    UInt32  _uBitBuff;
    Int32   _nBitBuff;
};

// Reads a bit and places it into ouBit
#define ReadBit(ouBit)                         \
    if (_nBitBuff==0) {                        \
        getNextCodeText(_uBitBuff, _nBitBuff); \
    }                                          \
    ouBit = (_uBitBuff >> 31);                 \
    _uBitBuff <<= 1;                           \
    _nBitBuff--;

// Reads a bit and ORs it into bit 0 of ouBit
#define ReadBit0(ouBit)                        \
    if (_nBitBuff==0) {                        \
        getNextCodeText(_uBitBuff, _nBitBuff); \
    }                                          \
    ouBit |= (_uBitBuff >> 31);                \
    _uBitBuff <<= 1;                           \
    _nBitBuff--;
// Writes bit 0 of uBit
#define WriteBit(uBit)                         \
    if (_nBitBuff==32) {                       \
        addCodeText(_uBitBuff, 32);            \
        _uBitBuff = _nBitBuff = 0;             \
    }                                          \
    _uBitBuff <<= 1;                           \
    _uBitBuff  |= (UInt32(uBit) & 0x1);        \
    _nBitBuff++;

Bool ArithmeticCodec::encode(const VecValue &vValues,
                             Veci           &ovOOBValues,
                             Vecu           &ovCodeText,
                             Int32          &onBitsCodeText,
                             ProbContextV   *pProbCntx)
{
    // Initialize output state
    ovOOBValues.setLength(0);
    ovCodeText.setLength(0);
    onBitsCodeText = 0;
    _pvCodeText = &ovCodeText;
    _pcCodeTextLen = &onBitsCodeText;

    // Initialize the encoder state
    _low  = 0x0000;
    _high = 0xffff;
    _nUnderflowBits = 0;

    // Prime the bit buffer
    _uBitBuff = 0;
    _nBitBuff = 0;

    const ValueType *paiValues = vValues.ptr();
    const CntxEntry *pEntry;
    Int32 nValues = vValues.length();
    Int32 cTotalCount = pProbCntx->totalCount();
    for (Int32 i = 0; i < nValues; i++ ) {
        // Look up the value in the prob context
        pProbCntx->lookupValue(paiValues[i], pEntry );

        // If this is not the null context, then we emit an escape symbol,
        // move the context it specifies, and restart the tranlation of
        // the same value.  Thus, a value may emit more than one symbol.
        if (pEntry->isEscape()) {
            ovOOBValues.append(paiValues[i]);
        }

        _encodeSymbol(pEntry->_cCumCount,
                      pEntry->_cCumCount + pEntry->_cCount,
                      cTotalCount));
    }

    _flushEncoder();

    return True;
}

Bool ArithmeticCodec2::_encodeSymbol(UInt16 uLowCt, UInt16 uHighCt, UInt16 uScale )
{
    // These three lines rescale _high and _low for the new symbol.
    UInt32 uRange = UInt32(_high - _low) + 1;
    _high  = _low + (uRange * uHighCt) / uScale - 1;
    _low   = _low + (uRange * uLowCt ) / uScale;

    // This loop turns out new bits until _high and _low are far enough
    // apart to have stabilized.
    for (;;) {
        // If this test passes, it means that the most signif digits match,
        // and can be sent to the output stream.
        if ( (_high & 0x8000) == (_low & 0x8000) )
        {
            // Flush the bit buff if the MSB and underflow bits
            // won't all fit in what's left
            if (1+_nUnderflowBits > 32 - _nBitBuff) {
                addCodeText(_uBitBuff, _nBitBuff);
                _uBitBuff = 0;
                _nBitBuff = 0;
            }
            // Write the MSB and all uflow bits at once
            if (1+_nUnderflowBits <= 32 - _nBitBuff) {
                _uBitBuff <<= (1 + _nUnderflowBits);
                _uBitBuff |= (1 << _nUnderflowBits)
                          + (Int32(Int16(~_high)) >> 15);
                _nBitBuff += 1 + _nUnderflowBits;
                _nUnderflowBits = 0;
            }
            else {
                // We're writing more than 32 bits!
                _uBitBuff = (1 << 31)
                          + (Int32(Int16(~_high)) >> 15);
                addCodeText(_uBitBuff, 32);
                _nBitBuff = 0;
                _nUnderflowBits -= 31;
                // Emit the rest of the underflow bits
                _uBitBuff = (_uBitBuff << 1) | (_uBitBuff & 1);
                while (_nUnderflowBits >= 32) {
                    addCodeText(_uBitBuff, 32);
                    _nUnderflowBits -= 32;
                }
                addCodeText(_uBitBuff, _nUnderflowBits);
                _nUnderflowBits = 0;
                _uBitBuff = 0;
            }
        }

        // If this test passes, the numbers are in danger of underflow, because
        // the most sigif digits don't match, and the 2nd digits are just one apart.
        //
        // _low =  01... and _high = 10...
        else if ( ( _low & 0x4000 ) && !( _high & 0x4000 ))
        {
            _nUnderflowBits++;
            _low  &= 0x3fff;
            _high |= 0x4000;
        }
        else
            break;

        //Shift all bits left.  Move 0 into _low and 1 into _high.
        _low  <<= 1;
        _high <<= 1;
        _high |=  1;
    }

    return True;
}

Bool ArithmeticCodec2::_flushEncoder()
{
    // Write out some underflow bits and misc.
    WriteBit((_low & 0x4000)>>14)
    _nUnderflowBits++;
    while (_nUnderflowBits-- > 0)
        WriteBit((~_low & 0x4000)>>14)

    //Need 16 zeros at the end, for this decoding algorithm
    UInt32 zeroBit = 0x0000;
    for (Int32 ii=0; ii<16; ii++) {
        WriteBit(zeroBit)
    }

    // Flush out the local buffer
    addCodeText(_uBitBuff, _nBitBuff);

    return True;
}

template <class ValueType>
Bool ArithmeticCodec2::decode( Int32           nValues,
                               const VecValue &vOOBValues,
                               const Vecu     &vCodeText,
                               Int32           nBitsCodeText,
                               Veci           &ovValues,
                               ProbContextV   *pProbCntx     )
{
    ovValues.setLength(0);
    const ValueType *paiOOBValues = vOOBValues.ptr();
    ovValues.setLength(nValues);
    ValueType *paiValues = ovValues.ptr();
    Int32 cSymbolsCurrCtx = pProbCntx->totalCount();
    const CntxEntryV   *pCntxEntry = 0;

    // Initialize the arithmetic decoder state
    _iCurCodeText = 0;
    _pvCodeText = (Vecu*) &vCodeText;
    _pcCodeTextLen = &nBitsCodeText;
    getNextCodeText(_uBitBuff, _nBitBuff);
    _low  = 0;
    _high = 0xffff;
    _code = (_uBitBuff >> 16);
    _uBitBuff <<= 16;
    _nBitBuff -= 16;

    // Decode each symbol
    for (Int32 i = 0 ; i < nValues ; i++) {
        // Scale the current "code" into the range of counts presented by
        // the probcontext so we can look up the code.
        UInt16 rescaledCode = (((UInt32)(_code - _low) + 1) * (UInt32)cSymbolsCurrCtx -
1)
             / ((UInt32)(_high - _low) + 1);
        pProbCntx->lookupEntryByCumCount( (Int32)rescaledCode, pCntxEntry );

        // Emit the value corresponding to the symbol we just decoded
        if (!pCntxEntry->isEscape())
            *paiValues++ = pCntxEntry->_val;
        else
```

```
            *paiValues++ = *paiOOBValues++;

        // Set up the symbol's range and adjust the decoder state
        // to "remove" it.
        _removeSymbolFromStream( pCntxEntry->_cCumCount,
                                 pCntxEntry->_cCumCount + pCntxEntry->_cCount,
                                 cSymbolsCurrCtx );
    }

    _flushDecoder();

    return True;
}

Bool ArithmeticCodec2::_flushDecoder()
{
    UInt32 dummyBit;
    ReadBit(dummyBit)
    ReadBit(dummyBit)

    Assert( _uBitBuff == 0 );
    _nBitBuff = 0;

    return True;
}

Bool ArithmeticCodec2::_removeSymbolFromStream( UInt16 uLowCt, UInt16 uHighCt, UInt16
uScale )
{
    // First, the range is expanded to account for the symbol removal.
    UInt32 uRange = UInt32(_high - _low)  + 1;
    _high  = _low + (UInt32)((uRange * uHighCt) / uScale - 1);
    _low   = _low + (UInt32)((uRange * uLowCt ) / uScale);

    //Next, any possible bits are shipped out.
    for (;;) {
        // If the most signif digits match, the bits will be shifted out.
        if (UInt16(~(_high ^ _low)) >> 15){
        }
        // Else, if underflow is threatening, shift out the 2nd most signif digit.
        //else if ((_low & 0x4000) && !(_high & 0x4000))
        // If high=10xx and low=01xx
        else if (((_low >> 14) == 1) & ((_high >> 14) == 2)) {
            _code ^= 0x4000;
            _low  &= 0x3fff;
            _high |= 0x4000;
        }
        // Otherwise, nothing can be shifted out, so return.
        else {
            return True;
        }

        _low  <<= 1;
        _high <<= 1;
        _high |=  1;
        _code <<= 1;

        ReadBit0( _code )
    }
}
```

B.4 Deering Normal decoding classes

The following sub-sections contain a sample implementation of the decoding portion of the Deering Normal CODEC algorithm. A summary technical explanation of the Deering Normal CODEC can be found in 13.2.4 Deering Normal CODEC.

B.4.1 DeeringNormalLookupTable class

The DeeringNormalLookupTable class represents a lookup table used by the DeeringNormalCodec class for faster conversion from the compressed normal representation to the standard 3-float representation. The tables hold precomputed results of the trig functions called during conversion.

```
class DeeringNormalLookupTable
{
public:
   DeeringNormalLookupTable();

   // Lookup and return the result of converting iTheta and iPsi to
   // real angles and taking the sine and cosine of both.  This gives
   // a slight speedup for normal decoding.
   Bool lookupThetaPsi(Int32 iTheta,
                       Int32 iPsi,
                       UInt32 numberBits,
                       Float32 outCosTheta,
                       Float32 outSinTheta,
                       Float32 outCosPsi,
                       Float32 outSinPsi );

   UInt32 numBitsPerAngle() {return nBits;}

private:
   UInt32 nBits;
   Vector vCosTheta;
   Vector vSinTheta;
   Vector vCosPsi;
   Vector vSinPsi;
};

DeeringNormalLookupTable::DeeringNormalLookupTable()
{
   UInt32 numberbits = 8;
   nBits = min(numberbits, (UInt32)31);

   Int32 tableSize = (1 << nBits);

   vCosTheta.setLength(tableSize+1);
   vSinTheta.setLength(tableSize+1);
   vCosPsi.setLength(tableSize+1);
   vSinPsi.setLength(tableSize+1);

   Float32 fPsiMax = 0.615479709;
   Float32 fTableSize = (Float32)tableSize;

   for( Int32 ii = 0; ii <= tableSize; ii++ )
   {
   Float32 fTheta =
      asin(tan(fPsiMax * Float32(tableSize - ii) /  fTableSize));

      Float32 fPsi  = fPsiMax * (((Float32)ii) / fTableSize);
      vCosTheta[ii] = cos(fTheta);
      vSinTheta[ii] = sin(fTheta);
      vCosPsi[ii]   = cos(fPsi);
      vSinPsi[ii]   = sin(fPsi);
   }
}

Bool DeeringNormalLookupTable::lookupThetaPsi(Int32 iTheta,
                                              Int32 iPsi,
                                              UInt32 numberBits,
    Float32 outCosTheta,
    Float32 outSinTheta,
    Float32 outCosPsi,
    Float32 outSinPsi)
{
   Int32 offset = nBits - numberBits;

   outCosTheta = vCosTheta[iTheta << offset];
```

```
    outSinTheta = vSinTheta[iTheta << offset];
    outCosPsi   = vCosPsi[iPsi << offset];
    outSinPsi   = vSinPsi[iPsi << offset];

    return True;
}
```

B.4.2 DeeringNormalCodec class

The DeeringNormalCodec class converts a normal vector to and from the standard 3-float representation and a lower-precision representation. The precision can be adjusted using the nbits parameter.

```
class DeeringNormalCodec
{
public:
    DeeringNormalCodec(Int32 numberbits = 6)
    {
        numBits = numberbits;
    }

    // Converts a compressed normal into a vector.
    Bool convertCodeToVec(UInt32 code, Vector& outVec);

    // Converts a compressed normal into a vector.
    Bool convertCodeToVec (UInt32 iSextant,
                           UInt32 iOctant,
                           UInt32 iTheta,
                           UInt32 iPsi,
                           Vector& outVec);

    // Separates an encoded normal into its 4 pieces
    Bool unpackCode (UInt32  code,
                     UInt32& outSextant,
                     UInt32& outOctant,
                     UInt32& outTheta,
                     UInt32& outPsi );

    private:
        Int32 numBits;
}

Bool DeeringNormalCodec::convertCodeToVec(UInt32 code, Vector& outVec)
{
    UInt32 s=0, o=0, t=0, p=0;
    unpackCode(code, s, o, t, p);
    convertCodeToVec(s, o, t, p, outVec);
    return True;
}

Bool DeeringNormalCode::convertCodeToVec(UInt32 iSextant,
                                         UInt32 iOctant,
                                         UInt32 iTheta,
                                         UInt32 iPsi,
                                         Vector& outVec)
{
    // Size of code = 6+2*numBits, and max code size is 32 bits,
    // so numBits shall be <= 13.

    // Code layout: [sextant:3][octant:3][theta:numBits][psi:numBits]

    outVec.setValues(0,0,0);
    Float32 fPsiMax = 0.615479709;

    UInt32  iBitRange = 1<<numBits;
    Float32 fBitRange = Float32(iBitRange);

    // For sextants 1, 3, and 5, iTheta needs to be incremented
    iTheta += (iSextant & 1);

    Float32 fCosTheta, fSinTheta, fCosPsi, fSinPsi;

    DeeringNormalLookupTable LookupTable;
```

```
if( (LookupTable.numBitsPerAngle() < (UInt32)numBits) ||
     !LookupTable.lookupThetaPsi(iTheta, iPsi, numBits,
                                    fCosTheta, fSinTheta,
                                    fCosPsi, fSinPsi) )
{
   Float32 fTheta = asin(tan(fPsiMax * Float32(iBitRange - iTheta) /
                                 fBitRange));

   Float32 fPsi = fPsiMax * (iPsi / fBitRange);
   fCosTheta = cos(fTheta);
   fSinTheta = sin(fTheta);
   fCosPsi   = cos(fPsi);
   fSinPsi   = sin(fPsi);
}

Float32 x,y,z;
Float32 xx = x = fCosTheta * fCosPsi;
Float32 yy = y = fSinPsi;
Float32 zz = z = fSinTheta * fCosPsi;

//Change coordinates based on the sextant
switch( iSextant )
{
   case 0:      // No op
      break;

   case 1:      // Mirror about x=z plane
      z = xx;
      x = zz;
      break;

   case 2:      // Rotate CW
      z = xx;
      x = yy;
      y = zz;
      break;

   case 3:      // Mirror about x=y plane
      y = xx;
      x = yy;
      break;

   case 4:      // Rotate CCW
      y = xx;
      z = yy;
      x = zz;
      break;

   case 5:      // Mirror about y=z plane
      z = yy;
      y = zz;
      break;
};

//Change some more based on the octant

//if first bit is 0, negate x component
if( !(iOctant & 0x4) )
   x = -x;

//if second bit is 0, negate y component
if( !(iOctant & 0x2) )
   y = -y;

//if third bit is 0, negate z component
if( !(iOctant & 0x1) )
   z = -z;

outVec.setValues(x,y,z);

return True;
```

```
}

Bool DeeringNormalCodec::unpackCode(UInt32 code,
                                    UInt32& outSextant,
                                    UInt32& outOctant,
                                    UInt32& outTheta,
                                    UInt32& outPsi)
{
   UInt32 mask = (1<<numBits)-1;

   outSextant = (code >> (numBits+numBits+3)) & 0x7;
   outOctant  = (code >> (numBits+numBits))   & 0x7;
   outTheta   = (code >> (numBits))           & mask;
   outPsi     = (code)                        & mask;

   return True;
}
```

Annex C

Hashing – An Implementation

This Appendix provides a sample C++ implementation for the creation of hash values (as detailed in 13.2 Encoding Algorithms) used in the JT format.

```
unsigned int hash32( const unsigned int *pWords,
                     int nWords,
                     unsigned int uSeedHashValue )
{ return hash2(pWords, nWords, uSeedHashValue); }

unsigned int jthash16(const unsigned short *pBytes,
                      int nShort,
                      unsigned int uSeedHashValue)
{ return hash3(pBytes, nShort, uSeedHashValue); }

//---------------------------------------------------------------------
//  mix -- mix 3 32-bit values reversibly.
// For every delta with one or two bit set, and the deltas of all three
//   high bits or all three low bits, whether the original value of a,b,c
//   is almost all zero or is uniformly distributed,
// * If mix() is run forward or backward, at least 32 bits in a,b,c
//   have at least 1/4 probability of changing.
// * If mix() is run forward, every bit of c will change between 1/3 and
//   2/3 of the time.  (Well, 22/100 and 78/100 for some 2-bit deltas.)
// mix() was built out of 36 single-cycle latency instructions in a
//   structure that could supported 2x parallelism, like so:
//       a -= b;
//       a -= c; x = (c>>13);
//       b -= c; a ^= x;
//       b -= a; x = (a<<8);
//       c -= a; b ^= x;
//       c -= b; x = (b>>13);
//       ...
//   Unfortunately, superscalar Pentiums and Sparcs can't take advantage
//   of that parallelism.  They've also turned some of those single-cycle
//   latency instructions into multi-cycle latency instructions.  Still,
//   this is the fastest good hash I could find.  There were about 2^^68
//   to choose from.  I only looked at a billion or so.
--------------------------------------------------------------------

#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8); \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12);  \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5); \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}

--------------------------------------------------------------------
// hash() -- hash a variable-length key into a 32-bit value
//   k     : the key (the unaligned variable-length array of bytes)
//   len   : the length of the key, counting by bytes
//   level : can be any 4-byte value
// Returns a 32-bit value.  Every bit of the key affects every bit of
// the return value.  Every 1-bit and 2-bit delta achieves avalanche.
// About 36+6len instructions.

// The best hash table sizes are powers of 2.  There is no need to do
// mod a prime (mod is sooo slow!).  If you need less than 32 bits,
// use a bitmask.  For example, if you need only 10 bits, do
//   h = (h & hashmask(10));
// In which case, the hash table should have hashsize(10) elements.
//
// If you are hashing n strings (JtUInt8 **)k, do it like this:
//   for (i=0, h=0; i<n; ++i) h = hash( k[i], len[i], h);
//
// By Bob Jenkins, 1996.  bob_jenkins@burtleburtle.net.  You may use this
// code any way you wish, private, educational, or commercial.  It's free.
//
// See http://burtleburtle.net/bob/                  // 2010/02/12
// See http://burtleburtle.net/bob/hash/doobs.html   // 2010/02/12
//
// Use for hash table lookup, or anything where one collision in 2^32 is
// acceptable.  Do NOT use for cryptographic purposes.
//--------------------------------------------------------------------

//--------------------------------------------------------------------
// This works on all machines.  hash2() is identical to hash() on
// little-endian machines, except that the length has to be measured
// in ub4s instead of bytes.  It is much faster than hash().  It
// requires
// -- that the key be an array of UInt32's, and
// -- that all your machines have the same endianness, and
// -- that the length be the number of UInt32's in the key
// --------------------------------------------------------------------
unsigned int hash(const usigned char *k,        // key
                  unsigned int        length,   // length of the key
                  unsigned int        initval)  // prev hash, or an arbitrary value
{
   register unsigned int a,b,c,len;

   /* Set up the internal state */
   len = length;
   a = b = 0x9e3779b9;  /* the golden ratio; an arbitrary value */
   c = initval;           /* the previous hash value */
   /*---------------------------------------- handle most of the key */
   while (len >= 12) {
      a += (k[0] +((UInt32)k[1]<<8) +((UInt32)k[2]<<16) +((UInt32)k[3]<<24));
      b += (k[4] +((UInt32)k[5]<<8) +((UInt32)k[6]<<16) +((UInt32)k[7]<<24));
      c += (k[8] +((UInt32)k[9]<<8) +((UInt32)k[10]<<16)+((UInt32)k[11]<<24));
      mix(a,b,c);
      k += 12; len -= 12;
   }
   /*------------------------------------- handle the last 11 bytes */
   c += length;
   switch(len) {            /* all the case statements fall through */
     case 11: c+=((UInt32)k[10]<<24);
     case 10: c+=((UInt32)k[9]<<16);
     case 9 : c+=((UInt32)k[8]<<8);
      /* the first byte of c is reserved for the length */
     case 8 : b+=((UInt32)k[7]<<24);
     case 7 : b+=((UInt32)k[6]<<16);
     case 6 : b+=((UInt32)k[5]<<8);
     case 5 : b+=k[4];
     case 4 : a+=((UInt32)k[3]<<24);
     case 3 : a+=((UInt32)k[2]<<16);
     case 2 : a+=((UInt32)k[1]<<8);
     case 1 : a+=k[0];
     /* case 0: nothing left to add */
   }
   mix(a,b,c);
   /*-------------------------------------------- report the result */
   return c;
}

unsigned int hash3(const unsigned short *k,       /* the key */
                   unsigned int          length,  /* the length of the key */
                   unsigned int          initval) /* the previous hash, or an arbitrary
value */
{
   unsigned int a,b,c,len;
   /* Set up the internal state */
   len = length;
   a = b = 0x9e3779b9;    /* the golden ratio; an arbitrary value */
   c = initval;           /* the previous hash value */

   /*---------------------------------------- handle most of the key */
   while (len >= 6)
   {
      a += (k[0] + (UInt32(k[1]) << 16));
      b += (k[2] + (UInt32(k[3]) << 16));
      c += (k[4] + (UInt32(k[5]) << 16));
      mix(a,b,c);
      k += 6; len -= 6;
   }

   /*-------------------------------------- handle the last 2 uint32s */
   c += length;
   switch(len)              /* all the case statements fall through */
   {
       case 5 : c+=(UInt32(k[4]) << 16);
       /* c is reserved for the length */
       case 4 : b+=(UInt32(k[3]) << 16);
       case 3 : b+=k[2];
       case 2 : a+=(UInt32(k[1]) << 16);
       case 1 : a+=k[0];
       /* case 0: nothing left to add */
   }
   mix(a,b,c);
   /*-------------------------------------------- report the result */
   return c;
}
```

Annex D

Polygon Mesh Topology Coder

The topology coding algorithm described here is used to code the *dual* of the desired mesh. Thus, for example, the reader will need to take the dual of the decoded mesh in order to obtain the original primal mesh. Presented below are classes suitable for representing the dual of a polygon mesh and the dual topology decoding algorithm.

At a high level, the topology coder works by traversing the dual mesh to be encoded one vertex and one face at a time. The coder maintains a queue of faces to be processed; the initial queue is created using the valence of an arbitrary vertex of the mesh followed by the degrees of the faces adjacent to that vertex, and adds the adjacent faces to the face queue. Each time it visits a face, it encodes the *degree* of that face and emits any incident vertices that have not yet been visited. Each time the coder visits a vertex, it encodes the *valence* of the vertex (usually 3 in the current case), and emits any incident faces that have not yet been visited. It works its way through the mesh in this fashion until all vertices and faces have been encoded. Thus, the primary output from the topology coder is a list of vertex valences and face degrees. These two fields plus two more encoding so-called *split faces*, coupled with the exact coder implementation completely encode the mesh topology in a very compact manner[1].

In addition to these two basic fields are added a number of other fields that organize the dual vertices into *vertex groups*, and also encode the *vertex attributes* (e.g. normals, colours, and texture coordinates) around each dual face's *degree ring*.

The topological coder can only encode *closed, manifold* meshes. It cannot encode *boundaries*; it can only encode edges with exactly two incident faces. But, as we know, real-world data is chock full of meshes with boundaries. In order to encode these types of meshes, it is necessary to add *cover faces* incident to all boundary loops whose sole job is to turn the mesh into a *closed* mesh. It is the dual of this closed, manifold mesh that is actually encoded. Thus, most meshes encoded in JT files contain a few cover faces. These faces may be of arbitrarily high degree, and they represent the only exceptions to the general rule that the numbers in the dual vertex valence array are usually three. It is necessary to flag all such artificially introduced cover faces so that they can be removed by the loader. These flags are encoded below in the Face Flags array. Primal faces are flagged with zero, while cover faces are flagged with one.

Now, let us make the connection between topological vertices and how vertex attributes relate to them. Several faces may be incident on the same topological mesh vertex. While this topological vertex has only a single 3D coordinate, it may have a different set of *vertex attributes* for each incident face. Vertex attributes include colour, normal, and texture coordinates. An important observation in real-world data is that adjacent faces tend to share the same vertex attributes. Thus, a natural way to encode which vertex attributes map to which faces within a given valence ring (the counter-clockwise ordered set of faces incident on a given vertex) is by way of a bit vector. The bit vector begins at the first face the coder encounters that is incident to the vertex, and proceeds counter clockwise around the vertex, allocating one bit per incident face. A value of 0 is assigned to the bit if all vertex attributes for the face are the same as the face immediately clockwise. A value of 1 is assigned if the vertex attributes for the face are different. Recall that these bits from the original primal mesh are encoded as face attributes in the dual mesh.

Thus, at the end of the coding process, there will be one such bit vector per topological vertex in the mesh. These bit vectors will be of disparate lengths because all vertex valences are not the same. Though there is no theoretical limit to the valence of any given vertex, in practice, the vertex valences seldom rise above six, and only rarely rise into the dozens. As a matter of practicality, then, we break this list of bit vectors into those of length 64 and smaller into one group, and all others into a list of so-called ―high-valence” bit vectors. The low-valence bit vectors are encoded into two fields of 32 bits each. The high-valence bit vectors are adjoined end-to-end into a single long bit vector, and encoded as a single array of integers. As an additional optimization, the low-valence bit vectors are grouped into 8 ―context groups” depending on the valence of the vertex being coded. This is done in order to improve compression performance because the valence bit

[1] Similar methods of topology coding are described in [7] and US patent # 7,098,916. The topology coding algorithm described herein differs from such methods in that while they utilize a queue of active *vertices*, the instant algorithm utilizes a queue of active *faces*. Other differences include the tracking of face group numbers and per-vertex attributes such as normals, colours, and texture coordinates.

vectors in each of the most common groups typically share similar statistics. Context group number 8 is the only one that encodes valence rings up to valence 64. Again, recall that these attribute bits from the original primal mesh are encoded as face attribute bits in the dual mesh.

D.1 DualVFMesh

The DualVFMesh (Dual Vertex-Facet Mesh) is a support class paired with the topology decoder itself, and represents a closed two-manifold polygon mesh. The topology decoder reconstructs the encoded dual mesh into a DualVFMesh, building it one vertex and one facet at a time. When the decoder is finished, it will have visited each vertex and each face of the dual mesh exactly once. DualVFMesh is not intended as a work-horse in-memory storage container because its way of encoding the topological connections between faces and vertices is memory-intensive.

