PUBLICLY AVAILABLE SPECIFICATION

ISO/PAS 14306

Industrial automation systems and integration — JT file format specification for 3D visualization

Systèmes d'automatisation industrielle et intégration — Spécification de format de fichier JT pour visualisation 3D

Contents

Tables

Figures

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

In other circumstances, particularly when there is an urgent market requirement for such documents, a technical committee may decide to publish other types of document:

— an ISO Publicly Available Specification (ISO/PAS) represents an agreement between technical experts in an ISO working group and is accepted for publication if it is approved by more than 50 % of the members of the parent committee casting a vote;

— an ISO Technical Specification (ISO/TS) represents an agreement between the members of a technical committee and is accepted for publication if it is approved by 2/3 of the members of the committee casting a vote.

An ISO/PAS or ISO/TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/PAS or ISO/TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn.

ISO/PAS 14306 was prepared by Technical Committee ISO/TC 184, *Automation systems and integration*, Subcommittee SC 4, *Industrial data*.

ISO/PAS 14306 is based on Siemens JT File Format Reference Version 8.1 Rev-C.

Introduction

This Publicly Available Specification was transposed by an ad hoc committee focused on industrial requirements for 3D product data visualization under the ISO/TC 184/SC 4 Harvesting Process, as defined in SC4 Standing Document (SC4N1198), Procedures for Transposing Externally Developed Specifications into ISO Deliverables.

The ad hoc committee was formed by members of ISO/TC 184/SC 4 in response to requests from the global industrial community for information on visualization formats. The group assessed several 3D visualization formats including COLLADA, JT, U3D and X3D against a list of 36 requirements. The final results concluded that these candidate formats are complementary to the ISO 10303 "STEP" series of standards concerning visualization data exchange. These formats are not intended for use for CAx data exchange or product data exchange.

The JT file format presented in this Publicly Available Specification is intended to provide data that can be used for further engineering activities in a PLM domain. The other formats were found to support product documentation (U3D) and visualisation data exchange based on XML (COLLADA, X3D) in a similar domain.

The International Organization for Standardization (ISO) draws attention to the fact that it is claimed that compliance with this document may involve the use of patent USA 20110199382.

ISO takes no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO that he is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO. Information may be obtained from:

Siemens PLM Software Product Development
345 Woodcliff Drive
Fairport
NY 14450
USA

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. ISO shall not be held responsible for identifying any or all such patent rights.

Industrial automation systems and integration — JT file format specification for 3D visualization

1 Scope

This Publicly Available Specification defines the syntax and semantics of the JT Version 8.1 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
- Wire harness information
- 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 trade off 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   References and Additional Information

[1]   *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.

[2]   *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).

[3]   *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.

[4]   *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.

[5]   Michael Deering, *Geometry Compression*, Computer Graphics, Proceedings SIGGRAPH '95, August 1995, pp. 13-20.

[6]   Michael Deering, Craig Gotsman, Stefan Gumhold, Jarek Rossignac, and Gabriel Taubin, *3D Geometry Compression*, Course Notes for SIGGRAPH 2000, July 25, 2000.

[7]   *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.

[8]   *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.

[9]   *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

[10]   K. Weiler. *Topological Structures for Geometric Modeling*, PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1986.

[11]   C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.

[12]   *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.

[13] 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.

[14] Glen G. Langdon Jr., *An Introduction to Arithmetic Coding*, IBM Journal of Research and Development, Volume 28, Number 2, March 1984, pp. 135-149.

[15] 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.

[16] 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.

3   Definitions

3.1   Terms

It is assumed that readers of this document are familiar with concepts in the area of computer graphics and solid modeling. The intention of this section is not to provide comprehensive definitions, but is to provide a short introduction and clarification of the usage of terms within this document.

Assembly	– A related collection of *model* parts, represented in a JT format logical scene graph as a logical graph branch
Attribute	– Objects associated with nodes in a *logical scene graph* and specifying one of several appearances, positioning, or rendering characteristics of a *shape*
Boundary Representation	– A solid model representation where the solid volume is specified by its surface boundary (both its geometric and topological boundaries).
CodeText	– A collection of data in encoded form.
Directed Acyclic Graph	– A *graph* is a set of nodes, and a set of edges connecting the nodes in a tree like structure. 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). 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. So 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.
JT Enabled Application	– Application which supports reading and/or writing reference compliant JT Format files.
Level of Detail	– One alternative graphical representation for some *model* component (e.g. part).
Logical Scene Graph	– A *scene graph* representing the logical organization of a *model*. 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.

Mipmap	– A reduced resolution version of a texture map. 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.
Model	– Representation, in JT format, of a physical or virtual product, part, assembly; or collections of such objects.
Parasolid XT Format	– Parasolid boundary representation format
Product and Manufacturing Information	– Collection of information created on a 3D/2D CAD Model to completely document the product with respect to design, manufacturing, inspection, etc. This may includes 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)
Property	– An object associated with a logical scene graph node and identifying arbitrary application or enterprise specific information (meta-data) related to that node
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.
Scene Graph	– In the context of the JT format, a scene graph is a *directed acyclic graph* that arranges the logical and often (but not necessarily) spatial representation of a graphical scene.
Shader	– A user-definable program, expressed directly in a target assembly language, or in high-level form to be compiled. A shader program replaces a portion of the otherwise fixed-functionality graphics pipeline with some user-defined function. At present, hardware manufacturers have made it possible to run a shader for each vertex that is processed or each pixel that is rendered.

Streaming	– In the context of the JT format, streaming refers to both:
	  o Loading from disk based medium only the portions of data that are required by the user to perform the tasks at hand. The motivation being to more efficiently manage system memory.
	  o 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. The motivation being 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
	
	The desired end result of both being to *deliver only the JT data that the user needs, where the user needs it, when the user needs it.* A "just-in-time" approach to delivering JT format product data.

Shape	– A logical scene graph leaf node containing or referencing the geometric shape definition data (e.g. vertices, polygons, normals, etc.) of a model component.

| Texture Channel | – A Texture Unit plus the *texture environment*. In OpenGL® terms, Texture Channel basically controls "glActiveTexture" [4] |

| Texture Object | – JT format meaning is the same as in OpenGL [4] "*A 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 color, and texture priority.*" |

| Texture Unit | – JT format meaning is the same as in OpenGL [4], with the connotation that *texture parameters* go with the Texture Unit (through binding of a texture object) but *texture environment* (texturing function) does not. |

3.2 Coordinate Systems

The data contained within a JT file is defined within one of the following coordinate systems.   If not otherwise specified in a data field's description, it should be assumed that the data is defined in Local Coordinate System.

- *Local Coordinate System (LCS).* The coordinate system in which shape geometry is specified. It is the coordinate system used to specify the "raw" data with no transforms applied.

- *Node Coordinate System (NCS).* Local coordinates transformed by any transforms specified as attributes at the node. The NCS is also often referred to as Model Coordinate System (MCS).

- *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).

- *View Coordinate System (VCS).* World coordinates transformed by a view matrix.

4   Acronyms and Abbreviations

Abs	Absolute Value
BBox	Bounding Box
B-Rep	Boundary Representation
CAE	Computer Aided Engineering
Cg	C for Graphics
CODEC	Coder-Decoder
GD&T	Geometric Dimensioning and Tolerancing
GLSL	OpenGL Shader Language
GPU	Graphics Processing Unit
GUID	Globally Unique Identifier
HSV	Hue, Saturation, Value
HSVA	Hue, Saturation, Value, Alpha
LCS	Local Coordinate System
LOD	Level of Detail
LsbFirst	Least Significant Byte First
LSG	Logical Scene Graph
Max	Maximum
MCS	Model Coordinate System
Min	Minimum
MsbFirst	Most Significant Byte First

N/A	Not Applicable
NCS	Node Coordinate System
PCS	Parameter Coordinate Space
PLM	Product Lifecycle Management
PMI	Product and Manufacturing Information
RGB	Red, Green, Blue
RGBA	Red, Green, Blue, Alpha
TOC	Table of Contents.
VPCS	Viewpoint Coordinate System
URL	Uniform Resource Locator
WCS	World Coordinate System

5 Notational Conventions

5.1 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:
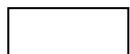
Rectangles represent a data field of one of the standard data types.

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.
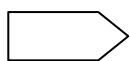
Rectangles with the right side corners clipped off represent information that has been compressed.

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 5.2 Data Types. In the example below the Data_Type is "I32" (a signed 32 bit integer) and Field_Name is "Count."

I32 : Count

- 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."

Graph Elements

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

    1. 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"

```
U32{6} : Count
```

2. 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:

o Int32CDP – The Int32CDP (i.e. Int32 Compressed Data Packet) represents the format used to encode/compress a collection of data into a series of Int32 based symbols. A complete description for Int32 Compressed Data Packet can be found in 7.1.1 Int32 Compressed Data Packet.

o Float64CDP – The Float64CDP (i.e. Float64 Compressed Data Packet) represents the format used to encode/compress a collection of data into a series of Float64 based symbols. A complete description for Float64 Compressed Data Packet can be found in 7.1.2 Float64 Compressed Data Packet.

The Int32 Compressed Data Packet type is used for compressing/encoding both "integer" and "float" (through quantization) data. While the Float64 Compressed Data Packet type is used for compressing/encoding "double" 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."

```
VecU32{Int32CDP, Lag1} : First Shell
```

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 or Float64CDP 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:Decoding Algorithms – An Implementation):

Predictor Type	Description
Lag1	Predicts as last value
Lag2	Predicts as value before last
Stride1	Predicts using stride from last two values
Stride2	Predicts using stride from values 2 and 4 back

StripIndex	This is a completely empirical predictor. Looks at the values two back and four back in the stream, and uses the stride between these two values to predict the current value if and only if the stride lays between -8 and 8 *noninclusive*, else it predicts the value as the one two back plus two. In pseudo-code form the predicted values is computed as follows:
	
	if(val2back - val4back < 8 && val2back - val4back > -8)
	    iPredicted = val2back + (val2back - val4back);
	else
	    iPredicted = val2back + 2;

Predictor Type	Description
Ramp	Predict value "i" as values "i's" index
Xor1	Predict as last, but use XOR instead of subtract to compute residual
Xor2	Predict as value before last, but use XOR instead of subtract to compute residual
NULL	No prediction applied

3. "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 5.2 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
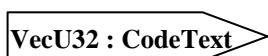
Note that for some JT file Segment Types there is ZLIB compression also applied to all bytes of element data stored in the segment. This ZLIB compression applied to all the segment's data is not indicated in the diagrams through the use of "rectangle box with clipped right side corners". Instead, one must examine information stored with the first Element in the file segment to determine if ZLIB compression is applied to all data in the segment. A complete description of the JT format data compression and encoding can be found in 6.1.3 Data Segment and 7 Data Compression.

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. Any undocumented bits are reserved.

- 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.
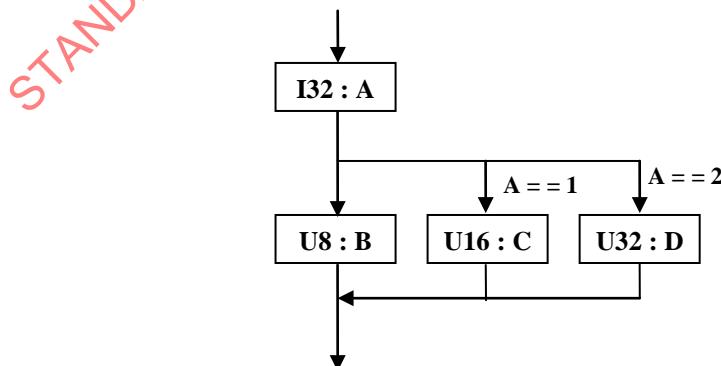
In cases where the same data type repeats, a loop construct is used where the number of iterations appears next to the loop line. There are two forms of this loop construct. The first form is used when the number of iterations is not controlled by some previous read count value. Instead the number of 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:
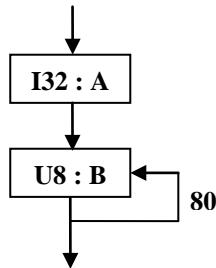
The second form of this loop construct is used when the number of iterations is based on data (e.g. count) previously read from the file. In this case it is valid for there to be zero data iterations (zero count). This second from of the loop construct looks as follows (data field D is repeated C value times).
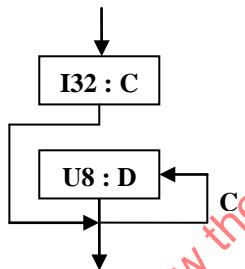
5.2 Data Types

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

Table 1: Basic Data Types lists the basic/standard data types which can occur in JT file.
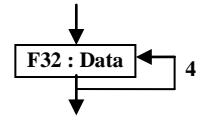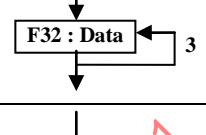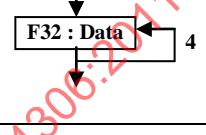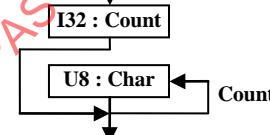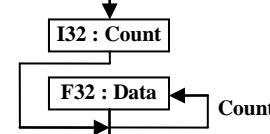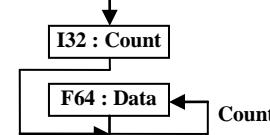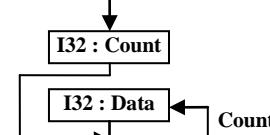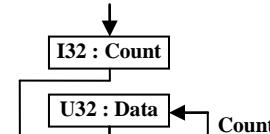
Table 1: Basic Data Types

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.
I16	A signed two's complement 16-bit integer value.
I32	A signed two's complement 32-bit integer value.
F32	An IEEE 32-bit floating point number.
F64	An IEEE 64-bit double precision floating point number

Table 2: Composite Data Types lists some composite data types which are used to represent some frequently occurring groupings of the basic data types (e.g. Vector, RGBA color). The composite data types are defined in this reference simply for convenience/brevity in describing the JT file contents.

Table 2: 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 : Min Corner CoordF32 : Max Corner
CoordF32	The CoordF32 type defines X, Y, Z coordinate values. So a CoordF32 is made up of three F32 base types.	F32 : Data 3
CoordF64	The CoordF64 type defines X, Y, Z coordinate values. So a CoordF64 is made up of three F64 base types.	F64 : Data 3
DirF32	The DirF32 type defines X, Y, Z components of a direction vector. So a DirF32 is made up of three F32 base types.	F32 : Data 3
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.)	U32 U16 2 U8 8
HCoordF32	The HCoordF32 type defines X, Y, Z, W homogeneous coordinate values. So an HCoordF32 is made up of four F32 base types.	F32 : Data 4
HCoordF64	The HCoordF64 type defines X, Y, Z, W homogeneous coordinate values. So an HCoordF64 is made up of four F64 base types	F64 : Data 4
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).	I32 : Count U16 : Char Count
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.	F32 : Data 16
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.	F32 : Data 4
Quaternion	The Quaternion type defines a 3-dimensional orientation (no	

Type	Description	Symbolic Diagram
	translation) in quaternion linear combination form ($a + bi + cj + dk$) 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.	F32 : Data — 4
RGB	The RGB type defines a color 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 color values typically range from 0.0 to 1.0.	F32 : Data — 3
RGBA	The RGBA type defines a color 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 color 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.	F32 : Data — 4
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).	I32 : Count / U8 : Char — Count
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.	I32 : Count / F32 : Data — Count
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.	I32 : Count / F64 : Data — Count
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.	I32 : Count / I32 : Data — Count
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.	I32 : Count / U32 : Data — Count

6 File Format

All objects represented in the JT format are assigned an "object identifier" (e.g. see 6.2.1.1.1.1.1 Base Node Data, or 6.2.1.1.2.1.1 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).

6.1   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 must always exist to have a reference compliant JT file is the 6.2.1 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. Although there are no JT format compliance rules about where the TOC Segment must be located within the file, in practice the TOC Segment is typically located either immediately following the File header (as shown in the below Figure) or at the very end of the file following all other Data Segments.

Figure 1: File Structure

```
        ┌──────────────────┐
        │   File Header     │
        └──────────────────┘
                 │
                 ▼
        ┌──────────────────┐
        │   TOC Segment     │
        └──────────────────┘
                 │
                 ▼
        ┌──────────────────┐◄──┐
        │   Data Segment    │   │
        └──────────────────┘───┘
                 │
                 ▼
```
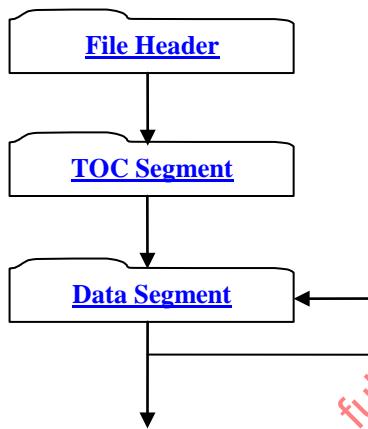
6.1.1   File Header

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 2: File Header data collection

```
        ┌──────────────────┐◄──┐
        │  UChar : Version  │   │  80
        └──────────────────┘───┘
                 │
                 ▼
        ┌────────────────────┐
        │ UChar : Byte Order  │
        └────────────────────┘
                 │
                 ▼
        ┌────────────────────┐
        │ I32 : File Attributes│
        └────────────────────┘
                 │
                 ▼
        ┌────────────────────┐
        │  I32 : TOC Offset   │
        └────────────────────┘
                 │
                 ▼
        ┌─────────────────────────┐
        │ GUID : LSG Segment ID    │
        └─────────────────────────┘
```
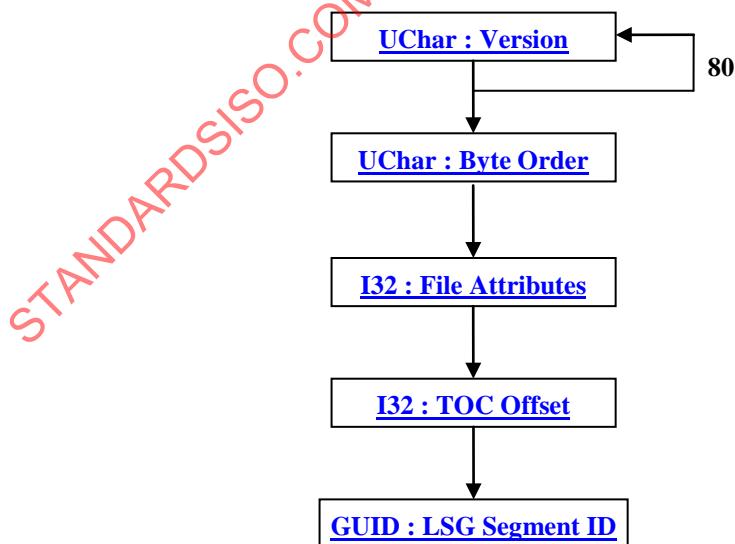
UChar : Version

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 unspecified information. The version string is padded with spaces to a length of 75 ASCII characters and then the final five characters must 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.

So for a JT Version 8.1 file this string will look as follows:

"Version 8.1 JT   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 : File Attributes
All bits in this field are reserved

I32 : 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 must be accessible from the root LSG Data Segment.

6.1.2 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 3: TOC Segment data collection

```
┌─────────────────────┐
│  I32 : Entry Count   │
└─────────────────────┘
        ┌──────────────────────┐
        │      TOC Entry        │ ◄──┐
        └──────────────────────┘    │ Entry Count
```
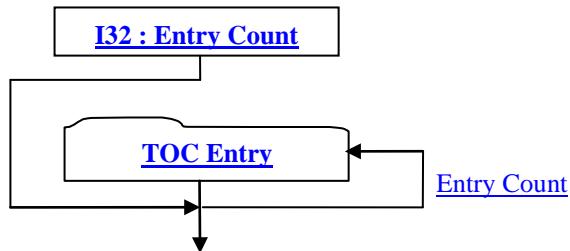
I32 : Entry Count

Entry Count is the number of entries in the TOC.

6.1.2.1 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 4: TOC Entry data collection

```
┌─────────────────────┐
│  GUID : Segment ID   │
└─────────────────────┘
          │
┌─────────────────────┐
│ I32 : Segment Offset │
└─────────────────────┘
          │
┌─────────────────────┐
│ I32 : Segment Length │
└─────────────────────┘
          │
┌──────────────────────────┐
│ U32 : Segment Attributes  │
└──────────────────────────┘
```
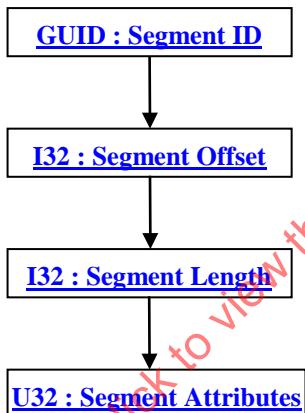
GUID : Segment ID

Segment ID is the globally unique identifier for the segment.

I32 : Segment Offset

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

I32 : 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.

Bits 0 - 23	Reserved for future use.
Bits 24 - 31	Segment type. Complete list of Segment types can be found in Table 3: Segment Types.

6.1.3 Data Segment

All data stored in a JT file must 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 a ZLIB compression conditionally applied to all the Data bytes of information persisted within the segment. Whether ZLIB compression is conditionally applied to a segment's Data bytes of information is indicated by information stored with the first "Element" in the segment. Also Table 3: Segment Types has a column indicating whether the Segment Type may have ZLIB compression applied to its Data bytes.

All Data Segments have the same basic structure.

Figure 5: Data Segment data collection
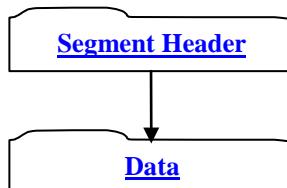
6.1.3.1 Segment Header

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

Figure 6: Segment Header data collection
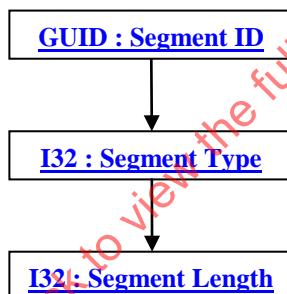
GUID : Segment ID

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; "2" denotes contents of a B-Rep, etc.

The complete list of segment types is as follows:

Table 3: Segment Types

Type	Data Contents	ZLIB Compression Conditionally Applied to all of the Segment's Element Data
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

Type	Data Contents	ZLIB Compression Conditionally Applied to all of the Segment's Element Data
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 Representation	Yes

Note: 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: 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:

1. Not a descendant of an LOD node.
2. Is referenced by (i.e. is a child of) more than one LOD node.
3. Shape has its own built-in LODs
4. 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.

6.1.3.2 Data

The interpretation of the Data section depends on the Segment Type. See 6.2 Data Segments for complete description for all Data Segment 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. ZLIB compression) may be conditionally applied to all bytes of information stored for all Elements within a particular Segment. Not all Segment types support ZLIB compression on all Segment data as indicated in Table 3: Segment Types. If a particular file Segment is of the type which supports ZLIB compression on all the Segment data, whether this compression is applied or not is indicated by data values stored in the Element Header ZLIB data collection of the first Element within the Segment. An in-depth description of JT file compression/encoding techniques can be found in 7 Data Compression.

Figure 7: Data data collection

For Segment Types that do **NOT** support ZLIB compression on all Segment Data. (see Table 3: Segment Types.)
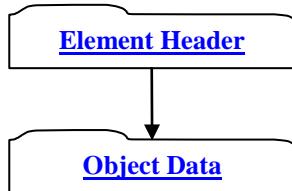
For Segment Types that support ZLIB compression on all Segment Data (see Table 3: Segment Types.)

Element Header → Object Data

Element Header ZLIB → Object Data
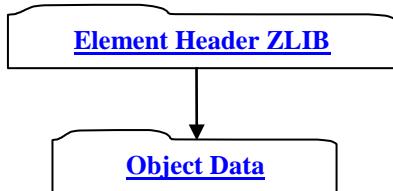
6.1.3.2.1 Element Header

Element Header contains data defining the length in bytes of the Element along with information describing the object type contained in the Element.

Figure 8: Element Header data collection

I32 : Element Length

↓

GUID : Object Type ID

↓

UChar : Object Base Type
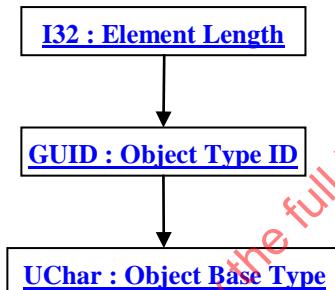
I32 : Element Length

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

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: Object Type Identifiers.

UChar : Object Base Type

Object Base Type identifies the base object type. This is useful when an unknown element type is encountered and thus the best the loader can do is to read the known Object Base Type data bytes (base type object data is always written first) and then skip (read pass) the bytes of unknown data using knowledge of number of bytes encompassing the Object Base Type data and the unknown types Length field. If the Object Base Type is unknown then the loader should simply skip (read pass) Element Length number of bytes.

Valid Object Base Types include the following:

Table 4: Object Base Types

Object Base Type	Description	Object Base Type's Data Format
255	Unknown Graph Node Object	none
0	Base Graph Node Object	6.2.1.1.1.1.1 Base Node Data
1	Group Graph Node Object	6.2.1.1.1.3.1Group Node Data
2	Shape Graph Node Object	6.2.1.1.1.10.1.1 Base Shape Data
3	Base Attribute Object	6.2.1.1.2.1.1 Base Attribute Data
4	Shape LOD	none
5	Base Property Object	6.2.1.2.1.1 Base Property Atom Data

Object Base Type	Description	Object Base Type's Data Format
6	JT Object Reference Object	6.2.1.2.5 JT Object Reference Property Atom Element without the Element Header ZLIB data collection.
8	JT Late Loaded Property Object	6.2.1.2.7 Late Loaded Property Atom Element without the Element Header ZLIB data collection.
9	JtBase (none)	none

6.1.3.2.2 Element Header ZLIB

Element Header ZLIB data collection is the format of Element Header data used by all Elements within Segment Types that support ZLIB compression on all data in the Segment. See Table 3: Segment Types for information on whether a particular Segment Type supports ZLIB compression on all data in the Segment.

Figure 9: Element Header ZLIB data collection
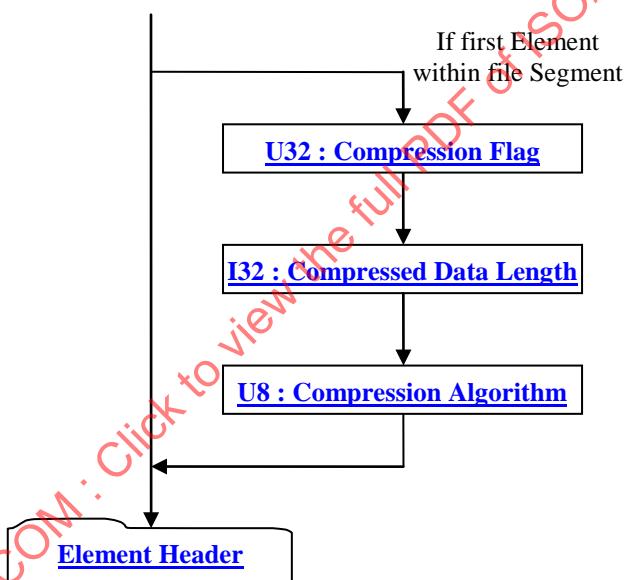
Complete description for Element Header can be found in 6.1.3.2.1Element Header. Note that if Compression Flag indicates that ZLIB compression is ON for all element data in the Segment, then the Element Header data collection is also compressed accordingly.

U32 : Compression Flag

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

= 2	– ZLIB compression is ON
!= 2	– ZLIB compression is OFF.

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:

= 1	– No compression
= 2	– ZLIB compression

6.1.3.2.3 Object Data

The interpretation of the Object Data section depends upon the Object Type ID stored in the Element Header (see 6.1.3.2.1 Element Header).

6.2   Data Segments

6.2.1   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).   It is the responsibility of the loader to insure that the acyclic property of the resulting LSG is maintained.

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

Figure 10: LSG Segment data collection
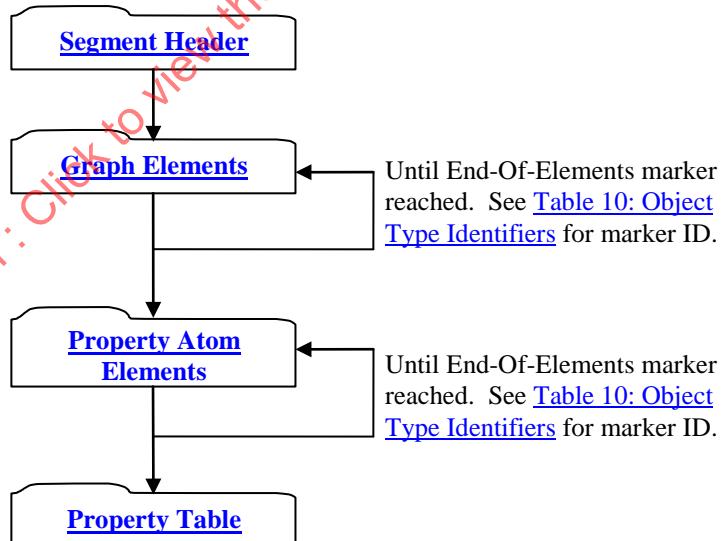
Complete description for Segment Header can be found in 6.1.3.1Segment Header.

6.2.1.1 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. color), 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 behavior. The following sub-sections describe each of the Node and Attribute Element types.

6.2.1.1.1 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.

6.2.1.1.1.1 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 behavior nor can it contain any children nodes.

Figure 11: Base Node Element data collection
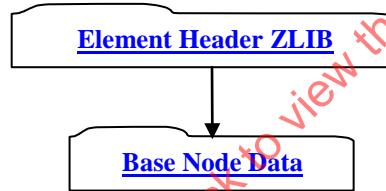
Complete description for Element Header ZLIB can be found in <u>6.1.3.2.2 Element Header ZLIB</u>.

6.2.1.1.1.1.1 Base Node Data

Figure 12: Base Node Data data collection
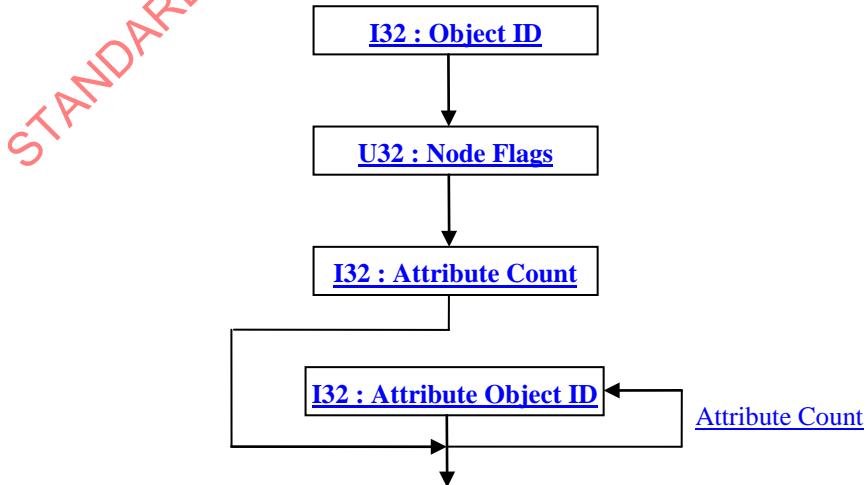
I32 : Object ID

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

U32 : Node Flags

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 undocumented bits are reserved.

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.

6.2.1.1.1.2 Partition Node Element

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

A 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). When the referenced JT file is opened, the Partition Node's children are really the children of the LSG root node for the underlying JT file. Usage of Partition Nodes in LSG also aids in supporting JT file loader/reader "best practice" of late loading data (i.e. can delay opening and loading the externally referenced JT file until the data is needed).

Figure 13: Partition Node Element data collection
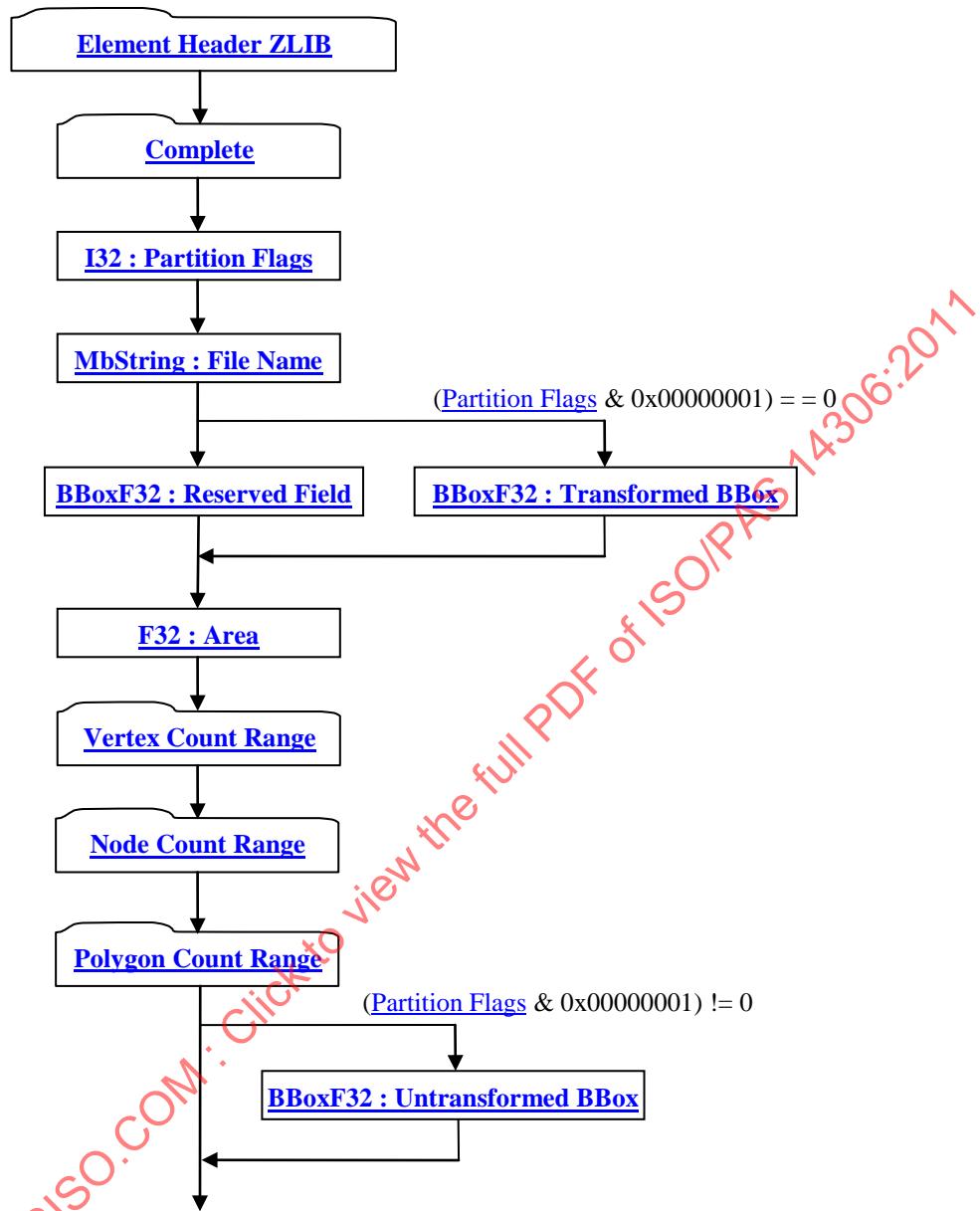
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Group Node Data can be found in 6.2.1.1.1.3.1Group Node Data.

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 undocumented bits are reserved.

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 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion

BBoxF32 : Transformed BBox

The Transformed BBox is an NCS 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 store in NCS coordinate space (i.e. values scaled by NCS 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).

6.2.1.1.1.2.1 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 14: Vertex Count Range data collection
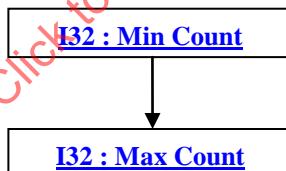
I32 : Min Count

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.

6.2.1.1.1.2.2 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 6.2.1.1.1.2.1Vertex Count Range.

6.2.1.1.1.2.3 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 6.2.1.1.1.2.1Vertex Count Range.

6.2.1.1.1.3 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 15: Group Node Element data collection
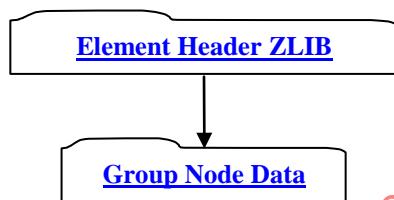
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

6.2.1.1.1.3.1 Group Node Data

Figure 16: Group Node Data data collection

Complete description for Base Node Data can be found in 6.2.1.1.1.1.1Base Node Data.

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.

6.2.1.1.1.4 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.

For example, a Group Node could use Instance Nodes to instance the same Shape Node several times, applying different material properties and matrix transformations to each instance. Note that this could also be done by using Group Nodes instead of Instance Nodes, but Instance Nodes require fewer resources.

Figure 17: Instance Node Element data collection
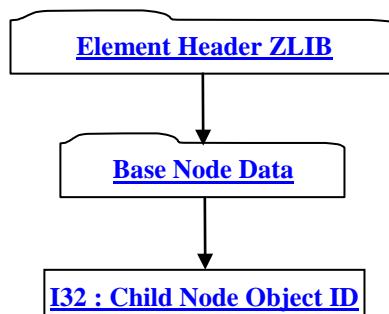
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Node Data can be found in 6.2.1.1.1.1Base Node Data.

I32 : Child Node Object ID
Child Node Object ID is the identifier for the instanced Node Object.

6.2.1.1.1.5 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 18: Part Node Element data collection
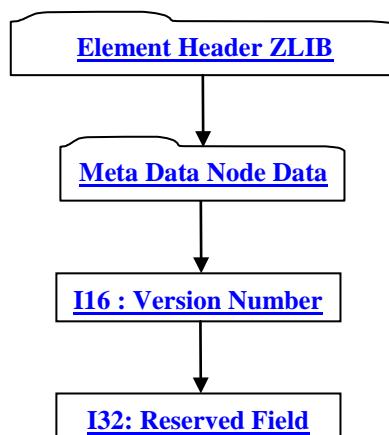
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Meta Data Node Data can be found in 6.2.1.1.1.6.1Meta Data Node Data.