```
class DualVFMesh
{
  public:
    // ========== Housekeeping Interface ==========
    DualVFMesh();
    DualVFMesh (const DualVFMesh &rhs);
    DualVFMesh &operator=(const DualVFMesh &rhs);

    // ========== Topology Interface ==========

    // Vtx creation
    bool          isValidVtx (Int32  iVtx) const;
    bool          newVtx     (Int32  iVtx,
                              Int32  iValence,
                              UInt16 uFlags = 0);
    bool          setVtxFlags(Int32  iVtx,
                              UInt16 uFlags);
    bool          setVtxGrp  (Int32  iVtx,
                              Int32  iVGrp);
    UInt16        vtxFlags   (Int32  iVtx) const;
    Int32         vtxGrp     (Int32  iVtx) const;

    // Face creation
    bool          isValidFace  (Int32  iFace) const;
    bool          newFace      (Int32  iFace,
                                Int32  cDegree,
                                Int32  cFaceAttrs = 0,
                                UInt64 uFaceAttrMask = 0,
                                UInt16 uFlags = 0);
    bool          newFace      (Int32  iFace,
                                Int32  cDegree,
                                Int32  cFaceAttrs,
                                const BitVec *pvbFaceAttrMask,
                                UInt16 uFlags);
    bool          setFaceFlags (Int32  iFace,
                                UInt16 uFlags);
    UInt16        faceFlags    (Int32  iVtx) const;
    bool          setFaceAttr  (Int32  iFace,
                                Int32  iAttrSlot,
                                Int32  iFaceAttr);
    Int32         faceAttr     (Int32  iFace,
                                Int32  iAttrSlot) const;

    // Topology connection
    bool          setVtxFace(Int32  iVtx,
                             Int32  iFaceSlot,
                             Int32  iFace);
    bool          setFaceVtx(Int32  iFace,
                             Int32  iVtxSlot,
                             Int32  iVtx);

    // Queries
    Int32         valence    (Int32  iVtx) const
        { return _vVtxEnts[iVtx].cVal; }
    Int32         degree    (Int32  iFace ) const
        { return _vFaceEnts[iFace].cDeg; }
    Int32         face     (Int32  iVtx,
                           Int32  iFaceSlot) const
          { return _viVtxFaceIndices[(_vVtxEnts[iVtx]).iVFI + iFaceSlot]; }
    Int32         vtx       (Int32  iFace,
                            Int32  iVtxSlot) const
          { return _viFaceVtxIndices[_vFaceEnts[iFace].iFVI + iVtxSlot]; }
    Int32         numVts         () const
        { return _vVtxEnts.length(); }
    Int32         numFaces       () const
        { return _vFaceEnts.length(); }
    Int32         numAttrs       () const
        { return _viFaceAttrIndices.length(); }
    Int32         numAttrs        (Int32 iFace) const
        { return _vFaceEnts[iFace].cFaceAttrs; }
    UInt64        attrMask        (Int32 iFace) const
        { return _vFaceEnts[iFace].u.uAttrMask; }
    const BitVec *attrMaskV       (Int32 iFace) const
        { return _vFaceEnts[iFace].u.pvbAttrMask; }
    Int32         findVtxSlot    (Int32 iFace,
                                 Int32 iTargVtx) const;
    Int32         findFaceSlot     (Int32 iVtx,
                                    Int32 iTargFace) const;
    Int32         emptyFaceSlots   (Int32 iFace) const
        { return _vFaceEnts[iFace].cEmptyDeg; }

    // ========== VFMesh Data Members ==========
  public:
    class VtxEnt {
      public:
        VtxEnt() : cVal(0), uFlags(0), iVGrp(-1), iVFI(-1) {}
        UInt16    cVal;   // Vtx valence
        UInt16    uFlags; // User flags
        Int32     iVGrp;  // Vtx group
        Int32     iVFI;   // Idx into _viVtxFaceIndices of cVal incident faces
    };

    // Number of optimized mask bits.
    static const Int32 cMBits = 64;

    class FaceEnt {
      public:
        FaceEnt() : cDeg(0), uFlags(0), cEmptyDeg(0),
                   cFaceAttrs(0), iFVI(-1), iFAI(-1) { u.uAttrMask = 0; }
        FaceEnt(const FaceEnt &rhs) : cDeg(rhs.cDeg), cEmptyDeg(rhs.cEmptyDeg),
                                     cFaceAttrs(rhs.cFaceAttrs), iFVI(rhs.iFVI),
                                     iFAI(rhs.iFAI)
        {
            if (cDeg <= cMBits)
                u.uAttrMask = rhs.u.uAttrMask;
            else
                JtWrapNew(u.pvbAttrMask, new BitVec(*rhs.u.pvbAttrMask));
        }
        ~FaceEnt() { if (cDeg > cMBits && u.pvbAttrMask) delete u.pvbAttrMask; }
        UInt16    cDeg;       // Face degree
        UInt16    cEmptyDeg;  // Empty degrees (opt for emptyFaceSlots())
        UInt16    cFaceAttrs;  // Number of face attributes
        UInt16    uFlags;     // User flags
        union {
            UInt64  uAttrMask;    // Degree-ring attr mask as a UInt64
            BitVec *pvbAttrMask;  // Degree-ring attr mask as a BitVec
        } u;
        Int32     iFVI; // Idx into _viFaceVtxIndices of cDeg incident vts
        Int32     iFAI; // Idx into _viFaceAttrIndices of cAttr attributes
    };

  protected:
    // Subscripted by atom number, the entry contains the vtx valence and
    // points to the location in _viVtxFaceIndices of valence consecutive
    // integers that in turn contain the indices of the incident faces
    // in _vFaceRecs to the vtx.
    JtVec<VtxEnt>  _vVtxEnts;
```

```
    // Subscripted by unique vertex record number, the entry contains the
    // face degree and points to the location in _viFaceVtxIndices of
    // cDeg consecutive integers that in turn contain the indices of the
    // vertices indicent upon the face, in CCW order, in _vVtxRecs.
    JtVec<FaceEnt>   _vFaceEnts;

    // Combined storage for all vtxs.
    JtVeci          _viVtxFaceIndices;

    // Combined storage for all faces.
    JtVeci          _viFaceVtxIndices;

    // Combined storage for all face attribute record identifiers
    JtVeci          _viFaceAttrIndices;
};

bool
DualVFMesh::isValidVtx(Int32 iVtx) const
{
    bool bRet = JtFalse;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        bRet = (rFE.cVal != 0);
    }
    return bRet;
}

bool
DualVFMesh::newVtx(Int32 iVtx,
                   Int32 iValence,
                   UInt16 uFlags)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    if (rFE.cVal != iValence) {
        rFE.cVal   = iValence;
        rFE.uFlags = uFlags;
        rFE.iVFI   = _viVtxFaceIndices.length();
        _viVtxFaceIndices.verify(rFE.iVFI + iValence - 1);
        for (Int32 i = rFE.iVFI ; i < rFE.iVFI + iValence ; i++)
            _viVtxFaceIndices[i] = -1;
    }
    return true;
}

bool
DualVFMesh::setVtxGrp(Int32  iVtx,
                     Int32  iVGrp)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    rFE.iVGrp = iVGrp;
    return true;
}

bool
DualVFMesh::setVtxFlags(Int32  iVtx,
                       UInt16 uFlags)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    rFE.uFlags = uFlags;
    return true;
}

Int32
DualVFMesh::vtxGrp   (Int32  iVtx) const
{
    Int32 u = -1;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        u = rFE.iVGrp;
    }
    return u;
}
```

```
UInt16
DualVFMesh::vtxFlags   (Int32  iVtx) const
{
    UInt16 u = 0;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        u = rFE.uFlags;
    }
    return u;
}


bool
DualVFMesh::isValidFace(Int32 iFace) const
{
    bool bRet = JtFalse;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rVE = _vFaceEnts[iFace];
        bRet = (rVE.cDeg != 0);
    }
    return bRet;
}

bool
DualVFMesh::newFace(Int32  iFace,
                    Int32  cDegree,
                    Int32  cFaceAttrs,
                    UInt64 uFaceAttrMask,
                    UInt16 uFlags)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    if (rVE.cDeg != cDegree) {
        rVE.cDeg        = cDegree;
        rVE.cEmptyDeg   = cDegree;
        rVE.cFaceAttrs   = cFaceAttrs;
        rVE.uFlags      = uFlags;
        rVE.u.uAttrMask = uFaceAttrMask;
        rVE.iFVI        = _viFaceVtxIndices.length();
        rVE.iFAI        = _viFaceAttrIndices.length();
        _viFaceVtxIndices.verify(rVE.iFVI + cDegree  - 1);
        if (cFaceAttrs > 0)
            _viFaceAttrIndices.verify(rVE.iFAI + cFaceAttrs - 1);
        for (Int32 i = rVE.iFVI ; i < rVE.iFVI + cDegree ; i++)
            _viFaceVtxIndices[i] = -1;
        for (Int32 i = rVE.iFAI ; i < rVE.iFAI + cFaceAttrs ; i++)
            _viFaceAttrIndices[i] = -1;
    }
    return true;
}

bool
DualVFMesh::newFace(Int32  iFace,
                    Int32  cDegree,
                    Int32  cFaceAttrs,
                    const BitVec *pvbFaceAttrMask,
                    UInt16 uFlags)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    if (rVE.cDeg != cDegree) {
        rVE.cDeg        = cDegree;
        rVE.cEmptyDeg   = cDegree;
        rVE.cFaceAttrs   = cFaceAttrs;
        rVE.uFlags      = uFlags;
        rVE.u.pvbAttrMask = new BitVec(*pvbFaceAttrMask);
        rVE.iFVI        = _viFaceVtxIndices.length();
        rVE.iFAI        = _viFaceAttrIndices.length();
        _viFaceVtxIndices.verify(rVE.iFVI + cDegree  - 1);
        if (cFaceAttrs > 0)
            _viFaceAttrIndices.verify(rVE.iFAI + cFaceAttrs - 1);
        for (Int32 i = rVE.iFVI ; i < rVE.iFVI + cDegree ; i++)
            _viFaceVtxIndices[i] = -1;
        for (Int32 i = rVE.iFAI ; i < rVE.iFAI + cFaceAttrs ; i++)
            _viFaceAttrIndices[i] = -1;
    }
    return true;
}

bool
DualVFMesh::setFaceFlags(Int32  iFace,
                         UInt16 uFlags)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    rVE.uFlags = uFlags;
    return true;
}

UInt16
DualVFMesh::faceFlags   (Int32  iFace) const
{
    UInt16 u = 0;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rVE = _vFaceEnts[iFace];
        u = rVE.uFlags;
    }
    return u;
}

bool
DualVFMesh::setFaceAttr(Int32  iFace,
                        Int32  iAttrSlot,
                        Int32  iFaceAttr)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    Int32 *paiFAI = _viFaceAttrIndices.ptr();
    paiFAI[rVE.iFAI + iAttrSlot] = iFaceAttr;
    return true;
}

Int32
DualVFMesh::faceAttr(Int32  iFace,
                     Int32  iAttrSlot) const
{
    Int32 u = 0;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rVE = _vFaceEnts[iFace];
        if (iAttrSlot >= 0 && iAttrSlot < rVE.cDeg) {
            const Int32 *paiFAI = _viFaceAttrIndices.ptr();
            u = paiFAI[rVE.iFAI + iAttrSlot];
        }
    }
    return u;
}

// Attaches VF face iFace to VF vertex iVtx in the vertex's
// face slot iFaceSlot
bool
DualVFMesh::setVtxFace(Int32 iVtx,
                       Int32 iFaceSlot,
                       Int32 iFace)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    _viVtxFaceIndices[rFE.iVFI + iFaceSlot] = iFace;
    return true;
}

// Attaches VF vertex iVtx to VF face iFace in the face's
// vertex slot iVtxSlot
bool
DualVFMesh::setFaceVtx(Int32 iFace,
                       Int32 iVtxSlot,
                       Int32 iVtx)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    Int32 *paiFVI = _viFaceVtxIndices.ptr();
    rVE.cEmptyDeg -= (paiFVI[rVE.iFVI + iVtxSlot] != iVtx);
    paiFVI[rVE.iFVI + iVtxSlot] = iVtx;
    return true;
}

// Searches the list of incident vts to face iFace for
// iTargVtx and returns the vtx slot at which it is found
// or -1 if iTargVtx is not found.
Int32
DualVFMesh::findVtxSlot(Int32 iFace,
                        Int32 iTargVtx) const
{
    const FaceEnt &rVE = _vFaceEnts[iFace];
    const Int32 *const pFaceVtxIndices = _viFaceVtxIndices.ptr() + rVE.iFVI;
    Int32 cDeg = rVE.cDeg;
    Int32 iSlot = -1;
    for (Int32 iVtxSlot = 0 ; iVtxSlot < cDeg ; iVtxSlot++) {
        if (pFaceVtxIndices[iVtxSlot] == iTargVtx) {
            iSlot = iVtxSlot;
            break;
        }
    }
    return iSlot;
}

// Searches the list of incident faces to vertex iVtx for
// iTargFace and returns the face slot at which it is found
// or -1 if iTargFace is not found.
Int32
DualVFMesh::findFaceSlot (Int32 iVtx,
                          Int32 iTargFace) const
{
    const VtxEnt &rFE = _vVtxEnts[iVtx];
    const Int32 *const pVtxFaceIndices = _viVtxFaceIndices.ptr() + rFE.iVFI;
    for (Int32 iFaceSlot = 0 ; iFaceSlot < rFE.cVal ; iFaceSlot++) {
        if (pVtxFaceIndices[iFaceSlot] == iTargFace) {
            return iFaceSlot;
        }
    }
    return -1;
}
```

D.2 Topology Decoder

Partial implementations of three classes are given here for MeshCoderDriver, MeshCodec, and MeshDecoder. MeshCodec contains the abstract implementation of the topology coder. MeshDecoder implements the functionality needed to *decode* a mesh from the input data read from a JT file (see Topologically Compressed Rep Data). MeshCoderDriver manages the input data, the output VFMesh, and the MeshDecoder itself, providing a simple three-step API.

D.2.1 MeshCoderDriver class

```
// This class serves as a coordinating driver for mesh coding and decoding.
class MeshCoderDriver
{
  public:
    MeshCoderDriver ();

    // ========== Operations Interface ==========
    void      setInputData(const Veci   vviOutValSyms[/*8*/],
                           const Veci   &viOutDegSyms,
                           const Veci   &viOutFGrpSyms,
                           const Vecus  &vuOutFaceFlags,
                           const Veclu  vvuOutAttrMasks[/*8*/],
                           const Vecu   &vuOutAttrMasksLrg,
                           const Veci   &viOutSplitVtxSyms,
                           const Veci   &viOutSplitPosSyms)
                           { /* Copy into 22 fields below */ }
    void      decode();
    VFMesh   *vfm() const { return _pOutVFM; }

    // ========== Utility Methods ==========
    Int32         _nextDegSymbol    (Int32 iCCntx);
    Int32         _nextValSymbol    ();
    Int32         _nextFGrpSymbol   ();
    UInt16        _nextVtxFlagSymbol();
    UInt64        _nextAttrMaskSymbol(Int32 iCCntx);     // <= 64-bit attrmask
    void          _nextAttrMaskSymbol(BitVec *iopvbAttrMask,
                                      Int32   cDegree);  // > 64 bit attrmask
    Int32         _nextSplitFaceSymbol();
    Int32         _nextSplitPosSymbol();
    Int32         _faceCntxt(Int32 iVtx, JtDualVFMesh *pVFM);

    // ========== Member Data ==========
  protected:
    SharedPtr<MeshCodec>     _pMC;      // The mesh coder or decoder being used
    SharedPtr<JtDualVFMesh>  _pOutVFM;  // Back-end VFMesh built by decoder
    SharedPtr<MeshDecoder>   _pMeshDecoder;

    // Coding symbols generated by encoding operation, auxiliary data such as
    // offsets, etc.
    Veci          _vviOutDegSyms[8];  // Face degree + SPLIT symbols for multiple contexts
    Veci          _viOutValSyms;      // Vtx valence symbols
    Veci          _viOutVGrpSyms;     // Vtx group of each encoded vtx
    Vecus         _vuOutVtxFlags;     // Vtx flags; parallel to _viOutValSyms.
    Veclu         _vvuOutAttrMasks[8];// Attribute bitmasks per face for multiple
contexts.
                                      //  One per non-split entry in _viOutValSyms.
    Vecu          _vuOutAttrMasksLrg; // > 64-bit attrmasks
    Veci          _viOutSplitFaceSyms;// Split face offsets
    Veci          _viOutSplitPosSyms; // Split face vtx slots

    // The next symbol to be consumed by _next*Symbol()
    Int32         _iValReadPos[8];
    Int32         _iDegReadPos;
    Int32         _iVGrpReadPos;
    Int32         _iFFlagReadPos;
    Int32         _iAttrMaskReadPos[8];
    Int32         _iAttrMaskLrgReadPos;
    Int32         _iSplitFaceReadPos;
    Int32         _iSplitPosReadPos;
};

void MeshCoderDriver::decode()
{
    // Allocate a coder
    if (!_pMeshDecoder) {
        _pMeshDecoder = new MeshDecoder(this);
    }
    _pMC = _pMeshDecoder;
    _pMC->setTopoDualMeshCoder(this);

    // Reset the symbol counters
    for (Int32 i = 0 ; i < 8 ; i++) {
        _iValReadPos[i] = 0;
        _iAttrMaskReadPos[i] = 0;
    }
    _iDegReadPos = 0;
    _iVGrpReadPos = 0;
    _iFFlagReadPos = 0;
    _iAttrMaskLrgReadPos = 0;
    _iSplitFaceReadPos = 0;
    _iSplitPosReadPos = 0;

    // Run the decoder
    _pMC->run();

    // Assert that ALL symbols have been consumed
    for (Int32 i = 0 ; i < 8 ; i++) {
        Assert(_iValReadPos[i]         == _vviOutDegSyms[i].length());
        Assert(_iAttrMaskReadPos[i]    == _vvuOutAttrMasks[i].length());
    }
    Assert(_iDegReadPos         == _viOutValSyms.length());
    Assert(_iVGrpReadPos        == _viOutVGrpSyms.length());
    Assert(_iFFlagReadPos       == _vuOutVtxFlags.length());
    Assert(_iAttrMaskLrgReadPos == _vuOutAttrMasksLrg.length());
    Assert(_iSplitFaceReadPos    == _viOutSplitFaceSyms.length());
    Assert(_iSplitPosReadPos    == _viOutSplitPosSyms.length());

    // Set output VFMesh
    _pOutVFM = _pMC->vfm();
}

Int32 MeshCoderDriver::_nextDegSymbol  (Int32 iCCntx)
{
    Int32 eSym = -1;
    if (_iValReadPos[iCCntx] < _vviOutDegSyms[iCCntx].length())
        eSym = _vviOutDegSyms[iCCntx].value(_iValReadPos[iCCntx]++);
    return eSym;
}

Int32
MeshCoderDriver::_nextValSymbol  ()
{
    Int32 eSym = -1;
    if (_iDegReadPos < _viOutValSyms.length())
        eSym = _viOutValSyms.value(_iDegReadPos++);
    return eSym;
}

Int32 MeshCoderDriver::_nextFGrpSymbol()
{
    Int32 eSym = -1;
    if (_iVGrpReadPos < _viOutVGrpSyms.length())
        eSym = _viOutVGrpSyms.value(_iVGrpReadPos++);
    return eSym;
}

UInt16 MeshCoderDriver::_nextVtxFlagSymbol  ()
{
    UInt16 eSym = 0;
    if (_iFFlagReadPos < _vuOutVtxFlags.length())
        eSym = _vuOutVtxFlags.value(_iFFlagReadPos++);
    return eSym;
}

UInt64 MeshCoderDriver::_nextAttrMaskSymbol  (Int32 iCCntx)
{
    UInt64 eSym = 0;
    if (_iAttrMaskReadPos[iCCntx] < _vvuOutAttrMasks[iCCntx].length())
        eSym = _vvuOutAttrMasks[iCCntx].value(_iAttrMaskReadPos[iCCntx]++);
    return eSym;
}

void MeshCoderDriver::_nextAttrMaskSymbol(BitVec *iopvbAttrMask, Int32 cDegree)
{
    if (_iAttrMaskLrgReadPos < _vuOutAttrMasksLrg.length()) {
        iopvbAttrMask->setLength(cDegree);
        UInt32 *pu = iopvbAttrMask->ptr();
        Int32 nWords = (cDegree + BitVec::cWordBits - 1) >> BitVec::cBitsLog2;
        memcpy(pu, &_vuOutAttrMasksLrg.value(_iAttrMaskLrgReadPos), nWords *
sizeof(UInt32));
        _iAttrMaskLrgReadPos += nWords;
    }
    else {
        iopvbAttrMask->setLength(0);
    }
}

Int32 MeshCoderDriver::_nextSplitFaceSymbol  ()
{
    Int32 eSym = -1;
    if (_iSplitFaceReadPos < _viOutSplitFaceSyms.length())
```

```
        eSym = _viOutSplitFaceSyms.value(_iSplitFaceReadPos++);
    return eSym;
}

Int32 MeshCoderDriver::_nextSplitPosSymbol  ()
{
    Int32 eSym = -1;
    if (_iSplitPosReadPos < _viOutSplitPosSyms.length())
        eSym = _viOutSplitPosSyms.value(_iSplitPosReadPos++);
    return eSym;
}

// Computes a "compression context" from 0 to 7 inclusive for
// faces on vertex iVtx.  The context is based on the vertex's
// valence, and the total _known_ degree of already-coded
// faces on the vertex at the time of the call.
Int32 MeshCoderDriver::_faceCntxt(JtInt32 iVtx, JtDualVFMesh *pVFM)
{
    // Here, we are going to gather data to be used to determine a
    // compression contest for the face degree.
    JtInt32 cVal = pVFM->valence(iVtx);
    JtInt32 nKnownFaces = 0;
    JtInt32 cKnownTotDeg = 0;
    for (JtInt32 i = 0 ; i < cVal ; i++) {
        JtInt32 iTmpFace = pVFM->face(iVtx, i);
        if (!pVFM->isValidFace(iTmpFace))
            continue;
        nKnownFaces++;
        cKnownTotDeg += pVFM->degree(iTmpFace);
    }
    JtInt32 iCCntxt = 0;
    if (cVal == 3) {
        // Regular tristrip-like meshes tend to have degree 6 faces
        iCCntxt = (cKnownTotDeg <  nKnownFaces * 6) ? 0 :
                  (cKnownTotDeg == nKnownFaces * 6) ? 1 : 2;
    }
    else if (cVal == 4) {
        // Regular quadstrip-like meshes tend to have degree 4 faces
        iCCntxt = (cKnownTotDeg <  nKnownFaces * 4) ? 3 :
                  (cKnownTotDeg == nKnownFaces * 4) ? 4 : 5;
    }
    else if (cVal == 5)
        // Pentagons are all lumped into context 6
        iCCntxt = 6;
    else
        // All other polygons are lumped into context 7
        iCCntxt = 7;

    return iCCntxt;
}
```

D.2.2 MeshCodec class

```
// This class serves as the abstract base class from which two concrete classes
// are derived to implement the core operations for a polygonal
// mesh coder or decoder.  An instance of this object is used by the
// MeshCoderDriver to encode and decode polygonal meshes.
//
// This class makes extensive use of DualVFMesh objects as the primary source and
// destination mesh topology storage data structures. This mediating data
// structure is necessary because the mesh coding scheme is deeply cooperative
// with and dependent upon such a vertex-facet data structure.  Please refer to
// DualVFMesh for more information.
class MeshCodec {
  public:
    // ========== Housekeeping Interface ==========
    MeshCodec (MeshCoderDriver *pTMC = NULL);
  protected:
    virtual ~MeshCodec() {}
  public:

    // ========== Setup and Apply Interface ==========
    void setMeshCoderDriver(MeshCoderDriver *pTMC) { _pTMC = pTMC; }
    JtDualVFMesh *vfm() const                      { return _pDstVFM; }
    void  run();

    // ========== Generic encode/decode Driver Chain ==========
    void  clear();
    void  runComponent(bool &obFoundComponent);
    void  initNewComponent(bool &obFoundComponent);
    void  completeV(Int32 iFace);
    Int32 activateV(Int32 iVtx,  Int32 iVSlot);
    Int32 activateF(Int32 iFace, Int32 iFSlot);
    void  completeF(Int32 iVtx,  Int32 jFSlot);
    void  addVtxToFace (Int32 iVtx, Int32 iVSlot,
                        Int32 iFace,  Int32 iFSlot);

    // Active face list management
    void  addActiveFace(Int32 iFace);
    Int32 nextActiveFace();
    void  removeActiveFace(Int32 iFace);
    Int32 activeFaceOffset(Int32 iFace) const;

  private:
    // ========== Polymorphic I/O Interface ==========
    virtual Int32 ioVtxInit   ()                          = 0;
    virtual Int32 ioVtx       (Int32 iFace, Int32 jFSlot) = 0;
    virtual Int32 ioFace      (Int32 iVtx,  Int32 iVSlot) = 0;
    virtual Int32 ioSplitFace (Int32 iVtx,  Int32 iVSlot) = 0;
    virtual Int32 ioSplitPos  (Int32 iVtx,  Int32 iVSlot) = 0;

    // ========== Member Data ==========
  protected:
    MeshCoderDriver         *_pTMC;          // TopoDualMeshCoder this codec is attached
to
    SharedPtr<JtDualVFMesh>  _pSrcVFM;       // Input VFMesh
    SharedPtr<JtDualVFMesh>  _pDstVFM;       // Output VFMesh
    Veci                     _viActiveFaces; // Stack of incomplete "active faces"
    BitVec                   _vbRemovedActiveFaces;   // Helper bitvec parallel to above
    // Used by decoder to assign running attr indices
    Int32              _iFaceAttrCtr;
};


// Runs the mesh encoder/decoder machine.
// If decoding is being performed, it consumes the mesh
// coding symbols from pre-filled member variables to produce
// the output VFMesh _pDstVFM.
void MeshCodec::run()
{
    // Assert state is consistent and ready to co/dec
    if (!_pDstVFM)
        _pDstVFM = new JtDualVFMesh();
    Assert(_pDstVFM);
    _pDstVFM->clear();
    clear();

    // Co/dec connected mesh components one at a time
    bool bFoundComponent = JtTrue;
    while (bFoundComponent) {
        Bool bRetVal = runComponent(bFoundComponent);
        Assert (bRetVal);
    }
}

void MeshCodec::clear()
{
    // Setup
    _viActiveFaces.setLength(0);
    _vbRemovedActiveFaces.setLength(0);
    _iFaceAttrCtr = 0;
}

// Decodes one "connected component" (contiguous group of polygons) into
// _pDstVFM. Because the polygonal model may be formed of multiple
//  disconnected mesh components, it may be necessary for run() to call this
// method multiple times.  This method returns obFoundComponent = True
// if it actually encoded a new mesh component, and obFoundComponent = False
// if it did not.
void MeshCodec::runComponent(bool &obFoundComponent)
{
    Int32 iFace;
    initNewComponent(obFoundComponent);
    if (!obFoundComponent)
        return;
    while ((iFace = nextActiveFace()) != -1) {
        completeF(iFace);
        removeActiveFace(iFace);
    }
}

// Locates an unencoded vertex and begins the encoding
// process for the newly-found mesh component.
void MeshCodec::initNewComponent(bool &obFoundComponent)
{
    obFoundComponent = JtTrue;

    // Call ioVtxInit() to start us off with the seed face
    // from a new "connected component" of polygons.
    Int32 iVtx, i;
    if ((iVtx = ioVtxInit()) == -1) {
        obFoundComponent = JtFalse;  // All vtxs are processed
        return;
    }
    Int32 cVal = _pDstVFM->valence(iVtx);
    for (i = 0 ; i < cVal ; i++) {
        Int32 iFace = activateF(iVtx, i);    // Process all faces
        if (iFace == -2) {
            Assert(0 && “Mesh traversal failed”);
            return false;
        }
    }
}

// Completes the VFMesh face iFace on _pDstVFM by calling activateV() and
// completeV() for each as-yet inactive incident vertexes in the face's
// degree ring.
void MeshCodec::completeF(Int32 iFace)
{
    // While there is an empty vtx slot on the face
    Int32 jVtxSlot, iVtx;
    Int32 iVSlot = 0;
    while ((jVtxSlot = _pDstVFM->findVtxSlot(iFace, -1)) != -1) {
        // Create and return a vtx iVtx, attaching it to iFace at vtx
        // slot jVtxSlot.
        iVtx = activateV(iFace, jVtxSlot);

        // Assert FV consistency
        Assert(_pDstVFM->vtx (iFace,  jVtxSlot) == iVtx &&
               _pDstVFM->face(iVtx,   iVSlot)   == iFace   );

        // Process the faces of iVtx starting from face slot
        // jVtxSlot where iVtx is incident on iFace.
        completeV(iVtx, jVtxSlot);

        // Invariant "VF": vtx(iVtx).face(iVSlot) == iFace &&
        //                 face(iFace).vtx(jVtxSlot) == iVtx
    }
}

// "Activates" the VFMesh face, on _pDstVFM, at face iFace vertex slot iVSlot
// by calling ioFace() to obtain a new vertex number and hooking it up to the
// topological structure.  If the face is a SPLIT face, then call
// ioSplitFace() and ioSplitPos() to get the information necessary to connect
// to an already-active face.  Note that we use the term "activate" here to
// mean "read" for mesh decoding.
```

```
Int32 MeshCodec::activateF(Int32 iVtx, Int32 iVSlot)
{
    Int32 jFSlot;
    // ioFace might return -2 as an error condition
    Int32 iFace = ioFace(iVtx, iVSlot);
    if (iFace >= 0) {   // If a new active face
        if (!_pDstVFM->setVtxFace(iVtx, iVSlot, iFace) ||
            !_pDstVFM->setFaceVtx(iFace, 0, iVtx)      ||
            !addActiveFace(iFace)                            )
        {
            return -2;
        }
    }
    else if (iFace == -1) {                    // Face already exists, so Split
        iFace = ioSplitFace(iVtx, iVSlot);    // v's index in ActiveSet, returns v
        jFSlot = ioSplitPos(iVtx, iVSlot);    // Position of iVtx in v
        if (iFace == -2 || iVSlot == -1)
            return -2;
        _pDstVFM->setVtxFace(iVtx, iVSlot, iFace);
        addVtxToFace(iVtx, iVSlot, iFace, jFSlot);
    }
    return iFace;
}

// "Activates" the VFMesh vertex, on _pDstVFM, at face iFace vertex slot iVSlot
// by calling ioFace() to obtain a new face number and hooking it up to the
// topological structure.  Note that we use the term "activate" here to
// mean "read" for mesh decoding.
Int32 MeshCodec::activateV(Int32 iFace, Int32 iVSlot)
{
    Int32 iVtx = ioVtx(iFace, iVSlot);   // I/O valence; create a vtx
    _pDstVFM->setVtxFace(iVtx, 0, iFace);
    addVtxToFace (iVtx, 0, iFace, iVSlot);
    return iVtx;
}

// Completes the vertex iVtx on _pDstVFM by activating all inactive faces
// incident upon it.  As an optimization, the user shall also pass in iVSlot
// which is the vertex slot on face 0 of iVtx where iVtx is located.  This
// method begins its examination of iVtx's faces at face 0 by working its
// way around the vertex in both CCW and CW directions, checking to see if there
// are any faces that can be hooked into iVtx without calling activateF().
// This can happen when a face is completed by a nearby vertex before coming
// here.  The situation can be detected by traversing the topology of the
// _pDstVFM over to the neighboring vertex and checking if it already has a
// face number for the corresponding face entry on iVtx.  If so, then
// iVtx and the already completed face are connected together, and the
// next face around iVtx is examined. When the process can go no further,
// this method calls _activateF() on the remaining unresolved span of faces
// around the vertex.
void MeshCodec::completeF(Int32 iVtx, Int32 iVSlot)
{
    JtDualVFMesh *pDstVFM = _pDstVFM;
    Int32 i, vp, vn, jp, jn,
            iVtx2,
            cVal = pDstVFM->valence(iVtx);

    // Walk CCW from face slot 0, attempting to link in as many
    // already-reachable faces as possible until we reach one
    // that is inactive.
    vp = pDstVFM->face(iVtx, 0);
    jp = iVSlot;
    i = 1;
    JtDebugOnly(_assertParallelValRings(vp);)
    while ((vn = pDstVFM->face(iVtx, i)) != -1) {  // Forces "FV" in the "next" direction
        DecModN(jp, pDstVFM->degree(vp));
        iVtx2 = pDstVFM->vtx(vp, jp);
        if (iVtx2 == -1)
            break;
        jn = pDstVFM->findVtxSlot(vn, iVtx2);
        Assert(jn > -1);
        DecModN(jn, pDstVFM->degree(vn));
        addVtxToFace(iVtx, i, vn, jn);
        vp = vn;
        jp = jn;
        i++;
        if (i >= cVal)
            return;
    }

    // Walk CW from face slot 0, attempting to link in as many
    // already-reachable faces as possible until we reach one
    // that is inactive.
    Int32 ilast = i;
    vp = pDstVFM->face(iVtx, 0);
    jp = iVSlot;
    i = pDstVFM->valence(iVtx) - 1;
    while ((vn = pDstVFM->face(iVtx, i)) != -1) { // Forces "VF" in "prev" direction
        IncModN(jp, pDstVFM->degree(vp));
        iVtx2 = pDstVFM->vtx(vp, jp);
        if (iVtx2 == -1)
            break;
        jn = pDstVFM->findVtxSlot(vn, iVtx2);
        Assert(jn > -1);
        IncModN(jn, pDstVFM->degree(vn));
        addVtxToFace(iVtx, i, vn, jn);
        vp = vn;
        jp = jn;
        i--;
        if (i < ilast)
            return;
    }

    // Activate the remaining faces on iVtx that cannot be decuced from
    // the already-assembled topology in the destination VFMesh.
    for (; ilast <= i ; ilast++) {
        Int32 iFace = activateV(iVtx, ilast);
        JtDemandState(iFace >= -1);
    }
}

// This method connects vertex iVtx into the topology of
// _pDstVFM at and around iFace.  First, it connects iVtx
// to iFace's degree ring at position iVSlot.  Next, it
// will connect iVtx into the faces at the other ends of
// the shared edges between iVtx and the next vertices CS and
// CCW about iFace if necessary.
void MeshCodec::addVtxToFace (Int32 iVtx,  Int32 jFSlot,
                              Int32 iFace, Int32 iVSlot)
{
    Int32   iVSlotCW  = iVSlot,
            iVSlotCCW = iVSlot,
            fp, ip,
            fn, in;
    JtDualVFMesh *pDstVFM = _pDstVFM;
    IncModN(iVSlotCCW, pDstVFM->degree(iFace));
    DecModN(iVSlotCW,  pDstVFM->degree(iFace));

    // Connect iVtx to iFace/iVSlot
    JtRethrow(pDstVFM->setFaceVtx(iFace, iVSlot, iVtx));

    // Connect iVtx across the shared edge between iVtx and the vtx CW
    // from iVtx at iFace.  Connect iVtx into the face at the other
    // end of this edge if it is not already connected there.
    if ((fp = pDstVFM->vtx(iFace, iVSlotCW)) != -1) {
        ip = pDstVFM->findFaceSlot(fp, iFace);
        Int32 iVSlotCCW = jFSlot;
        IncModN(iVSlotCCW, pDstVFM->valence (iVtx));
        if (pDstVFM->face(iVtx, iVSlotCCW) == -1) {
            DecModN(ip, pDstVFM->valence(fp));
            pDstVFM->setVtxFace(iVtx, iVSlotCCW, pDstVFM->face(fp, ip));
        }
    }
    // Connect iVtx across the shared edge between iVtx and the vtx CCW
    // from iVtx at iFace.  Connect iVtx into the face at the other
    // end of this edge if it is not already connected there.
    if ((fn = pDstVFM->vtx(iFace, iVSlotCCW)) != -1) {
        in = pDstVFM->findFaceSlot(fn, iFace);
        Int32 iVSlotCW  = jFSlot;
        DecModN(iVSlotCW,  pDstVFM->valence (iVtx));
        if (pDstVFM->face(iVtx, iVSlotCW) == -1) {
            IncModN(in, pDstVFM->valence(fn));
            pDstVFM->setVtxFace(iVtx, iVSlotCW, pDstVFM->face(fn, in));
        }
    }
}

void MeshCodec::addActiveFace(Int32 iFace)
{
    JtRethrow(_viActiveFaces.pushBack(iFace));
}

// Returns a face from the active queue to be completed. This needn't be the
// one at the end of the queue, because the choice of the next active face
// can affect how many SPLIT symbols are produced.  This method employs a
// fairly simple scheme of searching the most recent 16 active faces for the
// fist one with the smallest number of incomplete slots in its degree ring.
Int32 MeshCodec::nextActiveFace()
{
    Int32 iFace = -1;
    // Search the 16 face record at the end of the
    // queue for the one with lowest remaining degree.
    while (_viActiveFaces.length() > 0 &&
_vbRemovedActiveFaces.test(_viActiveFaces.back()))
        _viActiveFaces.popBack();
    Int32 cLowestEmptyDegree = 9999999;
    Int32 i, iFace0, cEmptyDeg;
    const Int32 cWidth = 16;
    JtDualVFMesh *pDstVFM = _pDstVFM;
    for (i = _viActiveFaces.length() - 1 ;
         i >= ::jtmax(0, _viActiveFaces.length() - cWidth) ;
         i--)
    {
        iFace0 = _viActiveFaces[i];
        if (_vbRemovedActiveFaces.test(iFace0)) {
            _viActiveFaces.remove(i); // TOXIC: O(N^2)
            continue;
        }
        cEmptyDeg = pDstVFM->emptyFaceSlots(iFace0);
        if (cEmptyDeg < cLowestEmptyDegree) {
            cLowestEmptyDegree = cEmptyDeg;
            iFace = iFace0;
        }
    }

    // Return the selected active face
    return iFace;
}

// Removes iFace from the active face queue.
void MeshCodec::removeActiveFace(Int32 iFace)
{
    _vbRemovedActiveFaces.set(iFace);
}

// Searches the active face queue for iFace and returns
// its index position from the _end_ of the queue.  This is
// needed by the ioFace() method when encoding a SPLIT
// symbol.
Int32 MeshCodec::activeFaceOffset(Int32 iFace) const
{
    Int32 iOffset = -1;
    Int32 i, cLen = _viActiveFaces.length();
    const Int32 *paiActiveFaces = _viActiveFaces.ptr();
    for (i = cLen - 1 ; i >= 0 ; i--) {
        if (paiActiveFaces[i] == iFace) {
            // The offset is how far FROM THE END of the active
            // face list we found iFace.  This serves the make
            // the iOffset a much smaller number, which is better
            // for compression!
            iOffset = cLen - i;
            break;
        }
    }
    return iOffset;
}
```

D.2.3 MeshDecoder class

```
// This class implements the five abstract methods from
// MeshCodec to realize a mesh decoder.
class MeshDecoder : public MeshCodec {
  public:
    // ========== Housekeeping Interface ==========
    MeshDecoder (MeshCoderDriver *pTMC = NULL);
  protected:
    virtual ~MeshDecoder() {}

  private:
    // ========== Polymorphic I/O Interface ==========
    virtual Int32 ioVtxInit  ()                          ;
    virtual Int32 ioVtx      (Int32 iFace, Int32 iVSlot);
    virtual Int32 ioFace     (Int32 iVtx , Int32 jFSlot);
    virtual Int32 ioSplitFace(Int32 iVtx , Int32 jFSlot);
    virtual Int32 ioSplitPos (Int32 iVtx , Int32 jFSlot);
};

// Begins decoding a new connected mesh component by calling
// ioVtx() to read the next vertex from the symbol stream.
Int32 MeshDecoder::ioVtxInit()
{
    return ioVtx(-1, -1);
}

// Read a vertex valence symbol, vertex group number, and vertex
// flags from the input symbols stream.  Create a new vertex
// on _pDstVFM with this data, and return the new vertex number.
// It is this method's responsibility to detect the end of
// the input symbol stream by returning -1 when that happens.
Int32 MeshDecoder::ioVtx (Int32 /*iFace*/ , Int32 /*iVSlot*/)
{
    // Obtain a VERTEX VALENCE symbol
    Int32 eSym = _pTMC->_nextValSymbol();
    Int32 iVtxVal, iVtx = -1;
    if (eSym > -1) {
        // Create a new vtxt on the VFMesh
        iVtx = _pDstVFM->numVts();
        iVtxVal = eSym;
        _pDstVFM->newVtx     (iVtx, iVtxVal);
        _pDstVFM->setVtxGrp  (iVtx, _pTMC->_nextFGrpSymbol());
        _pDstVFM->setVtxFlags(iVtx, _pTMC->_nextVtxFlagSymbol());
    }

    return iVtx;
}

// Read a face degree symbol, and attribute mask bit
// vector, create a new DualVFMesh face, initialize the
// face attribute record numbers from a running counter,
// and return the new face number.  If the degree symbol
// read from the input symbol stream is 0, signify this by
// returning -1.
Int32
MeshDecoder::ioFace     (Int32 iVtx, Int32 /*jFSlot*/)
{
    // Obtain a FACE DEGREE symbol
    Int32 iCntxt = _pTMC->_faceCntxt(iVtx, _pDstVFM);
    Int32 eSym = _pTMC->_nextDegSymbol(iCntxt);
    Int32 cDeg, iFace = -1;
    if (eSym != 0) {
        // Create a new face on the VFMesh
        iFace = _pDstVFM->numFaces();
        cDeg = eSym;
        Int32 nFaceAttrs = 0;
        if (cDeg <= JtDualVFMesh::cMBits) {
            UInt64 uAttrMask = _pTMC-
>_nextAttrMaskSymbol(/*iCntxt*/::jtmin(7,::jtmax(0,cDeg-2)));
            for (UInt64 uMask = uAttrMask ; uMask ; nFaceAttrs += (uMask & 1), uMask >>=
1);
            _pDstVFM->newFace(iFace, cDeg, nFaceAttrs, uAttrMask);
        }
        else {
            BitVec vbAttrMask;
            _pTMC->_nextAttrMaskSymbol(&vbAttrMask, cDeg);
            for (Int32 i = 0 ; i < cDeg ; i++) {
                if (vbAttrMask.test(i))
                    nFaceAttrs++;
            }
            _pDstVFM->newFace(iFace, cDeg, nFaceAttrs, &vbAttrMask, 0);
        }

        // Error check for a corrupt degree or attrmask
        if (nFaceAttrs > cDeg) {
            Assert (nFaceAttrs <= cDeg);
            return -2;
        }

        // Set up the face attributes
        for (Int32 iAttrSlot = 0 ; iAttrSlot < nFaceAttrs ; iAttrSlot++) {
            _pDstVFM->setFaceAttr(iFace, iAttrSlot, _iFaceAttrCtr++);
        }
    }

}

// Consumes a split offset symbol from the SPLIT offset
// symbol stream, and determines the face number referenced
// by the offset.  Returns the referenced face number.
Int32 MeshDecoder::ioSplitFace(Int32 /*iVtx*/, Int32 /*jFSlot*/)
{
    // Obtain a SPLITFACE symbol
    Int32 eSym = _pTMC->_nextSplitFaceSymbol();
    Assert(eSym >= -1);
    Int32 iOffset = -1, iFace = -1;
    if (eSym > -1) {
        // Use the offset to index into the active face queue
        // to determine the actual face number.
        iOffset = eSym;
        Int32 cLen = _viActiveFaces.length();
        // Error check for a corrupt offset
        if (iOffset <= 0 || iOffset > cLen) {
            Assert(iOffset > 0 && iOffset <= cLen);
            return -2;
        }
        iFace = _viActiveFaces[cLen - iOffset];
    }

    return iFace;
}

// Consumes a split position symbol from the associated symbol
// stream, and returns the vertex slot number on the current
// split face at which the topological split/merge occurred.
Int32 MeshDecoder::ioSplitPos   (Int32 /*iVtx*/, Int32 /*jFSlot*/)
{
    // Obtain a SPLITVTX symbol
    Int32 eSym = _pTMC->_nextSplitPosSymbol();
    Assert(eSym >= -1);
    Int32 iVSlot = -1;
    if (eSym > -1) {
        // Return the vtx slot number
        iVSlot = eSym;
    }

    return iVSlot;
}
```

Annex E

XT B- Rep data segment

E.1 Introduction to the XT B- Rep data segment

The XT B- Rep data segment in a JT file is a netural binary definition of the solid body. A proprietary kernel is not required to read the XT B-Rep data segment. The model definition in the XT B-Rep data segment of a JT file is a fully described geometric and topological representation.

E.2 Logical Layout

The logical layout of a XT B- Rep data segment is:

—A short flag sequence describing the data format, followed by modeller identification information and user field size.

- The various flag sequences (mixtures of text and numbers) are documented under `Physical layout'.
- The content of the modeller identification information is:

  The version of the Parasolid Kernel (if appropriate) used to write the data, as a text string of the form:

  : TRANSMIT FILE created by modeller version 1200123The schema version describing the field sequences of the part nodes as a text string of the form:

  SCH_1200123_12006

  This example above denotes XT data written by the Parasolid Kernel V12.0.123 using schema number 12006.

  Applications writing XT data segments without using the Parasolid Kernel should use version 1200000 and schema number 12006 (i.e.):

  : TRANSMIT FILE created by modeller version 1200000

  SCH_1200000_12006

- The user field size is a simple integer.

—The objects (known as ‗nodes‘) in the XT data in an unordered sequence, followed by a terminator.

- Every node in the XT data is assigned an integer index from 1 upwards (some indices may not be used). Pointer fields are output as these indices, or as zero for a null pointer.
- Each node entry begins with the node type. If the node is of variable length (see below), this is followed by the length of the variable field. The index of the node is then output, followed by the fields of the node.
- The terminator which follows the sequence of nodes is a two-byte integer with value 1, followed by an index with value 0. The index is output as a 2-byte integer with value 1 in binary XT data.
- The node with index 1 is the root node of the dataas follows:

Table F.1 — Object Nodes

Contents of XT data	Type of root node
Body	BODY
Assembly	ASSEMBLY
Array of parts	POINTER_LIS_BLOCK
Partition	WORLD

E.2.1 Schema

XT permanent structures are defined in a language akin to C which generates the appropriate files for a C compiler, the runtime information used by the Parasolid Kernel, along with an optional schema file that can be used during transmit and receive. The schema file for version 12.0 is named sch_12006 and is distributed with the Parasolid kernel. However, it is not necessary for applications reading and writing XT data directly to have a copy of this schema file to understand the XT

E.2.2 Embedded schemas

XT parts, partitions and deltas can be transmitted with extra information that is intended to replace the schema used to describe the data layout. This information contains the differences between its schema and a defined base schema.The only fields that are included in this information are those which can be referenced in a cut-down version of the schema pertaining only to the XT part data that is present. Specifically, a full schema definition can contain fields that are not relevant in the context of the transmitted data and these fields are excluded.

Fields that are included are referred to as effective fields, and are either transmittable (`xmt_code` == 1) or have variable-length (`n_elts` == 1)

E.2.2.1 Physical layout

Most of the XT data are composed of integers, logical flags, and strings, but are of restricted ranges and so transmitted specially in binary format. The binary representation is given in bold type, such as —integer (**byte**)”. This is relevant to applications that attempt to read or write XT data directly. Two important elements are

- short strings

These are transmitted as an integer length (byte) followed by the characters (without trailing zero).

- positive integers

These are transmitted similarly to the pointer indices which link individual objects together, i.e., small values 0..32766 are transmitted as a single **short** integer, larger ones encoded into two.

E.2.2.2 XT format

Presence of the new format is indicated by a change to the standard header: the archive name is extended by the number of the base schema, e.g., `SCH_1400068_14000_13006`, and then the maximum number of node types is inserted (short).

Transmission then continues as normal, except that when transmitting the first node of any particular type, extra information is inserted between the nodetype and the variable-length, index data as follows:

— The arrays of effective fields in the base schema node and the current schema node are assembled.

— If the nodetype does not exist in the base schema then it is output as follows:

- number of fields (byte),
- name and description (short strings), and
- fields one by one as shown in the table F.2.

Table F.2 — Field types in order one by one

name	short string	
ptr_class	Short	
n_elts	Positive integer	
type	short string	The field type. Allowed values are described in —Field types”, below. Omitted if

		ptr_class non-zero
xmt_code	logical (byte)	Omitted for fixed-length (n_elts != 1)

—

—If the two arrays match (equal length and all fields match in `name`, `xmt_code`, `ptr_class`, `n_elts` and `type`) then output the flag value 255 (**byte** 0xff).

—If the two arrays do not match, output the number of effective fields in the current schema (byte), and an edit sequence as follows:

– Initialize pointers to the first base field and first current field, then while there are still unprocessed base and current fields, output a sequence of Copy, Delete and Insert instructions.

– If the base field matches the current field, output 'C' (char) to indicate an unchanged (Copied) field and advance to the next base and current fields.

– If the base field does not match any unprocessed current field, output 'D' (char) to indicate a Deleted field and advance to the next base field.

– Otherwise, output 'I' (char) to indicate an Inserted field, followed by the current field in the above format, and advance to the next current field.

– If there are any unprocessed current fields, then output an Append sequence, each instruction being 'A' (**char**) followed by the field.

— Finally, output 'Z' (**char**) to signal the end.

E.2.3 Field types

The XT format is not itself a binary protocol, and so does not define data sizes; the only requirement is that a runtime implementation has sufficient room for the information. The available implementations run with 8bit ASCII characters, 8bit unsigned bytes (0..255), 16bit short integers (0..65535 or -32768..32767), 32bit integers (0..4G-1, -2G..2G-1) and IEEE reals. The implementation used in the given binary XT data is specified by the "PS<code>" at the start of the XT data. See the chapter on —Pysical Layout” for more information.

The full list of field types used in XT segment data is as follows:

u unsigned byte 0-255

c char

l unsigned byte 0-1 (i.e. logical)

```
            typedef char logical;
```

n short int

w unicode character, output as a short int

d Int

p pointer-index

Small indices (less than 32767) are treated specially in binary XT data to save space. See the section below on binary format.

f double

i These correspond to a region of the real line:

```
typedef struct { double low, high; }interval;
```

v array [3] of doubles

These correspond to a 3-space position or direction:

```
typedef struct { double x,y,z; } vector;
```

b array [6] of doubles

These correspond to a 3-spce region:

```
typedef struct { interval x,y,z; } box;
```

h array [3] of doubles

These represent points of intersection between two surfaces; only the position vector is written to the XT data.The structure is documented further in the section on intersection curves.

E.2.4 Variable-length nodes

Variable-length nodes differ from fixed-length nodes in that their last field is of variable length, i.e. different nodes of the same type may have different lengths.

The number of entries in each such node is indicated by an integer in the XT data between its nodetype and index, so an example might be

```
83 3 15 1 2 3
```

E.2.5 Unresolved indices

In some cases a node will contain an index field which does not correspond to a node in the XT data, in this case the index is to be interpreted as zero.

E.3 Physical Layout

E.3.1 Binary

The flag sequence is followed by the length of the modeller version as a 2-byte integer, the characters of the modeller version, the length of the schema version as a 4-byte integer, the characters of the schema version, and finally the userfield size as a 4-byte integer.

There are two special numeric values (-32764 for integral values, -3.14158e13 for floating point) which are used to mark an ‗unset‘ or ‗null‘ value.

E.3.1.1 Neutral Binary

In the XT Segment neutral binary, data is represented in big-endian format, with IEEE floating point numbers and ASCII characters. The flag sequence is the 4-byte sequence "PS" followed by two zero bytes, i.e., 'P' 'S' '\0' '\0'. The initial letters are ASCII, thus '\120' '\123'.The nodetype at the start of a node is a 2-byte integer, the variable length which may follow it is a 4-byte integer.

Logical values (0,1) are represented as themselves in 1 byte.

Small pointer indices (in the range 0-32766) are implemented as a 2-byte integer, larger indices are represented as a pair, thus:

```
if (index < 32767)
      {                               // case: small index

      op_short( index + 1 );          // offset so is > 0

      }

else
```

```
          {                             // case: big index

          op_short( -(index % 32767 +    // remainder: add 1 so > 0
          1) );

          op_short( index / 32767 );     // nonzero quotient

          }
```

where op_short outputs a 2-byte integer.

The inverse is performed on reading:

```
    short q = 0, r;
    ip_short( &r );
    if (r < 0)
          {

          ip_short( &q );

          r = -r;

          }

    index = q * 32767 + r - 1;
```

where ip_short reads a 2-byte integer.

E.4 Model Structure

E.4.1 Topology

This section describes the XT Topology model, it gives an overview of how the nodes in theXT data are joined together.

The topological representation allows for:

- non-manifold solids,

- solids with internal partitions,

- bodies of mixed dimension (i.e. with wire, sheet, and solid `bits'),

- pure wire-frame bodies, and

- disconnected bodies.

Each entity is described, and its properties and links to other entities given.

E.4.2 General points

In this section a set is called finite if it can be enclosed in a ball of finite radius - not that it has a finite number of members.

A set of points in 3-dimensional space is called open if it does not contain its boundary.

Back-pointers, next and previous pointers in a chain, and derived pointers are not described explicitly here. For information on this see the following description of the schema-level model.

E.4.3 Entity definitions

E.4.3.1 Assembly

An assembly is a collection of instances of bodies or assemblies. It may also contain construction geometry. An assembly has the following fields:

— a set of instances, and

— a set of geometry (surfaces, curves and points).

E.4.3.2 Instance

An instance is a reference to a body or an assembly, with an optional transform:

— Body or assembly.

— Transform. If null, the identity transform is assumed.

E.4.3.3 Body

A body is a collection of faces, edges and vertices, together with the 3-dimensional connected regions into which space is divided by these entities. Each region is either **solid** or **void** (indicating whether it represents material or not).

The point-set represented by the body is the disjoint union of the point-sets represented by its solid regions, faces, edges, and vertices. This point-set need not be connected, but it shall be finite.

A body has the following fields:

— A set of regions.

> A body has one or more regions. These, together with their boundaries, make up the whole of 3-space, and do not overlap, except at their boundaries. One region in the body is distinguished as the exterior region, which shall be infinite; all other regions in the body shall be finite.

— A set of geometry (surfaces, curve and/or points).

— A body-type. This may be wire, sheet, solid or general.

E.4.3.4 Region

A region is an open connected subset of 3-dimensional space whose boundary is a collection of vertices, edges, and oriented faces.

Regions are either solid or void, and they may be non-manifold. A solid region contributes to the point-set of its owning body; a void region does not (although its boundary will).

Two regions may share a face, one on each side.

A region may be infinite, but a body shall have exactly one infinite region. The infinite region of a body shall be void.

A region has the following fields:

— A logical indicating whether the region is solid.

— A set of shells. The positive shell of a region, if it has one, is not distinguished.

The shells of a region do not overlap or share faces, edges or vertices.

A region may have no shells, in which case it represents all space (and will be the only region in its body, which will have no faces, edges or vertices).

E.4.3.5 Shell

A shell is a connected component of the boundary of a region. As such it will be defined by a collection of faces, each used by the shell on one `side', or on both sides; and some edges and vertices.

A shell has the following fields:

- A set of (face, logical) pairs.

  Each pair represents one side of a face (where true indicates the front of the face, i.e. the side towards which the face normal points), and means that the region to which the shell belongs lies on that side of the face. The same face may appear twice in the shell (once with each orientation), in which case the face is a 2-dimensional cut subtracted from the region which owns the shell.

- A set of wireframe edges.

  Edges are called **wireframe** if they do not bound any faces, and so represent 1-dimensional cuts in the shell's region. These edges are not shared by other shells.

- A vertex.

  This is only non-null if the shell is an **acorn** shell, i.e. it represents a 0-dimensional hole in its region, and has one vertex, no edges and no faces.

A shell shall contain at least one vertex, edge, or face.

E.4.3.6 Face

A face is an open finite connected subset of a surface, whose boundary is a collection of edges and vertices. It is the 2-dimensional analogy of a region.

A face has the following fields:

- A set of loops. A face may have zero loops (e.g. a full spherical face), or any number.
- Surface. This may be null, and may be used by other faces.
- Sense. This logical indicates whether the normal to the face is aligned with or opposed to that of the surface.

E.4.3.7 Loop

A loop is a connected component of the boundary of a face. It is the 2-dimensional analogy of a shell. As such it will be defined by a collection of fins and a collection of vertices.

A loop has the following fields:

- An ordered ring of fins.

  Each fin represents the oriented use of an edge by a loop. The sense of the fin indicates whether the loop direction and the edge direction agree or disagree. A loop may not contain the same edge more than once in each direction.

  The ordering of the fins represents the way in which their owning edges are connected to each other via common vertices in the loop (i.e. nose to tail, taking the sense of each fin into account).

  The loop direction is such that the face is locally on the left of the loop, as seen from above the face and looking in the direction of the loop.

—A vertex.

This is only non-null if the loop is an isolated loop, i.e. has no fins and represents a 0-dimensional hole in the face.

Consequently, a loop shall consist either of:

—A single fin whose owning ring edge has no vertices, or

—At least one fin and at least one vertex, or

—A single vertex.

E.4.3.8 Fin

A fin represents the oriented use of an edge by a loop.

A fin has the following fields:

—A logical **sense** indicating whether the fin's orientation (and thus the orientation of its owning loop) is the same as that of its owning edge, or different.

—A curve. This is only non-null if the fin‘s edge is tolerant, in which case every fin of that edge will reference a trimmed SP-curve. The underlying surface of the SP-curve shall be the same as that of the corresponding face. The curve shall not deviate by more than the edge tolerance from curves on other fins of the edge, and its ends shall be within vertex tolerance of the corresponding vertices.

E.4.3.9 Edge

An edge is an open finite connected subset of a curve; its boundary is a collection of zero, one or two vertices. It is the 1-dimensional analogy of a region.

An edge has the following fields:

—Start vertex.

—End vertex. If one vertex is null, then so is the other; the edge will then be called a **ring** edge.

—An ordered ring of distinct fins.

—The ordering of the fins represents the spatial ordering of their owning faces about the edge (with a right-hand screw rule, i.e. looking in the direction of the edge the fin ordering is clockwise). The edge may have zero or any number of fins; if it has none, it is called a **wireframe** edge.

—A curve. This will be null if the edge has a tolerance. Otherwise, the vertices shall lie within vertex tolerance of this curve, and if it is a Trimmed Curve, they shall lie within vertex tolerance of the corresponding ends of the curve. The curve shall also lie in the surfaces of the faces of the edge, to within modeller resolution.

—Sense. This logical indicates whether the direction of the edge (start to end) is the same as that of the curve.

—A tolerance. If this is null-double, the edge is **accurate** and is regarded as having a tolerance of half the modeller linear resolution, otherwise the edge is called **tolerant**.

E.4.3.10 Vertex

A vertex represents a point in space. It is the 0-dimensional analogy of a region.

A vertex has the following fields:

- A geometric point.

- A tolerance. If this is null-double, the vertex is **accurate** and is regarded as having a tolerance of half the modeller linear resolution.

E.4.3.11 Attributes

An attribute is an entity which contains data, and which can be attached to any other entity except attributes, fins, lists, transforms or attribute definitions. An attribute has the following fields:

- Definition. An attribute definition is an entity which defines the number and type of the data fields in a specific type of attribute, which entities may have such an attribute attached, and what happens to the attribute when its owning entity is changed. XT data shall not contain duplicate attribute definitions. Each attribute of a given type should reference the same instance of the attribute definition for that type. It is incorrect, for example, to create a copy of an attribute definition for each instance of the attribute of that type. Only those attribute definitions referenced by attributes in the part occur in thedata.

- Owner.

- Fields. These are data fields consisting of one or more integers, doubles, vectors etc.

There are a number of system attribute definitions which may be present in the XT data. These are documented in the section `System Attribute Definitions'. User attribute definitions can also be created. These are included in the XT data along with any attributes that use them.

E.4.3.12 Groups

A group is a collection of entities in the same part. Groups in assemblies may contain instances, surfaces, curves and points. Groups in bodies may contain regions, faces, edges, vertices, surfaces, curves and points. Groups have:

- Owning part.

- A set of member entities.

- Type. The type of the group specifies the allowed type of its members, e.g. a ‗face‘ group in a body may only contain faces, whereas a ‗mixed‘ group may have any valid members.

E.4.3.13 Node-ids

All entities in a part, other than fins, have a non-zero integer node-id which is unique within a part. This is intended to enable the entity to be identified within the XT data.

E.4.4 Entity matrix

Thus the relations between entities can be represented in matrix form as follows in Table F.3. The numbers represent the number of distinct entities connected (either directly or indirectly) to the given one.

Table F.3 — Entity Matrix relations

	Body	Region	Shell	Face	Loop	Fin	Edge	Vertex
Body	-	>0	any	any	any	any	any	any
Region	1	-	any	any	any	any	any	any
Shell	1	1	-	any	any	any	any	any
Face	1	1-2	1-2	-	any	any	any	any
Loop	1	1-2	1-2	1	-	any	any	any
Fin	1	1-2	1-2	1	1	-	1	0-2
Edge	1	any	any	any	any	any	-	0-2
Vertex	1	any	any	any	any	any	any	-

E.4.5 Representation of manifold bodies

E.4.5.1 Body types

XT bodies have a field body_type which takes values from an enumeration indicating whether the body is:

- **solid**, representing a manifold 3-dimensional volume, possibly with internal voids. It need not be connected;

- **sheet**, representing a 2-dimensional subset of 3-space which is either manifold or manifold with boundary (certain cases are not strictly manifold – see below for details). It need not be connected;

- **wire**, representing a 1-dimensional subset of 3-space which is either manifold or manifold with boundary, and which need not be connected. An **acorn** body, which represents a single 0-dimensional point in space, also has body-type wire;

- **general** - none of the above.

A general body is not necessarily non-manifold, but at the same time it is not constrained to be manifold, connected, or of a particular dimensionality (indeed, it may be of mixed dimensionality).

Restrictions on entity relationships for manifold body types

Solid, sheet, and wire bodies are best regarded as special cases of the topological model; for convenience we call them the manifold body types (although as stated above, a general body may also be manifold).

In particular, bodies of these manifold types shall obey the following constraints:

- An acorn body shall consist of a single void region with a single shell consisting of a single vertex.

- A wire body shall consist of a single void region, with one or more shells, consisting of one or more wireframe edges and zero or more vertices (and no faces). Every vertex in the body shall be used by exactly one or two of the edges (so, in particular, there are no acorn vertices).

- So each connected component will be either: closed, where every vertex has exactly two edges; or open, where all but two vertices have exactly two edges each.

- A wire is called open if all its components are open, and closed if all its components are closed.

- Solid and sheet bodies shall each contain at least one face; they may not contain any wireframe edges or acorn vertices.

- A solid body shall consist of at least two regions; at least one of its regions shall be solid. Every face in a solid body shall have a solid region on its negative side and a void region on its positive side (in other words, every face forms part of the boundary of the solid, and the face normals always point away from the solid).

— Every edge in a solid body shall have exactly two fins, which will have opposite senses. Every vertex in a solid body shall either belong to a single isolated loop, or belong to one or more edges; in the latter case, the faces which use those edges shall form a single edgewise-connected set (when considering only connections via the edges which meet at the vertex).

— These constraints ensure that the solid is manifold.

— All the regions of a sheet body shall be void. It is known as an open sheet if it has one region, and a closed sheet if it has no boundary.

— Every edge in a sheet body shall have exactly one or two fins; if it has two, these shall have opposite senses. In a closed sheet body, all the edges will have exactly two fins. Every vertex in a sheet body shall either belong to a single isolated loop, or belong to one or more edges; in the latter case, the faces which use those edges shall either form a single edgewise-connected set where all the edges involved have exactly two fins, or any number of edgewise-connected sets, each of which shall involve exactly two edges with one fin each (again, considering only connections via the edges which meet at the vertex).

Note that, although the constraints on edges and vertices in a sheet body are very similar to those which apply to a solid, in this case they do not guarantee that the body will be manifold; indeed, the rather complicated rules about vertices in an open sheet body specifically allow bodies which are non-manifold (such as a body consisting of two square faces which share a single corner vertex).

E.5 Schema Definition

E.5.1 Underlying types

```
union CURVE_OWNER_u

     {

     struct EDGE_s              *edge;

     struct FIN_s               *fin;

     struct BODY_s              *body;

     struct ASSEMBLY_s          *assembly;

     struct WORLD_s             *world;

     };


union SURFACE_OWNER_u

     {

     struct FACE_s              *face;

     struct BODY_s              *body;

     struct ASSEMBLY_s          *assembly;

     struct WORLD_s             *world;

     };


union ATTRIB_GROUP_u

     {

     struct ATTRIBUTE_s         *attribute;
     struct GROUP_s              *group;

     struct MEMBER_OF_GROUP_s    *member_of_group;

     };

typedef union ATTRIB_GROUP_u  ATTRIB_GROUP;
```

E.5.2 Geometry

```
union CURVE_u
     {

     struct LINE_s               *line;

     struct CIRCLE_s             *circle;

     struct ELLIPSE_s            *ellipse;

     struct INTERSECTION_s       *intersection;

     struct TRIMMED_CURVE_s      *trimmed_curve;

     struct PE_CURVE_s           *pe_curve;

     struct B_CURVE_s            *b_curve;

     struct SP_CURVE_s           *sp_curve;

     };

typedef union CURVE_u      CURVE;

union SURFACE_u
     {

     struct PLANE_s              *plane;

     struct CYLINDER_s           *cylinder;

     struct CONE_s               *cone;

     struct SPHERE_s             *sphere;

     struct TORUS_s              *torus;

     struct BLENDED_EDGE_s       *blended_edge;

     struct BLEND_BOUND_s        *blend_bound;

     struct OFFSET_SURF_s        *offset_surf;

     struct SWEPT_SURF_s         *swept_surf;

     struct SPUN_SURF_s          *spun_surf;

     struct PE_SURF_s            *pe_surf;

     struct B_SURFACE_s          *b_surface;

     };

typedef union SURFACE_u    SURFACE;

union GEOMETRY_u
```

```
     {

     union  SURFACE_u            surface;

     union  CURVE_u              curve;

     struct POINT_s             *point;

     struct TRANSFORM_s         *transform;

     };

typedef union GEOMETRY_u GEOMETRY;
```

E.5.2.1 Curves

In the following field tables, ‗pointer0‘ means a reference to another node which may be null. ‗pointer‘ means a non-null reference.

All curve nodes share the following common fields:

Table F.4 — Curve node common fields

Field name	Data type	Description
node_id	int	Integer value unique to curve in part
attributes_groups	pointer0	Attributes and groups associated with curve
owner	pointer0	topological owner
next	pointer0	next curve in geometry chain
previous	pointer0	previous curve in geometry chain
geometric_owner	pointer0	geometric owner node
sense	char	sense of curve: ‗+‘ or ‗-‘ (see end of Geometry section)

```
struct ANY_CURVE_s            // Any Curve
    {

    int                         node_id;                    // $d

    union  ATTRIB_GROUP_u       attributes_groups;          // $p

    union  CURVE_OWNER_u        owner;                      // $p

    union  CURVE_u              next;                       // $p

    union  CURVE_u              previous;                   // $p

    struct GEOMETRIC_OWNER_s   *geometric_owner;            // $p

    char                        sense;                      // $c

    };

typedef struct ANY_CURVE_s *ANY_CURVE;
```

E.5.2.1.1 Line

A straight line has a parametric representation of the form:

R(t) = P + t D

where

—P is a point on the line.

—D is its direction.

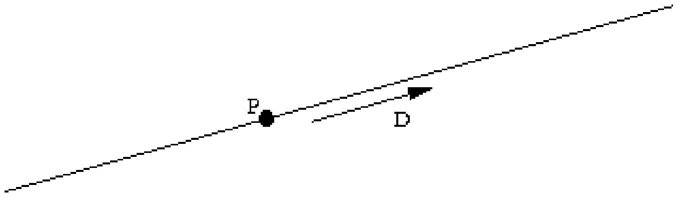


Table F.5 — Line Fields

Field name	Data type	Description
pvec	vector	point on the line
direction	vector	direction of the line (a unit vector)

```
struct LINE_s == ANY_CURVE_s     // Straight line
    {

    int                          node_id;                      // $d

    union  ATTRIB_GROUP_u        attributes_groups;            // $p

    union  CURVE_OWNER_u         owner;                        // $p

    union  CURVE_u               next;                         // $p

    union  CURVE_u               previous;                     // $p

    struct GEOMETRIC_OWNER_s    *geometric_ owner;             // $p

    char                         sense;                        // $c

    vector                       pvec;                         // $v

    vector                       direction;                    // $v

    };

typedef struct LINE_s      *LINE;
```

E.5.2.1.2 CIRCLE

A circle has a parametric representation of the form

R(t) = C+ r X cos(t) + r Y sin(t)

Where

—C is the centre of the circle.

—r is the radius of the circle.

—X and Y are the axes in the plane of the circle.

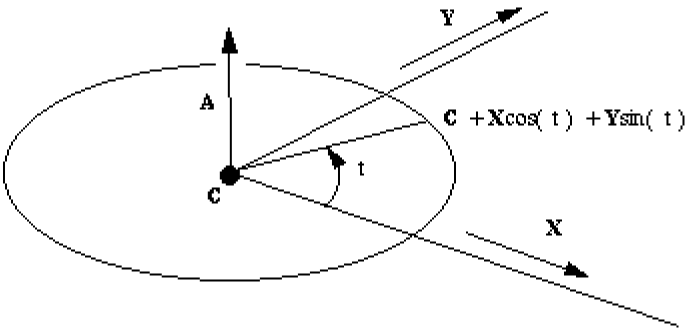


Table F.6 — Circle fields

Field name	Data type	Description
centre	vector	Centre of circle
normal	vector	Normal to the plane containing the circle (a unit vector)
x_axis	vector	X axis in the plane of the circle (a unit vector)
radius	double	Radius of circle

The Y axis in the definition above is the vector cross product of the normal and x_axis.