I16 : Version Number
Version Number is the version identifier for this node. Version number "0x0001" is currently the only valid value for Part nodes.

I32: Reserved Field
Reserved Field is a data field reserved for future JT format expansion

6.2.1.1.1.6 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 6.2.6 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 19: Meta Data Node Element data collection
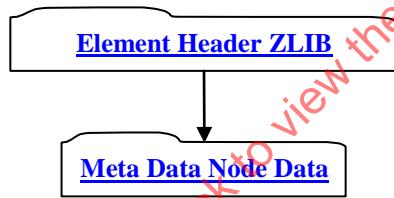
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

6.2.1.1.1.6.1 Meta Data Node Data

Figure 20: Meta Data Node Data data collection
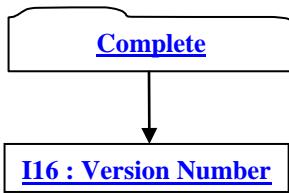
Complete description for Group Node Data can be found in 6.2.1.1.1.3.1Group Node Data.

I16 : Version Number
Version Number is the version identifier for this data. Version number "0x0001" is currently the only valid value.

6.2.1.1.1.7 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 21: LOD Node Element data collection
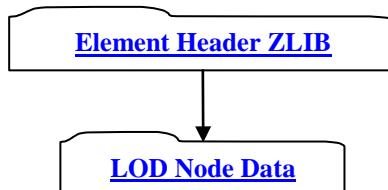
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

6.2.1.1.1.7.1 LOD Node Data

Figure 22: LOD Node Data data collection
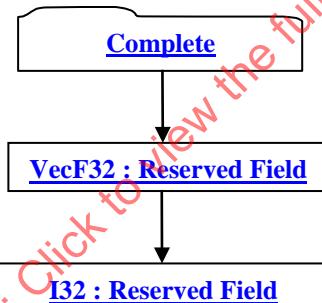
Complete description for Group Node Data can be found in 6.2.1.1.1.3.1Group Node Data.

VecF32 : Reserved Field

Reserved Field is a vector data field reserved for future JT format expansion.

I32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

6.2.1.1.1.8 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 center 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 23: Range LOD Node Element data collection
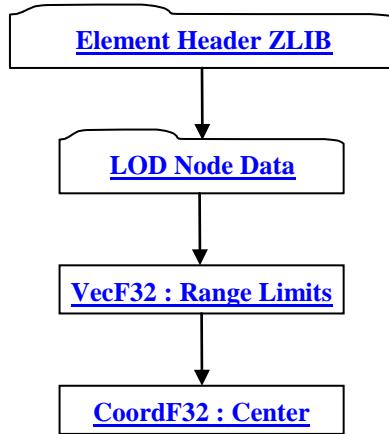
Complete description for Element Header ZLIB can be found in <u>6.1.3.2.2 Element Header ZLIB</u>.

Complete description for LOD Node Data can be found in <u>6.2.1.1.1.7.1 LOD Node Data</u>.

VecF32 : Range Limits

Range Limits indicate the WCS distance between a specified center 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 center 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 : Center

Center specifies the X,Y,Z coordinates for the NCS center point upon which alternative representation selection eye distance computations are based. Typically this location is the center 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

6.2.1.1.1.9 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 24: Switch Node Element data collection
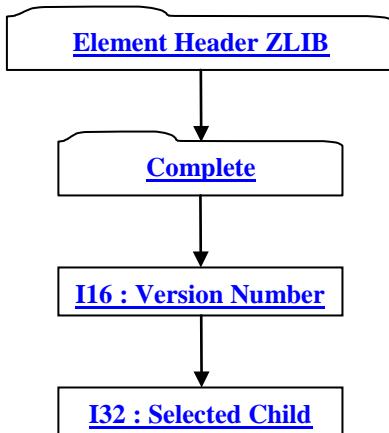
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Group Node Data can be found in 6.2.1.1.1.3.1 Group Node Data.

I16 : Version Number

Version Number is the version identifier for this node. Version number "0x0001" is currently the only valid value for Switch nodes.

I32 : 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 6.2.1.1.1.3.1 Group Node Data.

6.2.1.1.1.10 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 6.2.1.2.7 Late Loaded Property Atom Element and 6.2.2 Shape LOD Segment 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.

6.2.1.1.1.10.1 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 25: Base Shape Node Element data collection
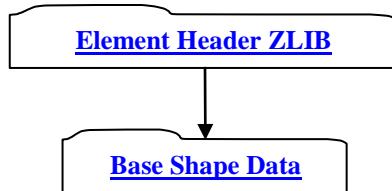
```
    ┌─────────────────────────────┐
    │   Element Header ZLIB       │
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │      Base Shape Data        │
    └─────────────────────────────┘
```

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

6.2.1.1.1.10.1.1   **Base Shape Data**

Figure 26: Base Shape Data data collection

```
    ┌─────────────────────────────┐
    │      Base Node Data         │
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │  BBoxF32 : Transformed BBox │
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │ BBoxF32 : Untransformed BBox│
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │        F32 : Area           │
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │    Vertex Count Range       │
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │     Node Count Range        │
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │    Polygon Count Range      │
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │        I32 : Size           │
    └─────────────────────────────┘
                  │
                  ▼
    ┌─────────────────────────────┐
    │   F32 : Compression Level   │
    └─────────────────────────────┘
```
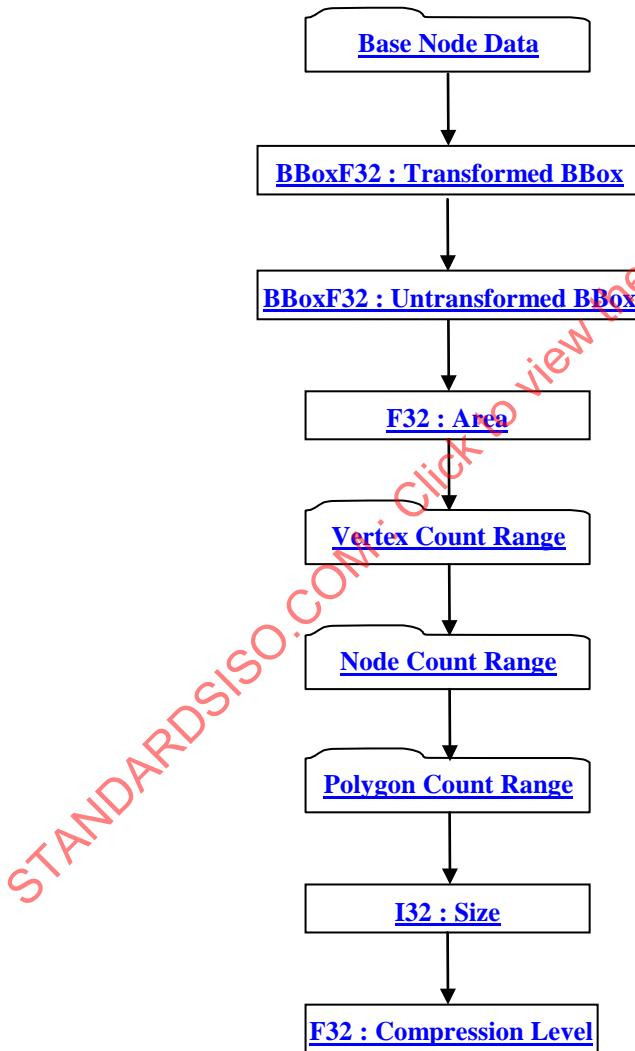
Complete description for Base Node Data can be found in 6.2.1.1.1.1.1Base Node Data

BBoxF32 : Transformed BBox
The Transformed BBox is an axis-aligned NCS bounding box and represents the transformed geometry extents for all geometry contained in the Shape Node.

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 NCS coordinate space (i.e. values scaled by NCS scaling).

I32 : 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 6.2.2.1Shape 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 6.2.2.1Shape 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. 6.2.1.1.10.2.1.1Quantization Parameters), and thus its not necessary to understand how to map this qualitative value to absolute compression values in order to uncompress/decode the data

= 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.

6.2.1.1.10.1.1.1 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 27: Vertex Count Range data collection

I32 : Min Count
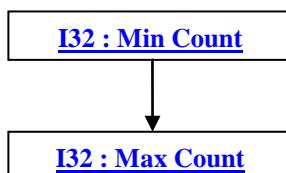
I32 : Max Count

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.

6.2.1.1.1.10.1.1.2   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 6.2.1.1.1.10.1.1.1Vertex Count Range.

6.2.1.1.1.10.1.1.3   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 6.2.1.1.1.10.1.1.1Vertex Count Range.

6.2.1.1.1.10.2   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 28: Vertex Shape Node Element data collection
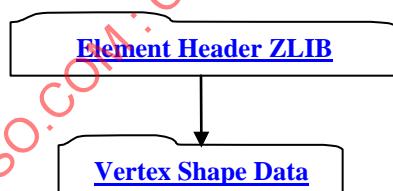
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

6.2.1.1.1.10.2.1 **Vertex Shape Data**

Figure 29: Vertex Shape Data data collection
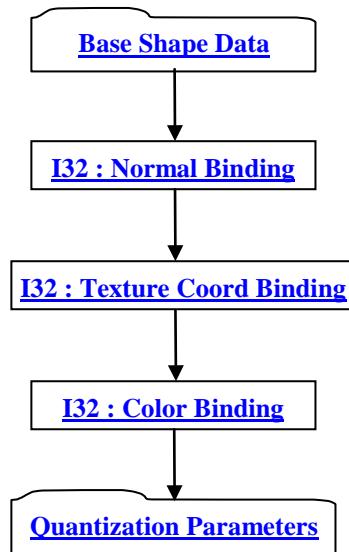
Complete description for Base Shape Data can be found in 6.2.1.1.10.1.1 Base Shape Data.

I32 : Normal Binding

Normal Binding specifies how (at what granularity) normal vector data is supplied ("bound") for the shape in the associated/referenced Shape LOD Element. See 6.2.2.1Shape LOD Element for complete description of Shape LOD Elements.

= 0	– None. Shape has no normal data.
= 1	– Per Vertex. Shape has a normal vector for every vertex.
= 2	– Per Facet. Shape has a normal vector for every face/polygon.
= 3	– Per Primitive. Shape has a normal vector for each shape primitive (e.g. a normal for each tri-strip in a tri-strip set).

I32 : Texture Coord Binding

Texture Coord Binding specifies how (at what granularity) texture coordinate data is supplied ("bound") for the shape in the associated/referenced Shape LOD Element. Valid values are the same as documented for I32 : Normal Binding data field.

I32 : Color Binding

Color Binding specifies how (at what granularity) color data is supplied ("bound") for the shape in the associated/referenced Shape LOD Element. Valid values are the same as documented for I32 : Normal Binding data field.

6.2.1.1.1.10.2.1.1 Quantization Parameters

Quantization Parameters specifies for each shape data type grouping (i.e. Vertex, Normal, Texture Coordinates, Color) 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 6.2.2.1Shape LOD Element for complete description of Shape LOD Elements.

Figure 30: Quantization Parameters data collection
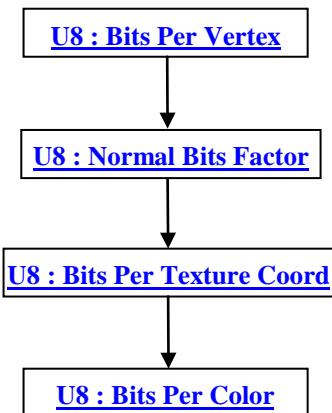
```
┌─────────────────────────┐
│  U8 : Bits Per Vertex   │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│ U8 : Normal Bits Factor │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│ U8 : Bits Per Texture Coord │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│   U8 : Bits Per Color   │
└─────────────────────────┘
```

U8 : Bits Per Vertex

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

U8 : Bits Per Color

Bits Per Color specifies the number of quantization bits per color component. Value must be within range [0:24] inclusive.

6.2.1.1.1.10.3 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 31: Tri-Strip Set Shape Node Element data collection
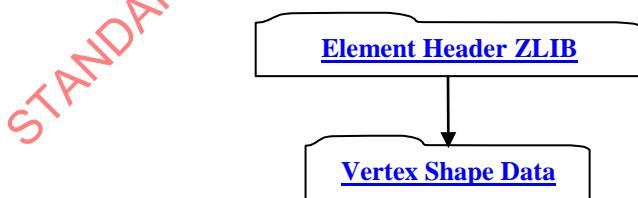
```
┌─────────────────────────┐
│  Element Header ZLIB    │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│   Vertex Shape Data     │
└─────────────────────────┘
```

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Vertex Shape Data can be found in 6.2.1.1.1.10.2.1Vertex Shape Data.

6.2.1.1.1.10.4 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 32: Polyline Set Shape Node Element data collection**
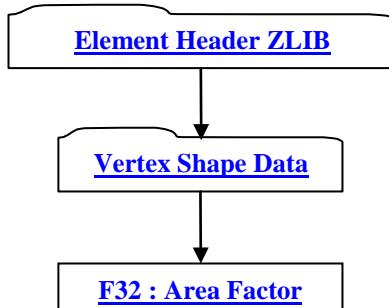
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Vertex Shape Data can be found in 6.2.1.1.1.10.2.1Vertex Shape Data.

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].

6.2.1.1.1.10.5 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 33: Point Set Shape Node Element data collection
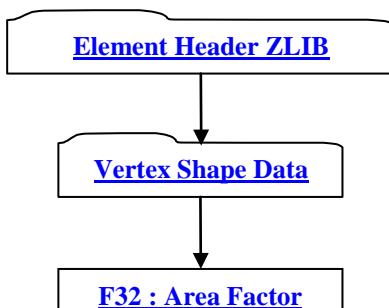
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Vertex Shape Data can be found in 6.2.1.1.1.10.2.1Vertex Shape Data.

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..

6.2.1.1.1.10.6 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 34: Polygon Set Shape Node Element data collection
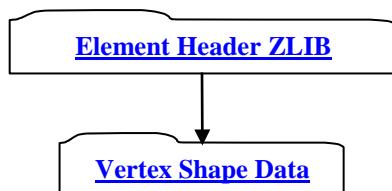
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Vertex Shape Data can be found in 6.2.1.1.1.10.2.1Vertex Shape Data.

6.2.1.1.1.10.7 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 35: NULL Shape Node Element data collection
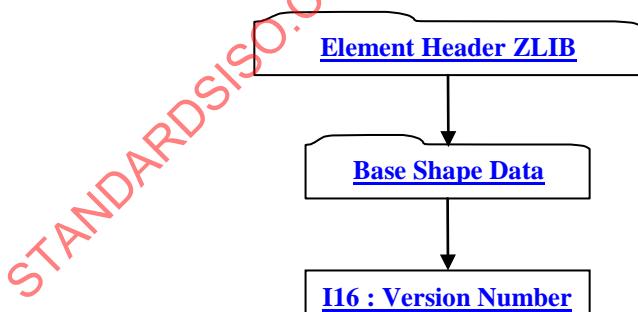
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Shape Data can be found in 6.2.1.1.1.10.1.1 Base Shape Data.

I16 : Version Number

Version Number is the version identifier for this node. Version number "0x0001" is currently the only valid value for NULL Shape Node Element.

6.2.1.1.1.10.8 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.) who's 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 center 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 Elements 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 6.2.1.2.7 Late Loaded Property Atom Element and 6.2.2.2 Primitive Set Shape Element respectively.

Figure 36: Primitive Set Shape Node Element data collection
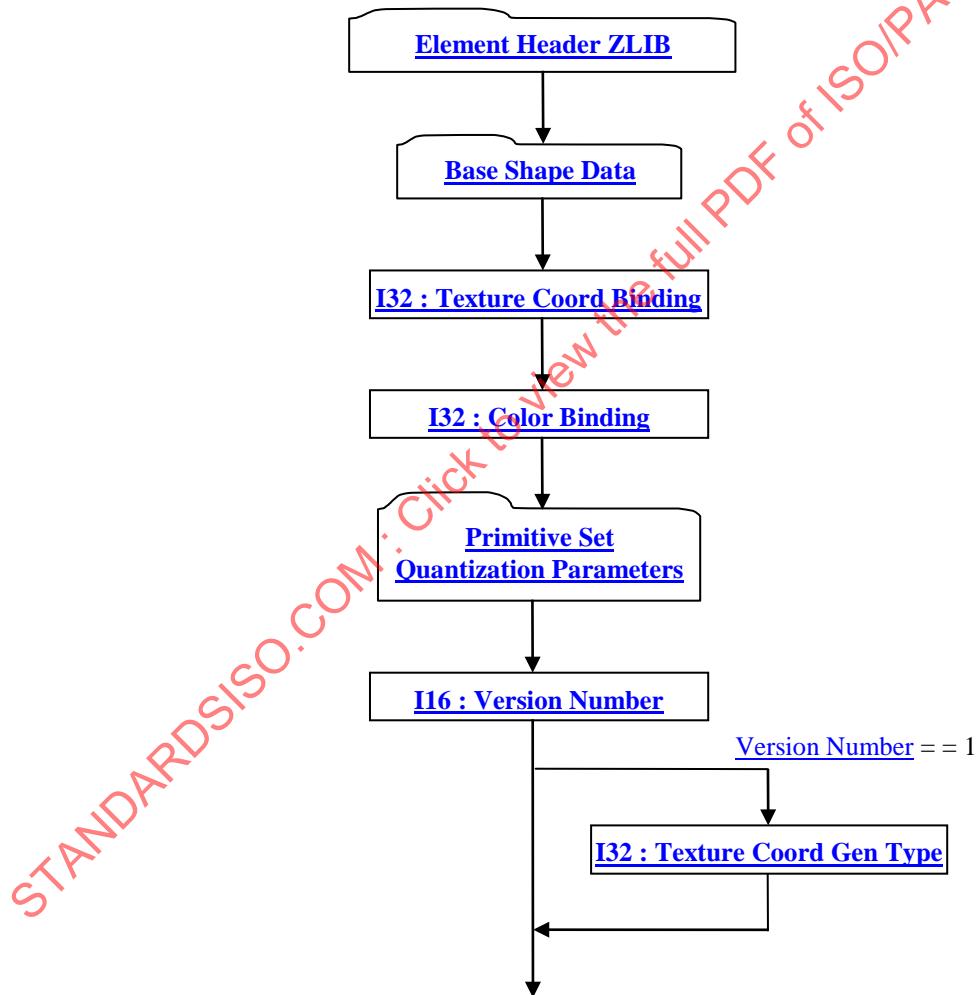
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Shape Data can be found in 6.2.1.1.1.10.1.1 Base Shape Data.

I32 : Texture Coord Binding

Texture Coord Binding specifies how (at what granularity) texture coordinate data is supplied ("bound") for the shape in the associated/referenced Shape LOD Element. Valid values are as follows:

| = 0 | – None. Shape has no texture coordinate data. |
| = 1 | – Per Vertex. Shape has texture coordinates for every vertex. |

I32 : Color Binding

Color Binding specifies how (at what granularity) color data is supplied ("bound") for the shape in the associated/referenced Shape LOD Element. Valid values are the same as documented for Texture Coord Binding data field.