```
struct CIRCLE_s == ANY_CURVE_s                        // Circle
    {
    int                           node_id;                        // $d
    union  ATTRIB_GROUP_u         attributes_groups;              // $p
    union  CURVE_OWNER_u          owner;                          // $p
    union  CURVE_u                next;                           // $p
    union  CURVE_u                previous;                       // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;                // $p
    char                          sense;                          // $c
    vector                        centre;                         // $v
    vector                        normal;                         // $v
    vector                        x_axis;                         // $v
    double                        radius;                         // $f
    };
typedef struct CIRCLE_s    *CIRCLE;
```

E.5.2.1.3 ELLIPSE

An ellipse has a parametric representation of the form

R(t) = C+ a X cos(t) + b Y sin(t)

where

- C is the centre of the circle.
- X is the major axis.

—r is the major radius.

—Y and b are the minor axis and minor radius respectively.

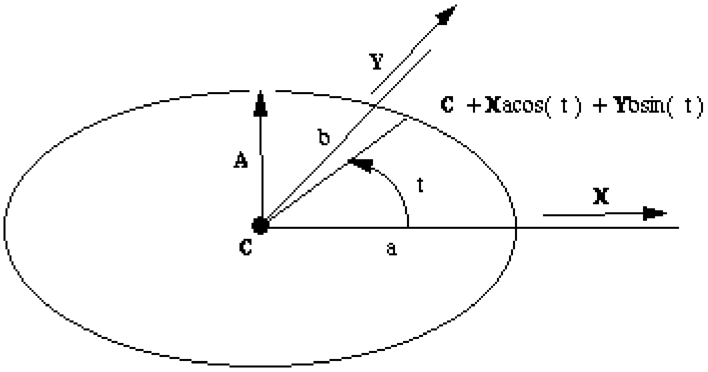


Table F.7 — Ellipse fields

Field name	Data type	Description
centre	Vector	Centre of ellipse
normal	Vector	Normal to the plane containing the ellipse (a unit vector)
x_axis	Vector	major axis in the plane of the ellipse (a unit vector)
major_radius	Double	major radius
minor_radius	Double	minor radius

The minor axis (Y) in the definition above is the vector cross product of the normal and x_axis.

```
struct ELLIPSE_s == ANY_CURVE_s      // Ellipse
    {
    int                             node_id;                  // $d
    union  ATTRIB_GROUP_u           attributes_groups;        // $p
    union  CURVE_OWNER_u            owner;                    // $p
    union  CURVE_u                  next;                     // $p
    union  CURVE_u                  previous;                 // $p
    struct GEOMETRIC_OWNER_s       *geometric_owner;          // $p
    vector                          centre;                   // $v
    char                            sense;                    // $c
    vector                          normal;                   // $v
    vector                          x_axis;                   // $v
    double                          major_radius;             // $f
    double                          minor_radius;             // $f
    };
typedef struct ELLIPSE_s   *ELLIPSE;
```

E.5.2.1.4 B_CURVE (B-spline curve)

XT supports B spline curves in full NURBS format. The mathematical description of these curves is:

—Non Uniform Rational B-splines as (NURBS), and

$$P(t) = \frac{\sum_{i=0}^{n-1} b_i(t) w_i V_i}{\sum_{i=0}^{n-1} b_i(t) w_i}$$

—the more simple Non Uniform B-spline

$$P(t) = \sum_{i=0}^{n-1} b_i(t) V_i$$

— Where:

n = number of vertices (n_vertices in the PK standard form)

$V_0 \ldots V_{n-1}$ are the B-spline vertices

$w_0 \ldots w_{n-1}$ are the weights

$b_i\,(t), I = 0 \ldots n-1$ are the B-spline basis functions

Knot Vectors

The parameter t above is global. The user supplies an ordered set of values of t at specific points. The points are called knots and the set of values of t is called the knot vector. Each successive value in the set shall be greater than or equal to its predecessor. Where two or more such values are the same we say that the knots are coincident, or that the knot has multiplicity greater than 1. In this case it is best to think of the knot set as containing a null or zero length span. The principal use of coincident knots is to allow the curve to have less continuity at that point than is formally required for a spline. A curve with a knot of multiplicity equal to its *degree* can have a discontinuity of first derivative and hence of tangent direction. This is the highest permitted multiplicity except at the first or last knot where it can go as high as *(degree+1)*.

In order to avoid problems associated, for example with rounding errors in the knot set, XT stores an array of distinct values and an array of integer multiplicities. This is reflected in the standard form used by the PK for input and output of B-curve data.

Most algorithms in the literature, and the following discussion refer to the expanded knot set in which a knot of multiplicity n appears explicitly n times.

THE NUMBER OF KNOTS AND VERTICES

The knot set determines a set of basis functions which are bell shaped, and non zero over a span of *(degree+1)* intervals. One basis function starts at each knot, and each one finishes *(degree +1)* knots higher. The control vectors are the coefficients applied to these basis functions in a linear sum to obtain positions on the curve. Thus it can be seen that we require the number of knots n_knots = n_vertices + degree + 1

THE VALID RANGE OF THE B-CURVE

So if the knot set is numbered {t_0 to $t_{n\_knots-1}$ } it can be seen then that it is only after t_{degree} that sufficient (*degree + 1*) basis functions are present for the curve to be fully defined, and that the B-curve ceases to be fully defined after $t_{n\_knots - 1 - degree}$.

The first *degree* knots and the last *degree* knots are known as the imaginary knots because their parameter values are outside the defined range of the B-curve.

PERIODIC B-CURVES

When the end of a B-curve meets its start sufficiently smoothly XT allows it to be defined to have periodic parametrization. That is to say that if the valid range were from t_{degree} to $t_{n\_knots - 1 - degree}$ then the difference between these values is called the period and the curve can continue to be evaluated with the same point reoccurring every period.

The minimal smoothness requirement for periodic curves in XT is tangent continuity, but we strongly recommend C $_{degree-1}$, or continuity in the (*degree-1*)th derivative. This in turn is best achieved by repeating the first *degree* vertices at the end, and by matching knot intervals so that counting from the start of the defined range, t_{degree}, the first *degree* intervals between knots match the last *degree* intervals, and similarly matching the last *degree* knot intervals before the end of the defined range to the first *degree* intervals.

CLOSED B-CURVES

A periodic B-curve shall also be closed, but is permitted to have a closed Bcurve that is not periodic.

In this case the rules for continuity are relaxed so that only C_0 or positional continuity is required between the start and end. Such closed non-periodic curves are not able to be attached to topology.

RATIONAL B-CURVE

In the rational form of the curve, each vertex is associated with a weight, which increases or decreases the effect of the vertex without changing the curve hull. To ensure that the convex hull property is retained, the curve equation is divided by a denominator which makes the coefficients of the vertices sum to one.

$$P(t) = \frac{\sum_{i=0}^{n-1} b_i(t) w_i V_i}{\sum_{i=0}^{n-1} b_i(t) w_i}$$

Where $w_0 \ldots w_{n-1}$ are weights.

Each weight may take any positive value, and the larger the value, the greater the effect of the associated vertex. However, it is the relative sizes of the weights which is important, as may be seen from the fact that in the equation given above, all the weights may be multiplied by a constant without changing the equation.

In XT the weights are stored with the vertices by treating these as having an extra dimension. In the usual case of a curve in 3-d cartesian space this means that vertex_dim is 4, the x, y, z values are multiplied through by the corresponding weight and the 4th value is the weight itself.

```
struct B_CURVE_s == ANY_CURVE_s                     // B curve
    {
    int                            node_id;                  // $d
    union  ATTRIB_GROUP_u          attributes_groups;        // $p
    union  CURVE_OWNER_u           owner;                    // $p
    union  CURVE_u                 next;                     // $p
    union  CURVE_u                 previous;                 // $p
    struct GEOMETRIC_OWNER_s      *geometric_owner;          // $p
    char                           sense;                    // $c
    struct NURBS_CURVE_s          *nurbs;                    // $p
    struct CURVE_DATA_s           *data;                     // $p
    };
typedef struct B_CURVE_s       *B_CURVE;
```

The data stored in the XT data for a NURBS_CURVE is

Table F.8 — NURB curve fields

Field name	Data type	Description
degree	Short	degree of the curve
n_vertices	Int	number of control vertices (‗poles')
vertex_dim	Short	dimension of control vertices
n_knots	Int	number of distinct knots
knot_type	Byte	form of knot vector
periodic	Logical	true if curve is periodic
closed	Logical	true if curve is closed
rational	Logical	true if curve is rational
curve_form	Byte	shape of curve, if special
bspline_vertices	Pointer	control vertices node
knot_mult	Pointer	knot multiplicities node
knots	Pointer	knots node

The knot_type enum is used to describe whether or not the knot vector has a certain regular spacing or other common property:

```
typedef enum

   {

   SCH_unset = 1,                // Unknown

   SCH_non_uniform = 2,          // Known to be not special

   SCH_uniform = 3,              // Uniform knot set

   SCH_quasi_uniform = 4,        // Uniform apart from bezier ends

   SCH_piecewise_bezier = 5,     // Internal multiplicity of order-1

   SCH_bezier_ends = 6           // Bezier ends, no other property

   }

    SCH_knot_type_t;
```

A uniform knot set is one where all the knots are of multiplicity one and are equally spaced. A curve has bezier ends if the first and last knots both have multiplicity ‗order'.

The curve_form enum describes the geometric shape of the curve. The parameterization of the curve is not relevant.

```
typedef enum

   {

   SCH_unset          = 1,     // Form is not known

   SCH_arbitrary      = 2,     // Known to be of no particular shape

   SCH_polyline       = 3,

   SCH_circular_arc   = 4,

   SCH_elliptic_arc   = 5,

   SCH_parabolic_arc  = 6,

   SCH_hyperbolic_arc = 7
```

```
    }

    SCH_curve_form_t;

struct NURBS_CURVE_s                              // NURBS curve
    {
    short                         degree;                 // $n
    int                           n_vertices;             // $d
    short                         vertex_dim;             // $n
    int                           n_knots;                // $d
    SCH_knot_type_t               knot_type;              // $u
    logical                       periodic;               // $l
    logical                       closed;                 // $l
    logical                       rational;               // $l
    SCH_curve_form_t              curve_form;             // $u
    struct BSPLINE_VERTICES_s    *bspline_vertices;       // $p
    struct KNOT_MULT_s           *knot_mult;              // $p
    struct KNOT_SET_s            *knots;                  // $p
    };

typedef struct NURBS_CURVE_s *NURBS_CURVE;
```

The bspline vertices node is simply an array of doubles; _vertex_dim‘ doubles together define one control vertex. Thus the length of the array is n_vertices * vertex_dim.

```
struct BSPLINE_VERTICES_s            // B-spline vertices
    {
    double                      vertices[ 1 ];                // $f[]
    };

typedef struct BSPLINE_VERTICES_s *BSPLINE_VERTICES;
```

The knot vector of the NURBS _CURVE is stored as an array of distinct knots and an array describing the multiplicity of each distinct knot. Hence the two nodes

```
struct KNOT_SET_s                    // Knot set
    {
    double                      knots[ 1 ];                   // $f[]
    };

typedef struct KNOT_SET_s *KNOT_SET;
```

and

```
struct KNOT_MULT_s                   // Knot multiplicities
    {
    short                       mult[ 1 ];                    // $n[]
    };

typedef struct KNOT_MULT_s *KNOT_MULT;
```

The data stored in the XT data for a CURVE_DATA node is:

```
typedef enum

    {

    SCH_unset = 1,                         // check has not been performed

    SCH_no_self_intersections = 2,         // passed checks

    SCH_self_intersects = 3,               // fails checks

    SCH_checked_ok_in_old_version = 4      // see below

    }

     SCH_self_int_t;

struct CURVE_DATA_s                      // curve_data

     {

     SCH_self_int_t             self_int;                        // $u

     Struct HELIX_CU_FORM_s     *analytic_form                   // $p

     };

typedef struct CURVE_DATA_s *CURVE_DATA;
```

The self-intersection enum describes whether or not the geometry has been checked for self-intersections, and whether such self-intersections were found to exist:

If the analytic_form field is not null, it will point to a HELIX_CU_FORM node, which indicates that the curve has a helical shape, as follows:

```
struct HELIX_CU_FORM_s

     {

     vector                     axis_pt                          // $v

     vector                     axis_dir                         // $v

     vector                     point                            // $v

     char                       hand                             // $c

     interval                   turns                            // $i

     double                     pitch                            // $f

     double                     tol                              // $f

     };

typedef struct HELIX_CU_FORM_s *HELIX_CU_FORM;
```

The axis_pt and axis_dir fields define the axis of the helix. The hand field is ‗+‘ for a right-handed and ‗-‘ for a left-handed helix. A representative point on the helix is at turn position zero. The turns field gives the extent of the helix relative to the point. For instance, an interval [0 10] indicates a start position at the point and an end 10 turns along the axis. Pitch is the distance travelled along the axis in one turn. Tol is the accuracy to which the owning bcurve fits this specification.

E.5.2.1.5 INTERSECTION

An intersection curve is one of the branches of a surface / surface intersection. XT represents these curves exactly; the information held in an intersection curve node is sufficient to identify the particular intersection branch involved, to identify the behaviour of the curve at its ends, and to evaluate precisely at any point in the curve. Specifically, the data is:

- The two surfaces involved in the intersection.
- The two ends of the intersection curve. These are referred to as the ‗limits‘ of the curve. They identify the particular branch involved.
- An ordered array of points along the curve. This array is referred to as the ‗chart‘ of the curve. It defines the parameterization of the curve, which increases as the array index increases.

The natural tangent to the curve at any point (i.e. in the increasing parameter direction) is given by the vector cross-product of the surface normals at that point, taking into account the senses of the surfaces.

Singular points where the cross-product of the surface normals is zero, or where one of the surfaces is degenerate, are called terminators. Intersection curves do not contain terminators in their interior. At terminators, the tangent to the curve is defined by the limit of the curve tangent as the curve parameter approaches the terminating value.

Table F.9 — Curve intersection fields

Field name	Data type	Description
Surface	pointer array [2]	Surfaces of intersection curve
chart	Pointer	array of hvecs on the curve – see below
start	Pointer	start limit of the curve
end	Pointer	end limit of the curve

```
struct INTERSECTION_s == ANY_CURVE_s              // Intersection
   {
   int                          node_id;                    // $d
   union  ATTRIB_GROUP_u        attributes_groups;          // $p
   union  CURVE_OWNER_u         owner;                      // $p
   union  CURVE_u               next;                       // $p
   union  CURVE_u               previous;                   // $p
   struct GEOMETRIC_OWNER_s    *geometric_owner;            // $p
   char                         sense;                      // $c
   union  SURFACE_u             surface[ 2 ];               // $p[2]
   struct CHART_s              *chart;                      // $p
   struct LIMIT_s              *start;                      // $p
   struct LIMIT_s              *end;                        // $p
   };
typedef struct INTERSECTION_s  *INTERSECTION;
```

A point on an intersection curve is stored in a data structure called an ‗hvec‘ (hepta-vec, or 7-vector):

```
typedef struct hvec_s                    // hepta_vec
```

```
    {

    vector              Pvec;                   // position

    double              u[2];                   // surface parameters

    double              v[2];

    vector              Tangent;                // curve tangent

    double              t;                      // curve parameter

    } hvec;
```

where

- pvec is a point common to both surfaces;
- u[] and v[] are the u and v parameters of the pvec on each of the surfaces;
- tangent is the tangent to the curve at pvec. This will be equal to the (normalized) vector cross product of the surface normals at pvec, when this cross product is non-zero. These surface normals take account of the surface sense fields;
- t is the parameter of the pvec on the curve.

Note that only the pvec part of an hvec is actually transmitted.

The chart data structure essentially describes a piecewise-linear (chordal) approximation to the true curve. As well as containing the ordered array of hvecs defining this approximation, it contains extra information pertaining to the accuracy of the approximation:

```
struct CHART_s                                  // Chart
    {

    double              Base_parameter;          // $f

    double              Base_scale;              // $f

    int                 Chart_count;             // $d

    double              Chordal_error;           // $f

    double              Angular_error;           // $f

    double              Parameter_error[2];      // $f[2]

    hvec                Hvec[ 1 ];               // $h[]

    };
```

where

- base_parameter is the parameter of the first hvec in the chart;
- base_scale determines the scale of the parameterization (see below);
- chart_count is the length of the hvec array;
- chordal_error is an estimate of the maximum deviation of the curve from the piecewise-linear approximation given by the hvec array. It may be null;
- angular_error is the maximum angle between the tangents of two sequential hvecs. It may be null.

— parameter_error[] is always [null, null];

— hvec[] is the ordered array of hvecs.

The limits of the intersection curve are stored in the following data structure:

```
struct LIMIT_s                             // Limit
     {

     char                     type;                     // $c

     hvec                     hvec[ 1 ];                // $h[]

     };
```

The ‗type' field may take one of the following values

```
     const char SCH_help               = 'H';              // help hvec

     const char SCH_terminator         = 'T';              // terminator

     const char SCH_limit              = 'L';              // arbitrary limit

     const char SCH_boundary           = 'B';              // spine boundary
```

The length of the hvec array depends on the type of the limit:

— a SCH_help limit is an arbitrary point on a closed intersection curve. There will be one hvec in the hvec array, locating the curve.

— a SCH_terminator limit is a point where one of the surface normals is degenerate, or where their cross-product is zero. Typically, there will be more than one branch of intersection between the two surfaces at these singularities. There will be two values in the hvec array. The first will be the exact position of the singularity, and the second will be a point on the curve a small distance away from the terminator. This ‗branch point' identifies which branch relates to the curve in question. The branch point is the one which appears in the chart, at the corresponding end – so the singularity lies just outside the parameter range of the chart.

— a SCH_limit limit is an artificial boundary of an intersection curve on an otherwise potentially infinite branch. The single hvec describes the end of the curve.

— a SCH_boundary limit is used to describe the end of a degenerate rolling-ball blend. It is not relevant to intersection curves.

The parameterization of the curve is given as follows. If the chart points are Pi, i = 0 to n, with parameters ti, and natural tangent vectors Ti, then define

$C_i = | P_{i+1} – P_i |$

$\cos(a_i) = T_i . (P_{i+1} – P_i) / C_i$

$\cos(b_i) = T_i . (P_i – P_{i-1}) / C_{i-1}$

Then at any chart point P_i the angles a_i and b_i are the deviations between the tangent at the chart point and the next and previous chords respectively.

Let f_0 = base_scale

$f_i = (\cos(b_i) / \cos(a_i)) f_{i-1}$

Then t_0 = base_parameter

$t_i = t_{i-1} + C_{i-1} f_{i-1}$

The factors f_i are chosen so that the parameterization is C1. The parameter of a point between two chart points is given by projecting the point onto the chord between the previous and next chart points.

The point on the intersection curve corresponding to a given parameter is defined as follows:

- For a parameter equal to that of a chart point, it is the position of the chart point.
- For a parameter interior to the chart, it is the local point of intersection of three surfaces: the two surfaces of the intersection, and a plane defined by the chart. If the parameter t lies between chart parameters t_i, t_{i+1}, then the chord point corresponding to t lies at

$$(t_{i+1} - t) / (t_{i+1} - t_i) P_i + (t - t_i) / (t_{i+1} - t_i) P_{i+1}$$

  The plane lies through this point and is orthogonal to the chord (P_{i+1}, P_i).

- For a parameter between a branch chart point and a terminator, it is the local point of intersection of three surfaces: one of the intersection surfaces and two planes. Surface[0] is used unless it is singular at the terminator and surface[1] is not singular at the terminator. The first plane contains the chord between the branch and the terminator, and the normal of the chosen intersection surface at the terminator or the curve tangent at the branch chart point if the surface normal cannot be defined. The second plane is the plane orthogonal to the chord between the branch and terminator points through the chord point as calculated above.

E.5.2.1.6 TRIMMED_CURVE

A trimmed curve is a bounded region of another curve, referred to as its basis curve. It is defined by the basis curve and two points and their corresponding parameters. Trimmed curves are most commonly attached to fins (fins) of tolerant edges in order to specify which portion of the underlying basis curve corresponds to the tolerant edge. They are necessary since the tolerant vertices of the edge do not necessarily lie exactly on the basis curve; the ‗point' fields of the trimmed curve lie exactly on the basis curve, and within tolerance of the relevant vertex.

The rules governing the parameter fields and points are:

- point_1 and point_2 correspond to parm_1 and parm_2 respectively.
- If the basis curve has positive sense, parm_2 > parm_1.
- If the basis curve has negative sense, parm_2 < parm_1.

In addition,

For open basis curves.

- Both parm_1 and parm_2 shall be in the parameter range of the basis curve.
- point_1 and point_2 shall not be equal.

For periodic basis curves.

- parm_1 shall lie in the base range of the basis curve.
- If the whole basis curve is required then parm_1 and parm_2 should be a period apart and point_1 = point_2. Equality of parm_1 and parm_2 is not permitted.
- parm_1 and parm_2 shall not be more than a period apart.

For closed but non-periodic basis curves.

—Both parm_1 and parm_2 shall be in the parameter range of the basis curve.

—If the whole of the basis curve is required, parm_1 and parm_2 shall lie close enough to each end of the valid parameter range in order that point_1 and point_2 are coincident to XT tolerance (1.0e-8 by default).

The sense of a trimmed curve is positive.

Table F.10 — Trimmed curve fields

Field name	Data type	Description
basis_curve	pointer	Basis curve
point_1	vector	start of trimmed portion
point_2	vector	end of trimmed portion
parm_1	double	parameter on basis curve corresponding to point_1
parm_2	double	parameter on basis curve corresponding to point_2

```
struct TRIMMED_CURVE_s == ANY_CURVE_s             // Trimmed Curve
    {
    int                            node_id;                     // $d
    union  ATTRIB_GROUP_u          attributes_groups;           // $p
    union  CURVE_OWNER_u           owner;                       // $p
    union  CURVE_u                 next;                        // $p
    union  CURVE_u                 previous;                    // $p
    struct GEOMETRIC_OWNER_s      *geometric_owner;             // $p
    char                           sense;                       // $c
    union  CURVE_u                 basis_curve;                 // $p
    vector                         point_1;                     // $v
    vector                         point_2;                     // $v
    double                         parm_1;                      // $f
    double                         parm_2;                      // $f
    };
typedef struct TRIMMED_CURVE_s      *TRIMMED_CURVE;
```

E.5.2.1.7 PE_CURVE (Foreign Geometry curve)

Foreign geometry curve is a type used for representing customers‘ in-house proprietary data. The definition of the data structure is included here for completeness.

Foreign geometry in XT is a type used for representing customers‘ in-house proprietary data. It is also known as PE (parametrically evaluated) geometry. It can also be used for representing geometry connected with this data (for example, offsets of foreign surfaces). These two types of foreign geometry usage are referred to as ‗external‘ and ‗internal‘ PE data respectively. Internal PE curves are not used at present.

Table F.11 — Foreign Geometry curve fields

Field name	Data type	Description
type	char	whether internal or external

data	pointer	internal or external data
tf	pointer0	transform applied to geometry
internal geom	pointer array	reference to other related geometry

```
union PE_DATA_u                                  // PE_data_u
    {
    struct EXT_PE_DATA_s        *external;                      // $p
    struct INT_PE_DATA_s        *internal;                      // $p
    };
typedef union PE_DATA_u PE_DATA;
```

The PE internal geometry union defined below is used by internal foreign geometry only.

```
union PE_INT_GEOM_u
    {
    union SURFACE_u              surface;                       // $p
    union CURVE_u                curve;                         // $p
    };
typedef union PE_INT_GEOM_u PE_INT_GEOM;
struct PE_CURVE_s == ANY_CURVE_s                 // PE_curve
    {
    int                           node_id;                      // $d
    union  ATTRIB_GROUP_u         attributes_groups;            // $p
    union  CURVE_OWNER_u          owner;                        // $p
    union  CURVE_u                next;                         // $p
    union  CURVE_u                previous;                     // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;              // $p
    char                          sense;                        // $c
    char                          type;                         // $c
    union  PE_DATA_u              data;                         // $p
    struct TRANSFORM_s           *tf;                           // $p
    union  PE_INT_GEOM_u          internal_geom[ 1 ];           // $p[]
    };
typedef struct PE_CURVE_s       *PE_CURVE;
```

The type of the foreign geometry (whether internal or external) is identified in the PE curve node by means of the char ‗type‘ field, taking one of the values

```
    const char SCH_external  = 'E';      // external PE geometry
    const char SCH_interna   = 'I';      // internal PE geometry
```

The PE_data union is used in a PE curve or surface node to identify the internal or external evaluator corresponding to the geometry, and also holds an array of real and/or integer parameters to be passed to the evaluator. The data stored corresponds exactly to that passed to the PK routine PK_FSURF_create when the geometry is created.

```
struct EXT_PE_DATA_s                         // ext_PE_data
    {
    struct KEY_s              *key;                          // $p
    struct REAL_VALUES_s      *real_array;                   // $p
    struct INT_VALUES_s       *int_array;                    // $p
    };
typedef struct EXT_PE_DATA_s *EXT_PE_DATA;
struct INT_PE_DATA_s                         // int_PE_data
    {
    int                        geom_type;                   // $d
    struct REAL_VALUES_s      *real_array;                  // $p
    struct INT_VALUES_s       *int_array;                   // $p
    };
typedef struct INT_PE_DATA_s *INT_PE_DATA;
```

The only internal pe type in use at the moment is the offset PE surface, for which the geom_type is 2.

E.5.2.1.8 SP_CURVE

An SP curve is the 3D curve resulting from embedding a 2D curve in the parameter space of a surface.

The 2D curve shall be a 2D BCURVE; that is it shall either be a rational B curve with a vertex dimensionality of 3, or a non-rational B curve with a vertex dimensionality of 2.

Table F.12 — SP curve fields

Field name	Data type	Description
surface	pointer	surface
b_curve	pointer	2D Bcurve
original	pointer0	not used
tolerance_to_original	double	not used

```
struct   SP_CURVE_s == ANY_CURVE_s                  // SP curve
    {
    int                         node_id;                    // $d
    union  ATTRIB_GROUP_u       attributes_groups;          // $p
    union  CURVE_OWNER_u        owner;                      // $p
    union  CURVE_u              next;                       // $p
    union  CURVE_u              previous;                   // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;              // $p

    char                          sense;                        // $c

    union  SURFACE_u              surface;                      // $p

    struct B_CURVE_s             *b_curve;                      // $p

    union  CURVE_u                original;                     // $p

    double                        tolerance_to_original;        // $f

    };

typedef struct SP_CURVE_s   *SP_CURVE;
```

E.5.2.2 Surfaces

All surface nodes share the following common fields:

Table F.13 — Surface node fields

Field name	Data type	Description
node_id	int	Integer value unique to surface in part
attributes_groups	pointer0	Attributes and groups associated with surface
owner	pointer	topological owner
next	pointer0	next surface in geometry chain
previous	pointer0	previous surface in geometry chain
geometric_owner	pointer0	geometric owner node
sense	char	sense of surface: ‗+‘ or ‗-‘(see end of Geometry section)

```
struct ANY_SURF_s                                     // Any Surface
    {

    int                           node_id;                      // $d

    union  ATTRIB_GROUP_u         attributes_groups;            // $p

    union  SURFACE_OWNER_u        owner;                        // $p

    union  SURFACE_u              next;                         // $p

    union  SURFACE_u              previous;                     // $p

    struct GEOMETRIC_OWNER_s     *geometric_owner;              // $p

    char                          sense;                        // $c

    };

typedef struct ANY_SURF_s  *ANY_SURF;
```

E.5.2.2.1 PLANE

A plane has a parametric representation of the form

R(u, v) = P + uX + vY

where

— P is a point on the plan.

—X and Y are axes in the plane.

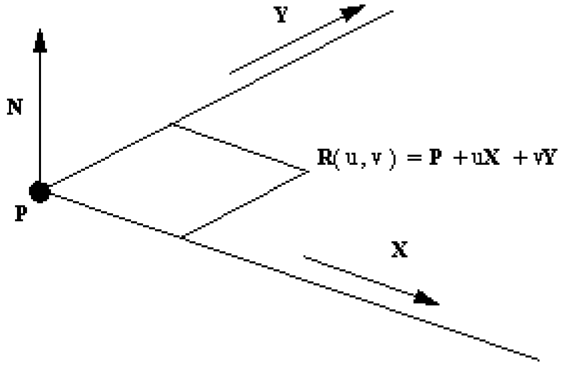


Table F.14 — Plane fields

Field name	Data type	Description
pvec	vector	point on the plane
normal	vector	normal to the plane (a unit vector)
x_axis	vector	X axis of the plane (a unit vector)

The Y axis in the definition above is the vector cross product of the normal and x_axis.

```
struct PLANE_s == ANY_SURF_s                       // Plane
    {
    int                          node_id;                  // $d
    union  ATTRIB_GROUP_u        attributes_groups;        // $p
    union  SURFACE_OWNER_u       owner;                    // $p
    union  SURFACE_u             next;                     // $p
    union  SURFACE_u             previous;                 // $p
    struct GEOMETRIC_OWNER_s    *geometric_owner;          // $p
    char                         sense;                    // $c
    vector                       pvec;                     // $v
    vector                       normal;                   // $v
    vector                       x_axis;                   // $v
    };
typedef struct PLANE_s     *PLANE;
```

E.5.2.2.2 CYLINDER

A cylinder has a parametric representation of the form:

R(u,v) = P + rXcos(u) + rYsin(u) + vA

where

—P is a point on the cylinder axis.

—r is the cylinder radius.

—A is the cylinder axis.

—X and Y are unit vectors such that A, X and Y form an orthonormal set.

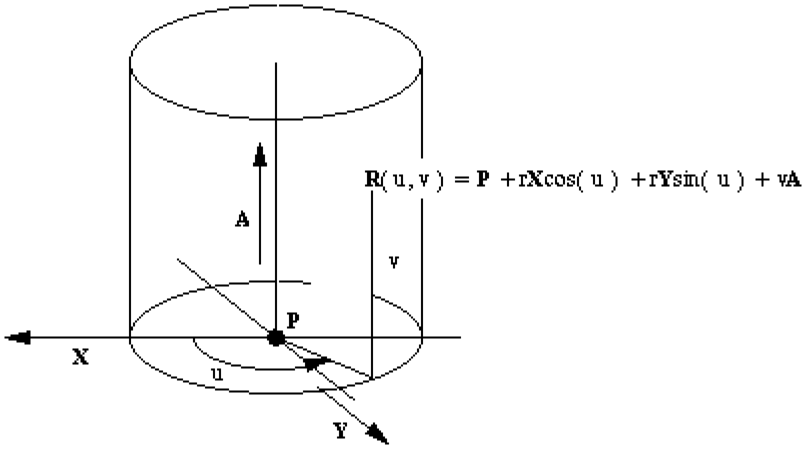


Table F.15 — Cylinder fields

Field name	Data type	Description
pvec	vector	point on the cylinder axis
axis	vector	direction of the cylinder axis (a unit vector)
radius	double	radius of cylinder
x_axis	vector	X axis of the cylinder (a unit vector)

The Y axis in the definition above is the vector cross product of the axis and x_axis.

```
struct CYLINDER_s == ANY_SURF_s                    // Cylinder
    {
    int                           node_id;                     // $d
    union  ATTRIB_GROUP_u         attributes_groups;           // $p
    union  SURFACE_OWNER_u        owner;                       // $p
    union  SURFACE_u              next;                        // $p
    union  SURFACE_u              previous;                    // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;             // $p
    char                          sense;                       // $c
    vector                        pvec;                        // $v
    vector                        axis;                        // $v
    double                        radius;                      // $f
    vector                        x_axis;                      // $v
    };
typedef struct CYLINDER_s  *CYLINDER;
```

E.5.2.2.3 CONE

A cone in XT is only half of a mathematical cone. By convention, the cone axis points away from the half of the cone in use. A cone has a parametric representation of the form:

$$R(u, v) = P - vA + (X\cos(u) + Y\sin(u))(r + v\tan(a))$$

where

—P is a point on the cone axis.

—r is the cone radius at the point P.

—A is the cone axis.

—X and Y are unit vectors such that A, X and Y form an orthonormal set, i.e. Y = A x X.

—a is the cone half angle.

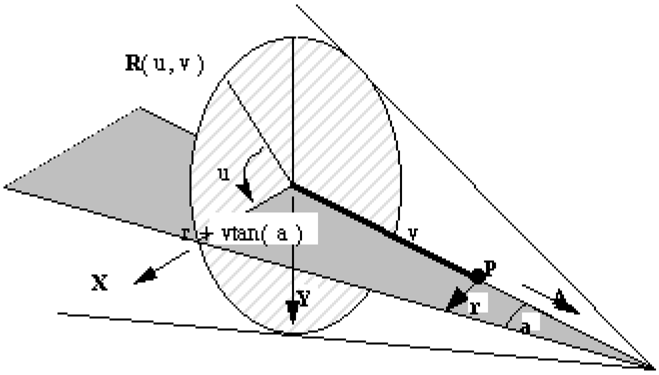


Table F.16 — Cone fields

Field name	Data type	Description
pvec	vector	point on the cone axis
axis	vector	direction of the cone axis (a unit vector)
radius	double	radius of the cone at its pvec
sin_half_angle	double	sine of the cone‘s half angle
cos_half_angle	double	cosine of the cone‘s half angle
x_axis	vector	X axis of the cone (a unit vector)

The Y axis in the definition above is the vector cross product of the axis and x_axis.

```
struct CONE_s == ANY_SURF_s                        // Cone
    {
    int                          node_id;                     // $d
    union  ATTRIB_GROUP_u        attributes_groups;           // $p
    union  SURFACE_OWNER_u       owner;                       // $p
    union  SURFACE_u             next;                        // $p
    union  SURFACE_u             previous;                    // $p
    struct GEOMETRIC_OWNER_s    *geometric_owner;             // $p
    char                         sense;                       // $c
    vector                       pvec;                        // $v
    vector                       axis;                        // $v
    double                       radius;                      // $f
    double                        sin_half_angle;              // $f

    double                        cos_half_angle;              // $f

    vector                        x_axis;                      // $v

    };

typedef struct CONE_s      *CONE;
```

E.5.2.2.4 SPHERE

A sphere has a parametric representation of the form:

R(u, v) = C + (Xcos(u) + Ysin(u)) rcos(v) + rAsin(v)

where

— C is centre of the sphere.

— r is the sphere radius.

— A, X and Y form an orthonormal axis set.

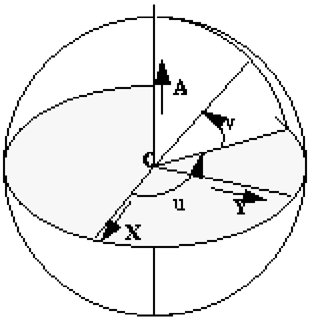


Table F.17 — Sphere fields

Field name	Data type	Description
Centre	vector	centre of the sphere
Radius	double	radius of the sphere
Axis	vector	A axis of the sphere (a unit vector)
x_axis	vector	X axis of the sphere (a unit vector)

The Y axis of the sphere is the vector cross product of its A and X axes.

```
struct SPHERE_s == ANY_SURF_s                      // Sphere
    {

    int                           node_id;                     // $d

    union  ATTRIB_GROUP_u         attributes_groups;           // $p

    union  SURFACE_OWNER_u        owner;                       // $p

    union  SURFACE_u              next;                        // $p

    union  SURFACE_u              previous;                    // $p

    struct GEOMETRIC_OWNER_s     *geometric_owner;             // $p

    char                          sense;                       // $c
    vector                      centre;                     // $v

    double                      radius;                     // $f

    vector                      axis;                       // $v

    vector                      x_axis;                     // $v

    };

typedef struct SPHERE_s    *SPHERE;
```

E.5.2.2.5 TORUS

A torus has a parametric representation of the form

$$R(u, v) = C + (X \cos(u) + Y \sin(u))(a + b \cos(v)) + b A \sin(v)$$

where.

- C is centre of the torus.
- A is the torus axis.
- a is the major radius.
- b is the minor radius.
- X and Y are unit vectors such that A, X and Y form an orthonormal set.

In XT, there are three types of torus:

Doughnut - the torus is not self-intersecting (a > b)

Apple - the outer part of a self-intersecting torus (a <= b, a > 0)

Lemon - the inner part of a self-intersecting torus (a < 0, |a| < b)

The limiting case a = b is allowed; it is called an ‗osculating apple‘, but there is no ‗lemon‘ surface corresponding to this case.

The limiting case a = 0 cannot be represented as a torus; this is a sphere.

Table F.18 — Torus fields

Field name	Data type	Description
centre	vector	centre of the torus
axis	vector	axis of the torus (a unit vector)
major_radius	double	major radius
minor_radius	double	minor radius
x_axis	vector	X axis of the torus (a unit vector)

The Y axis in the definition above is the vector cross product of the axis of the torus and the x_axis.

```
struct TORUS_s == ANY_SURF_s                       // Torus
    {
        int                        node_id;                      // $d
    union  ATTRIB_GROUP_u          attributes_groups;            // $p
    union  SURFACE_OWNER_u         owner;                        // $p
    union  SURFACE_u               next;                         // $p
    union  SURFACE_u               previous;                     // $p
    struct GEOMETRIC_OWNER_s      *geometric_owner;              // $p
    char                           sense;                        // $c
    vector                         centre;                       // $v
    vector                         axis;                         // $v
    double                         major_radius;                 // $f
    double                         minor_radius;                 // $f
    vector                         x_axis;                       // $v
    };

typedef struct TORUS_s     *TORUS;
```

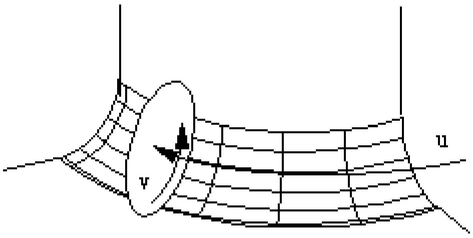
E.5.2.2.6 BLENDED_EDGE (Rolling Ball Blend)

XT supports exact rolling ball blends. They have a parametric representation of the form

$R(u, v) = C(u) + rX(u)\cos(v\, a(u)) + rY(u)\sin(va(u))$

where

- C(u) is the spine curve.
- r is the blend radius.
- X(u) and Y(u) are unit vectors such that $C'(u) . X(u) = C'(u) . Y(u) = 0$.
- a(u) is the angle subtended by points on the boundary curves at the spine.



X, Y and a are expressed as functions of u, as their values change with u.

The spine of the rolling ball blend is the centre line of the blend; i.e. the path along which the centre of the ball moves.

Table F.19 — Blended edge fields

Field name	Data type	Description
type	char	type of blend: _R‘ or _E‘
surface	pointer[2]	supporting surfaces (adjacent to original edge)
spine	pointer	spine of blend
range	double[2]	offsets to be applied to surfaces
thumb_weight	double[2]	always [1,1]
boundary	pointer0[2]	always [0, 0]
start	pointer0	Start LIMIT in certain degenerate cases
end	pointer0	End LIMIT in certain degenerate cases

```
struct BLENDED_EDGE_s == ANY_SURF_s             // Blended edge
    {
    int                         node_id;                    // $d
    union  ATTRIB_GROUP_u       attributes_groups;          // $p
    union  SURFACE_OWNER_u      owner;                      // $p
    union  SURFACE_u            next;                       // $p
    union  SURFACE_u            previous;                   // $p
    struct GEOMETRIC_OWNER_s   *geometric_owner;            // $p
    char                        sense;                      // $c
    char                        blend_type;                 // $c
    union  SURFACE_u            surface[2];                 // $p[2]
    union  CURVE_u              spine;                      // $p
    double                      range[2];                   // $f[2]
    double                      thumb_weight[2];            // $f[2]
    union  SURFACE_u            boundary[2];                // $p[2]
    struct LIMIT_s             *start;                      // $p
    struct LIMIT_s             *end;                        // $p
    };
typedef struct BLENDED_EDGE_s *BLENDED_EDGE;
```

The parameterization of the blend is as follows. The u parameter is inherited from the spine, the constant u lines being circles perpendicular to the spine curve. The v parameter is zero at the blend boundary on the first surface, and one on the blend boundary on the second surface; unless the sense of the spine curve is negative, in which case it is the other way round. The v parameter is proportional to the angle around the circle.

XT data can contain blends of the following types:

```
     const char SCH_rolling_ball = 'R';       //  rolling ball blend

     const char SCH_cliff_edge   = 'E';       //  cliff edge blend
```

For rolling ball blends, the spine curve will be the intersection of the two surfaces obtained by offsetting the supporting surfaces by an amount given by the respective entry in range[]. Note that the offsets to be applied may be positive or negative, and that the sense of the surface is significant; i.e. the offset vector is the natural unit surface normal, times the range, times –1 if the sense is negative.

For cliff edge blends, one of the surfaces will be a blended_edge with a range of [0,0]; its spine will be the cliff edge curve, and its supporting surfaces will be the surfaces of the faces adjacent to the cliff edge. Its type will be R.

The limit fields will only be non-null if the spine curve is periodic but the edge curve being blended has terminators – for example if the spine is elliptical but the blend degenerates. In this case the two LIMIT nodes, of type ‗L', determine the extent of the spine.

E.5.2.2.7 BLEND_BOUND (Blend boundary surface)

A blend_bound surface is a construction surface, used to define the boundary curve where a blend becomes tangential to its supporting surface. It is an implicit surface defined internally so that it intersects one of the supporting surfaces along the boundary curve. It is orthogonal to the blend and the supporting surface along this boundary curve. The supporting surface corresponding to the blend_bound is

Blend_bound -> blend.blended_edge -> surface[1-blend_bound->boundary]

Blend boundary surfaces have no parameterization, but are defined by the distance function

f(X) = f0 (X + r1 * grad_f1 (X)) – r0

Where

- f0 is the surface distance function of the supporting surface corresponding to the blend_bound.
- r0 is the blend radius corresponding to that supporting surface.
- f1 is the surface distance function of the other supporting surface of the blend.
- r1 is the blend radius corresponding to the other supporting surface.

Blend boundary surfaces are most commonly referenced by the intersection curve representing the boundary curve of the blend.

The data stored in the XT data for a blend_bound is only that necessary to identify the relevant blend and supporting surface:

Table F.20 — Blend boundary surface fields

Field name	Data type	Description
boundary	short	index into supporting surface array
blend	pointer	corresponding blend surface

```
struct BLEND_BOUND_s == ANY_SURF_s          // Blend boundary
```

```
    {

    int                           node_id;                     // $d

    union  ATTRIB_GROUP_u         attributes_groups;           // $p

    union  SURFACE_OWNER_u        owner;                       // $p

    union  SURFACE_u              next;                        // $p

    union  SURFACE_u              previous;                    // $p

    struct GEOMETRIC_OWNER_s     *geometric_owner;             // $p

    char                          sense;                       // $c

    short                         boundary;                    // $n

    union  SURFACE_u              blend;                       // $p

    };

typedef struct BLEND_BOUND_s  *BLEND_BOUND;
```

The supporting surface corresponding to the blend_bound is

blend_bound->blend.blended_edge->surface[1 - blend_bound->boundary].

E.5.2.2.8 OFFSET_SURF

An offset surface is the result of offsetting a surface a certain distance along its normal, taking into account the surface sense. It inherits the parameterization of this underlying surface.

Table F.21 — Offset surface fields

Field name	Data type	Description
Check	char	check status
true_offset	logical	not used
surface	pointer	underlying surface
offset	double	signed offset distance
scale	double	for internal use only – may be set to null

```
struct OFFSET_SURF_s == ANY_SURF_s       // Offset surface
    {

    int                            node_id;                    // $d

    union  ATTRIB_GROUP_u          attributes_groups;          // $p

    union  SURFACE_OWNER_u         owner;                      // $p

    union  SURFACE_u               next;                       // $p

    union  SURFACE_u               previous;                   // $p

    struct GEOMETRIC_OWNER_s      *geometric_owner;            // $p

    char                           sense;                      // $c

    char                           check;                      // $c

    logical                        true_offset;                // $l
    union  SURFACE_u               surface;                  // $p
    double                         offset;                   // $f
    double                         scale;                    // $f
    };
typedef struct OFFSET_SURF_s    *OFFSET_SURF;
```

The offset surface is subject to the following restrictions:

- The offset distance shall not be within modeller linear resolution of zero.
- The sense of the offset surface shall be the same as that of the underlying surface.
- Offset surfaces may not share a common underlying surface.

The ‗check‘ field may take one of the following values:

```
    const char SCH_valid       = 'V';       // valid
    const char SCH_invalid     = 'I';       // invalid
    const char SCH_unchecked   = 'U';       // has not been checked
```

E.5.2.2.9 B_SURFACE

XT supports B spline surfaces in full NURBS format.

B-SURFACE DEFINITION

$$P(u,v) = \frac{\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} b_i(u) \quad b_j(v) \quad w_{ij}V_{ij}}{\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} b_i(u) \quad b_j(v) \quad w_{ij}}$$

The B-surface definition is best thought of as an extension of the B-curve definition into two parameters, usually called u and v. Two knot sets are required and the number of control vertices is the product of the number that would be required for a curve using each knot vector. The rules for periodicity and closure given in the B-curve documentation are extended to surfaces in an obvious way.

For attachment to topology a B-surface is required to have G_1 continuity. That is to say that the surface normal direction shall be continuous.

Surfaces that are self-intersecting or contain cusps are not permitted to be attached to topology.

Table F.22 — B-Surface fields

Field name	Data type	Description
nurbs	pointer	Geometric definition
data	pointer0	Auxiliary information

```
struct B_SURFACE_s == ANY_SURF_s                    // B surface
    {
    int                            node_id;                  // $d
    union ATTRIB_GROUP_u          attributes_groups;          // $p
    union  SURFACE_OWNER_u        owner;                      // $p
    union  SURFACE_u              next;                       // $p
    union  SURFACE_u              previous;                   // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;            // $p
    char                          sense;                      // $c
    struct NURBS_SURF_s          *nurbs;                      // $p
    struct SURFACE_DATA_s        *data;                       // $p
    };
typedef struct B_SURFACE_s       *B_SURFACE;
```

The data stored in the XT data for a NURBS surface is

Table F.23 — NURB Surface fields

Field name	Data type	Description
u_periodic	logical	true if surface is periodic in u parameter
v_periodic	logical	true if surface is periodic in v parameter
u_degree	short	u degree of the surface
v_degree	short	v degree of the surface
n_u_vertices	int	number of control vertices (‗poles‘) in u direction
n_v_vertices	int	number of control vertices (‗poles‘) in v direction
u_knot_type	byte	form of u knot vector – see —Bcurve”
v_knot_type	byte	form of v knot vector
n_u_knots	int	number of distinct u knots
n_v_knots	int	number of distinct v knots
Rational	logical	true if surface is rational
u_closed	logical	true if surface is closed in u
v_closed	logical	true if surface is closed in v
surface_form	byte	shape of surface, if special
vertex_dim	short	dimension of control vertices
bspline_vertices	pointer	control vertices (poles) node
u_knot_mult	pointer	multiplicities of u knot vector
v_knot_mult	pointer	multiplicities of v knot vector
u_knots	pointer	u knot vector
v_knots	pointer	v knot vector

The surface form enum is defined below.

```
typedef enum
   {
   SCH_unset = 1,                        // Unknown
   SCH_arbitrary = 2,                    // No particular shape
   SCH_planar = 3,
   SCH_cylindrical = 4,
   SCH_conical = 5,
   SCH_spherical = 6,
   SCH_toroidal = 7,
   SCH_surf_of_revolution = 8,
   SCH_ruled = 9,
   SCH_quadric = 10,
   SCH_swept = 11
   }
   SCH_surface_form_t;
struct NURBS_SURF_s                              // NURBS surface
   {
   logical                         u_periodic;                   // $l
   logical                         v_periodic;                   // $l
   short                           u_degree;                     // $n
   short                           v_degree;                     // $n
   int                             n_u_vertices;                 // $d
   int                             n_v_vertices;                 // $d
   SCH_knot_type_t                 u_knot_type;                  // $u
   SCH_knot_type_t                 v_knot_type;                  // $u
   int                             n_u_knots;                    // $d
   int                             n_v_knots;                    // $d
   logical                         rational;                     // $l
   logical                         u_closed;                     // $l
   logical                         v_closed;                     // $l
   SCH_surface_form_t              surface_form;                 // $u
   short                           vertex_dim;                   // $n
   struct BSPLINE_VERTICES_s      *bspline_vertices;             // $p
   struct KNOT_MULT_s             *u_knot_mult;                  // $p
   struct KNOT_MULT_s             *v_knot_mult;                  // $p
```

```
   struct KNOT_SET_s             *u_knots;                     // $p

   struct KNOT_SET_s             *v_knots;                     // $p

   };

typedef struct NURBS_SURF_s *NURBS_SURF;
```

The ‗bspline_vertices‘, ‗knot_set‘ and ‗knot_mult‘ nodes and the ‗knot_type‘ enum are described in the documentation for B_CURVE.

The ‗surface data‘ field in a B surface node is a structure designed to hold auxiliary or ‗derived‘ data about the surface: it is not a necessary part of the definition of the B surface. It may be null, or the majority of its individual fields may be null.

```
struct  SURFACE_DATA_s           // auxiliary surface data
    {

    interval                    original_uint;                  // $i

    interval                    original_vint;                  // $i

    interval                    extended_uint;                  // $i

    interval                    extended_vint;                  // $i

    SCH_self_int_t              self_int;                       // $u

    char                        original_u_start;               // $c

    char                        original_u_end;                 // $c

    char                        original_v_start;               // $c

    char                        original_v_end;                 // $c

    char                        extended_u_start;               // $c

    char                        extended_u_end;                 // $c

    char                        extended_v_start;               // $c

    char                        extended_v_end;                 // $c

    char                        analytic_form_type;             // $c

    char                        swept_form_type;                // $c

    char                        spun_form_type;                 // $c

    char                        blend_form_type;                // $c

    void                       *analytic_form;                  // $p

    void                       *swept_form;                     // $p

    void                       *spun_form;                      // $p

    void                       *blend_form;                     // $p

    };

typedef struct SURFACE_DATA_s *SURFACE_DATA;
```

The ‗original_‘ and ‗extended_‘ parameter intervals and corresponding character fields original_u_start etc. are all connected with the ability to extend B surfaces when necessary – functionality which is commonly exploited in ―local operation” algorithms for example. This is done automatically without the need for user intervention.

In cases where the required extension can be performed by adding rows or columns of control points, then the nurbs data will be modified accordingly – this is referred to as an ‗explicit‘ extension. In some rational B surface cases, explicit extension is not possible - in these cases, the surface will be ‗implicitly‘ extended. When a B surface is implicitly extended, the nurbs data is not changed, but it will be treated as being larger by allowing out-of-range evaluations on the surface. Whenever an explicit or implicit extension takes place, it is reflected in the following fields:

- ―original_u_int” and ―original_v_int” are the original valid parameter ranges for a B surface before it was extended.

- ―extended_u_int” and ―extended_v_int” are the valid parameter ranges for a B surface once it has been extended.

The character fields ‗original_u_start‘ etc. all refer to the status of the corresponding parameter boundary of the surface before or after an extension has taken place. For B surfaces, the character can have one of the following values:

```
const char SCH_degenerate = 'D';      // Degenerate edge

const char SCH_periodic   = 'P';      // Periodic parameterization

const char SCH_bounded    = 'B';      // Parameterization bounded

const char SCH_closed     = 'C';      // Closed, but not periodic
```

The separate fields original_u_start and extended_u_start etc. are necessary because an extension may cause the corresponding parameter boundary to become degenerate.

If the surface_data node is present, then the original_u_int, original_v_int, original_u_start, original_u_end, original_v_start and original_v_end fields should be set to their appropriate values. If the surface has not been extended, the extended_u_int and extended_v_int fields should contain null, and the extended_u_start etc. fields should contain

```
const char SCH_unset_char = '?'; // generic uninvestigated value
```

As soon as any parameter boundary of the surface is extended, all the fields should be set, regardless of whether the corresponding boundary has been affected by the extension.

The SCH_self_int_t enum is documented in the corresponding curve_data structure under B curve.

The ‗swept_form_type‘, ‗spun_form_type‘ and ‗blend_form_type‘ characters and the corresponding pointers swept_form, spun_form and blend_form, are not implemented in XT. The character fields should be set to SCH_unset_char (‗?‘) and the pointers should be set to null pointer.

If the analytic_form field is not null, it will point to a HELIX_SU_FORM node, which indicates that the surface has a helical shape. In this case the analytic_form_type field will be set to ‗H‘.

```
struct HELIX_SU_FORM_s
    {

    vector                      axis_pt                          // $v

    vector                      axis_dir                         // $v

    char                        hand                             // $c

    interval                    turns                            // $i

    double                      pitch                            // $f
    double                    gap                           // $f

    double                    tol                           // $f

    };

typedef struct HELIX_SU_FORM_s *HELIX_SU_FORM;
```

The axis_pt and axis_dir fields define the axis of the helix. The hand field is ‗+‘ for a right-handed and ‗-‘ for a left-handed helix. The turns field gives the extent of the helix relative to the profile curve which was used to generate the surface. For instance, an interval [0 10] indicates a start position at the profile curve and an end 10 turns along the axis. Pitch is the distance travelled along the axis in one turn. Tol is the accuracy to which the owning bsurface fits this specification. Gap is for future expansion and will currently be zero. The v parameter increases in the direction of the axis.

E.5.2.2.10 SWEPT_SURF

A swept surface has a parametric representation of the form:

R(u, v) = C(u) + vD

where

- C(u) is the section curve.
- D is the sweep direction (unit vector).
- C shall not be an intersection curve or a trimmed curve.

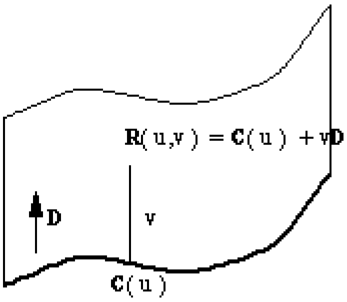


Table F.24 — Swept surface fields

Field name	Data type	Description
section	pointer	section curve
sweep	vector	sweep direction (a unit vector)
scale	double	for internal use only – may be set to null

```
struct SWEPT_SURF_s == ANY_SURF_s            // Swept surface
    {

    int                         node_id;                    // $d

    union  ATTRIB_GROUP_u       attributes_groups;          // $p

    union  SURFACE_OWNER_u      owner;                      // $p

    union  SURFACE_u            next;                       // $p

    union  SURFACE_u            previous;                   // $p

    struct GEOMETRIC_OWNER_s   *geometric_owner;            // $p

    char                        sense;                      // $c
    union   CURVE_u                 section;                        // $p

    vector                          sweep;                          // $v

    double                          scale;                          // $f

    };

typedef struct SWEPT_SURF_s *SWEPT_SURF;
```

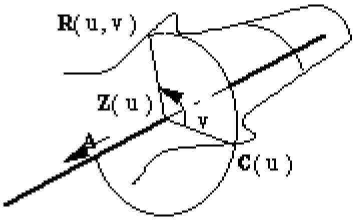
E.5.2.2.11 SPUN_SURF

A spun surface has a parametric representation of the form:

R(u, v) = Z(u) + (C(u) - Z(u))cos(v) + A X (C(u) - Z(u)) sin(v)

where

- C(u) is the profile curve
- Z(u) is the projection of C(u) onto the spin axis.
- A is the spin axis direction (unit vector).
- C shall not be an intersection curve or a trimmed curve.



NOTE: Z(u) = P + ((C(u) - P) . A)A where P is a reference point on the axis.

Table F.25 — Spun surface fields

Field name	Data type	Description
profile	pointer	profile curve
base	vector	point on spin axis
axis	vector	spin axis direction (a unit vector)
start	vector	position of degeneracy at low u (may be null)
end	vector	position of degeneracy at low v (may be null)
start_param	double	curve parameter at low u degeneracy (may be null)
end_param	double	curve parameter at high u degeneracy (may be null)
x_axis	vector	unit vector in profile plane if common with spin axis
scale	double	for internal use only – may be set to null

```
struct SPUN_SURF_s == ANY_SURF_s                     // Spun surface
    {

    int                             node_id;                        // $d

    union   ATTRIB_GROUP_u          attributes_groups;              // $p

    union   SURFACE_OWNER_u         owner;                          // $p

    union   SURFACE_u               next;                           // $p
    union  SURFACE_u             previous;                    // $p
    struct GEOMETRIC_OWNER_s    *geometric_owner;             // $p
    char                         sense;                       // $c
    union  CURVE_u               profile;                     // $p
    vector                       base;                        // $v
    vector                       axis;                        // $v
    vector                       start;                       // $v
    vector                       end;                         // $v
    double                       start_param;                 // $f
    double                       end_param;                   // $f
    vector                       x_axis;                      // $v
    double                       scale;                       // $f
    };
typedef struct SPUN_SURF_s *SPUN_SURF;
```

The ‗start' and ‗end' vectors correspond to physical degeneracies on the spun surface caused by the profile curve crossing the spin axis at that point. The values start_param and end_param are the corresponding parameters on the curve. These parameter values define the valid range for the u parameter of the surface. If either value is null, then the valid range for u is infinite in that direction. For example, for a straight line profile curve intersecting the spin axis at the parameter t=1, values of null for start_param and 1 for end_param would define a cone with u parameterization (-infinity, 1].

If the profile curve lies in a plane containing the spin axis, then x_axis shall be set to a vector perpendicular to the spin axis and in the plane of the profile, pointing from the spin axis to a point on the profile curve in the valid range. If the profile curve is not planar, or its plane does not contain the spin axis, then x_axis should be set to null.

E.5.2.2.12 PE_SURF (Foreign Geometry surface)

Foreign geometry surface is a type used for representing customers' in-house proprietary data. The definition of the data structure is included here for completeness.

Foreign (or ‗PE') geometry in XT data is a type used for representing customers' in-house proprietary data. It can also be used for representing geometry connected with this data (for example, offset foreign surfaces). These two types of foreign geometry usage are referred to as ‗external' and ‗internal' respectively. The only internal PE surface is the offset PE surface.

Table F.26 — Foreign Geometry suface fields

Field name	Data type	Description
type	char	whether internal or external
data	pointer	internal or external data
tf	pointer0	transform applied to geometry
internal geom	pointer array	reference to other related geometry