I16 : 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.

| = 0 | – Version 0 Format |
| = 1 | – Version 1 Format |

I32 : Texture Coord Gen Type

Texture Coord Gen Type specifies how texture coordinates are to be generated.

| = 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. |

6.2.1.1.1.10.8.1 Primitive Set Quantization Parameters

Primitive Set Quantization Parameters specifies for the two shape data type grouping (i.e. Vertex, Color) 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 6.2.2.1Shape LOD Element for complete description of Shape LOD Elements.

Figure 37: Primitive Set Quantization Parameters data collection
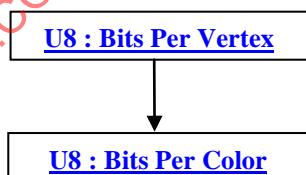
U8 : Bits Per Vertex

U8 : Bits Per Color

U8 : Bits Per Vertex

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

U8 : Bits Per Color

Bits Per Color specifies the number of quantization bits per color component. Value must be within range [0:24] inclusive.

6.2.1.1.1.10.9 Wire Harness Set Shape Node Element

Object Type ID: 0x4cc7a521, 0x728, 0x11d3, 0x9d, 0x8b, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2

A Wire Harness Set Shape Node Element represents a list of wire harness shapes. Where a wire harness is defined as a single manufactured wire unit consisting of several physical electrical wires all bound together into a branching structure of wire bundles that terminate at connectors. A Wire Harness Set Shape Node Element is meant to procedurally generate its LODs. "Procedurally generate" means that the raw geometric shape definition data (e.g. vertices, polygons, normals, etc) for LODs is not directly stored; instead some descriptive shape information is stored from which LODs can be generated (if desired) at load time.

Wire Harness Set Shape Node Elements actually do not even directly contain this description shape definition data, but instead reference (using Late Loaded Property Atoms) Wire Harness Set Shape Element within the file for the actual descriptive shape definition data. Storing the descriptive 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 Wire Harness Set Shape Element can be found in 6.2.1.2.7 Late Loaded Property Atom Element and 6.2.2.3 Wire Harness Set Shape Element respectively.

Figure 38: Wire Harness Set Shape Node Element data collection
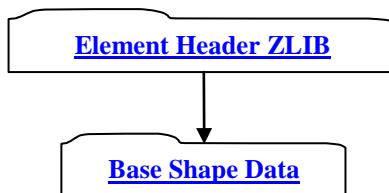
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Shape Data can be found in 6.2.1.1.1.10.1.1 Base Shape Data.

6.2.1.1.2 Attribute Elements

Attribute Elements (e.g. color, 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.

Attributes can be declared "final" (see 6.2.1.1.2.1.1Base Attribute Data), which terminates accumulation of that attribute type at that attribute and propagates the accumulated values there to all descendants of the associated node. Descendants can explicitly do a one-shot override of "final" using the attribute "force" flag (see 6.2.1.1.2.1.1Base 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." 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!

6.2.1.1.2.1 Base Attribute Element

Object Type ID: 0x10dd1001, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

Base Attribute Element represents the simplest form of an attribute that can exist within the LSG. A Base Attribute Element within a LSG has no implied appearance or positioning semantics.

Figure 39: Base Attribute Element data collection
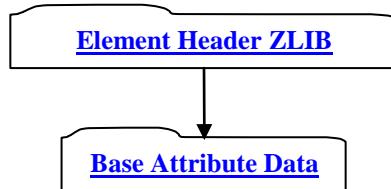
```
┌─────────────────────────┐
│  Element Header ZLIB     │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│  Base Attribute Data     │
└─────────────────────────┘
```

Complete description for Element Header ZLIB can be found in <u>6.1.3.2.2 Element Header ZLIB</u>.

6.2.1.1.2.1.1 Base Attribute Data

Figure 40: Base Attribute Data data collection
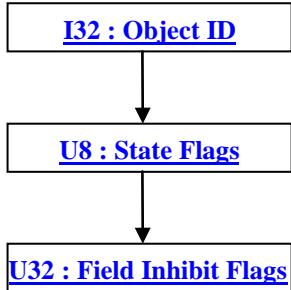
```
┌─────────────────────────┐
│  I32 : Object ID         │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│  U8 : State Flags        │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│  U32 : Field Inhibit Flags │
└─────────────────────────┘
```

I32 : Object ID
Object ID is the identifier for this Object. Other objects referencing this particular object do so using the Object ID.

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 undocumented bits are reserved.

0x01	– Accumulation Final flag. Provides a means to terminate a particular attribute type's accumulation at any node of the LSG and thereby force all descendants to have that value of the attribute. = 0 – Accumulation is to occur normally = 1 – Accumulation is "final"
0x02	– Accumulation Force flag. Provides a way to assign nodes in LSG, attributes that must 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.

U32 : Field Inhibit Flags
Field Inhibit Flags is a collection of flags. The flags are combined using the binary OR operator and store the per attribute value accumulation flag. Each value 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.

6.2.1.1.2.2   Material Attribute Element

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

Material Attribute Element defines the reflective characteristics of a material. JT format LSG traversal semantics dictate that material attributes accumulate down the LSG by replacement.

The Field Inhibit flag (see 6.2.1.1.2.1.1Base Attribute Data) bit assignments for the Material Attribute Element data fields, are as follows:

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

Figure 41: Material Attribute Element data collection
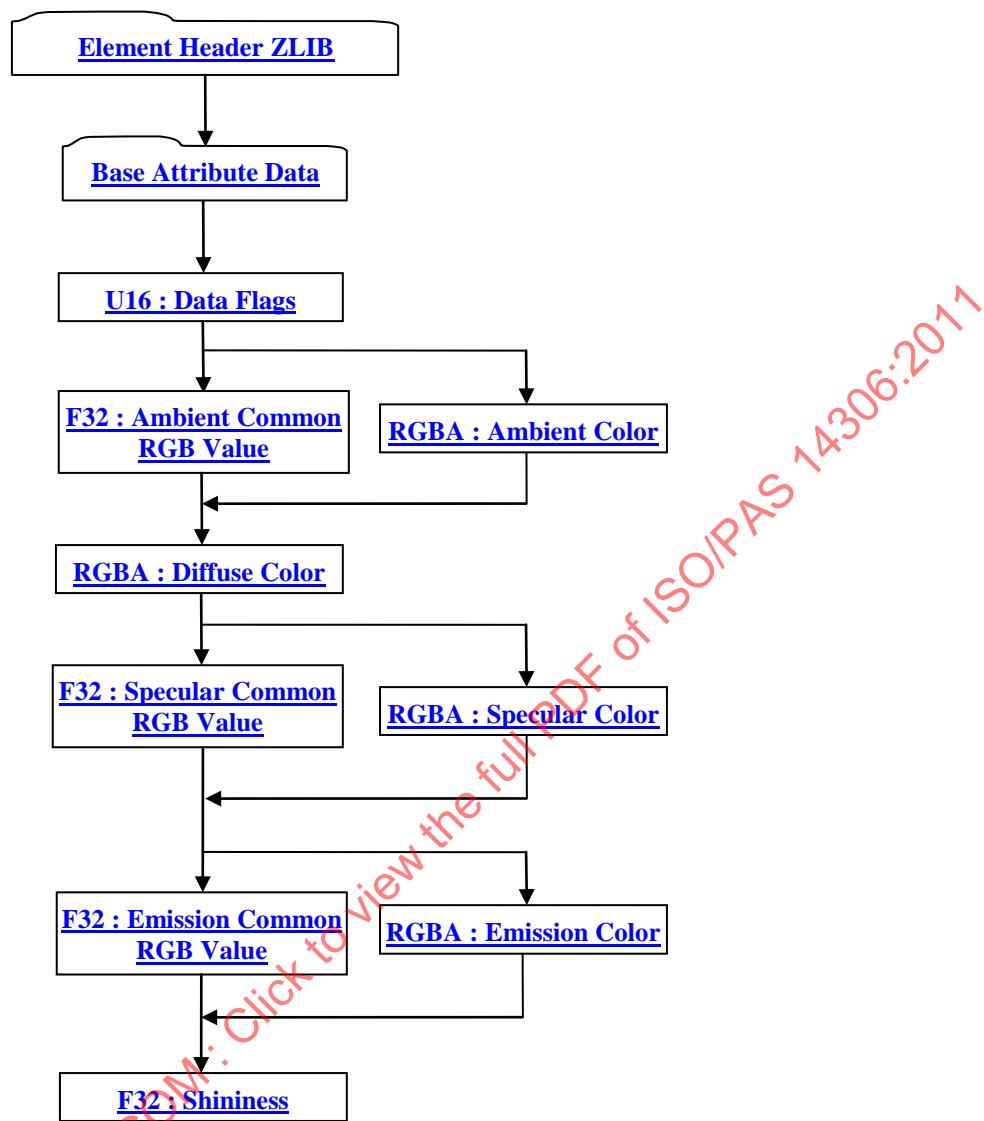
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

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 undocumented bits are reserved.

The Ambient/Emission/Specular Pattern Flags are used to optimize color data storage size to a single F32 for the common case where the color is defined as [c, c, c, 1.0] (i.e. RGB values are the same "c" value and Alpha is always "1.0").

0x0001	– Pattern flag bits are used flag (i.e. Ambient/Emission/Specular pattern flags) = 0 – Pattern bits are to be ignored. = 1 – Pattern bits are valid.

0x0002	– Ambient Pattern Flag = 0 – Ambient data stored as four F32. = 1 – Ambient data stored as one F32 and resultant color equals [c, c, c, 1.0]
0x0004	– Emission Pattern Flag = 0 – Emission data stored as four F32. = 1 – Emission data stored as one F32 and resultant color equals [c, c, c, 1.0]
0x0008	– Specular Pattern Flag = 0 – Specular data stored as four F32. = 1 – Specular data stored as one F32 and resultant color equals [c, c, c, 1.0]
0x0010	– Blending Flag. Blending is a color combining operation in the graphics pipeline that happens just before writing a color to the framebuffer. If Blending is ON then incoming fragment RGBA color values are used (based on Source Blend Factor) and existing framebuffer's RGBA color 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 color to write into the framebuffer. If Blending is OFF then incoming fragment RGBA color is written directly into framebuffer unmodified (i.e. completely overriding existing framebuffer RGBA color). Additional information on how one might leverage the Blending Flag and Blending Factors to render an image can be found in the references listed in section 2 References and Additional Information. = 0 – Blending OFF. = 1 – Blending ON
0x0020	– Override Vertex Colors Flag. If ON, then a shape's per vertex colors are to be overridden by the accumulated Material color. = 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 color values are to be used to blend with the current framebuffer's (i.e. the destination) RGBA color 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 [4] listed in section 2 References and Additional Information. = 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_COLOR** Blending Factor = 3 – Interpret same as OpenGL **GL_SRC_COLOR** Blending Factor = 4 – Interpret same as OpenGL **GL_ONE_MINUS_DST_COLOR** Blending Factor = 5 – Interpret same as OpenGL **GL_ONE_MINUS_SRC_COLOR** 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 color values are to be used to blend with the incoming fragment's (the source) RGBA color 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 [4] listed in section 2 References and Additional Information. = 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_COLOR** Blending Factor = 3 – Interpret same as OpenGL **GL_SRC_COLOR** Blending Factor = 4 – Interpret same as OpenGL **GL_ONE_MINUS_DST_COLOR** Blending Factor = 5 – Interpret same as OpenGL **GL_ONE_MINUS_SRC_COLOR** 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

F32 : Ambient Common RGB Value

Ambient Common RGB Value is the assigned value for the Red, Green, and Blue components of the ambient color (i.e. Red, Green, and Blue are all equal to this same value; R = G = B = value). Also the Alpha component is always assumed to be equal to "1.0". Ambient Common RGB Value is only present if Ambient Pattern Flag equals 1.

RGBA : Ambient Color

Ambient Color specifies the ambient red, green, blue, alpha color values of the material. Ambient Color is only present if Ambient Pattern Flag equals 0.

RGBA : Diffuse Color

Diffuse Color specifies the diffuse red, green, blue, alpha color values of the material.

F32 : Specular Common RGB Value

Specular Common RGB Value is the assigned value for the Red, Green, and Blue components of the specular color (i.e. Red, Green, and Blue are all equal to this same value; R = G = B = value). Also the Alpha component is always assumed to be equal to "1.0". Specular Common RGB Value is only present if Specular Pattern Flag equals 1.

RGBA : Specular Color

Specular Color specifies the specular red, green, blue, alpha color values of the material. Specular Color is only present if Specular Pattern Flag equals 0.

F32 : Emission Common RGB Value

Emission Common RGB Value is the assigned value for the Red, Green, and Blue components of the emissive color (i.e. Red, Green, and Blue are all equal to this same value; R = G = B = value). Also the Alpha component is always assumed to be equal to "1.0". Emission Common RGB Value is only present if Emission Pattern Flag equals 1.

RGBA : Emission Color

Emission Color specifies the emissive red, green, blue, alpha color values of the material. Emission Color is only present if Emission Pattern Flag equals 0.

F32 : Shininess

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

6.2.1.1.2.3 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 dictate that texture image attributes accumulate down the LSG by replacement.

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 section 2 References and Additional Information.

The Field Inhibit flag (see 6.2.1.1.2.1.1Base Attribute Data) bit assignments for the Texture Image Attribute Element data fields, are as follows:

Field Inhibit Flag Bit	Data Field(s) Bit Applies To
0	Texture Type, Image Texel Data, Mipmap Image Texel Data, External Storage Name, Shared Image Flag
1	Border Mode, Border Color
2	Mipmap Minification Filter, Mipmap Magnification Filter
3	S-Dimen Wrap Mode, T-Dimen Wrap Mode, R-Dimen Wrap Mode
4	Blend Type, Blend Color
5	Texture Transform
6	Tex Coord Gen Mode, Tex Coord Reference Plane, Environment Mapping Flag
8	Internal Compression Level

Figure 42: Texture Image Attribute Element data collection
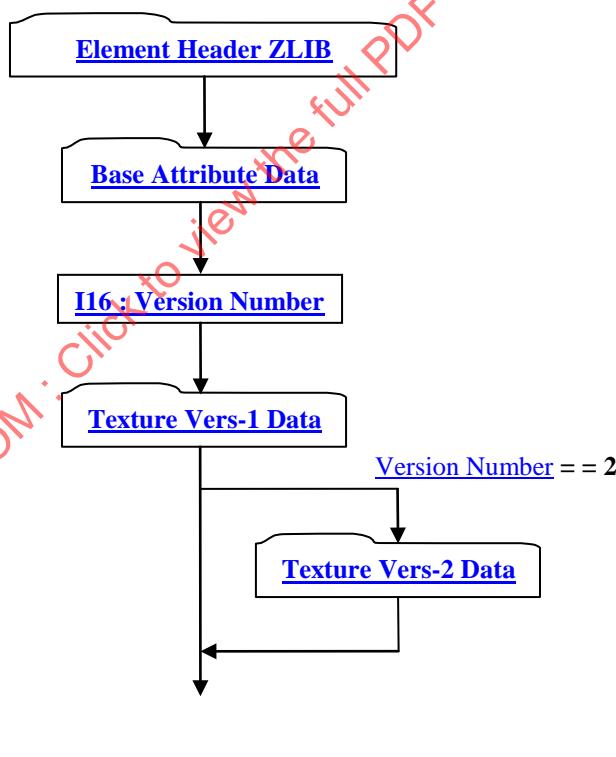
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

I16 : 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.

= 1	– Version-1 Format
= 2	– Version-2 Format

6.2.1.1.2.3.1 Texture Vers-1 Data

Texture Vers-1 Data format is stored in JT file if the Texture Image Element is a vanilla/basic texture image (i.e. if texture does not use any advanced features as described in 6.2.1.1.2.3.2 Texture Vers-2 Data).

Figure 43: Texture Vers-1 Data data collection
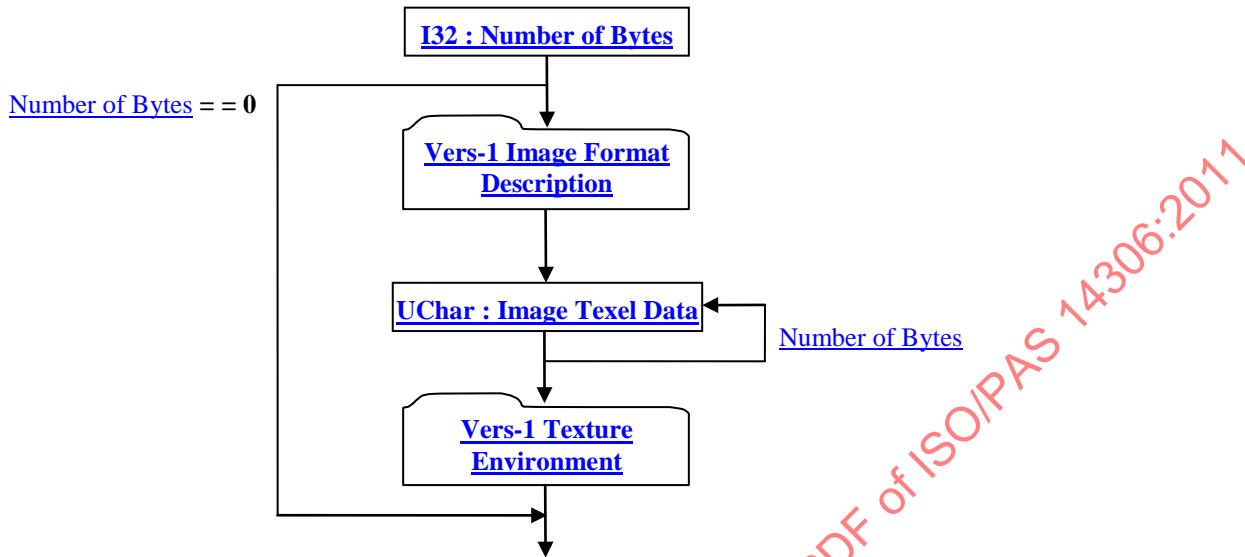
Complete details for Vers-1 Image Format Description can be found in 6.2.1.1.2.3.1.1 Vers-1 Image Format Description.

Complete details for Vers-1 Texture Environment can be found in 6.2.1.1.2.3.1.2 Vers-1 Texture Environment.

I32 : Number of Bytes

Number of Bytes specifies the length, in bytes, of the on-disk representation of the texture image. The texture image in a JT file is a single monolithic/contiguous block of data beginning with the highest-level mip image, and processing through the mipmaps down to a one-by-one texel image. If there are no mipmaps, then the number of bytes is for a single texture image. If Number of Bytes is zero then no other data is stored.

UChar : Image Texel Data

Image Texel Data is the single monolithic/contiguous block of image data. The length of this field in bytes is specified by the value of data field Number of Bytes.

6.2.1.1.2.3.1.1   Vers-1 Image Format Description

The Vers-1 Image Format Description is a collection of data defining the pixel format, data type, size, and other miscellaneous characteristics of the monolithic block of image data.

Figure 44: Vers-1 Image Format Description data collection

U32 : Pixel Format

↓

U32 : Pixel Data Type

↓

I16 : Dimensionality

↓

I32 : Width

↓

I32 : Height

↓

U32 : Mipmaps Flag

↓

U32 : Shared Image Flag

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.