```
struct PE_SURF_s == ANY_SURF_s                   // PE_surface
    {
    int                            node_id;                   // $d
    union  ATTRIB_GROUP_u           attributes_groups;           // $p
    union  SURFACE_OWNER_u          owner;                       // $p
    union  SURFACE_u                next;                        // $p
    union  SURFACE_u                previous;                    // $p
    struct GEOMETRIC_OWNER_s       *geometric_owner;             // $p
    char                            sense;                       // $c
    char                            type;                        // $c
    union  PE_DATA_u                data;                        // $p
    struct TRANSFORM_s             *tf;                          // $p
    union  PE_INT_GEOM_u            internal_geom[ 1 ];          // $p[]
    };
typedef struct PE_SURF_s  *PE_SURF;
```

The PE_DATA and PE_INT_GEOM unions are defined under ‗PE curve‘.

E.5.2.3 Point

Table F.27 — Point fields

Field name	Data type	Description
node_id	int	integer unique within part
attributes_groups	pointer0	attributes and groups associated with point
owner	pointer	Owner
next	pointer0	next point in chain
previous	pointer0	previous point in chain
pvec	vector	position of point

```
union POINT_OWNER_u
    {
    struct VERTEX_s               *vertex;
    struct BODY_s                 *body;
    struct ASSEMBLY_s             *assembly;
    struct WORLD_s                *world;
    };

struct POINT_s                                     // Point
    {
    int                          node_id;                         // $d
    union  ATTRIB_GROUP_u        attributes_groups;               // $p
    union  POINT_OWNER_u         owner;                          // $p

    struct POINT_s              *next;                           // $p

    struct POINT_s              *previous;                       // $p

    vector                       pvec;                           // $v

    };

typedef struct POINT_s    *POINT;
```

E.5.2.4 Transform

Table F.28 — Transform fields

Field name	Data type	Description
node_id	int	integer unique within part
owner	pointer	owning instance or world
next	pointer0	next transform in chain
previous	pointer0	previous pointer in chain
rotation_matrix	double[3][3]	rotation component
translation_vector	vector	translation component
scale	double	scaling factor
flag	byte	binary flags indicating non-trivial components
perspective_vector	vector	perspective vector (always null vector)

The transform acts as

x‘ = (rotation_matrix . x + translation_vector) * scale

The ‗flag‘ field contains various bit flags which identify the components of the transformation:

Table F.29 — Transform action fields

Flag Name	Binary Value	Description
translation	00001	set if translation vector non-zero
rotation	00010	set if rotation matrix is not the identity
scaling	00100	set if scaling component is not 1.0
reflection	01000	set if determinant of rotation matrix is negative
general affine	10000	set if the rotation_matrix is not a rigid rotation

```
union TRANSFORM_OWNER_u
    {

    struct INSTANCE_s          *instance;

    struct WORLD_s             *world;

    };


struct TRANSFORM_s                       // Transformation
    {

    int                         node_id;                      // $d

    union  TRANSFORM_OWNER_u    owner;                        // $p

    struct TRANSFORM_s         *next;                         // $p
    struct TRANSFORM_s           *previous;                    // $p
    double                        rotation_matrix[3][3];       // $f[9]
    vector                        translation_vector;          // $v
    double                        scale;                       // $f
    unsigned                      flag;                        // $d
    vector                        perspective_vector;          // $v
    };
typedef struct TRANSFORM_s *TRANSFORM;
```

E.5.2.5 Curve and Surface Senses

The ‗natural‘ tangent to a curve is that in the increasing parameter direction, and the ‗natural‘ normal to a surface is in the direction of the cross-product of dP/du and dP/dv. For some purposes these are modified by the curve and surfaces senses, respectively – for example in the definition of blend surfaces, offset surfaces and intersection curves.

At the PK interface, the edge/curve and face/surface sense orientations are regarded as properties of the topology/geometry combination. In the XT format, this orientation information resides in the curves, surfaces and faces as follows:

The edge/curve orientation is stored in the curve->sense field. The face/surface orientation is a combination of sense flags stored in the face->sense and surface->sense fields, so the face/surface orientation is true (i.e. the face normal is parallel to the natural surface normal) if neither, or both, of the face and surface senses are positive.

E.5.2.6 Geometric_owner

Where geometry has dependants, the dependants point back to the referencing geometry by means of Geometric Owner nodes. Each geometric node points to a doubly-linked ring of Geometric Owner nodes which identify its referencing geometry. Referenced geometry is as follows:

Intersection:	2 surfaces
SP-curve:	Surface
Trimmed curve:	basis curve
Blended edge:	2 supporting surfaces, 2 blend_bound surfaces, 1 spine curve
Blend bound:	blend surface
Offset surface:	underlying surface
Swept surface:	section curve
Spun surface:	profile curve

Note that the 2D B-curve referenced by an SP-curve is not a dependent in this sense, and does not need a geometric owner node.

Table F.30 — Geometry owner fields

Field name	Data type	Description
owner	pointer	referencing geometry
next	pointer	next in ring of geometric owners referring to the same geometry

previous	pointer	previous in above ring
shared_geometry	pointer	referenced (dependent) geometry

```
struct GEOMETRIC_OWNER_s          //  geometric owner of geometry
    {

    union  GEOMETRY_u              owner;                      //  $p

    struct GEOMETRIC_OWNER_s      *next;                       //  $p

    struct GEOMETRIC_OWNER_s      *previous;                   // $p

    union  GEOMETRY_u              shared_geometry;            //  $p

    };

typedef struct GEOMETRIC_OWNER_s *GEOMETRIC_OWNER;
```

E.5.3 Topology

In the following tables, ‗ignore‘ means this may be set to null (zero) and should be ignored.

Unless otherwise stated, all chains of nodes are doubly-linked and null-terminated.

E.5.3.1 WORLD

Table F.31 — World topology fields

Field name	Type	Description
assembly	pointer0	Head of chain of assemblies
attribute	pointer0	Ignore
body	pointer0	Head of chain of bodies
transform	pointer0	Head of chain of transforms
surface	pointer0	Head of chain of surfaces
curve	pointer0	Head of chain of curves
point	pointer0	Head of chain of points
alive	logical	True unless partition is at initial pmark
attrib_def	pointer0	Head of chain of attribute definitions
highest_id	int	Highest pmark id in partition
current_id	int	Id of current pmark
index_map_offset	int	Shall be set to 0
index_map	pointer0	Shall be set to null
schema_embedding_map	pointer0	Shall be set to null

The World node is only used when a partition is transmitted. Because some of the attribute definitions may be referenced by nodes which have been deleted, but which may reappear on rollback, the attribute definitions are chained off the World node rather than simply being referenced by attributes.

The fields index_map_offset, index_map, and schema_embedding_map are used for Indexed Transmit; applications writing XT data shall set them to 0 and null.

```
struct WORLD_s                                 // World
    {

    struct ASSEMBLY_s          *assembly;                      // $p

    struct ATTRIBUTE_s         *attribute;                     // $p

    struct BODY_s              *body;                          // $p
    struct TRANSFORM_s          *transform;                      // $p
    union  SURFACE_u             surface;                        // $p
    union  CURVE_u               curve;                          // $p
    struct POINT_s              *point;                          // $p
    logical                      alive;                          // $l
    struct ATTRIB_DEF_s         *attrib_def;                     // $p
    int                          highest_id;                     // $d
    int                          current_id;                     // $d
    };
typedef struct WORLD_s  *WORLD;
```

E.5.3.2 ASSEMBLY

Table F.32 — Assembly fields

highest_node_id	int	Highest node-id in assembly
attributes_groups	pointer0	Head of chain of attributes of, and groups in, assembly
attribute_chains	pointer0	List of attributes, one for each attribute definition used in the assembly
list	pointer0	Null
surface	pointer0	Head of construction surface chain
curve	pointer0	Head of construction curve chain
point	pointer0	Head of construction point chain
key	pointer0	Ignore
res_size	double	Value of ‗size box‘ when transmitted (normally 1000)
res_linear	double	Value of modeller linear precision when transmitted (normally 1.0e-8).
ref_instance	pointer0	Head of chain of instances referencing this assembly
next	pointer0	Ignore
previous	pointer0	Ignore
state	byte	Set to 1.
owner	pointer0	Ignore
type	byte	Always 1.
sub_instance	pointer0	Head of chain of instances in assembly

The value of the ‗state‘ field should be ignored, as should any nodes of type ‗KEY‘ referenced by the assembly. If XT data is constructed without use of the Parasolid Kernel, the state field should be set to 1, and the key to null.

The highest_node_id gives the highest node-id of any node in the assembly. Certain nodes within the assembly (namely instances, transforms, geometry, attributes and groups) have unique node-ids which are non-zero integers.

```
typedef enum
    {
    SCH_collective_assembly  = 1,
    SCH_conjunctive_assembly = 2,
    SCH_disjunctive_assembly = 3
    }
    SCH_assembly_type;

typedef enum
    {
    SCH_new_part       = 1,
    SCH_stored_part    = 2,
    SCH_modified_part  = 3,
    SCH_anonymous_part = 4,
    SCH_unloaded_part  = 5
    }
    SCH_part_state;

struct ASSEMBLY_s                               //  Assembly
    {
    int                         highest_node_id;                //  $d
    union ATTRIB_GROUP_u        attributes_groups;              //  $p
    struct LIST_s              *attribute_chains;               //  $p
    struct LIST_s              *list;                           //  $p
    union SURFACE_u             surface;                        //  $p
    union CURVE_u               curve;                          //  $p
    struct POINT_s             *point;                          //  $p
    struct KEY_s               *key;                            //  $p
    double                      res_size;                       //  $f
    double                      res_linear;                     //  $f
    struct INSTANCE_s          *ref_instance;                   //  $p
    struct ASSEMBLY_s          *next;                           //  $p
    struct ASSEMBLY_s          *previous;                       //  $p
    SCH_part_state              state;                          //  $u
    struct WORLD_s             *owner;                          //  $p
    SCH_assembly_type           type;                           //  $u
    struct INSTANCE_s          *sub_instance;                   //  $p
    };
typedef struct ASSEMBLY_s *ASSEMBLY;
```

```
struct KEY_s                                  // Key
     {
     string[1];                  char                             // $c[]
     };
typedef struct KEY_s *KEY;
```

E.5.3.3 INSTANCE

Table F.33 — Instance fields

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of instance and member_of_groups of instance
type	byte	Always 1
part	pointer	Part referenced by instance
transform	pointer0	Transform of instance
assembly	pointer	Assembly in which instance lies
next_in_part	pointer0	Next instance in assembly
prev_in_part	pointer0	Previous instance in assembly
next_of_part	pointer0	Next instance of instance->part
prev_of_part	pointer0	Previous instance of instance->part

```
typedef enum
     {
     SCH_positive_instance = 1,
     SCH_negative_instance = 2
     }
     SCH_instance_type;

union  PART_u
     {
     struct BODY_s             *body;
     struct ASSEMBLY_s         *assembly;
     };
typedef union PART_u       PART;
struct INSTANCE_s                                       //  Instance
     {
     int                          node_id;                        //  $d
     union  ATTRIB_GROUP_u        attributes_groups;              //  $p
     SCH_instance_type            type;                           //  $u
     union  PART_u                part;                           //  $p
     struct TRANSFORM_s          *transform;                      //  $p
```

```
    struct ASSEMBLY_s          *assembly;                    //  $p
    struct INSTANCE_s          *next_in_part;                //  $p
    struct INSTANCE_s          *prev_in_part;                //  $p
    struct INSTANCE_s          *next_of_part;                //  $p
    struct INSTANCE_s          *prev_of_part;                //  $p
    };
typedef struct INSTANCE_s *INSTANCE;
```

E.5.3.4 BODY

Table F.34 — Body fields

Field name	Type	Description
highest_node_id	int	Highest node-id in body
attributes_groups	pointer0	Head of chain of attributes of, and groups in, body
attribute_chains	pointer0	List of attributes, one for each attribute definition used in the body
surface	pointer0	Head of construction surface chain
curve	pointer0	Head of construction curve chain
point	pointer0	Head of construction point chain
key	pointer0	Ignore
res_size	double	Value of ‗size box‘ when transmitted (normally 1000)
res_linear	double	Value of modeller linear precision when transmitted (normally 1.0e-8)
ref_instance	pointer0	Head of chain of instances referencing this part
next	pointer0	Ignore
previous	pointer0	Ignore
state	byte	Set to 1 (see below)
owner	pointer0	Ignore
body_type	byte	Body type
nom_geom_state	byte	Set to 1 (not documented)
shell	pointer0	For general bodies: null For solid bodies: the first shell in one of the solid regions For other bodies: the first shell in one of the regions This field is **obsolete**, and should be ignored by applications reading XT data. When writing XT data, it shall be set as above.
boundary_surface	pointer0	Head of chain of surfaces attached directly or indirectly to faces or edges or fins
boundary_curve	pointer0	Head of chain of curves attached directly or indirectly to edges or faces or fins
boundary_point	pointer0	Head of chain of points attached to vertices
region	pointer	Head of chain of regions in body; this is the infinite region
edge	pointer0	Head of chain of all non-wireframe edges in body
vertex	pointer0	Head of chain of all vertices in body
index_map_offset	int	Shall be set to 0
index_map	pointer0	Shall be set to null

node_id_index_map	pointer0	Shall be set to null
schema_embedding_ma p	pointer0	Shall be set to null

The value of the ‗state‘ field should be ignored, as should any nodes of type ‗KEY‘ referenced by the body. If the XT data is constructed without using the Parasolid Kernel, the state field should be set to 1, and the key to null.

The highest_node_id gives the highest node of any node in this body. Most nodes in a body have node-ids, which are non-zero integers unique to that node within the body. Applications writing XT data shall ensure that node-ids are present and distinct. The details of which nodes have node ids are given in an appendix.

The fields index_map_offset, index_map, node_id_index_map, and schema_embedding_map are used for Indexed Transmit; applications writing XT data shall ensure that these fields are set to 0 and null.

```
typedef enum
     {
     SCH_solid_body      = 1,
     SCH_wire_body       = 2,
     SCH_sheet_body      = 3,
     SCH_general_body    = 6
     }
     SCH_body_type;

typedef short short enum
        {
        SCH_nom_geom_off = 1,              --- Entirely off
        SCH_nom_geom_on  = 2               --- Entirely on
        }
        SCH_nom_geom_state_t;

struct BODY_s                                      //  Body
     {
     int                          highest_node_id;              //  $d
     union ATTRIB_GROUP_u         attributes_groups;            //  $p
     struct LIST_s               *attribute_chains;             //  $p
     union SURFACE_u              surface;                      //  $p
     union CURVE_u                curve;                        //  $p
     struct POINT_s              *point;                        //  $p
     struct KEY_s                *key;                          //  $p
     double                       res_size;                     //  $f
```

```
    double                     res_linear;                 //  $f
    struct INSTANCE_s         *ref_instance;               //  $p
    struct BODY_s             *next;                       //  $p
    struct BODY_s             *previous;                   //  $p
    SCH_part_state             state;                      //  $u
    struct WORLD_s            *owner;                      //  $p
    SCH_body_type              body_type;                  //  $u
    SCH_nom_geom_state_t      nom_geom_state;              //  $u
    struct SHELL_s            *shell;                      //  $p
    union SURFACE_u            boundary_surface;           //  $p
    union CURVE_u              boundary_curve;             //  $p
    struct POINT_s            *boundary_point;             //  $p
    struct REGION_s           *region;                     //  $p
    struct EDGE_s             *edge;                       //  $p
    struct VERTEX_s           *vertex;                     //  $p
    int                       index_map_offset;            //  $d
    struct INT_VALUES_s       *index_map;                  //  $p
    struct INT_VALUES_s       *node_id_index_map;          //  $p
    struct INT_VALUES_s       *schema_embedding_map;       //  $p
    };
typedef struct BODY_s     *BODY;
```

Attaching Geometry to Topology

The faces which reference a surface are chained together, surface->owner is the head of this chain. Similarly the edges which reference the same curve are chained together. Fins do not share curves.

Geometry in parts may be chained into one of the three boundary geometry chains, or one of the three construction geometry chains. A geometric node will fall into one of the following cases:

Table F.35 — Geometry to Topology attachment

Geometry	Owner	Whether chained
Attached to face	face	In boundary_surface chain
Attached to edge or fin	edge or fin	In boundary_curve chain
Attached to vertex	vertex	In boundary_point chain
Indirectly attached to face or edge or fin	body	In boundary_surface chain or boundary_curve chain
Construction geometry	body or assembly	In surface, curve or point chain
2D B-curve in SP-curve	null	Not chained

Here ‗indirectly attached' means geometry which is a dependent of a dependent of (... etc) of geometry attached to an edge, face or fin.

Geometry in a construction chain may reference geometry in a boundary chain, but not vice-versa.

E.5.3.5 REGION

Table F.36 — Region fields

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of region and member_of_groups of region
body	pointer	Body of region
next	pointer0	Next region in body
prev	pointer0	Previous region in body
shell	pointer0	Head of singly-linked chain of shells in region
type	char	Region type – solid (‗S') or void (‗V')

```
struct REGION_s                                   //  Region
    {

    int                           node_id;                      //  $d

    union  ATTRIB_GROUP_u         attributes_groups;            //  $p

    struct BODY_s                *body;                         //  $p

    struct REGION_s              *next;                         //  $p

    struct REGION_s              *previous;                     //  $p

    struct SHELL_s               *shell;                        //  $p

    char                          type;                         //  $c

    };

typedef struct REGION_s   *REGION;
```

E.5.3.6 SHELL

Table F.37 — Shell fields

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of shell
body	pointer0	For shells in wire and sheet bodies, and for shells bounding a solid region of a solid body, this is set to the body of the shell. For shells in general bodies, or void shells in solid bodies, it is null. This field is **obsolete**, and should be ignored by applications reading XT data. When writing XT data, it shall be set as above.
next	pointer0	Next shell in region
face	pointer0	Head of chain of back-faces of shell (i.e. faces with face normal pointing out of region of shell).
edge	pointer0	Head of chain of wire-frame edges of shell
vertex	pointer0	If shell consists of a single vertex, this is it; else null
region	pointer	Region of shell
front_face	pointer0	Head of chain of front-faces of shell (i.e. faces with face normal pointing into region of shell)

```
struct SHELL_s                                     //  Shell
     {
     int                           node_id;                      //  $d
     union  ATTRIB_GROUP_u         attributes_groups;            //  $p
     struct BODY_s                *body;                         //  $p
     struct SHELL_s               *next;                         //  $p
     struct FACE_s                *face;                         //  $p
     struct EDGE_s                *edge;                         //  $p
     struct VERTEX_s              *vertex;                       //  $p
     struct REGION_s              *region;                       //  $p
     struct FACE_s                *front_face;                   //  $p
     };
typedef struct SHELL_s    *SHELL;
```

E.5.3.7 FACE

Table F.38 — Face fields

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of face and member_of_groups of face
tolerance	double	Not used (null double)
next	pointer0	Next back-face in shell
previous	pointer0	Previous back-face in shell
loop	pointer0	Head of singly-linked chain of loops

shell	pointer	Shell of which this is a back-face
surface	pointer0	Surface of face
sense	char	Face sense – positive (_+‘) or negative (_-‘)
next_on_surface	pointer0	Next in chain of faces sharing the surface of this face
previous_on_surface	pointer0	Previous in chain of faces sharing the surface of this face
next_front	pointer0	Next front-face in shell
previous_front	pointer0	Previous front-face in shell
front_shell	pointer	Shell of which this is a front-face

```
struct FACE_s                                       //  Face
    {
    int                            node_id;                        //  $d
    union  ATTRIB_GROUP_u          attributes_groups;              //  $p
    double                         tolerance;                      //  $f
    struct FACE_s                 *next;                           //  $p
    struct FACE_s                 *previous;                       //  $p
    struct LOOP_s                 *loop;                           //  $p
    struct SHELL_s                *shell;                          //  $p
    union  SURFACE_u               surface;                        //  $p
    char                           sense;                          //  $c
    struct FACE_s                 *next_on_surface;                //  $p
    struct FACE_s                 *previous_on_surface;            //  $p
    struct FACE_s                 *next_front;                     //  $p
    struct FACE_s                 *previous_front;                 //  $p
    struct SHELL_s                *front_shell;                    //  $p
    };
typedef struct FACE_s     *FACE;
```

E.5.3.8 LOOP

Table F.39 — Loop fields

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of loop
fin	pointer	One of ring of fins of loop
face	pointer	Face of loop
next	pointer0	Next loop in face

Isolated loops

An isolated loop (one consisting of a single vertex) does not refer directly to a vertex, but points to a fin which refers to that vertex. This isolated fin has fin->forward = fin->backward = fin, and fin->other = fin->curve = fin->edge = null. Its sense is not significant. The fin is chained into the chain of fins referencing the isolated vertex.

```
struct LOOP_s                                      //  Loop
    {
    int                          node_id;                     //  $d
    union  ATTRIB_GROUP_u        attributes_groups;           //  $p
    struct FIN_s                *fin;                         //  $p
    struct FACE_s               *face;                        //  $p
    struct LOOP_s               *next;                        //  $p
    };
typedef struct LOOP_s     *LOOP;
```

E.5.3.9 FIN

Table F.40 — Fin fields

Field name	Type	Description
attributes_groups	pointer0	Head of chain of attributes of fin
loop	pointer0	Loop of fin
forward	pointer0	Next fin around loop
backward	pointer0	Previous fin around loop
vertex	pointer0	Forward vertex of fin
other	pointer0	Next fin around edge, clockwise looking along edge
edge	pointer0	Edge of fin
curve	pointer0	For a non-dummy fin of a tolerant edge, this will be a trimmed SP-curve, otherwise null.
next_at_vx	pointer0	Next fin referencing the vertex of this fin
sense	char	Positive (‗+‘) if the fin direction is parallel to that of its edge, else negative (_-‘)

Dummy fins

An application will see edges as having any number of fins, including zero. However internally, they have at least two. This is so that the forward and backward vertices of an edge can always be found as edge->fin->vertex and edge->fin->other->vertex respectively - the first one being a positive fin, the second a negative fin. If an edge does not have both a positive and a negative externally-visible fin, **dummy** fins will exist for this purpose. Dummy fins have fin->loop = fin->forward = fin->backward = fin->curve = fin->next_at_vx = null. For example the boundaries of a sheet always have one dummy fin.

```
struct FIN_s                                        //  Fin
    {
    union  ATTRIB_GROUP_u        attributes_groups;            //  $p
    struct LOOP_s               *loop;                         //  $p
    struct FIN_s                *forward;                      //  $p
    struct FIN_s                *backward;                     //  $p
    struct VERTEX_s             *vertex;                       //  $p
    struct FIN_s                *other;                        //  $p
    struct EDGE_s               *edge;                         //  $p
    union  CURVE_u              curve;                          //  $p

    struct FIN_s               *next_at_vx;                     //  $p

    char                        sense;                          //  $c

    };

typedef struct FIN_s *FIN;
```

E.5.3.10 VERTEX

Table F.41 — Vertex fields

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of vertex and member_of_groups of vertex
fin	pointer0	Head of singly-linked chain of fins referencing this vertex
previous	pointer0	Previous vertex in body
next	pointer0	Next vertex in body
point	pointer	Point of vertex
tolerance	double	Tolerance of vertex (null-double for accurate vertex)
owner	pointer	Owning body (for non-acorn vertices) or shell (for acorn vertices)

```
union SHELL_OR_BODY_u
    (

    struct BODY_s              *body;

    struct SHELL_s             *shell;

    };

typedef union SHELL_OR_BODY_u SHELL_OR_BODY;

struct VERTEX_s                                    //  Vertex
    {

    int                         node_id;                        //  $d

    union  ATTRIB_GROUP_u       attributes_groups;              //  $p

    struct FIN_s               *fin;                            //  $p

    struct VERTEX_s            *previous;                       //  $p

    struct VERTEX_s            *next;                           //  $p

    struct POINT_s             *point;                          //  $p

    double                      tolerance;                      //  $f

    union  SHELL_OR_BODY_u      owner;                          //  $p

    };

typedef struct VERTEX_s   *VERTEX;
```

E.5.3.11 EDGE

Table F.42 — Edge fields

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of edge and member_of_groups of edge
tolerance	double	Tolerance of edge (null-double for accurate edges)
fin	pointer	One of singly-linked ring of fins around edge
previous	pointer0	Previous edge in body or shell
next	pointer0	Next edge in body or shell
curve	pointer0	Curve of edge, zero for tolerant edge. If edge is accurate, but any of its vertices are tolerant, this will be a trimmed curve
next_on_curve	pointer0	Next in chain of edges sharing the curve of this edge
previous_on_curve	pointer0	Previous in chain of edges sharing the curve of this edge
owner	pointer	Owning body (for non-wireframe edges) or shell (for wireframe edges)

```
struct EDGE_s                                       //  Edge
    {
    int                         node_id;                        //  $d
    union  ATTRIB_GROUP_u       attributes_groups;              //  $p
    double                      tolerance;                      //  $f
    struct FIN_s               *fin;                            //  $p
    struct EDGE_s              *previous;                       //  $p
    struct EDGE_s              *next;                           //  $p
    union  CURVE_u              curve;                          //  $p
    struct EDGE_s;             *next_on_curve                   //  $p
    struct EDGE_s              *previous_on_curve;              //  $p
    union  SHELL_OR_BODY_u      owner;                          //  $p
    };
typedef struct EDGE_s     *EDGE;
```

E.5.4 Associated Data

E.5.4.1 LIST

Table F.43 — Associated List

Field name	Type	Description
node_id	int	Zero
list_type	byte	Always 4
notransmit	logical	Ignore
owner	pointer	Owning part
next	pointer0	Ignore
previous	pointer0	Ignore
list_length	int	Length of list (>= 0)
block_length	int	Length of each block of list. Always 20
size_of_entry	int	Ignore
finger_index	int	Any integer between 1 and list->list_length (set to 1 if length is zero). Ignore
finger_block	pointer	Any block e.g. the first one. Ignore
list_block	pointer	Head of singly-linked chain of pointer list blocks

Lists only occur in part data as the list of attributes referenced by a part.

```
typedef enum
    {
    LIS_pointer    = 4
    }
LIS_type_t;
union LIS_BLOCK_u
    {
    struct POINTER_LIS_BLOCK_s      *pointer_block;
    };
typedef union LIS_BLOCK_u       LIS_BLOCK;
union LIST_OWNER_u
    {
    struct BODY_s                   *body;
    struct ASSEMBLY_s               *assembly;
    struct WORLD_s                  *world;
    };
typedef union LIST_OWNER_u LIST_OWNER;
struct LIST_s                                // List Header
    {
    int                              node_id;                // $d
    LIS_type_t                       list_type;              // $u
    logical                          notransmit;             //  $l
    union LIST_OWNER_u               owner;                  //  $p
    struct LIST_s                   *next;                   //  $p
    struct LIST_s                   *previous;               //  $p
    int                              list_length;            //  $d
    int                              block_length;           //  $d
    int                              size_of_entry;          //  $d
    int                              finger_index;           //  $d
    union LIS_BLOCK_u                finger_block;           //  $p
    union LIS_BLOCK_u                list_block;             //  $p
    };
typedef struct LIST_s *LIST;
```

E.5.4.2 POINTER_LIS_BLOCK:

Table F.44 — Pointer List Block

Field name	Type	Description
n_entries	int	Number of entries in this block (0 <= n_entries <= 20). Only the first block may have n_entries = 0.
index_map_offset	int	Shall be set to 0
next_block	pointer0	Next pointer list block in chain
Entries[20]	pointer0	Pointers in block, those beyond n_entries shall be zero

When the pointer_lis_block is used as the root node in XT data containing more than one part, the restriction n_entries <= 20 does not apply.

The index_map_offset field is used for Indexed Transmit; applications writing XT data shall ensure this field is set to 0.

```
struct POINTER_LIS_BLOCK_s                      //  Pointer List
    {
    int                              n_entries;            //  $d
    int                             index_map_offset       // $d
    struct POINTER_LIS_BLOCK_s      *next_block;           //  $p
    void                            *entries[ 1 ];         //  $p[]
    };
typedef struct POINTER_LIS_BLOCK_s *POINTER_LIS_BLOCK;
```

E.5.4.3 ATT_DEF_ID

Table F.45 — Attribute Definition ID

Field name	Type	Description
string[]	char	String name e.g. "SDL/TYSA_COLOUR"

```
struct ATT_DEF_ID_s            //  name field type for attrib def.
    {

    char                             String[1];              //  $c[]

    };

typedef struct ATT_DEF_ID_s *ATT_DEF_ID;
```

E.5.4.4 FIELD_NAMES

Table F.46 — Field Names

Field name	Type	Description
names[]	pointer	Array of field names – unicode or char

```
typedef union FIELD_NAME_u
    {

    struct CHAR_VALUES_s        *name

    struct UNICODE_VALUES_s     *uname

    };

      FIELD_NAME_t;

struct FIELD_NAME_s            //  attribute field name
    {

    union FIELD_NAME_u               names[1];               //  $p[]

    };

typedef struct FIELD_NAME_s *FIELD_NAME;
```

E.5.4.5 ATTRIB_DEF

Table F.47 — Attribut definition

Field name	Type	Description
next	pointer0	Next attribute definition. This can be ignored, except in partition data.
identifier	pointer	Pointer to string name
type_id	int	Numeric id, e.g. 8001 for colour. 9000 for user-defined attribute definitions
actions[8]	byte	Required actions on various events
field_names	pointer0	Names of fields (unicode or char)
legal_owners[14]	logical	Allowed owner types
fields[]	byte	Array of field types. Note that the number of fields is given by the length of the variable length part of this node, i.e. the integer following the node type in the XT data.

The legal_owners array is an array of logicals determining which node types may own this type of attribute.

e.g. if faces are allowed attrib_def -> legal_owners [SCH_fa_owner] = true.

Note that if the XT data contains user fields, the ‗fields‘ field of an attribute definition may contain extra values, set to zero. These are to be ignored.

The ‗actions‘ field in an attribute definition defines the behaviour of the attribute when an event (rotate, scale, translate, reflect, split, merge, transfer, change) occurs. The actions are in table F.49:

Table F.48 — Attribute definition action fields

do_nothing	Leave attribute as it is
delete	Delete the attribute
transform	Transform the transformable fields (point, vector, direction, axis) by appropriate part of transformation
propagate	Copy attribute onto split-off node
keep_sub_dominant	Move attribute(s) from deleted node onto surviving node in a merge, but any such attributes already on the surviving node are deleted.
keep_if_equal	Keep attribute if present on both nodes being merged, with the same field values.
combine	Move attribute(s) from deleted node onto surviving node, in a merge

The XT attribute classes 1-7 correspond as follows:

Table F.49 — Corresponding attribute classes

	split	**merge**	**transfer**	**change**	**Rotate**	**scale**	**translate**	**reflect**
class 1	propagate	keep_equal	do_nothing	do_nothing	do_nothing	do_nothing	do_nothing	do_nothing
class 2	delete	delete	delete	delete	do_nothing	delete	do_nothing	do_nothing
class 3	delete	delete	delete	delete	Delete	delete	delete	delete
class 4	propagate	keep_equal	do_nothing	do_nothing	Transform	transform	transform	transform
class 5	delete	delete	delete	delete	Transform	transform	transform	transform
class 6	propagate	combine	do_nothing	do_nothing	do_nothing	do_nothing	do_nothing	do_nothing
class 7	propagate	combine	do_nothing	do_nothing	Transform	transform	transform	transform

```
typedef enum
    {

    SCH_rotate      = 0,

    SCH_scale       = 1,

    SCH_translate   = 2,

    SCH_reflect     = 3,

    SCH_split       = 4,

    SCH_merge       = 5,

    SCH_transfer    = 6,

    SCH_change      = 7,
    SCH_max_logged_event     //  last entry; value in $d[] code for
                             actions

    }

    SCH_logged_event_t;


typedef enum
    {

    SCH_do_nothing   = 0,

    SCH_delete       = 1,

    SCH_transform    = 2,

    SCH_propagate              = 3,

    SCH_keep_sub_dominant = 4,

    SCH_keep_if_equal   = 5,

    SCH_combine      = 6

    }

    SCH_action_on_fields_t;


typedef enum
    {

    SCH_as_owner  = 0,

    SCH_in_owner  = 1,

    SCH_by_owner  = 2,

    SCH_sh_owner  = 3,

    SCH_fa_owner  = 4,

    SCH_lo_owner  = 5,

    SCH_ed_owner  = 6,

    SCH_vx_owner  = 7,

    SCH_fe_owner  = 8,

    SCH_sf_owner  = 9,

    SCH_cu_owner  = 10,

    SCH_pt_owner  = 11,

    SCH_rg_owner  = 12,

    SCH_fn_owner  = 13,

    SCH_max_owner            //  last entry; value in $l[] for
                             .legal_owners
    } SCH_attrib_owners_t;

typedef enum
    {
    SCH_int_field            = 1,
    SCH_real_field           = 2,
    SCH_char_field           = 3,
    SCH_point_field          = 4,
    SCH_vector_field         = 5,
    SCH_direction_field      = 6,
    SCH_axis_field           = 7,
    SCH_tag_field            = 8,
    SCH_pointer_field        = 9,
    SCH_unicode_field        = 10
    } SCH_field_type_t;

struct ATTRIB_DEF_s           //  attribute definition
    {
    struct ATTRIB_DEF_s        *next;                       //  $p
    struct ATT_DEF_ID_s        *identifier;                 //  $p
    int                         type_id;                    //  $d
    SCH_action_on_fields_t      actions                     //  $u[8]
                               [(int)SCH_max_logged_event]
                               ;
    struct FIELD_NAMES_s       *field_names                 //  $p
    logical                     legal_owners                //  $l[14]
                                [(int)SCH_max_owner];
    SCH_field_type_t            fields[1];                  //  $u[]
    };

typedef struct ATTRIB_DEF_s    *ATTRIB_DEF;
```

E.5.4.6 ATTRIBUTE

Table F.50 — Attribute fields

Field name	Type	Description
node_id	int	Node-id
definition	pointer	Attribute definition
owner	pointer	Attribute owner
next	pointer0	Next attribute, group, or member_of_group
previous	pointer0	Previous ditto
next_of_type	pointer0	Next attribute of this type in this part

previous_of_type	pointer0	Previous attribute of this type in this part
fields[]	pointer	Fields, of type int_values etc. The number of fields is given by the length of the variable part of the node. There may be no fields.

The attributes of a node are chained using the next and previous pointers in the attribute. The attribute_groups pointer in the node points to the head of this chain. This chain also contains the member_of_groups of the node.

Attributes within the same part, with the same attribute definition, are chained together by the next_of_type and previous_of_type pointers. The part points to the head of this chain as follows. The attribute_chains pointer in the part points to a list which contains the heads of these attribute chains, one for each attribute definition which has attributes in the part. The list may be null.

Note that the attributes_groups chains in parts, groups and nodes contain the following types of node:

Part:	attributes and groups
Group:	attributes
Node:	attributes and member_of_groups

```
union ATTRIBUTE_OWNER_u
      {
    struct ASSEMBLY_s         *assembly;
    struct INSTANCE_s         *instance;
    struct BODY_s             *body;
    struct SHELL_s            *shell;
    struct REGION_s           *region;
    struct FACE_s             *face;
    struct LOOP_s             *loop;
    struct EDGE_s             *edge;
    struct FIN_s              *fin;
    struct VERTEX_s           *vertex;
    union  SURFACE_u           Surface;
    union  CURVE_u             Curve;
    struct POINT_s            *point;
    struct GROUP_s            *group;
    };
typedef union ATTRIBUTE_OWNER_u ATTRIBUTE_OWNER;
union FIELD_VALUES_u
    {
    struct INT_VALUES_s          *int_values;
    struct REAL_VALUES_s         *real_values;
    struct CHAR_VALUES_s         *char_values;

    struct POINT_VALUES_s        *point_values;

    struct VECTOR_VALUES_s       *vector_values;

    struct DIRECTION_VALUES_s    *direction_values;

    struct AXIS_VALUES_s         *axis_values;

    struct TAG_VALUES_s          *tag_values;

    struct UNICODE_VALUES_s      *unicode_values;

    };

typedef union FIELD_VALUES_u FIELD_VALUES;

struct ATTRIBUTE_s                           //  Attribute
    {

    int                             node_id;                 //  $d

    struct ATTRIB_DEF_s            *definition;              //  $p

    union  ATTRIBUTE_OWNER_u        owner;                   //  $p

    union  ATTRIB_GROUP_u           next;                    //  $p

    union  ATTRIB_GROUP_u           previous;                //  $p

    struct ATTRIBUTE_s             *next_of_type;            //  $p

    struct ATTRIBUTE_s             *previous_of_type;        //  $p

    union  FIELD_VALUES_u           fields[1];               //  $p[]

    };

typedef struct ATTRIBUTE_s *ATTRIBUTE;
```

E.5.4.7 INT_VALUES

Table F.51 — Integer values

values[]	int	Integer values

```
struct INT_VALUES_s                      //  Int values
    {

    int                      values[1];                      //  $d[]

    };

typedef struct INT_VALUES_s *INT_VALUES;
```

E.5.4.8 REAL_VALUES

Table F.52 — Real values

values[]	double	Real values

```
struct REAL_VALUES_s                     //  Real values
```

```
    {
    double                   values[1];                     //  $f[]
    };
typedef struct REAL_VALUES_s *REAL_VALUES;
```

E.5.4.9 CHAR_VALUES

Table F.53 — Character values

values[]	char	Character values

```
struct CHAR_VALUES_s                    //  Character values
    {
    char                     values[1];                     //  $c[]
    };
typedef struct CHAR_VALUES_s *CHAR_VALUES;
```

E.5.4.10 UNICODE_VALUES

Table F.54 — Unicode values

values[]	short	Unicode character values

```
struct UNICODE_VALUES_s                    //  Unicode character values
    {
    short                    values[1];                     //  $w[]
    };
typedef struct UNICODE_VALUES_s *UNICODE_VALUES;
```

E.5.4.11 POINT_VALUES

Table F.55 — Point values

values[]	vector	Point values

```
struct POINT_VALUES_s                   //  Point values
     {
    vector                values[1];                      // $v[]
    };
typedef struct POINT_VALUES_s *POINT_VALUES;
```

E.5.4.12 VECTOR_VALUES

Table F.56 — Vector values

values[]	vector	Vector values

```
struct VECTOR_VALUES_s                  //  Vector values
    {
    vector                values[1];                      //  $v[]
    };

typedef struct VECTOR_VALUES_s *VECTOR_VALUES;
```

E.5.4.13 DIRECTION_VALUES

Table F.57 — Direction values

values[]	vector	Direction values

```
struct DIRECTION_VALUES_s                //  Direction values
    {
    vector                 values[1];                      //  $v[]
    };

typedef struct DIRECTION_VALUES_s *DIRECTION_VALUES;
```

E.5.4.14 AXIS_VALUES

Table F.58 — Axis values

values[]	vector	Axis values

Note that an axis takes up two vectors.

```
struct AXIS_VALUES_s                     //  Axis values
    {
    vector                 values[1];                      //  $v[]
    };

typedef struct AXIS_VALUES_s *AXIS_VALUES;
```

E.5.4.15 TAG_VALUES

Table F.59 — Tag values

values[]	int	Integer tag values

The tag field type and the tag_values node are not available for use in user-defined attributes, they occur only in certain system attributes.

```
struct TAG_VALUES_s                      //  Tag values
    {
    int                    values[1];                      //  $t[]
    };

typedef struct TAG_VALUES_s *TAG_VALUES;
```

E.5.4.16 GROUP

Table F.60 — Group fields

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of this group
owner	pointer	Owning part

next	pointer0	Next group or attribute
previous	pointer0	Previous group or attribute
type	byte	Type of node allowed in group
first_member	pointer0	Head of chain of member_of_group nodes in group

The groups in a part are chained by the next and previous pointers in a group. The attributes_groups pointer in the part points to the head of the chain. This chain also contains the attributes attached directly to the part - groups and attributes are intermingled in this chain, the order is not significant.

Each group has a chain of member_of_groups. These are chained together using the next_member and previous_member pointers. The first_member pointer in the group points to the head of the chain. Each member_of_group has an owning_group pointer which points back to the group.

Each member_of_group has an owner pointer which points to a node. Thus the group references its member nodes via the member_of_groups.

The member_of_groups which refer to a particular node are chained using the next and previous pointers in the member_of_group. The attributes_groups pointer in the node points to the head of this chain. This chain also contains the attributes attached to the node.

```
typedef enum
    {
    SCH_instance_fe   = 1,
    SCH_face_fe       = 2,
    SCH_loop_fe       = 3,
    SCH_edge_fe       = 4,
    SCH_vertex_fe     = 5,
    SCH_surface_fe    = 6,
    SCH_curve_fe      = 7,
    SCH_point_fe      = 8,
    SCH_mixed_fe      = 9,
    SCH_region_fe     = 10
    }  SCH_group_type_t;

struct GROUP_s                        //  Group
    {
    int                            node_id;                  //  $d
    union   ATTRIB_GROUP_u         attributes_groups;        //  $p
    union   PART_u                 owner;                    //  $p
    union   ATTRIB_GROUP_u         next;                     //  $p
    union   ATTRIB_GROUP_u         previous;                 //  $p
    SCH_group_type_t               type;                     //  $u
    struct  MEMBER_OF_GROUP_s     *first_member;                //  $p
    };
typedef struct GROUP_s *GROUP;
```

E.5.4.17 MEMBER_OF_GROUP

Table F.61 — Group member fields

Field name	Type	Description
dummy_node_id	int	Entity label
owning_group	pointer	Owning group
owner	pointer	Referenced member of group
next	pointer0	Next attribute, group or member_of_group
previous	pointer0	Previous ditto
next_member	pointer0	Next member_of_group in this group
previous_member	pointer0	Previous ditto

```
union GROUP_MEMBER_u
     {
    struct INSTANCE_s           *instance;
    struct FACE_s               *face;
    struct REGION_s             *region;
    struct LOOP_s               *loop;
    struct EDGE_s               *edge;
    struct VERTEX_s             *vertex;
    union  SURFACE_u             surface;
    union  CURVE_u               curve;
    struct POINT_s              *point;
    };
typedef union GROUP_MEMBER_u GROUP_MEMBER;
struct MEMBER_OF_GROUP_s             //  Member of group
    {
    int                           dummy_node_id;                //  $d
    struct GROUP_s               *owning_group;                 //  $p
    union  GROUP_MEMBER_u         owner;                        //  $p
    union  ATTRIB_GROUP_u         next;                         //  $p
    union  ATTRIB_GROUP_u         previous;                     //  $p
    struct MEMBER_OF_GROUP_s     *next_member;                  //  $p
    struct MEMBER_OF_GROUP_s     *previous_member;              //  $p
    };
```

```
typedef struct MEMBER_OF_GROUP_s *MEMBER_OF_GROUP;
```

E.6 Node Types

Table F.62 — Node types

Node name	Node type	Visible at PK	Has node-id
ASSEMBLY	10	Yes	No
INSTANCE	11	Yes	Yes
BODY	12	Yes	No
SHELL	13	Yes	Yes
FACE	14	Yes	Yes
LOOP	15	Yes	Yes
EDGE	16	Yes	Yes
FIN	17	Yes	No
VERTEX	18	Yes	Yes
REGION	19	Yes	Yes
POINT	29	Yes	Yes
LINE	30	Yes	Yes
CIRCLE	31	Yes	Yes
ELLIPSE	32	Yes	Yes
INTERSECTION	38	Yes	Yes
CHART	40	No	
LIMIT	41	No	
BSPLINE_VERTICES	45	No	
PLANE	50	Yes	Yes
CYLINDER	51	Yes	Yes
CONE	52	Yes	Yes
SPHERE	53	Yes	Yes
TORUS	54	Yes	Yes
BLENDED_EDGE	56	Yes	Yes
BLEND_BOUND	59	No	
OFFSET_SURF	60	Yes	Yes
SWEPT_SURF	67	Yes	Yes
SPUN_SURF	68	Yes	Yes
LIST	70	Yes	Yes
POINTER_LIS_BLOCK	74	No	
ATT_DEF_ID	79	No	
ATTRIB_DEF	80	Yes	No
ATTRIBUTE	81	Yes	Yes
INT_VALUES	82	No	
REAL_VALUES	83	No	
CHAR_VALUES	84	No	
POINT_VALUES	85	No	
VECTOR_VALUES	86	No	
AXIS_VALUES	87	No	
TAG_VALUES	88	No	
DIRECTION_VALUES	89	No	
GROUP	90	Yes	Yes

MEMBER_OF_GROUP	91	No	
UNICODE_VALUES	98	No	
FIELD_NAMES	99	No	
TRANSFORM	100	Yes	Yes
WORLD	101	No	
KEY	102	No	
PE_SURF	120	Yes	Yes
INT_PE_DATA	121	No	
EXT_PE_DATA	122	No	
B_SURFACE	124	Yes	Yes
SURFACE_DATA	125	No	
NURBS_SURF	126	No	
KNOT_MULT	127	No	
KNOT_SET	128	No	
PE_CURVE	130	Yes	Yes
TRIMMED_CURVE	133	Yes	Yes
B_CURVE	134	Yes	Yes
CURVE_DATA	135	No	
NURBS_CURVE	136	No	
SP_CURVE	137	Yes	Yes
GEOMETRIC_OWNER	141	No	
HELIX_CU_FORM	163	No	
HELIX_SU_FORM	184	No	

E.7 Node Classes

Table F.63 — Node classes

Node class name	Node class
GEOMETRY	1003
PART	1005
SURFACE	1006
SURFACE_OWNER	1007
CURVE	1008
CURVE_OWNER	1010
POINT_OWNER	1011
LIS_BLOCK	1012
LIST_OWNER	1013
ATTRIBUTE_OWNER	1015
GROUP_OWNER	1016
GROUP_MEMBER	1017
FIELD_VALUES	1018
ATTRIB_GROUP	1019
TRANSFORM_OWNER	1023
PE_DATA	1027
PE_INT_GEOM	1028
SHELL_OR_BODY	1029
FIELD_NAME	1037

E.8 System Attribute Definitions

All system attribute definitions are of class 1.

E.8.1 Hatching Attributes

E.8.1.1 Hatching

Table F.64 — Hatching

Identifier	SDL/TYSA_HATCHING	
Type_id	8003	
Entity types	face	
Fields	real	real 1
		real 2
		real 3
		real 4
	integer	Hatching type
Set by	Application	

For **planar hatching** - the four real values define the hatch orientation as a vector and a spacing between consecutive planes.

For **radial hatching** - the first three real values define the spacing of the hatch lines. The fourth value is not used.

For **parametric hatching** - the first two real values define the spacing in *u* and *v* respectively. The last two values are not used.

E.8.1.2 Planar Hatch

Table F.65 — Planar Hatch

Identifier	SDL/TYSA_PLANAR_HATCH		
Type_id	8021		
Entity types	face		
Fields	real	x component	‚direction‘ or plane normal
		y component	
		z component	
		‚pitch‘ or separation	
		x component	position vector
		y component	
		z component	
Set by	Application		

For planar hatching, an attribute with this definition takes precedence over an attribute with the SDL/TYSA_HATCHING definition, if a face has both types of attribute attached.

E.8.1.3 Radial Hatch

Table F.66 — Radial Hatch

Identifier	SDL/TYSA_RADIAL_HATCH

Type_id	8027	
Entity types	face	
Fields	real	radial around
		radial along
		radial about
		radial around start
		radial along start
		radial about start
Set by	Application	

For radial hatching, an attribute with this definition takes precedence over an attribute with the SDL/TYSA_HATCHING definition, if a face has both types of attribute attached.

E.8.1.4 Parametric Hatch

Table F.67 — Parametric Hatch

Identifier	SDL/TYSA_PARAM_HATCH	
Type_id	8028	
Entity types	face	
Fields	real	u spacing
		v spacing
		u start
		v start
Set by	Application	

For parametric hatching, an attribute with this definition takes precedence over an attribute with the SDL/TYSA_HATCHING definition, if a face has both types of attribute attached.

E.8.2 Density Attributes

There are density attributes for each of regions, faces, edges and vertices in addition to the system attribute for density of a body.

The region/face/edge/vertex attributes will be taken into account when finding the mass, centre of gravity and moment of inertia of a body or of the entity to which the attribute is attached:

- The mass of a region will not include that of any of its faces or edges, and the same applies to faces and edges and their boundaries.

- A void region will always have zero mass whatever its density and a solid region will inherit its density from the body if it does not have a density of its own.

- The default density for faces, edges and vertices is always zero.

E.8.2.1 Density (of a body)

Table F.68 — Body Density

Identifier	SDL/TYSA_DENSITY	
Type_id	8004	
Entity types	body	
Fields	real	Density
	string	Units
Set by	Application	

A body without a density attribute is taken to have, by default, a density of 1.0.

The character field units can be set and read by the application.

E.8.2.2 Region Density

Table F.69 — Region Density

Identifier	SDL/TYSA_REGION_DENSITY	
Type_id	8023	
Entity types	region	
Fields	real	Density of region
	string	Units
Set by	Application	

This attribute only makes sense for solid regions; void regions always have a mass of zero.

A solid region without a density attribute is taken to have, by default, the same density as its owning body.

The character field units can be set and read by the user.

E.8.2.3 Face Density

Table F.70 — Face Density

Identifier	SDL/TYSA_FACE_DENSITY	
Type_id	8024	
Entity types	face	
Fields	real	Density of face
	string	Units
Set by	Application	

The value of this attribute is treated as a mass per unit area.

A mass will be calculated for a face only when a face possesses this attribute. In all other cases the mass of a face is not defined.

The character field units can be set and read by the user.

E.8.2.4 Edge Density

Table F.71 — Edge Density

Identifier	SDL/TYSA_EDGE_DENSITY	
Type_id	8025	
Entity types	edge	
Fields	real	Density of edge
	string	Units
Set by	Application	

The value of this attribute is treated as a mass per unit length.

A mass will be calculated for an edge only when an edge possesses this attribute. In all other cases the mass of an edge is not defined.

The character field units can be set and read by the user.

E.8.2.5Vertex Density

Table F.72 — Vertex Density

Identifier	SDL/TYSA_VERTEX_DENSITY	
Type_id	8026	
Entity types	vertex	
Fields	real	Mass of vertex
	string	Units
Set by	Application	

The value of this attribute is treated as a point mass.

A mass will be calculated for a vertex only when a vertex possesses this attribute. In all other cases the mass of a vertex is not defined.

The character field units can be set and read by the user.

E.8.3 Region

Table F.73 — Region

Identifier	SDL/TYSA_REGION	
Type_id	8013	
Entity types	face	
Fields	string	Unused
Set by	Application	

Regional data will allow the application to analyze a hidden-line picture for distinct regions in the 2D view.

E.8.4 Colour

Table F.74 — Colour

Identifier	SDL/TYSA_COLOUR		
Token	8001		
Entity types	face edge		
Fields	real	Red value	These three values should be in the range 0.0 to 1.0
		Green value	
		Blue value	
Set by	Application		

E.8.5 Reflectivity

Table F.75 — Reflectivity

Identifier	SDL/TYSA_REFLECTIVITY	
Token	8014	
Entity types	face	
Fields	real	Coefficient of specular reflection
		Proportion of coloured light in highlights
		Coefficient of diffuse reflection
		Coefficient of ambient reflection

	integer	Reflection power
Set by	Application	

The attribute types for Reflectivity and Translucency are also used by the Parasolid routine RRPIXL, but the use of this routine is not recommended.

E.8.6 Translucency

Table F.76 — Translucency

Identifier	SDL/TYSA_TRANSLUCENCY		
Token	8015		
Entity types	face		
Fields	real	Transparency coefficient	range 0.0 to 1.0, where 0 is opaque and 1 is transparent
Set by	Application		

E.8.7 Name

Table F.77 — Name

Identifier	SDL/TYSA_NAME	
Token	8017	
Entity types	assembly, body, instance, shell, face, loop, edge, vertex, group, surface, curve, point	
Fields	string	Name of entity
Set by	Application	

.

E.8.8 Incremental faceting

Table F.78 — Incremental faceting

Identifier	SDL/TYSA_INCREMENTAL_FACETTING	
Token	8030	
Entity types	face	
Fields	string	Unused
Set by	XT incremental faceting/Application	

E.8.9 Transparency

Table F.79 — Transparancy

Identifier	SDL/TYSA_TRANSPARENCY	
Token	8029	
Entity types	Body, face	
Fields	integer	Non-zero transparency coefficient

		value is transparent
Set by	Application	

A body may be rendered transparent if it has an attached transparency attribute with a non-zero transparency coefficient

E.8.10Non-mergeable edges

Table F.80 — Non-mergable edges

Identifier	SDL/TYSA_NO_MERGE	
Token	8032	
Entity types	Edge	
Fields	string	Unused
Set by	Application	

If an edge has an attribute of this definition attached, it indicates that the edge should not be merged in any modelling operations.

E.8.11Group merge behaviour

Table F.81 — Group merge behaviour

Identifier	SDL/TYSA_GROUP_MERGE	
Token	8037	
Entity types	Group	
Fields	string	Unused
Set by	Application	

If a group has an attribute of this definition attached, it indicates that alternative behaviour should be used if an entity in the group is merged with an entity not in that group.

E.8.12Unicode name

Table F.82 — Unicode name

Identifier	SDL/TYSA_UNAME	
Token	8038	
Entity types	assembly, body, instance, shell, face, loop, edge, vertex, group, surf, curve, point, region	
Fields	ustring	Name of entity
Set by	Application	

If a group has an attribute of this definition attached, it indicates that alternative behaviour should be used if an entity in the group is merged with an entity not in that group.

Annex F

XT data description – questions and answers

F.1 intersection curves

Intersection curves are represented procedurally in the same manner as (for example) swept, spun or offset surfaces. As with all procedural geometry, some computations using common, standard mathematical techniques are required to fully represent and interpret intersection curves.

In addition to this (and similar to b-spline curves), intersection curves are stored in discrete form. Properties of the curve between the discrete stored points can be interpolated from the stored information again using common, standard mathematical techniques.

The information in this section:

- Provides additional clarification with regards to the meaning of the intersection curve data that is stored within the transmitted XT Data
- Explains how the non-transmitted information for each chart point can be fully derived using standard techniques
- Clarifies how curve data can be interpolated to uniquely determine the positions of intermediate points (points on the curve between transmitted chart points)

Question G1.1; XT Intersection Curve: How can the hvec data that is not transmitted be determined?

The hvec data structure is as follows*:*

```
Typdef struct hvec_s     // hepta_vec
Vector                   Pvec;           // position
Double                   u[2];           // surface parameters
double                   v[2];
vector                   Tangent;        // curve tangent
double                   t;              // curve parameter
} hvec;
```

The only hvec data that is transmitted is the model-space position vectors. Each limit point and each chart point in the curve lies on both surfaces on which the definition of the intersection curve is based. The limit and chart points are chosen to uniquely identify which branch of the intersection the node refers to. There is a unique parameterization of the points within the branch.

The remaining elements of the hvec structure can all be derived using the transmitted data as follows:

- The [u,v] parameters and surface normals at each position can be calculated by standard surface evaluation techniques for those surfaces
    - Since the chart points themselves lie exactly on both underlying surfaces, no model-space projection of the point into the underlying surfaces is required. The mathematical formulae representing those surfaces can be used to accurately calculate the corresponding 2D surface parameters from those 3D points
- The curve tangent can be calculated from the surface normals following the guidelines in the XT Data Description. Note that there are special rules for calculating the tangent at a terminator. Please refer to the extract below:
    - The natural tangent to the curve at any point (i.e. in the increasing parameter direction) is given by calculating the [normalized] vector cross-product of the two surface normals [the surface normal of each of the two surfaces in the intersection curve structure] at that point.

      With the intersection curve structure being as follows:

      ```
          struct INTERSECTION_s == ANY_CURVE_s // Intersection
          {
          int node_id; // $d
      ```

```
union ATTRIB_GROUP_u attributes_groups; // $p
union CURVE_OWNER_u owner; // $p
union CURVE_u next; // $p
union CURVE_u previous; // $p
struct GEOMETRIC_OWNER_s *geometric_owner; // $p
char sense; // $c
union SURFACE_u surface[ 2 ]; // $p[2]
struct CHART_s *chart; // $p
struct LIMIT_s *start; // $p
struct LIMIT_s *end; // $p
};
typedef struct INTERSECTION_s *INTERSECTION;
```

And natural normal, surface sense, and surface normal being defined as:

- The ‗natural‘ normal to a surface is the direction of the cross-product of dP/du and dP/dv.
- The surface sense is a field ‗sense‘ consisting of a single character (‗+‘ or ‗-‗) in the data structure for every surface in the XT data. A positive (+) surface sense indicates the surface normal lies parallel to the natural surface normal and a negative (-) surface sense indicates the surface normal lies anti-parallel to the natural surface normal.
- The surface normal is calculated using a combination of the natural normal to a surface and the surface sense.

- Singular points where the cross-product of the surface normals is zero, or where one of the surfaces is degenerate, are called terminators. Intersection curves do not contain terminators in their interior. At terminators, the tangent to the curve is defined by the limit of the curve tangent as the curve parameter approaches the terminating value.

The parameterization of a chart point can be calculated using the following methodology

The chart structure is as follows. Information from the chart is required to calculate the parameterization of a chart point. In particular, base_parameter and base_scale are used in later formulae.

```
struct CHART_s // Chart {
double          Base_parameter;          // $f
double          Base_scale;              // $f
int             Chart_count;             // $d
double          Chordal_error;           // $f
double          Angular_error;           // $f
double          Parameter_error[2];      // $f[2]
hvec            Hvec[ 1 ];               // $h[]
};
```

where

- base_parameter is the parameter of the first hvec in the chart
- base_scale determines the scale of the parameterization (see below)
- chart_count is the length of the hvec array
- chordal_error is an estimate of the maximum deviation of the curve from the piecewise-linear approximation given by the hvec array. It may be null.
- angular_error is the maximum angle between the tangents of two sequential hvecs. It may be null.
- parameter_error[] is always [null, null].
- hvec[] is the ordered array of hvecs.

The method for calculating the parameterization of a chart point is as follows (extract from the XT Data Description):

The parameterization of the curve is given as follows. If the chart points are Pi, i = 0 to n, with parameters ti, and natural tangent vectors Ti, then define

Ci = | Pi+1 – Pi |
cos(ai) = Ti . (Pi+1 – Pi) / Ci
cos(bi) = Ti . (Pi – Pi-1) / Ci-1

Then at any chart point Pi the angles ai and bi are the deviations between the tangent at the chart point and the next and previous chords respectively.

```
        Let    f0 = base_scale
               fi = ( cos(bi) / cos(ai) ) fi-1
        Then  t0 = base_parameter
               ti = ti-1 + Ci-1 fi-1
```

The above algorithm allows the parameters of the hvecs to be calculated in order.

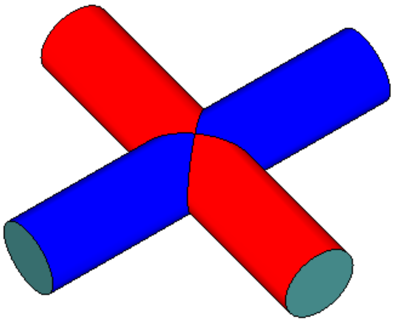
Question G1.2; XT Intersection Curve: Do start and end points in the chart data belong to the curve or are these points just used to determine tangential directions?

This question concerns the start and end limits of the intersection curve. These limits are a separate part of the intersection curve structure to the chart points themselves. See the extract from the XT Data Description below that describes the structure.