= 0	– No format specified. Texture mapping is not applied.
= 1	– A red color component followed by green and blue color components
= 2	– A red color component followed by green, blue, and alpha color components
= 3	– A single luminance component
= 4	– A luminance component followed by an alpha color component.
= 5	– A single stencil index.
= 6	– A single depth component
= 7	– A single red color component
= 8	– A single green color component
= 9	– A single blue color component
= 10	– A single alpha color component
= 11	– A blue color component, followed by green and red color components
= 12	– A blue color component, followed by green , red, and alpha color components

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 "1" with Pixel Data Type of "7" would require 12 bytes of storage for each texel).

= 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:

= 1	– One-dimensional texture
= 2	– Two-dimensional texture

I32 : Width

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

I32 : 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.

U32 : Mipmaps Flag

Mipmaps Flag is a flag indicating whether the texture image has mipmaps.

= 0	– No mipmaps
= 1	– Yes has mipmaps. Image Texel Data is assumed to contain multiple textures, each a mipmap of the base texture. All texture in power of two must be provided between the base texture and a one-by-one texture.

U32 : Shared Image Flag

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

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

6.2.1.1.2.3.1.2 Vers-1 Texture Environment

The Vers-1 Texture Environment is a collection of data defining various aspects of how a texture image is to be mapped/applied to a surface.

Figure 45: Vers-1 Texture Environment data collection

I32 : Mipmap Magnification Filter

↓

I32 : Mipmap Minification Filter

↓

I32 : S-Dimen Wrap Mode

↓

I32 : T-Dimen Wrap Mode

↓

I32 : Texture Function Data

↓

RGBA : Blend Color

↓

Mx4F32 : Texture Transform

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.

= 0	− None.
= 1	− Nearest. Texel with coordinates nearest the center of the pixel is used.
= 2	− Linear. A weighted linear average of the 2 x 2 array of texels nearest to the center 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.

= 0	− None.
= 1	− Nearest. Texel with coordinates nearest the center of the pixel is used.
= 2	− Linear. A weighted linear average of the 2 x 2 array of texels nearest to the center 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 center 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 center 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 center 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 center 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].

= 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 grater 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 : Texture Function Data

Texture Function Data contains information indicating how the values in the texture map are to be modulated/combined/blended with the original color of the surface or some other alternative color to compute the final color to be painted on the surface. This information is encoded within a single I32 using the following bit allocations.

Bits 0 - 2	Texture Environment Mode. Additional information on the interpretation of the Texture Environment Mode values and how one might leverage them to render an image can be found in reference [4] listed in section 2 References and Additional Information. = 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.
Bit 3	Environment Mapping Flag. Note that if this flag is ON (i.e. = 1), then applications processing this JT data for 3D graphical visualization should automatically turn ON texture coordinate generation for spherical environment maps. = 0 – OFF = 1 – ON
Bits 4 - 31	Reserved for future use.

RGBA : Blend Color

Blend Color specifies the color to be used for "Blend" Texture Environment Mode operations.

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.

6.2.1.1.2.3.2 Texture Vers-2 Data

Texture Vers-2 Data collection supports texturing effects not representable in the Texture Vers-1 Data format (e.g. multiple textures (i.e. channels), texture image storage location external to the JT file, three-dimensional textures, other than unsigned-byte data formats, mirror and edge/border coordinate clamp modes, etc.). Any Texture Image Attribute Element using the Texture Vers-2 Data format will contain a "degenerate" Texture Vers-1 Data block, where Number of Bytes data field has a value of "0".

Figure 46: Texture Vers-2 Data data collection
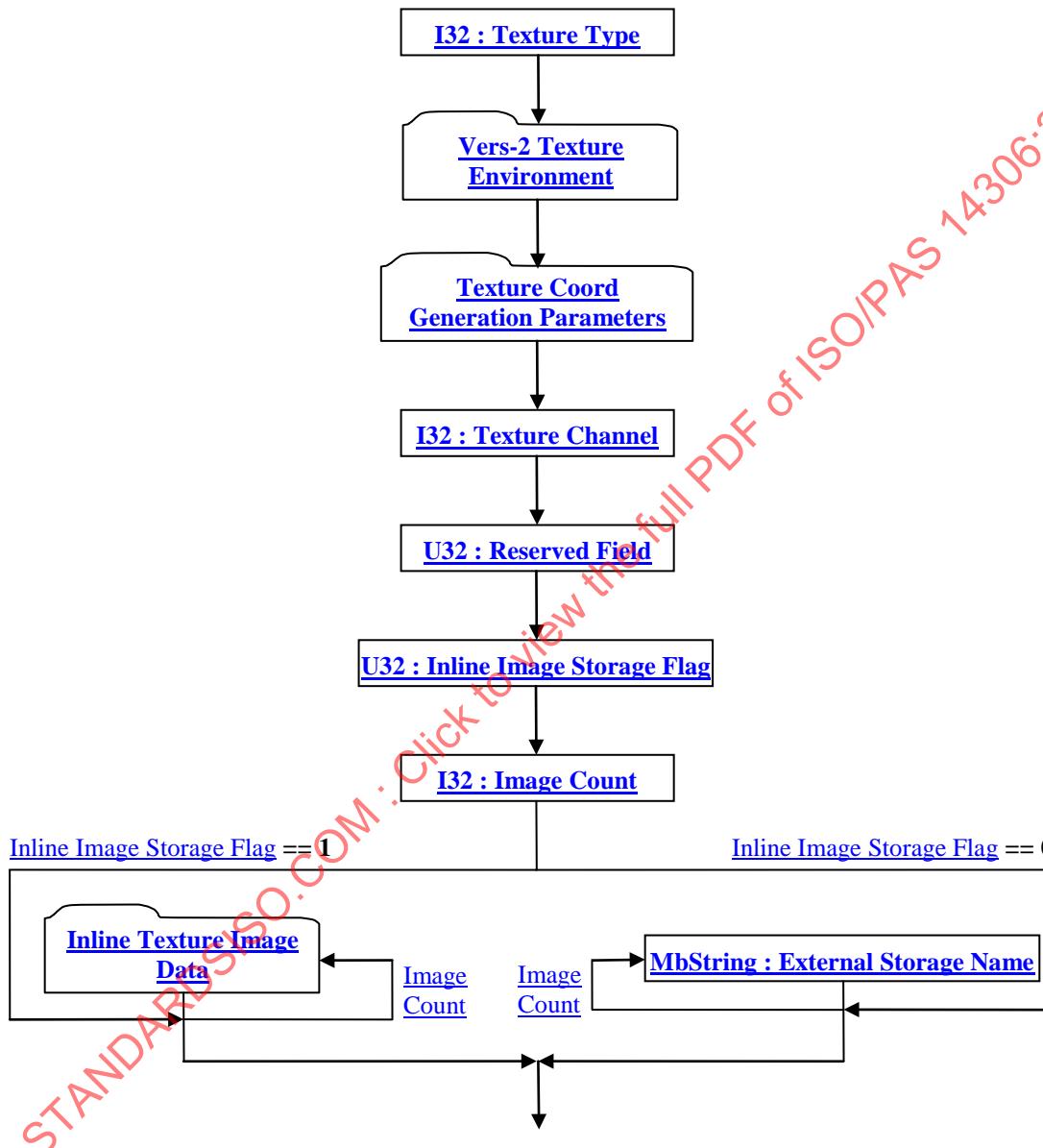
Complete details for Vers-2 Texture Environment can be found in 6.2.1.1.2.3.2.1 Vers-2 Texture Environment.

Complete details for Texture Coord Generation Parameters can be found in 6.2.1.1.2.3.2.2 Texture Coord Generation Parameters.

Complete details for Inline Texture Image Data can be found in 6.2.1.1.2.3.2.3 Inline Texture Image Data.

I32 : Texture Type
Texture Type specifies the type of texture.

= 0	– None.

= 1	– One-Dimensional. A one-dimensional texture has a height (T-Dimension) and depth (R-Dimension) equal to "1" and no top or bottom border.
= 2	– Two-Dimensional. A two-dimensional texture has a depth (R-Dimension) equal to "1."
= 3	– Three-Dimensional. A three-dimensional texture can be thought of as layers of two-dimensional sub image rectangles arranged in a sequence.
= 4	– Bump Map. A bump map texture is a texture where the image texel data (e.g. RGB color values) represents surface normal XYZ components.
= 5	– Cube Map. A cube map texture is a texture cube centered at the origin and formed by a set of six two-dimensional texture images.

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 directly to the OpenGL concept of a "texture unit." The Texture Channel value must be between 0 and 31 inclusive. Best practices suggest that renderer of JT data ignore all but channel-0 if the renderer does not support multi-textured geometry. Also for purposes of blending, renderer of JT data should assume that higher numbered texture channels "blend over" lower numbered ones.

U32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

U32 : 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.

| = 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" Texture Type must 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 Inline Image Storage Flag equals "0." If present there will be data field Image Count 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.

6.2.1.1.2.3.2.1   Vers-2 Texture Environment

The Vers-2 Texture Environment is a collection of data defining various aspects of how a texture image is to be mapped/applied to a surface.

Figure 47: Vers-2 Texture Environment data collection

```
┌─────────────────────────────────┐
│    I32 : Border Mode            │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│  I32 : Mipmap Magnification Filter │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│  I32 : Mipmap Minification Filter │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│    I32 : S-Dimen Wrap Mode       │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│    I32 : T-Dimen Wrap Mode       │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│    I32 : R-Dimen Wrap Mode       │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│    I32 : Blend Type              │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│  I32 : Internal Compression Level │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│    RGBA : Blend Color            │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│    RGBA : Border Color           │
└─────────────────────────────────┘
                │
                ▼
┌─────────────────────────────────┐
│   Mx4F32 : Texture Transform     │
└─────────────────────────────────┘
```
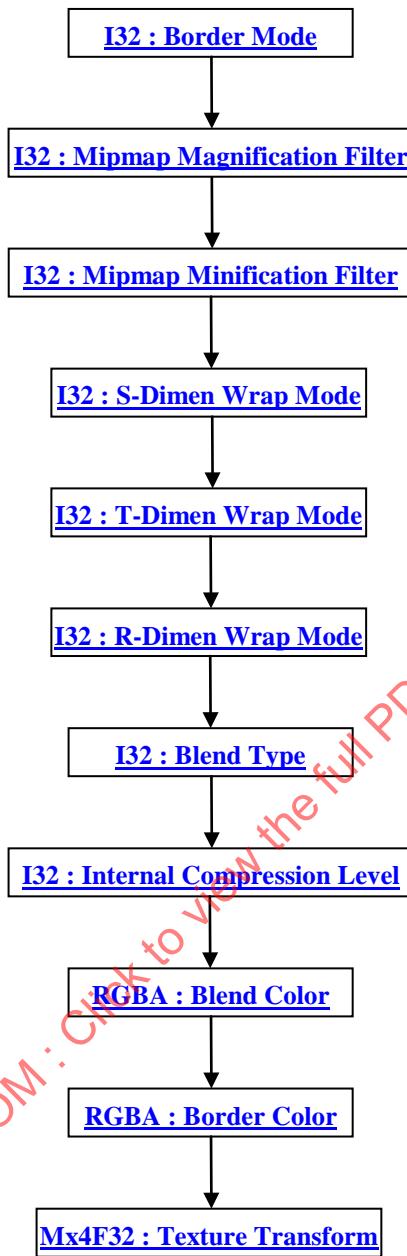
I32 : Border Mode

Border Mode specifies the texture border mode.

= 0	– No border.
= 1	– Constant Border Color. Indicates that the texture has a constant border color whose value is defined in data field Border Color.
= 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.

= 0	– None.
= 1	– Nearest. Texel with coordinates nearest the center of the pixel is used.

| = 2 | – Linear. A weighted linear average of the 2 x 2 array of texels nearest to the center 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.

= 0	– None.
= 1	– Nearest. Texel with coordinates nearest the center of the pixel is used.
= 2	– Linear. A weighted linear average of the 2 x 2 array of texels nearest to the center 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 center 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 center 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 center 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 center 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].

= 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 grater 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 color of the surface or some other alternative color to compute the final color 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 [4] listed in section 2 References and Additional Information.

= 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.

= 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 Color

Blend Color specifies the color to be used for the "Blend" mode of Blend Type operations.

RGBA : Border Color

Border Color specifies the constant border color 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.

6.2.1.1.2.3.2.2 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 48: Texture Coord Generation Parameters data collection
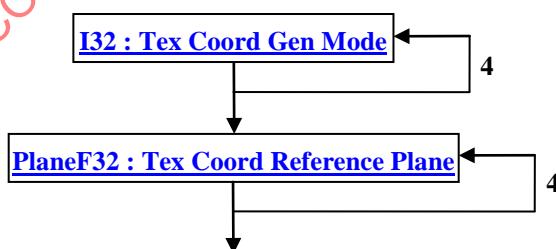
I32 : Tex Coord Gen Mode

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.

= 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.

6.2.1.1.2.3.2.3 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 Inline Image Storage Flag equals "1." If present there will be data field Image Count number of Inline Texture Image Data instances.

Figure 49: Inline Texture Image Data data collection
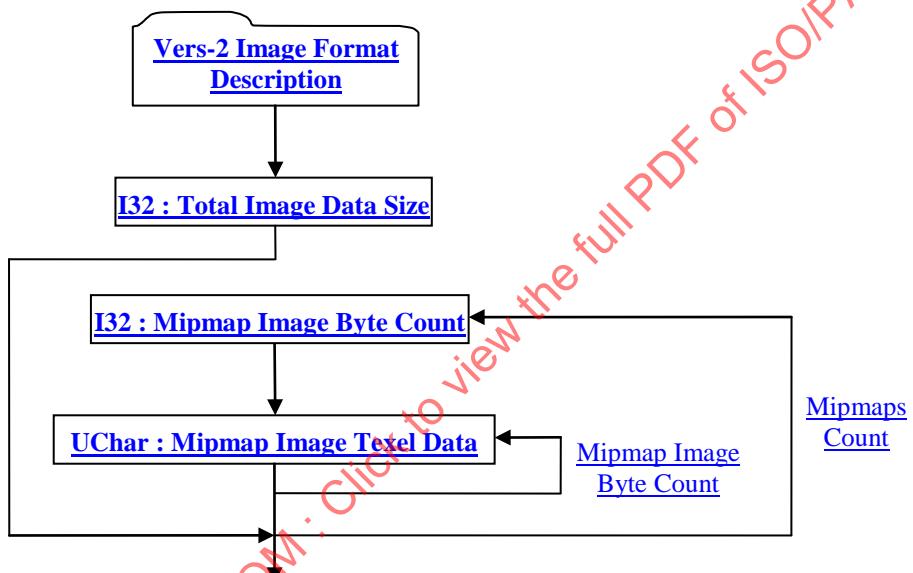
Complete description for Vers-2 Image Format Description can be found in 6.2.1.1.2.3.2.3.1 Vers-2 Image Format Description.

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.

6.2.1.1.2.3.2.3.1 Vers-2 Image Format Description

The Vers-2 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 50: Vers-2 Image Format Description data collection

```
┌─────────────────────────┐
│   U32 : Pixel Format     │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│   U32 : Pixel Data Type  │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│   I16 : Dimensionality   │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│   I16 : Row Alignment    │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│      I16 : Width         │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│      I16 : Height        │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│      I16 : Depth         │
└─────────────────────────┘
            │
            ▼
┌──────────────────────────────┐
│ I16 : Number Border Texels    │
└──────────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│  U32 : Shared Image Flag │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│   I16 : Mipmaps Count    │
└─────────────────────────┘
```

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.

= 0	– No format specified. Texture mapping is not applied.
= 1	– A red color component followed by green and blue color components
= 2	– A red color component followed by green, blue, and alpha color components
= 3	– A single luminance component
= 4	– A luminance component followed by an alpha color component.
= 5	– A single stencil index.
= 6	– A single depth component
= 7	– A single red color component
= 8	– A single green color component
= 9	– A single blue color component
= 10	– A single alpha color component
= 11	– A blue color component, followed by green and red color components
= 12	– A blue color component, followed by green , red, and alpha color components

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 "1" with Pixel Data Type of "3" would require 3 bytes of storage for each texel).

= 3	– Unsigned 8-bit integer

I16 : Dimensionality

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

= 1	– One-dimensional texture
= 2	– Two-dimensional texture

I16 : Row Alignment

Row Alignment specifies the byte alignment for image data rows. This data field must 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 eight for row alignment equal "8"). The actual formula (using C syntax) to determine number of bytes per row is as follows:

$$BytesPerRow = (numBytesPerPixel * ImageWidth + RowAlignmnet - 1) \& \sim(RowAlignment - 1)$$

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" or "1."

U32 : Shared Image Flag

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

= 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.

6.2.1.1.2.4 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 dictate that draw style attributes accumulate down the LSG by replacement.

The Field Inhibit flag (see 6.2.1.1.2.1.1Base Attribute Data) bit assignments for the Draw Style Attribute Element data fields, are as follows:

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 51: Draw Style Attribute Element data collection
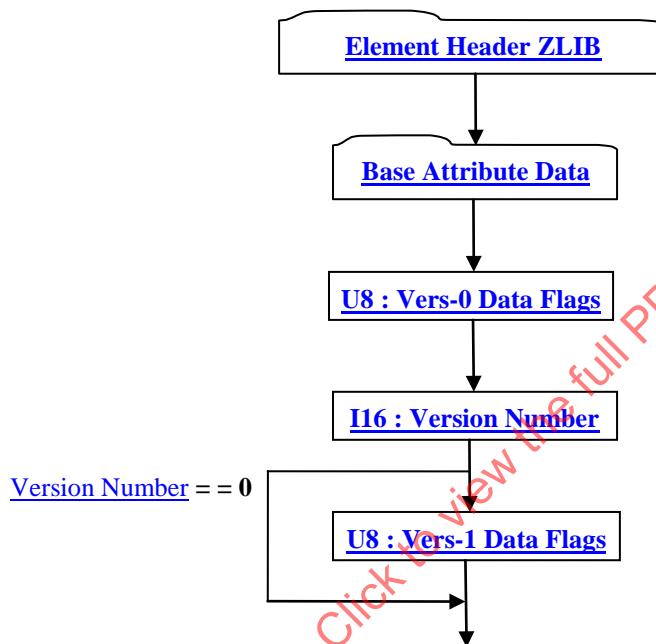
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

U8 : Vers-0 Data Flags

Vers-0 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 undocumented bits are reserved.

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 back-facing polygons are illuminated. = 0 – Disable two sided lighting. = 1 – Enable two sided lighting.

0x04	− Outlined Polygons Flag Indicates if polygons should be draw in "wire frame mode" i.e. not filled; only outlines drawn. = 0 – Polygons drawn as filled. = 1 – Only polygon's outline drawn.

I16 : Version Number

Version Number is the version identifier for this element. The value of this Version Number indicates if additional data follows.

= 0	− Version-0 Format
= 1	− Version-1 Format

U8 : Vers-1 Data Flags

Vers-1 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. Vers-1 Data Flags field is only present if Version Number equals "1." The Vers-1 Data Flags includes the Vers-0 Data Flags data (thus some data flags are repeated/duplicated) along with some additional flags. All undocumented bits are reserved.

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 back-facing polygons are illuminated. = 0 – Disable two sided lighting. = 1 – Enable two sided lighting.
0x04	− Outlined Polygons Flag Indicates if polygons should be draw in "wire frame mode" i.e. not filled; only outlines drawn. = 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 color (flat shading) or with many different color (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 color should be delayed until after texturing. If no texture mapping then this flag setting is irrelevant. = 0 – Apply specular color contribution before texture mapping. = 1 – Apply specular color contribution after texture mapping.

6.2.1.1.2.5 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 dictate 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 6.2.1.1.2.1.1Base Attribute Data) bit assignments.

Figure 52: Light Set Attribute Element data collection
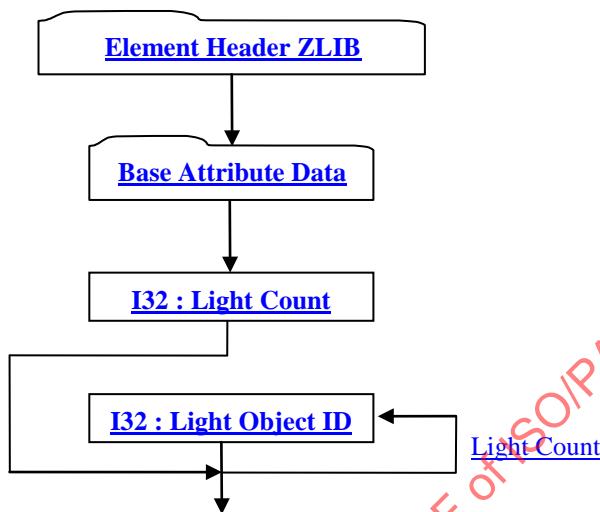
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

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.

6.2.1.1.2.6 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 dictate 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 6.2.1.1.2.1.1Base Attribute Data) bit assignments.

Figure 53: Infinite Light Attribute Element data collection

```
        ┌─────────────────────────┐
        │   Element Header ZLIB   │
        └─────────────────────────┘
                    │
                    ▼
        ┌─────────────────────────┐
        │     Base Light Data     │
        └─────────────────────────┘
                    │
                    ▼
        ┌─────────────────────────┐
        │   DirF32 : Direction    │
        └─────────────────────────┘
                    │
                    ▼
        ┌─────────────────────────┐
        │   I16 : Version Number  │
        └─────────────────────────┘
                    │           Version Number = = 1
                    │                   │
                    │                   ▼
                    │       ┌─────────────────────────┐
                    │       │   I32 : Coord System    │
                    │       └─────────────────────────┘
                    │◄──────────────────┘
                    ▼
```
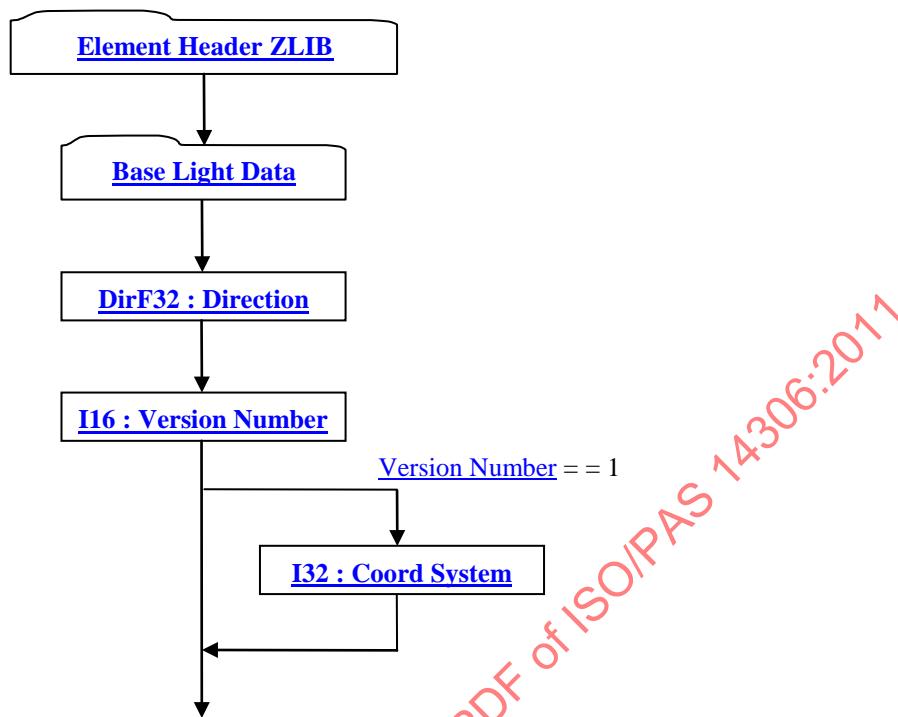
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

DirF32 : Direction
Direction specifies the direction the light is pointing in.

I16 : 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.

= 0	– Version-0 Format
= 1	– Version-1 Format

I32 : Coord System
Coord System specifies the coordinate space in which Light source is defined. Valid values include the following:

= 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.

6.2.1.1.2.6.1 Base Light Data

Figure 54: Base Light Data data collection
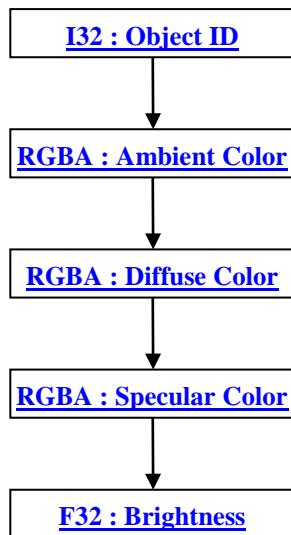
```
┌─────────────────────┐
│   I32 : Object ID    │
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│ RGBA : Ambient Color │
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│ RGBA : Diffuse Color │
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│ RGBA : Specular Color│
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│   F32 : Brightness   │
└─────────────────────┘
```

I32 : Object ID
Object ID is the identifier for this Object. Other objects referencing this particular object do so using the Object ID.

RGBA : Ambient Color
Ambient Color specifies the ambient red, green, blue, alpha color values of the light.

RGBA : Diffuse Color
Diffuse Color specifies the diffuse red, green, blue, alpha color values of the light.

RGBA : Specular Color
Specular Color specifies the specular red, green, blue, alpha color values of the light.

F32 : Brightness
Brightness specifies the Light brightness. The Brightness value must be greater than or equal to "-1".

6.2.1.1.2.7 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 dictate 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 6.2.1.1.2.1.1Base Attribute Data) bit assignments.

Figure 55: Point Light Attribute Element data collection
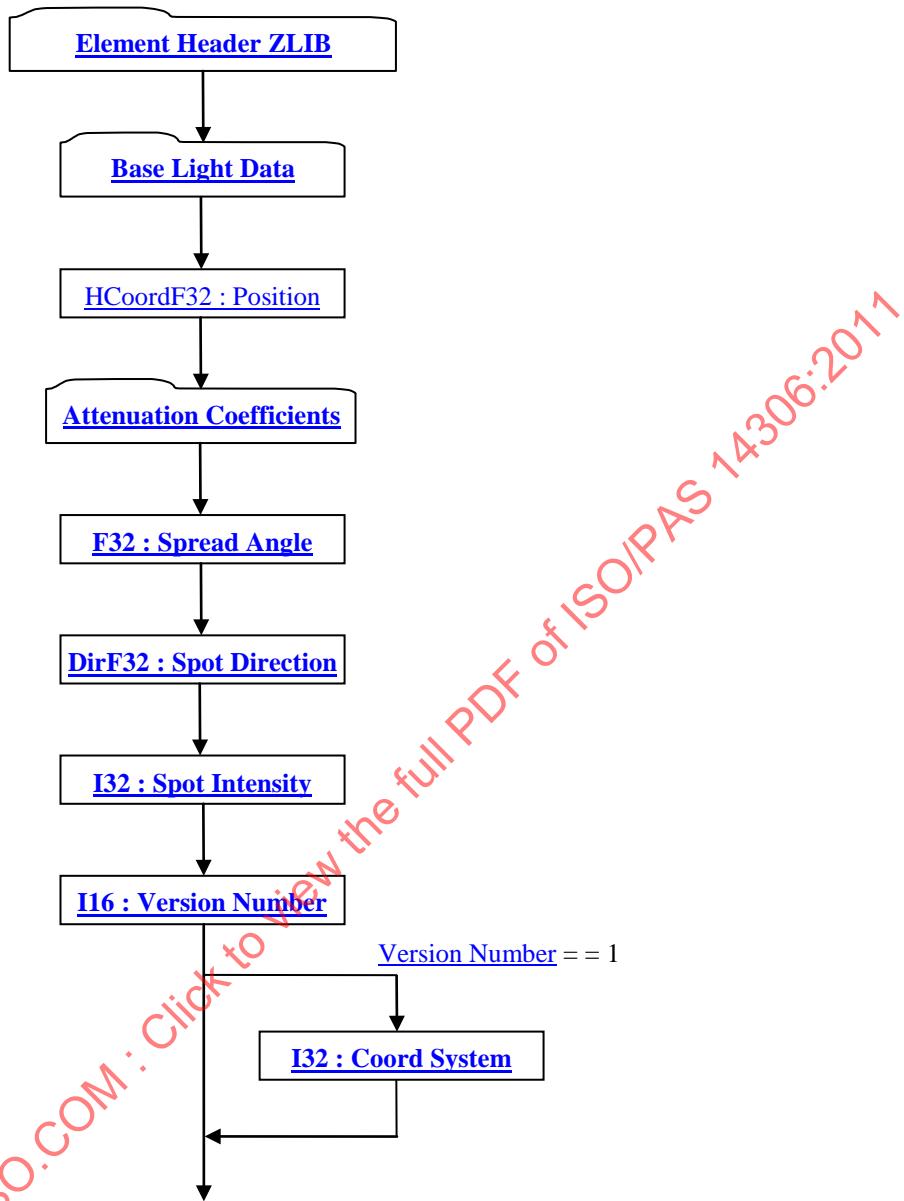
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Light Data can be found in 6.2.1.1.2.6.1 Base Light Data.

HCoordF32 : Position
Position specifies the light position in homogeneous coordinates.

F32 : Spread Angle
Spread Angle, as shown in Figure 56 below, specifies in degrees the half angle of the light cone. Valid Spread Angle values are clamped and interpreted as follows:

angle = = 180.0	– Simple point light
0.0 >= angle <= 90.0	– Spot Light
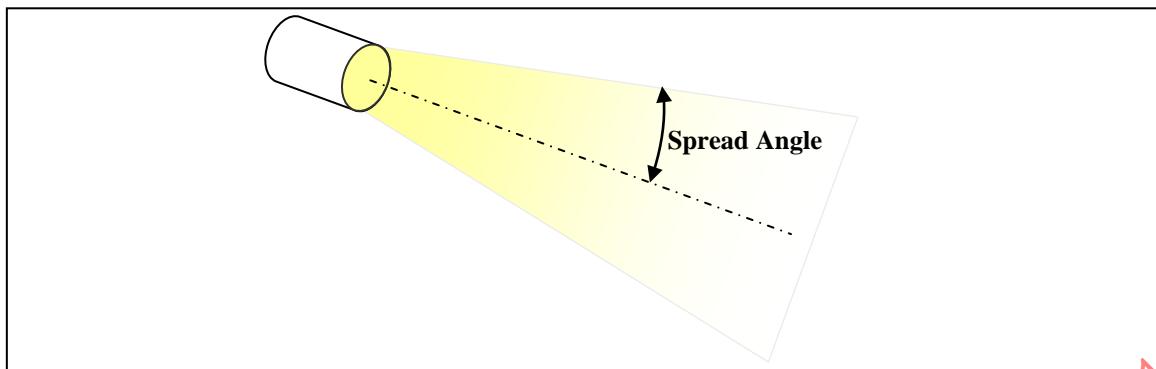
Figure 56: Spread Angle value with respect to the light cone

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 center. The larger the value, the more focused the light source. Only non-negative Spot intensity values are valid.

I16 : 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.

= 0	– Version-0 Format
= 1	– Version-1 Format

I32 : Coord System

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

= 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.

6.2.1.1.2.7.1 Attenuation Coefficients

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

Figure 57: Attenuation Coefficients data collection
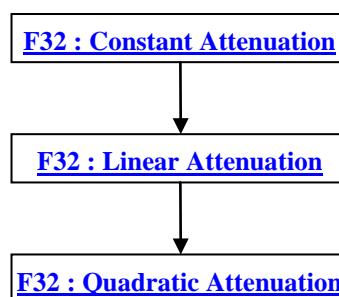
F32 : Constant Attenuation

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

F32 : Linear Attenuation

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

F32 : Quadratic Attenuation

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

6.2.1.1.2.8 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 that should be used for rendering polylines.   JT format LSG traversal semantics dictate that linestyle attributes accumulate down the LSG by replacement.

Linestyle Attribute Element does not have any Field Inhibit flag (see 6.2.1.1.2.1.1Base Attribute Data) bit assignments.

Figure 58: Linestyle Attribute Element data collection
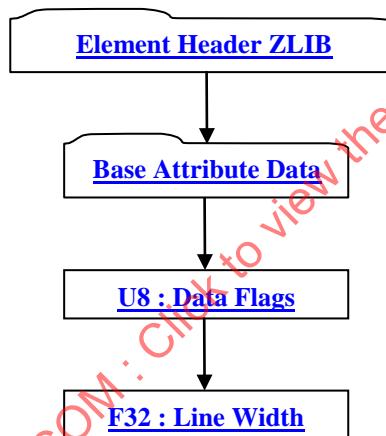
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

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 undocumented bits are reserved.

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 – Center_Dash = 7 – Center_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

6.2.1.1.2.9 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 dictate that pointstyle attributes accumulate down the LSG by replacement.

Pointstyle Attribute Element does not have any Field Inhibit flag (see 6.2.1.1.2.1.1Base Attribute Data) bit assignments.

Figure 59: Pointstyle Attribute Element data collection
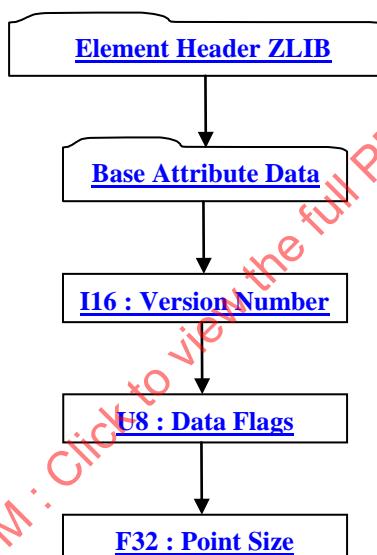
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

I16 : Version Number

Version Number is the version identifier for this element. Version number "0x0001" is currently the only valid value.

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 undocumented bits are reserved.

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.

       67

F32 : Point Size

Point Size specifies the size in pixels that should be used for rendering points.

6.2.1.1.2.10 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 matrix that positions the associated LSG node's coordinate system relative to its parent LSG node. JT format LSG traversal semantics dictate 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.

Geometric Transform Attribute Element does not have any Field Inhibit flag (see 6.2.1.1.2.1.1Base Attribute Data) bit assignments.

Figure 60: Geometric Transform Attribute Element data collection
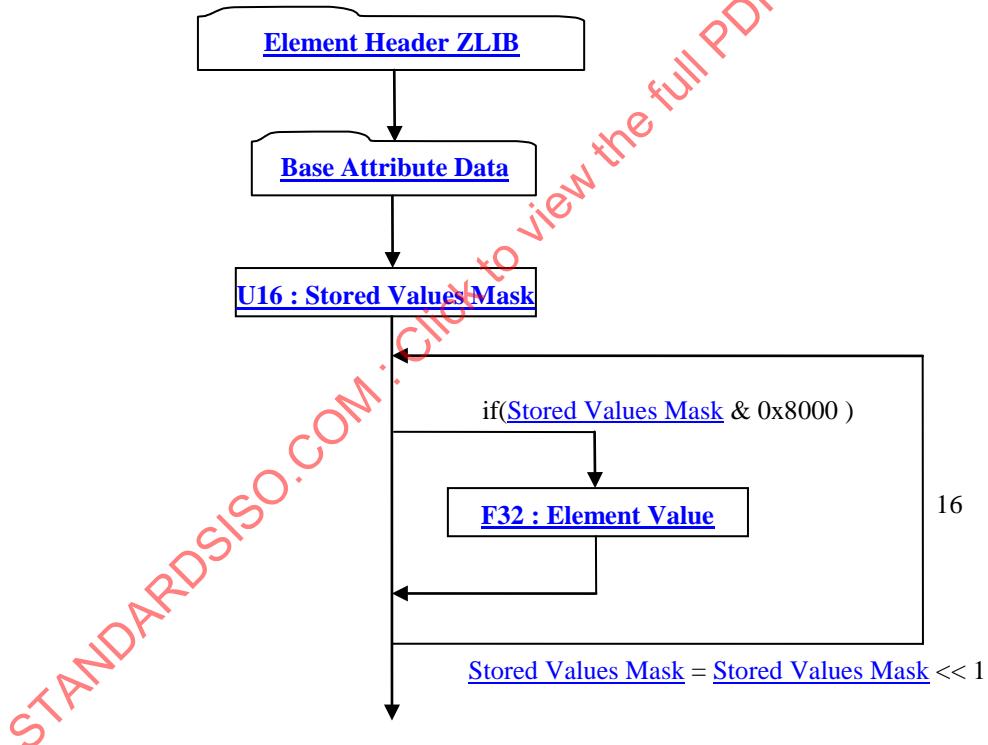
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

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:

= 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.

6.2.1.1.2.11 Shader Effects Attribute Element
Object Type ID: 0xaa1b831d, 0x6e47, 0x4fee, 0xa8, 0x65, 0xcd, 0x7e, 0x1f, 0x2f, 0x39, 0xdb

Shader Effects Attribute Element contains information specifying "high-level" shader functionality (e.g. Phong shading, bump mapping, etc.) that should be used for rendering the geometry this attribute element is associated with.

JT format LSG traversal semantics dictate that shader effects attributes accumulate down the LSG by replacement.

Shader Effects Attribute Element does not have any Field Inhibit flag (see 6.2.1.1.2.1.1Base_Attribute_Data) bit assignments.