```
struct INTERSECTION_s == ANY_CURVE_s // Intersection
{
int                            node_id;              // $d
union ATTRIB_GROUP_u           attributes_groups;    // $p
union CURVE_OWNER_u            owner;                // $p
union CURVE_u                  next;                 // $p
union CURVE_u                  previous;             // $p
struct GEOMETRIC_OWNER_s       *geometric_owner;     // $p
char                           sense;                // $c
union SURFACE_u                surface[ 2 ];         // $p[2]
struct CHART_s                 *chart;               // $p
struct LIMIT_s                 *start;               // $p
struct LIMIT_s                 *end;                 // $p
};
```

The limits define the types of the two ends of the intersection curve, and their position (which may be the same in the case of a periodic intersection curve). In this way, the limits are also used to identify the branch of the intersection between the surfaces to which the curve relates.

For example, there are four possible branches of the intersection between the red and blue cylindrical b-spline surfaces in the image below.



A single hvec is provided where the limit type is a help hvec (H) or an arbitrary limit (L):

- This position represents the start/end of the curve. This position will also appear in the chart.

Two hvecs are provided where the limit type is a terminator (T):

- The first hvec will be the exact position of the singularity. This position will not appear in the chart.
- The second hvec will be a small distance along the curve in the correct branch of the intersection. This position will appear in the chart

See the extract below from the XT Data Description for further details:

- The two ends of the intersection curve. These are referred to as the ‗limits‘ of the curve. They identify the particular branch involved.

The limits of the intersection curve are stored in the following data structure:

```
struct LIMIT_s // Limit
{
char                type;               // $c
hvec                hvec[ 1 ];          // $h[]
};
```

The ‗type‘ field may take one of the following values

```
                                                // help hvec
const char SCH_help = 'H';
const char SCH_terminator = 'T';                // terminator
const char SCH_limit = 'L';                     // arbitrary limit
const char SCH_boundary = 'B';                  // spine boundary
```

The length of the hvec array depends on the type of the limit.

- a SCH_help limit is an arbitrary point on a closed intersection curve. There will be one hvec in the hvec array, locating the curve.
- a SCH_terminator limit is a point where one of the surface normals is degenerate, or where their cross-product is zero. Typically, there will be more than one branch of intersection between the two surfaces at these singularities. There will be two values in the hvec array. The first will be the exact position of the singularity, and the second will be a point on the curve a small distance away from the terminator. This ‗branch point‘ identifies which branch relates to the curve in question. The branch point is the one which appears in the chart, at the corresponding end – so the singularity lies just outside the parameter range of the chart.
- a SCH_limit limit is an artificial boundary of an intersection curve on an otherwise potentially infinite branch. The single hvec describes the end of the curve.
- a SCH_boundary limit is used to describe the end of a degenerate rolling-ball blend. It is not relevant to intersection curves.

The rules for calculating curve tangents at the chart points are covered in the response to the previous question.

Question G1.3; What are the necessary steps to determine the position vector p(t) for an arbitrary curve parameter t between two adjacent hvecs of an only implicitly defined intersection curve? Are the chart parameters ‗chordal_error‘ and ‗angular_error‘ only for information or are they needed to calculate p(t) values?

The following extract from the XT Data Description explains how the position vector for an arbitrary curve parameter t can be determined. The chordal_error and angular_error parameters are not required for the calculation of the position vectors.

The point on the intersection curve corresponding to a given parameter is defined as follows:

- For a parameter equal to that of a chart point, it is the position of the chart point.
- For a parameter interior to the chart, it is the local point of intersection of three surfaces: the two surfaces of the intersection, and a plane defined by the chart. If the parameter t lies between chart parameters ti, ti+1, then the chord point corresponding to t lies at

  $$(t_{i+1} - t) / (t_{i+1} - t_i) P_i + (t - t_i) / (t_{i+1} - t_i) P_{i+1}$$

  The plane lies through this point and is orthogonal to the chord (Pi+1, Pi).

- For a parameter between a branch chart point and a terminator, it is the local point of intersection of three surfaces: one of the intersection surfaces and two planes. Surface[0] is used unless it is singular at the terminator and surface[1] is not singular at the terminator. The first plane contains the chord between the branch and the terminator, and the normal of the chosen intersection surface at the terminator or the curve tangent at the branch chart point if the surface normal cannot be defined. The second plane is the plane orthogonal to the chord between the branch and terminator points through the chord point as calculated above.

The approach therefore should be as follows:

1. Calculate the parameters of each of the chart points
2. Categorize which of the three scenarios above applies for the arbitrary curve parameter t in question
   a. If the second scenario applies, create a plane using the point and normal information using standard techniques. Then intersect the two surfaces underlying the curve and the plane to find the unique intersection point
   b. If the third scenario applies, create two planes from the information described above using standard techniques. Then intersect the two planes with the appropriate surface to find the unique intersection point.

As covered above, the chordal_error and angular_error parameters are not required for the calculation of the p(t) values. To ensure there is a unique solution to the evaluation of a curve at a parameter t, a sufficient number of chart points must be present. The number of chart points for a given intersection curve is correlated to the chordal_error and angular_error values which provide an indication of the deviation of the chordal polyline from the true intersection

Question G1.4; XT Intersection Curve: How do I avoid instabilities when performing calculations in the vicinity of a terminator?

The following definition of a terminator is from the XT Data description:

- Singular points where the cross-product of the surface normals is zero, or where one of the surfaces is degenerate, are called terminators.

As covered earlier in this material, the data structure for each intersection curve will include at least one limit point. Where there are multiple branches of the intersection between the two surfaces, the information contained within the limit point(s) uniquely identifies the particular branch of the surface-surface intersection on which the intersection curve lies.
In such circumstances the limit point(s) will have a terminator limit type ‗SCH_terminator' as described below in the description:

- a SCH_terminator limit is a point where one of the surface normals is degenerate, or where their cross-product is zero. Typically, there will be more than one branch of intersection between the two surfaces at these singularities. There will be two values in the hvec array. The first will be the exact position of the singularity, and the second will be a point on the curve a small distance away from the terminator. This 'branch point' identifies which branch relates to the curve in question. The branch point is the one which appears in the chart, at the corresponding end – so the singularity lies just outside the parameter range of the chart.

Terminator limit types specifically exist to help avoid the potential instabilities when moving towards singularities. For an intersection curve ‗containing' the terminator, the branch point rather than the terminator point itself will be present in the chart point array.

Special rules should be applied when determining or calculating properties of the curve in the vicinity of a terminator as described in various sections of the XT Data description.
Intersection curves do not contain terminators in their interior.

The point on the intersection curve corresponding to a given parameter is defined as follows:

- For a parameter interior to the chart, it is the local point of intersection of three surfaces: the two surfaces of the intersection, and a plane defined by the chart. If the parameter t lies between chart parameters t_i, t_{i+1}, then the chord point corresponding to t lies at

  $(t_{i+1} - t) / (t_{i+1} - t_i) P_i + (t - t_i) / (t_{i+1} - t_i) P_{i+1}$

  The plane lies through this point and is orthogonal to the chord (P_{i+1}, P_i).

- For a parameter between a branch chart point and a terminator, it is the local point of intersection of three surfaces: one of the intersection surfaces and two planes. Surface[0] is used unless it is singular at the terminator and surface[1] is not singular at the terminator. The first plane contains the chord between the branch and the terminator, and the normal of the chosen intersection surface at the terminator or the curve tangent at the branch chart point if the surface normal cannot be defined. The second plane is the plane orthogonal to the chord between the branch and terminator points through the chord point as calculated above.

‘as calculated above’ refers to the formulae supplied above for when the parameter is interior to the chart). So for example:

For clarity, the three surfaces that are intersected to give the chart point are:

1. Surface[0] unless it is singular at the terminator and surface [1] is not singular at the terminator
2. Plane #1 – a plane that contains:
   - The chord between branch point and terminator point
   - One of the following
     - normal of the chosen surface at the terminator OR (if the surface normal at the terminator cannot be defined)
     - The curve tangent at the branch chart point
3. Plane #2 – a plane that:
   - Is orthogonal to the chord between the branch and terminator points
   - Goes through the chord point

Curve tangent

- At terminators, the tangent to the curve is defined by the limit of the curve tangent as the curve parameter approaches the terminating value.

For example, an intersection curve with (n+1) chart points and two terminators (one at the ‘start’ and one at the ‘end’ of the curve) has the following:

- The chart points of the curve are P0 to Pn
- P0 is the branch point of the start limit
- Pn is the branch point of the end limit
- P-1 represents the start terminator point for the calculation of the parameterization
- Pn+1 represents is the end terminator point for the calculation of the parameterization

The representative parameterization of a terminator can be calculated using the standard formula quoted earlier

- ti = ti-1 + Ci-1 fi-1

Question G1.5; How would I handle a case where the intersection curve passes through a self-intersection in one of the underlying surfaces?

Intersection curves may not pass through a self-intersection in an underlying surface. This would lead to a self-intersecting curve and ambiguous evaluation. This is not permitted in XT.

F.2 rolling ball blend surface

Question G2.1; How does the blend radius r, correspond to the data stored in the XT Data? And is the parametric representation correct – can‘t r be positive for one surface and negative for another?

The blend radius r is the magnitude of the offset distance of each underlying surface and directly relates to the range values stored in the rolling ball blend surface data. The range value for each surface can be +r or –r. The + or - indicates the direction of the offset of the underlying surface taking into account the surface sense.

When interpreting the parametric representation quoted in the XT Data Description, the current formula is correct since the blend radius r itself is always a positive value (even though the offset value in the range field may be negative):

R(u, v) = C(u) + rX(u)cos(v a(u)) + rY(u)sin(va(u))

However, to avoid confusion you may prefer to think of the formula as follows:

R(u, v) = C(u) +|r|X(u)cos(v a(u)) + |r|Y(u)sin(va(u))

Quesiton G2.2; What is the spine curve of a rolling ball blend surface?

Conceptually, for a rolling ball blend surface, the spine curve describes the path of the centre of the ball as it moves along the two underlying surfaces, maintaining point contact with both surfaces at any one time as it does so.

This is explained in the XT Data Description as follows:
For rolling ball blends, the spine curve will be the intersection of the two surfaces obtained by offsetting the supporting surfaces by an amount given by the respective entry in range[]. Note that the offsets to be applied may be positive or negative, and that the sense of the surface is significant; i.e. the offset vector is the natural unit surface normal, times the range, times –1 if the sense is negative.

Question G2.3; What type of curves can be referenced as the spine of the rolling ball blend surface?

For a rolling ball blend surface that has been created using the Parasolid Kernel, the most common types of spine curves are ellipses, intersection curves, rational b-spline curves which exactly represent a portion of an ellipse.

Additionally, for cliff blends (type = E) created by the Parasolid Kernel, it is also common to see lines, b-curves and circles representing the spine of the supporting (zero radius rolling ball blend) surface.

The format does not preclude using any exact 3D curve as the spine curve within the blended_edge data structure.

Question G2.4; How is the rolling ball blend surface parameterized?

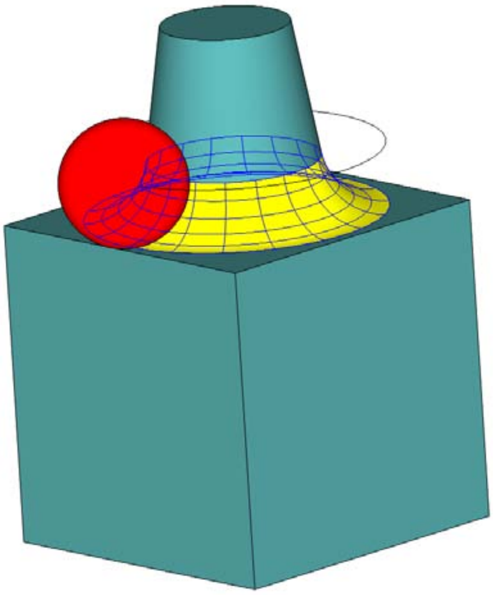
The XT Data Description explains the parameterization as follows:
The u parameter is inherited from the spine, the constant u lines being circles perpendicular to the spine curve. The v parameter is zero [0] at the blend boundary on the first surface, and one [1] on the blend boundary on the second surface; unless the sense of the spine curve is negative, in which case it is the other way round. The v parameter is proportional to the angle around the circle.

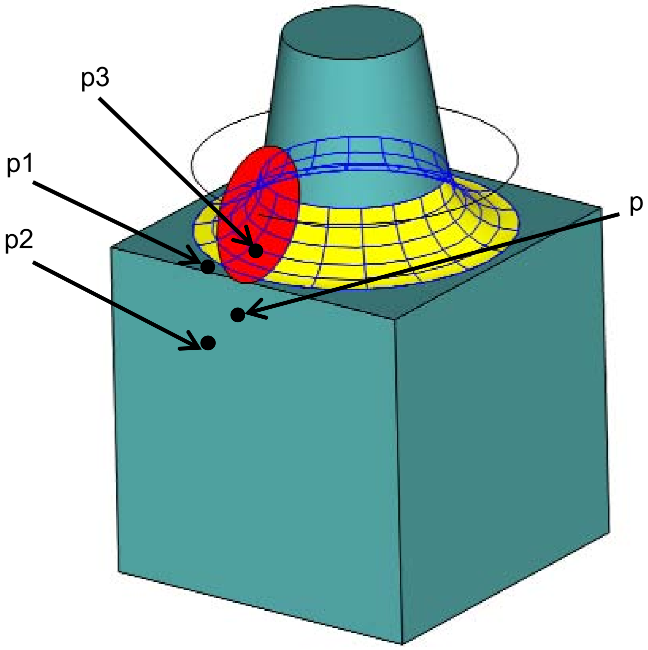
Question G2.5; Given an [x,y,z] point on the blend surface, how do I determine the corresponding [u,v] parameterization?

A blend surface is such that the following is always true:

1. Given a point p on the blend surface, find the closest point p1 on the blend spine. The parameter of that point p1 on the curve, provides the U parameterization of p
2. Given the point, p1 on the spine curve, find the closest points p2 and p3 on each of the underlying surfaces. These points p2 and p3 will both lie on the corresponding blend boundaries
3. One of the points p2 and p3 will have v parameter [0] and the other will have v parameter [1] dependent on the sense of the spine curve
4. Points p, p2 and p3 will all lie in the same circular arc on the underlying blend surface. The v parameterization of p will be proportional to its position along the arc between p2 and p3

The following pictures are representative of the above explanation. The red disc in the second picture represents the cross section of the rolling ball in a plane perpendicular to both the underlying conical and planar surfaces.



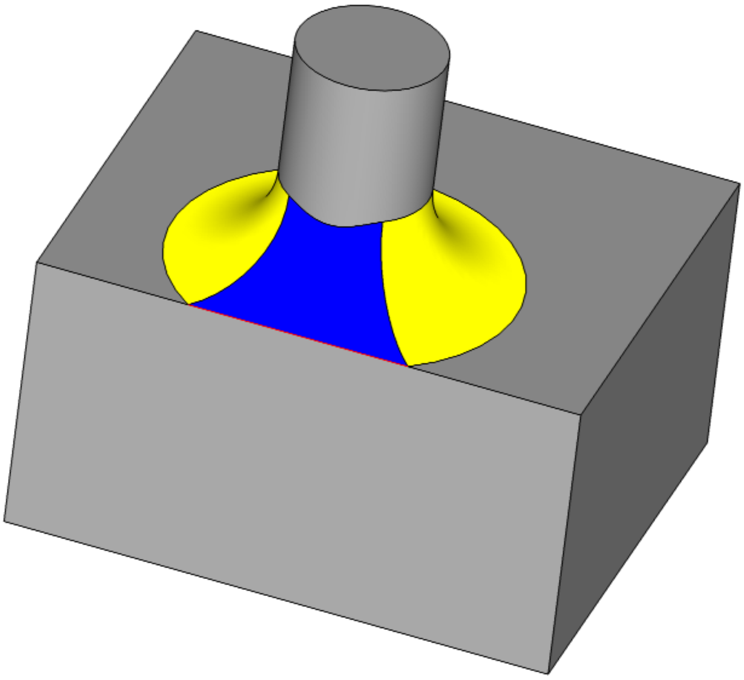


Question G2.6; What is a ‗cliff-edge blend‘? How does it relate to the blended_edge structure?

The presence of a ‗cliff-edge blend‘ (referred to in the following explanation as just ‗cliff blend‘) is denoted when the blend_type field of the blended_edge structure is set to ‗E‘ (rather than ‗R‘). This indicates that the blended_edge structure represents a cliff blend.

Conceptually, a cliff blend is one where a rolling ball blend surface would overflow a face boundary. In order to stay within the face boundary, the rolling ball moves along this ‗cliff‘ edge whilst also maintaining contact with the other surface and such that the resultant cliff blend surface meets that other surface tangentially.

A simple example of a rolling ball blend surface (in yellow) and a cliff blend surface (in blue) are shown in the picture below. The cliff (edge/curve) is also shown (in red). Note that this is shown for representation purposes only; if such a part had been constructed using the Parasolid Kernel, the yellow surface would be simplified exactly to a torus as part of the blend construction.



For a cliff blend, the spine curve is represented in a different way to the spine of a rolling ball blend surface. This is explained in the XT Data Description as follows:

For cliff edge blends, one of the surfaces will be a blended_edge with a range of [0,0]; its spine will be the cliff edge curve, and its supporting surfaces will be the surfaces of the faces adjacent to the cliff edge. Its type will be R.

For example in the case pictured above, the elements of the structure include the following:

For the rolling ball blend surface in yellow:

- One of the elements of the surface array is the cylinder
- The other element of the surface array is the planar surface of the top face
- The spine is a curve representing the intersection of the offsets of the two surfaces above
- The range is [+/-r, +/-r] (the sign for each takes into account the surface normal and surface sense)
- The blend_type is R

For the cliff blend surface in blue:

- One of the elements of the surface array is the cylinder
- The other element of the surface array is a rolling ball blend surface with range = [0,0]
    - Conceptually this is a representation of the cliff curve as a zero radius blend (see below for more details)
- The spine is derived in exactly the same way as for a rolling ball blend (based on the intersection of the offsets of the two surfaces)
- The range is [+/-r,+/-r]
- The blend_type is E

For the other element of the surface array above:

- One of the elements of its surface array is the front-facing planar surface adjacent to the edge
- The other element of its surface array is the top-facing planar surface adjacent to the edge in the original model
- Note that these two surfaces intersect to give the cliff curve
- The range is [0,0]
- The spine is the cliff curve (formed by intersecting the two surfaces), and can be conceptually thought of as the curve of the cliff edge
- The blend_type is R

F.3 procedural geometry – general

Question G3.1; How do I ensure that the answer(s) I compute for procedural geometry using the XT Data Description as my guide are the same as the answer that another such person would compute, or the same as an adopter of the Parasolid Kernel?

The procedural geometry present in XT Data is ‗exact‘ to the linear precision of 1.0e-08 units and angular precision of 1.0e-011 radians.

A typical mapping of 1 unit = 1 metre is used by adopters of the format. This is the convention used by all Siemens applications and almost all other adopters.

Elsewhere in this document it has been discussed that for some procedural geometry (e.g. intersection curves), the data is stored in discrete form. Data that is not stored can be computed exactly using the methods outlined. There is a unique answer in all cases.

Common techniques can be applied to compute this data. There are many textbooks, old and new, covering the techniques that can be applied. One such example of a textbook is ‗Computational Geometry for Design and Manufacture‘ I.D. Faux and M.J. Pratt (ISBN: 0853121141, 047026473X or 0470270691).

For example, in the case of an intersection curve between two b-spline surfaces, calculation of point values might require the intersection between two b-spline surfaces and a plane to be computed. Implementors can use common mathematical techniques to compute this unique intersection point. The method involves an iterative algorithm (such as a Newton Raphson approach) until the solution converges.

Question G3.2; If the procedural geometry were replaced by b-spline geometry, would this allow all implementors to obtain more consistent and better answers?

B-spline data is only as accurate as the algorithm that has been used to compute the b-spline approximation. By definition the data is explicit but is not always exact, since many shapes cannot be represented exactly using a b-spline. Intersection curves and blend surfaces are examples of such geometries that cannot in general be represented exactly using a b-spline.

Since b-spline data is explicit, all implementers will have the same representation of the b-spline geometry. Although the answers will be the same, they will only be faithful to the original data to the accuracy that has been used to compute the b-spline approximation. This would typically need to be of the order of 1.0e-05 to 1.0e-06 units.

Evaluating either a b-spline or procedural geometry is a numerical process. It is possible that variations in algorithm design, compiler, o/s or hardware can lead to very subtle differences in the results across different implementations. In both cases (b-spline and procedural geometry) these differences would typically be of the order of machine resolution.

Pragmatically speaking, the results obtained by any implementer using the XT Data Description are effectively the same and are also the same as that obtained by an implementer using the Parasolid Kernel. This applies to whether b-spline geometry or procedural geometry is used.

Taking all this into account, given the precise description of procedural geometry, implementers can ensure that precision of their geometry is the same accuracy as the original XT Data and is generally more precise than a b-spline approximation.

F.4 bounded Geometry

Question G.4.1; Please can you clarify how simple non-rational B spline curves and surfaces are supported . Is a 4D control point still required in this case, and if so is the weight value required to be 1.0?

There are the following scenarios to cover:

For a B curve in 2-D parameter space:

- The dimension of the control points is 2 (vertex_dim = 2) if the curve is non-rational (u,v)
- The dimension of the control points is 3 if the curve is rational (u,v,w)

For a non-rational curve or surface in 3-D Cartesian space

- The dimension of the control points is 3 if the curve is non-rational (x,y,z)
- The dimension of the control points is 4 if the curve is rational (x,y,z,w)

w above is the weight value of the control point.

The control point should be of the appropriate dimension as described above, and the weight does not need to be supplied for non-rational curves or surfaces.

F.5 geometry - general

Question G.5.1; My system / format only uses procedural geometry types like swept, spun and offset surfaces. How do I write procedural geometry such as blend surfaces and intersection curves into the XT Data?

Siemens PLM Software only recommends writing blend surfaces and intersection curves into the XT Data in the following circumstances:

1. When your system / format has equivalent data types and there is a straightforward mapping to one of these XT Data types
2. When you are using the Parasolid Kernel to write the XT Data

In other circumstances you should look at how the data type is represented in your own system/format and map this to the corresponding XT Data type. For example if blend data is always represented as b-surfaces in your system these should be written as b-surfaces rather than blend surfaces when writing the XT Data.

Question G.5.2; My system/format has data types which do not have a direct equivalent XT Data type. How should I represent these in the XT Data?

It may sometimes be the case that your system has data types which do not have a directly equivalent XT Data type. The recommended approach for such data types is to compute an appropriate alternative representation and write that into the XT Data. Some typical examples are as follows:

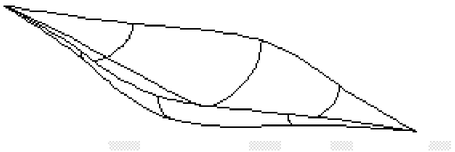
- Coons Patch – calculate the corresponding b-surface
- Gregory Patch – calculate the corresponding b-surface
- Elliptical Cylinder – calculate the corresponding b-surface or swept surface (swept ellipse)

Question G.5.3; What types of degenerate surface does XT Data support?

Degenerate surfaces are allowed in the XT data, but only in certain situations. A degeneracy is a place where a region of the parameter space of the surface reduces to a single point of Cartesian space. In this region one or both first partial derivatives are zero. The following restrictions apply:

- Interior Degeneracies are not allowed. Degeneracies are only allowed along the boundary of a surface. They may not occur in the interior, nor may they extend from the boundary inwards.
- Degeneracies into the surface are not allowed: The zero derivative of any degeneracy must be along the boundary, not into the surface itself.
- Partially degenerate boundaries are not allowed
- At any point a surface may only be degenerate in one of its parameters. In particular a corner of a surface may not be degenerate in both parameters, and a surface may not have two adjacent completely degenerate boundaries.

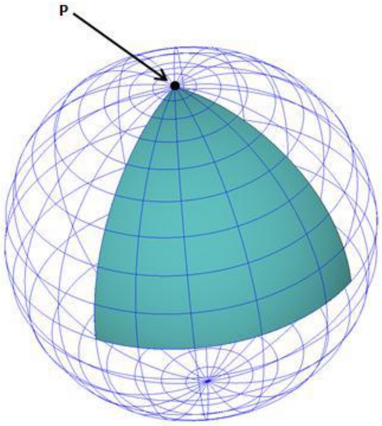
Below is an example where both boundaries in one of the parameter directions are completely degenerate. This is one example of a degeneracy that is permissible in XT.



Question G.5.3; How is a degenerate (zero-length) edge represented in the XT Data?

An edge which is zero length in model space is considered invalid. Such an entity would be represented by a (single) vertex in the XT Data.

For example a face such as that below which is has an underlying spherical surface and touches the pole of the sphere in one ‗corner' at point P. At P there needs to be a vertex rather than a degenerate edge



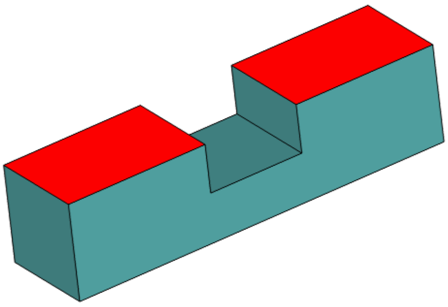
Question G.5.3; Do the XT Data structures support non-manifold and/or cellular topology?

In theory, the XT data structures could be used to represent non-manifold and / or cellular topology, but these data types are not permitted in the current version of the JT file format.

Question G.5.3; Can a face have more than one closed outer boundary loop in the XT Data?

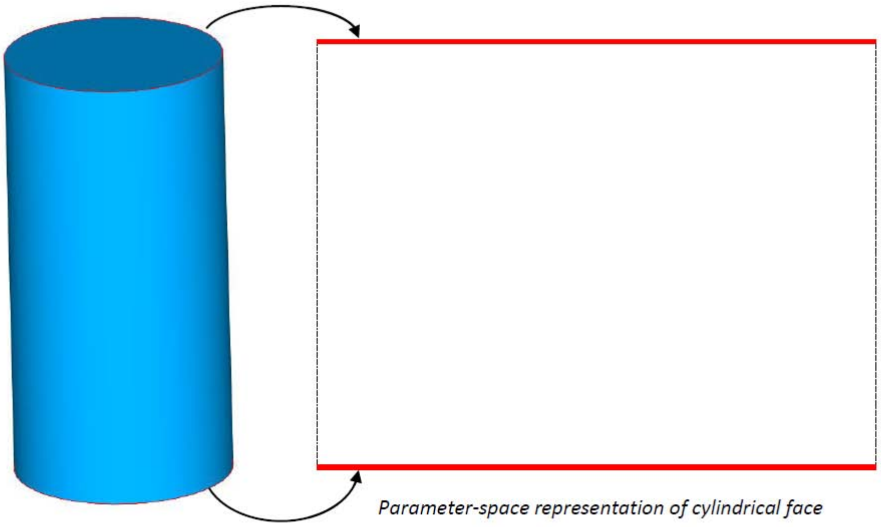
No, this is not permitted. A face may have at most one closed outer boundary loop. Some other 3D data model representations do support the concept of multiple outer boundaries, but it is not permitted in XT.

For example, in the model pictured below, the red faces could only be represented as two faces in the XT data, whereas in some other systems (for example Pro/ENGINEER), this could be represented as a single face with two outer loops.



Additionally, note that in the XT data, the two red faces may reference the same underlying surface, or they may each reference a different surface – either is permitted.

Note that in the description of the above, the reference is to closed outer boundaries in parameter space. In XT, a periodic face (e.g. a cylindrical face) can have more than one open (in parameter space) boundary. See the cylinder example below – the cylindrical face has two open boundaries in parameter space:



Parameter-space representation of cylindrical face

Question G.5.3; Can a topological face reference portions of the underlying surface outside the domain of that surface?

In some scenarios, it may be the case that a face references a surface outside of its natural range. It can also be the case that an intersection curve references a surface outside of the natural range of the surface.

In such scenarios, XT data assumes an implicit extension of the surface to compute such regions. The implicit extension must be calculated using a linear extension of the original surface.

F.6 face – surface connectivity

Question G.6.1 XT Faces: How are faces interpreted whose surface reference is 0?

A face with a surface reference of 0 has no geometric surface associated with the topological face. This is known as a rubber face. Some of the circumstances in which this could occur are as follows:

- The model is incomplete; the creator of the model intended there to be a surface associated with the face but none is present. In such circumstances, Siemens PLM Software recommends you notify the author of the XT Data of this observation and request that they investigate the cause.
- The representation is being used to pass around a purely topological model (no geometry)
- The model is a legacy XT Data structure which for example required back-facing rubber faces to complete the data structure of a sheet model

Bibliography

[1] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. OpenGL Programming Guide : The official guide to learning OpenGL Version 2, Fifth Edition. Addison-Wesley 2005.

[2] Michael Deering. Geometry Compression, Computer Graphics, Proceedings SIGGRAPH '95. August 1995, pp. 13-20.

[3] GNU C Library. Floating-Point Conversions. Available from World Wide Web: http://www.gnu.org/software/hello/manual/libc/Floating_002dPoint-Conversions.html#Floating_002dPoint-Conversions

[4] Michael Deering, Craig Gotsman, Stefan Gumhold, Jarek Rossignac, and Gabriel Taubin. 3D Geometry Compression Course Notes for SIGGRAPH 2000, July 25, 2000.

[5] C. M. Hoffmann. Geometric and Solid Modeling: An Introduction. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.

[6] Les Piegl and Wayne Tiller, The NURBS Book, Springer-Verlag, 1997.

[7] Andrei Khodakovsky, Pierre Alliez, Mathieu Desbrun, and Peter Schröder, Near-Optimal Connectivity Encoding of 2-Manifold Polygon Meshes, Graphical Models, Vol. 64, No. 3-4, Pages: 147 - 168, 2002.

[8] Greg Roelofs, Mark Adler, Jean-loup Gailly. zLib compression library. Available from World Wide Web: http://www.zlib.net/

[9] *JT Open Program* (http://www.jtopen.com) --- A program to help members leverage the benefits of open collaboration across the extended enterprise through the adoption of the JT format, a technology that makes it possible to view and share product information throughout the product lifecycle. Membership in the JT Open Program provides access to the JT Open Toolkit library, which among other things, provides read and write access to JT data and enforces certain JT conventions to ensure data compatibility with other JT-enabled applications.

[10] *JT2Go download* (http://www.jt2go.com) --- JT2Go is the no-charge 3D JT viewer from Siemens. JT2Go puts 3D data at your fingertips by allowing anyone to download the no-charge viewer. JT2Go also allows anyone to embed 3D JT data directly into Microsoft Office documents. JT2Go offers full 3D interactivity on parts, assemblies, and even 2D drawings (CGM & TIF).

[11] *Siemens: PLM Components: Parasolid: XT Pipeline* (http://www.ugs.com/products/open/parasolid/pipeline.shtml) --- This web page provides information on the Parasolid precise boundary representation format (XT) and how this XT format fits within the Siemens vision of seamless exchange of digital product models across enterprises, between different disciplines, using their PLM applications of choice.

[12] *OpenGL Programming Guide : The official guide to learning OpenGL Version 2*, Fifth Edition, by OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis (Addison-Wesley 2005) --- This book gives in-depth explanation of the OpenGL Specification and will provide further insight into the significance of some of the data (e.g. Materials, Textures) that can exist in a JT file. Information in this book may also serve as a guide for how one could process the data contained in a JT file to produce/render an image on the screen.

[13] Michael Deering, *Geometry Compression*, Computer Graphics, Proceedings SIGGRAPH ‗95, August 1995, pp. 13-20.

[14] Michael Deering, Craig Gotsman, Stefan Gumhold, Jarek Rossignac, and Gabriel Taubin, *3D Geometry Compression*, Course Notes for SIGGRAPH 2000, July 25, 2000.

[15] *OpenGL Shading Language Specification* (http://www.opengl.org/documentation/glsl/) --- OpenGL Shading Language (GLSL) as defined by the OpenGL Architectural Review Board, the governing body of OpenGL.

[16] *Cg Toolkit Users Manual* (http://developer.nvidia.com/object/cg_users_manual.html) --- Explains everything you need to learn and use the Cg language as well as the Cg runtime library.

[17] *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics,* Randima Fernando and Mark J. Kilgard, nVIDIA Corporation, Addison Wesley Publishing Company, April 2003

[18] K. Weiler. *Topological Structures for Geometric Modeling*, PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1986.

[19] C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.

[20] Les Piegl and Wayne Tiller, *The NURBS Book*, Springer-Verlag, 1997.

[21] *Planetmath.org - Huffman Coding* (http://planetmath.org/encyclopedia/HuffmanCoding.html) --- This web page provides a technical overview of Huffman coding which is one form of data encoding used within the JT format.

[22] Michael Schindler, *Practical Huffman Coding* (http://www.compressconsult.com/huffman/#encoding) --- This web page provides some coding hints for implementing Huffman coding which is one form of data encoding used within the JT format.

[23] Glen G. Langdon Jr., *An Introduction to Arithmetic Coding*, IBM Journal of Research and Development, Volume 28, Number 2, March 1984, pp. 135-149.

[24] Paul G. Howard and Jeffrey Scott Vitter, *Practical Implementation of Arithmetic Coding. Image and Text Compression*, ed. J. A. Storer, Kluwer Academic Publishers, April 1992, pp. 85-112.

[25] zlib.net (http://www.zlib.net/) --- This web page provides (either directly or through links) complete detailed information on ZLIB compression including frequently asked questions, technical documentation, source code downloads, etc.

[26] Andrei Khodakovsky, Pierre Alliez, Mathieu Desbrun, and Peter Schröder, *Near-Optimal Connectivity Encoding of 2-Manifold Polygon Meshes*, Graphical Models, Vol. 64, No. 3-4, Pages: 147 - 168, 2002.