Figure 61: Shader Effects Attribute Element data collection
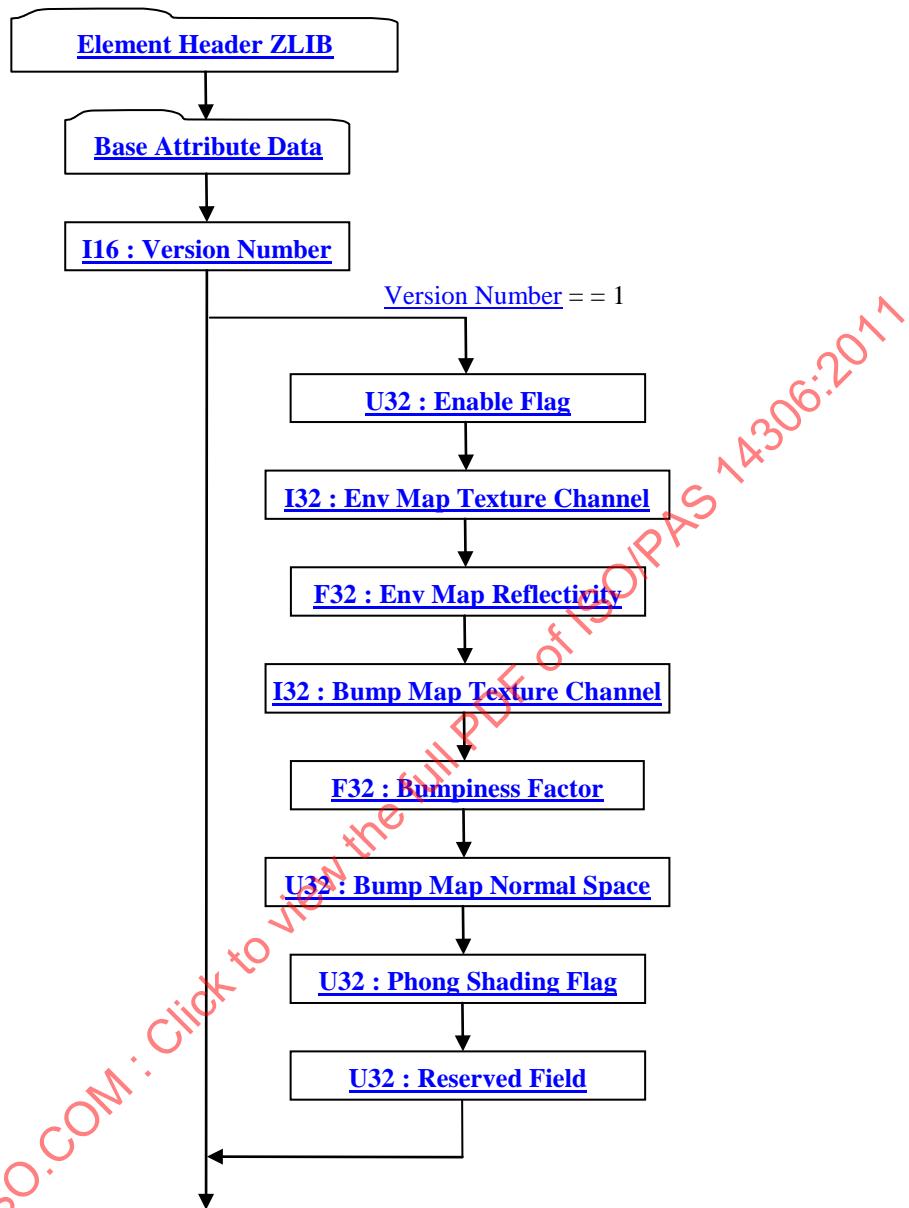
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

I16 : Version Number
Version Number is the version identifier for this element. Version number "0x0001" is currently the only valid value.

U32 : Enable Flag
Enable Flag specifies whether this Shader Effects Attribute is enabled. Valid values include the following:

= 0	− Shader Effects Attribute disabled
= 1	− Shader Effects Attribute enabled

I32 : Env Map Texture Channel
Env Map Texture Channel specifies the texture channel designated as containing an environment map. A value of "-1" disables environment mapping through the Shader Effects Attribute. Note that this will NOT disable a texture map that

is explicitly set up with sphere mapping or cube mapping through Material Attribute Element. Note that other irrelevant Material Attribute Element parameters (e.g. blending type, texture coordinate generation mode, border settings, etc.) are ignored for the environment map texture channel.

F32 : Env Map Reflectivity

Env Map Reflectivity specifies the fraction of the environment to be reflected (1 minus this fraction will show through form the underlying texture channel). Valid value must be in the range [0:1] inclusive.

I32 : Bump Map Texture Channel

Bump Map Texture Channel specifies the texture channel designated as containing a bump map. A value of "-1" disables bump mapping through the Shader Effects Attribute. Note that other irrelevant Material Attribute Element parameters (e.g. blending type, texture coordinate generation mode, border settings, etc.) are ignored for the bump map texture channel.

F32 : Bumpiness Factor

Bumpiness Factor specifies the degree of "bumpiness", or the relative "height" of the bump map. Larger values make the bumps appear deep and more severe. Negative values invert the sense of the bump map, making the surface appear engraved, rather then embossed. This value only has an effect with tangent space bump maps.; it has no effect on the appearance of object space bump maps.

U32 : Bump Map Normal Space

Bump Map Normal Space specifies what coordinate space the normal map is to be interpreted in. Valid values include the following:

= 0	− Normal Map Interpreted as an "object space" normal map
= 1	− Normal Map Interpreted as a "tangent space" normal map.

U32 : Phong Shading Flag

Phong Shading Flag specifies whether Phong Shading (i.e. per fragment lighting) is enabled. Valid values include the following:

= 0	− Phong Shading disabled
= 1	− Phong Shading enabled

U32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion

6.2.1.1.2.12 Vertex Shader Attribute Element

Object Type ID: 0x2798bcad, 0xe409, 0x47ad, 0xbd, 0x46, 0xb, 0x37, 0x1f, 0xd7, 0x5d, 0x61

Vertex Shader Attribute Element defines a per-vertex shader program in either the Cg or GLSL shading language. Complete descriptions of the Cg and GLSL shading languages can be found in references listed within the 2 References and Additional Information section of this document.

JT format LSG traversal semantics dictate that vertex shader attributes accumulate down the LSG by replacement; with the exception that if the new vertex shader attribute's shader language is not the same as current vertex shader attribute's shader language, then new vertex shader attribute is simply ignored.

In general, a shader program is used to replace a portion of the otherwise fixed functionality graphics pipeline with some user-defined function. Specifically a Vertex Shader program is a small user defined program to be run for each vertex that is sent down to the GPU and processed. A Vertex shader can alter vertex positions and normals, generate texture coordinates, perform Gouraud vertex lighting, etc

Figure 62: Vertex Shader Attribute Element data collection
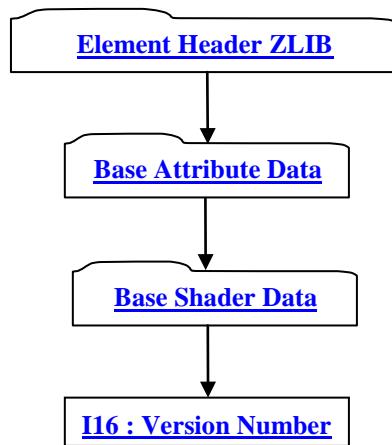
Element Header ZLIB

Base Attribute Data

Base Shader Data

I16 : Version Number

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

I16 : Version Number

Version Number is the version identifier for this element. Version number "0x0001" is currently the only valid value.

6.2.1.1.2.12.1 Base Shader Data

Figure 63: Base Shader Data data collection

I16 : Version Number

Version Number = = 1

I32 : Shader Language

U32 : Inline Source Flag

Inline Source Flag = = 1

MbString : Source Code Loc   MbString : Source Code

I32 : Shader Param Count

Shader Parameter

Shader Param Count
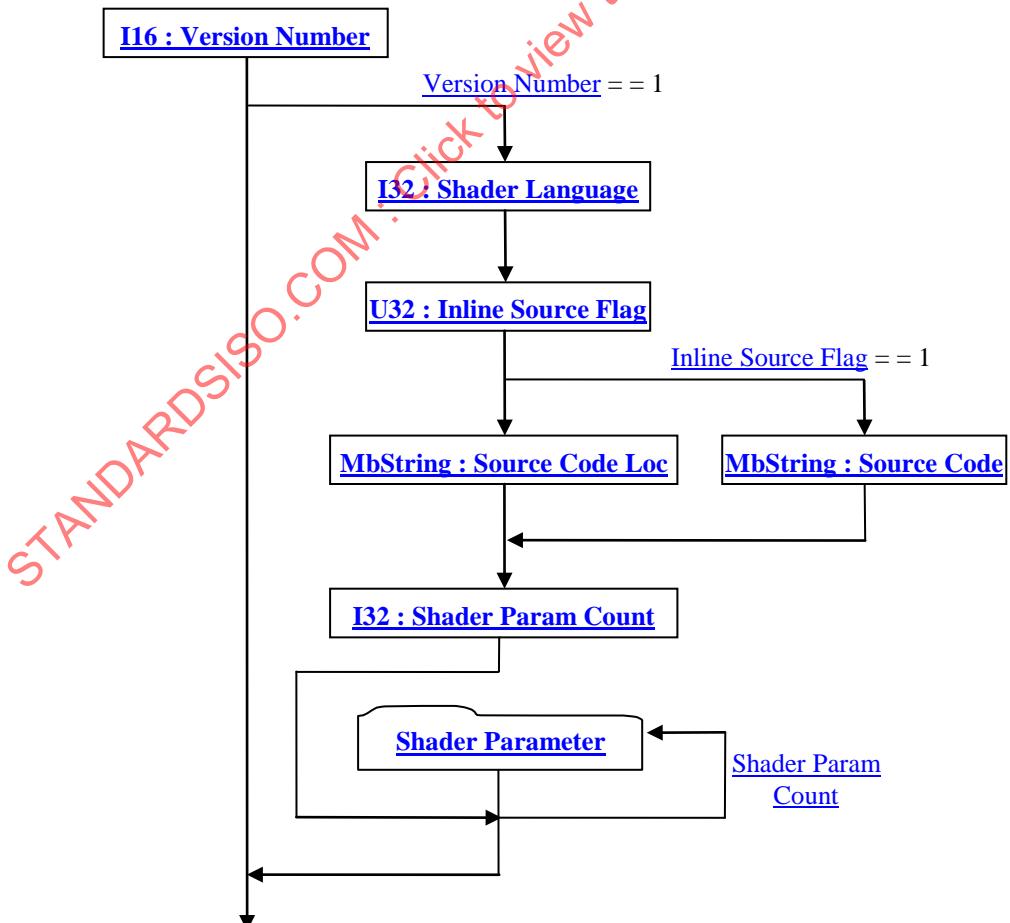
I16 : Version Number

Version Number is the version identifier for this data collection. Version number "0x0001" is currently the only valid value.

I32 : Shader Language

Shader Language specifies the Shader program language. Valid values include the following:

= 0	– None
= 1	– Cg ("C for graphics" is a high-level shading language created by nVIDIA for programming vertex and pixel shaders [8] [9].
= 2	– GLSL ("GL Shading Language" as defined by the Architectural Review Board of OpenGL, the governing body of OpenGL [7].

U32 : Inline Source Flag

Inline Source Flag specifies whether the shader's "source code" is stored within this JT file or in some other externally referenced file. Valid values include the following:

= 0	– Source code stored in an externally referenced file.
= 1	– Source code stored within this JT file.

MbString : Source Code

Source Code is the shader's source code in Shader Language programming language.

MbString : Source Code Loc

Source Code Loc specifies the file name for the external file containing the shader's source code.

I32 : Shader Param Count

Shader Param Count specifies the number of shader parameters.

6.2.1.1.2.12.1.1   Shader Parameter

Shader Parameter data collection defines a Shader input and/or output parameter. A list of Shader Parameters represents the runtime linkage of the shader program into the GPU's data streams.

Figure 64: Shader Parameter data collection
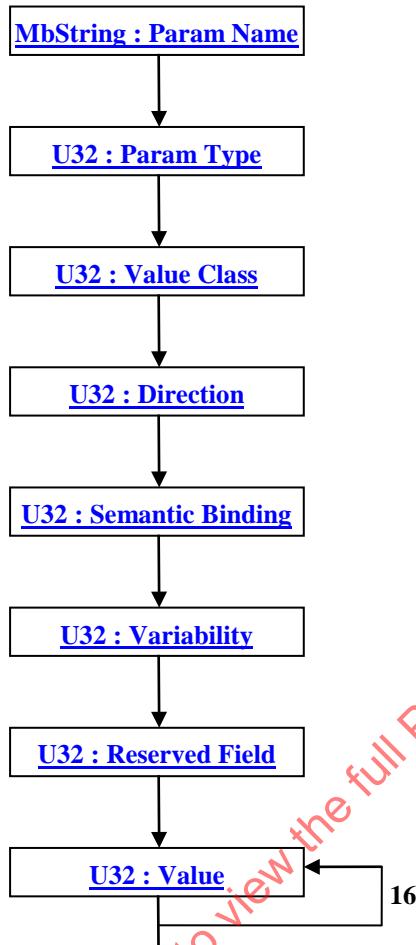
MbString : Param Name
Param Name specifies the shader parameter name.

U32 : Param Type
Param Type specifies the shader parameter type. Valid types include the following:

= 0	– Unknown
= 1	– Boolean
= 2	– Integer
= 3	– Float
= 4	– Vector of two Integer values.
= 5	– Vector of three Integer values
= 6	– Vector of four Integer values
= 7	– Vector of two Float values
= 8	– Vector of three Float values
= 9	– Vector of four Float values
= 10	– 2 x 2 matrix of Float values
= 11	– 3 x 3 matrix of Float values
= 12	– 4 x 4 matrix of Float values
= 13	– Texture Object/Unit number bound to current 1D texture sampler
= 14	– Texture Object/Unit number bound to current 2D texture sampler
= 15	– Texture Object/Unit number bound to current 3D texture sampler

= 16	– Texture Object/Unit number bound to current rect map texture sampler
= 17	– Texture Object/Unit number bound to current cube map texture sampler
= 18	– Texture Object/Unit number bound to current 1D shadow map texture sampler
= 19	– Texture Object/Unit number bound to current 2D shadow map texture sampler

U32 : Value Class

Value Class specifies the shader parameter "value class". Valid values include the following:

= 0	– Unknown class
= 1	– Immediate class.
= 2	– Semantic class (i.e. Shader Parameter is implicitly tied/bound to a piece of OpenGL graphics system state (e.g. OpenGL ModelView matrix) or JT graphics system state (e.g. diffuse material color)). The actual graphics state that the parameter is bound to is indicated by value in Value data field.

U32 : Direction

Direction specifies whether the shader parameter is an input, output, or input/output parameter. Direction value is only applicable for the Cg Shader Language. Valid values include the following:

= 0	– Unknown
= 1	– Input parameter
= 2	– Output parameter
= 3	– Both an Input and an Output parameter.

U32 : Semantic Binding

Semantic Binding specifies the "per vertex input and/or output" or the "per fragment input and/or output" this shader parameter is associated with (i.e. bound to). Semantic Binding value is only applicable for the Cg Shader Language. Valid values, including their input/output applicability to vertex and fragment shaders, are as follows (note that N/A indicates 'Not Applicable'):

Value	Binding Description	Vertex Shader Applicability	Fragment Shader Applicability
= 0	Unknown		
= 1	None		
= 2	Position	Input/Output	Input
= 3	Normal	Input	N/A
= 4	Binormal	Input	N/A
= 5	Blend Indices	Input	N/A
= 6	Blend Weight	Input	N/A
= 7	Tangent	Input	N/A
= 8	Point Size	Input/Output	Input
= 10	Texture Coordinate 0	Input/Output	Input
= 11	Texture Coordinate 1	Input/Output	Input
= 12	Texture Coordinate 2	Input/Output	Input
= 13	Texture Coordinate 3	Input/Output	Input
= 14	Texture Coordinate 4	Input/Output	Input
= 15	Texture Coordinate 5	Input/Output	Input
= 16	Texture Coordinate 6	Input/Output	Input
= 17	Texture Coordinate 7	Input/Output	Input
= 20	Fog Coordinate	Output	Input
= 21	Primary Color	Output	Input
= 22	Secondary Color	Output	Input
= 23	Primary Color	N/A	Output
= 24	Depth Value	N/A	Output

U32 : Variability

Variability specifies how often the value of the parameter is allowed to change. Valid values include the following:

= 0	– Unknown
= 1	– Constant (a parameter that takes on a single value and never changes)
= 2	– Uniform (a parameter that may take on a different value each time the shader is invoked but remains the same for all vertices or fragments processed by the shader)
= 3	– Varying (a parameter which may change with every vertex or fragment processed by the shader)

U32 : Reserved Field
Reserved Field is a data field reserved for future JT format expansion.

U32 : Value
Value specifies the shader parameter values treated as a U32 array of bytes. The maximum number of bytes required to store all possible Param Type and Value Class dependent values is 64 bytes and thus there are 16 U32 values stored. The interpretation of the Value data is Param Type and Value Class dependent as follows:

- For "Immediate" Value Class parameters (i.e. Value Class = = 1), the interpretation of the Value data is dependent upon the Param Type value.

- For "Semantic" Value Class parameters, the Value data is to be interpreted as a single U32 with all the possible values documented in Annex B:Semantic Value Class Shader Parameter Values.

6.2.1.1.2.13 Fragment Shader Attribute Element
Object Type ID: 0xad8dccc2, 0x7a80, 0x456d, 0xb0, 0xd5, 0xdd, 0x3a, 0xb, 0x8d, 0x21, 0xe7

Fragment Shader Attribute Element defines a per-fragment shader program in either the Cg or GLSL shading language. Complete descriptions of the Cg and GLSL shading languages can be found in references listed within the 2 References and Additional Information section of this document.

JT format LSG traversal semantics dictate that fragment shader attributes accumulate down the LSG by replacement; with the exception that if the new fragment shader attribute's shader language is not the same as current fragment shader attribute's shader language, then new fragment shader attribute is simply ignored.

In general, a shader program is used to replace a portion of the otherwise fixed functionality graphics pipeline with some user-defined function. Specifically, a Fragment Shader program is a small user defined program to be run for each fragment generated by the hardware's scan-conversion logic (where a fragment is a proto-pixel generated by triangle scan-conversion, but not let laid down into the frame buffer). A Fragment Shader can support sophisticated effects like Phong shading, shadow mapping, bump mapping, reflection mapping, etc.

Figure 65: Fragment Shader Attribute Element data collection
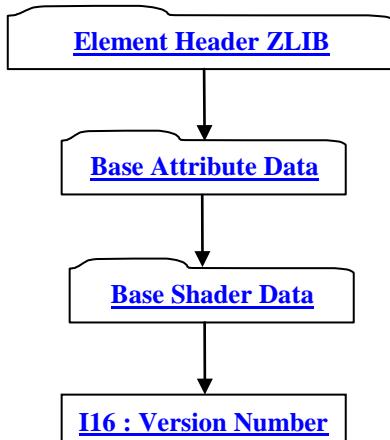
```
Element Header ZLIB
        |
        v
Base Attribute Data
        |
        v
Base Shader Data
        |
        v
I16 : Version Number
```

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Attribute Data can be found in 6.2.1.1.2.1.1Base Attribute Data.

Complete description for Base Shader Data can be found in 6.2.1.1.2.12.1 Base Shader Data.

I16 : Version Number

Version Number is the version identifier for this element. Version number "0x0001" is currently the only valid value.

6.2.1.2 Property Atom Elements

Property Atom Elements are meta-data objects associated with nodes. Property Atom Elements are not nodes themselves, but can be associated with any node to maintain arbitrary application or enterprise information (meta-data) pertaining to that node. Each Node Element in a LSG may hold zero or more properties and this relationship information is stored within 6.2.1.3 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 8.4 Metadata Conventions section of this reference.

6.2.1.2.1 Base Property Atom Element

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

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

Figure 66: Base Property Atom Element data collection
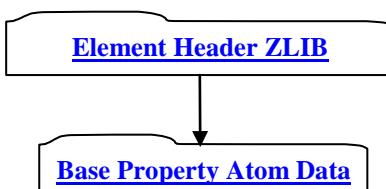
```
Element Header ZLIB
        |
        v
Base Property Atom Data
```

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

6.2.1.2.1.1 Base Property Atom Data

Figure 67: Base Property Atom Data data collection
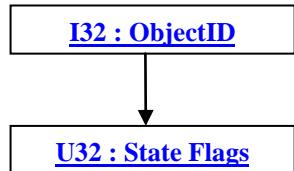
I32 : ObjectID

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

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 what ever property atom information desired. All other bits are reserved for future expansion of the file format.

6.2.1.2.2 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 68: String Property Atom Element data collection
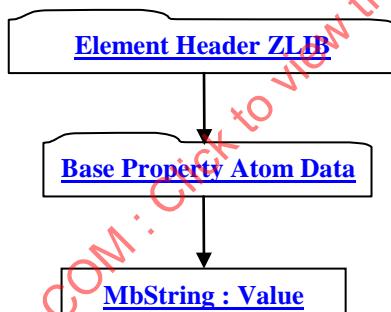
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Property Atom Data can be found in 6.2.1.2.1.1Base Property Atom Data.

MbString : Value

Value contains the character string value for this property atom.

6.2.1.2.3 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 69: Integer Property Atom Element data collection
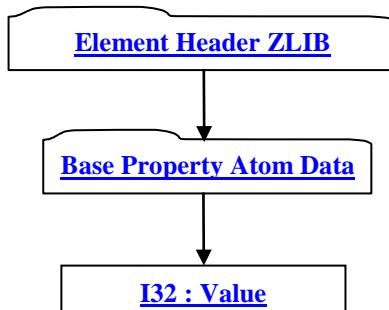
Element Header ZLIB

Base Property Atom Data

I32 : Value

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Property Atom Data can be found in 6.2.1.2.1.1Base Property Atom Data.

I32 : Value
Value contains the integer value for this property atom.

6.2.1.2.4 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 70: Floating Point Property Atom Element data collection
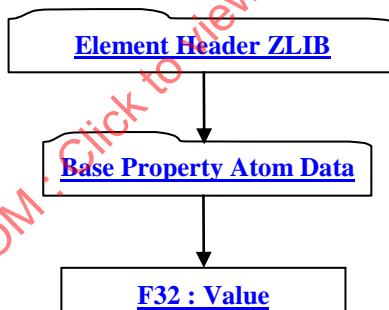
Element Header ZLIB

Base Property Atom Data

F32 : Value

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Property Atom Data can be found in 6.2.1.2.1.1Base Property Atom Data.

F32 : Value
Value contains the floating point value for this property atom.

6.2.1.2.5 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 71: JT Object Reference Property Atom Element data collection
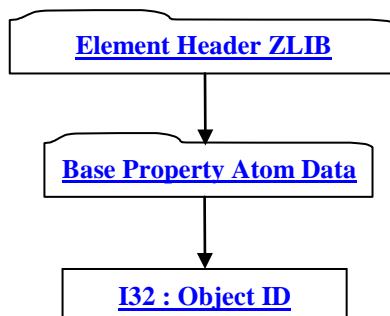
```
┌─────────────────────────────┐
│   Element Header ZLIB        │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│   Base Property Atom Data    │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│      I32 : Object ID         │
└─────────────────────────────┘
```

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Property Atom Data can be found in 6.2.1.2.1.1Base Property Atom Data.

I32 : Object ID
Object ID specifies the identifier within the JT file for the referenced object.

6.2.1.2.6 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 72: Date Property Atom Element data collection
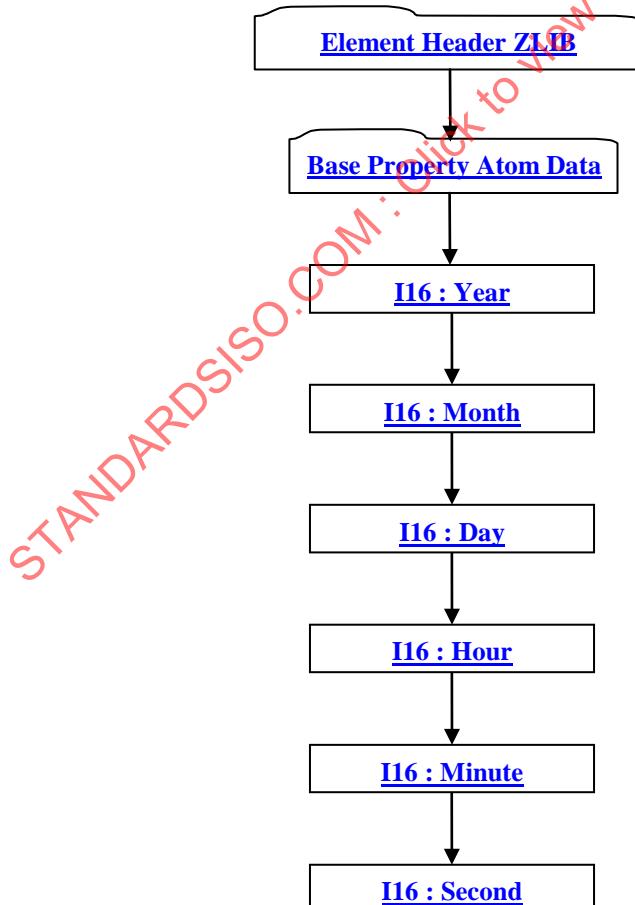
```
┌─────────────────────────────┐
│   Element Header ZLIB        │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│   Base Property Atom Data    │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│        I16 : Year            │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│        I16 : Month           │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│         I16 : Day            │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│        I16 : Hour            │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│        I16 : Minute          │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│        I16 : Second          │
└─────────────────────────────┘
```

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Property Atom Data can be found in 6.2.1.2.1.1Base Property Atom Data.

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.

6.2.1.2.7 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 6.2.2 Shape LOD Segment and 6.2.3 JT B-Rep Segment).

Figure 73: Late Loaded Property Atom Element data collection
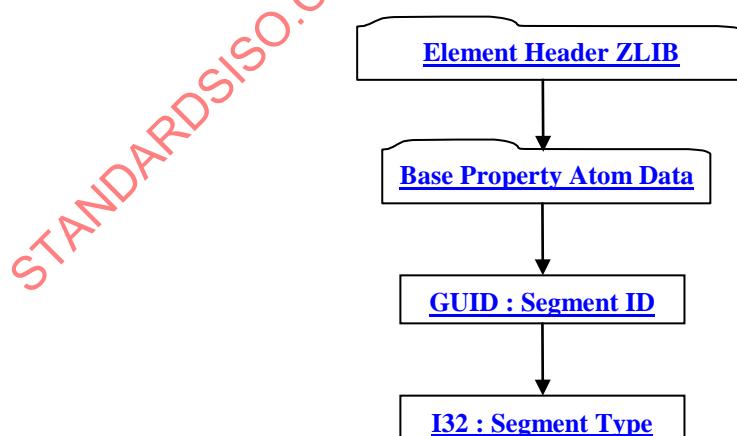
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

Complete description for Base Property Atom Data can be found in 6.2.1.2.1.1Base Property Atom Data.

GUID : Segment ID

Segment ID is the globally unique identifier for the associated data segment in the JT file. See 6.1.2 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.

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.

The complete list of segment types can be found in Table 3: Segment Types.

6.2.1.3 Property Table

The Property Table is where the data connecting Nodes with their associated Properties is stored. The Property Table contains a Node Property Table for each Node in the JT File which has associated Properties. A Node Property Table is a list of key/value Property Atom Element pairs for all Properties associated with a particular Node Element Object.

For a reference compliant JT File all Node 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 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 74: Property Table data collection
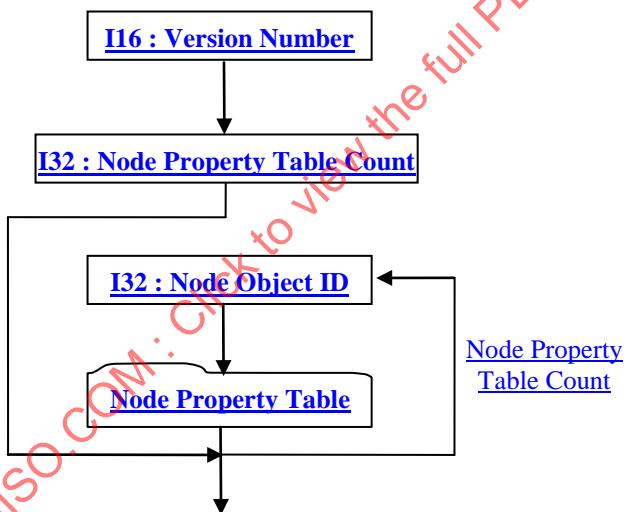
I16 : Version Number

Version Number is the version identifier for this Property Table. Version number "0x0001" is currently the only valid value.

I32 : Node Property Table Count

Node Property Table Count specifies the number of Node Property Tables to follow. This value is equivalent to the number of Node Elements (see 6.2.1.1.1Node Elements) that have associated Property Atom Elements (see 6.2.1.2Property Atom Elements).

I32 : Node Object ID

Node Object ID is the identifier for the Node Element object (see 6.2.1.1.1Node Elements) that the following Node Property Table is for (i.e. Node Element that all properties in the following Node Property Table are associated with).

6.2.1.3.1 Node Property Table

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

Figure 75: Node Property Table data collection
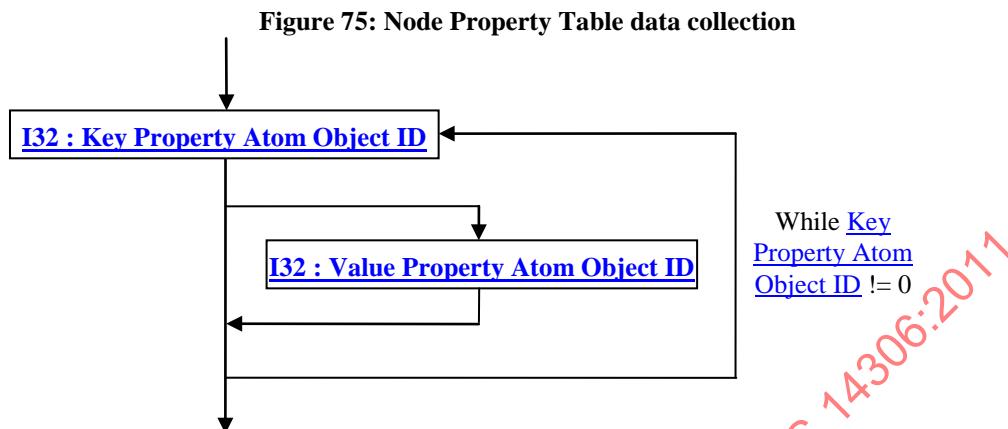
I32 : Key Property Atom Object ID

Key Property Atom Object ID is the identifier for the Property Atom Element object (see 6.2.1.2Property 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 6.2.1.2Property Atom Elements) representing the "value" part of the property key/value pair. A value is not stored if Key Property Atom Object ID has a value of "0".

6.2.2 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 6.2.1.1.1.10 Shape Node Elements and 6.2.1.2.7 Late Loaded Property Atom Element respectively).

Figure 76: Shape LOD Segment data collection
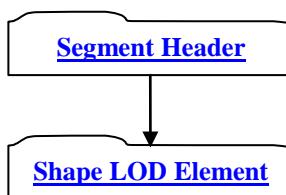
Complete description for Segment Header can be found in 6.1.3.1Segment Header.

6.2.2.1 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.

6.2.2.1.1 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 77: Vertex Shape LOD Element data collection
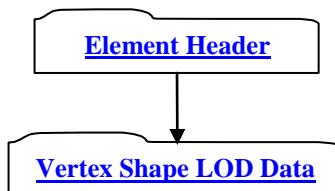
Complete description for Element Header can be found in 6.1.3.2.1Element Header.

6.2.2.1.1.1 Vertex Shape LOD Data

Vertex Shape LOD Data collection contains the bindings and quantization settings for all shape LODs defined by a collection of vertices.

Figure 78: Vertex Shape LOD Data data collection
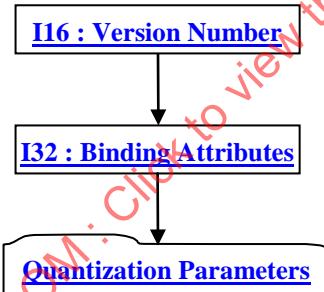
Complete description for Quantization Parameters can be found in 6.2.1.1.1.10.2.1.1Quantization Parameters.

I16 : Version Number
Version Number is the version identifier for this Vertex Shape LOD Data. Version number "0x0001" is currently the only valid value.

I32 : Binding Attributes
Binding Attributes is a collection of normal, texture coordinate, and color binding information encoded within a single I32 using the following bit allocation. All undocumented bits are reserved.

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

Bits 0 - 7	Normal Binding. Normal Binding specifies how (at what granularity) normal vector data is supplied ("bound") for the Shape LOD.
	= 0 – None. Shape has no normal data.
	= 1 – Per Vertex. Shape has a normal vector for every vertex.
	= 2 – Per Facet. Shape has a normal vector for every face/polygon.
	= 3 – Per Primitive. Shape has a normal vector for each shape primitive (e.g. a

	6.2.1.1.1.10.3 Tri-Strip Set Shape Node Element is made up of a collection of independent and unconnected triangle strips; where each strip constitutes one primitive of the shape and thus there would be a normal per triangle strip)
Bits 8 - 15	Texture Coordinate Binding. Texture Coordinate Binding specifies how (at what granularity) texture coordinate data is supplied ("bound") for the Shape LOD. Valid values are the same as documented for Normal Binding.
Bits 16 - 23	Color Binding. Color Binding specifies how (at what granularity) color data is supplied ("bound") for the Shape LOD. Valid values are the same as documented for Normal Binding.

6.2.2.1.2 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 (identified in Vertex Based Shape Compressed Rep Data as making up a single tri-strip primitive) in forming triangles, is the same as OpenGL's triangle strip definition [4].

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 6.2.1.1.1.10.3 Tri-Strip Set Shape Node Element and 6.2.1.2.7 Late Loaded Property Atom Element respectively).

Figure 79: Tri-Strip Set Shape LOD Element data collection
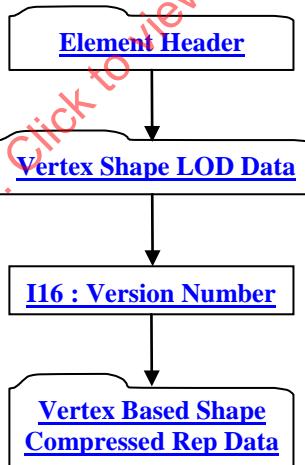
Complete description for Element Header can be found in 6.1.3.2.1Element Header.

Complete description for Vertex Shape LOD Data can be found in 6.2.2.1.1.1Vertex Shape LOD Data.

Complete description for Vertex Based Shape Compressed Rep Data can be found in 7.1.3Vertex Based Shape Compressed Rep Data.

I16 : Version Number

Version Number is the version identifier for this Tri-Strip Set Shape LOD. Version number "0x0001" is currently the only valid value.

6.2.2.1.3 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 6.2.1.1.1.10.5 Polyline Set Shape Node Element and 6.2.1.2.7 Late Loaded Property Atom Element respectively).

Figure 80: Polyline Set Shape LOD Element data collection
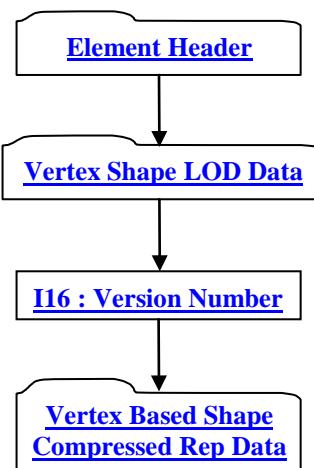
Complete description for Element Header can be found in 6.1.3.2.1Element Header.

Complete description for Vertex Shape LOD Data can be found in 6.2.2.1.1.1Vertex Shape LOD Data.

Complete description for Vertex Based Shape Compressed Rep Data can be found in 7.1.3Vertex Based Shape Compressed Rep Data.

I16 : Version Number
Version Number is the version identifier for this Polyline Set Shape LOD. Version number "0x0001" is currently the only valid value.

6.2.2.1.4 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 6.2.1.1.1.10.5 Point Set Shape Node Element and 6.2.1.2.7 Late Loaded Property Atom Element respectively).

Figure 81: Point Set Shape LOD Element data collection
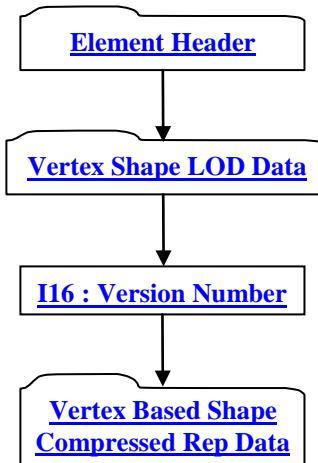
Complete description for Element Header can be found in 6.1.3.2.1Element Header.

Complete description for Vertex Shape LOD Data can be found in 6.2.2.1.1.1Vertex Shape LOD Data.

Complete description for Vertex Based Shape Compressed Rep Data can be found in 7.1.3Vertex Based Shape Compressed Rep Data.

I16 : Version Number

Version Number is the version identifier for this Point Set Shape LOD. Version number "0x0001" is currently the only valid value.

6.2.2.1.5 Polygon Set Shape LOD Element

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

A Polygon 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 polygons. Each polygon constitutes one primitive of the set.

A Polygon Set Shape LOD Element is typically referenced by a Polygon Set Shape Node Element using Late Loaded Property Atom Elements (see 6.2.1.1.1.10.6 Polygon Set Shape Node Element and 6.2.1.2.7 Late Loaded Property Atom Element respectively).

Figure 82: Polygon Set Shape LOD Element data collection
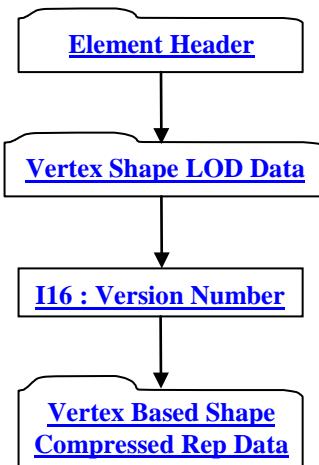
```
┌─────────────────────┐
│   Element Header     │
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│ Vertex Shape LOD Data│
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│ I16 : Version Number │
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│  Vertex Based Shape  │
│ Compressed Rep Data  │
└─────────────────────┘
```

Complete description for Element Header can be found in 6.1.3.2.1Element Header.

Complete description for Vertex Shape LOD Data can be found in 6.2.2.1.1.1Vertex Shape LOD Data.

Complete description for Vertex Based Shape Compressed Rep Data can be found in 7.1.3Vertex Based Shape Compressed Rep Data.

I16 : Version Number

Version Number is the version identifier for this Polygon Set Shape LOD. Version number "0x0001" is currently the only valid value.

6.2.2.1.6 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 6.2.1.1.1.10.7 NULL Shape Node Element and 6.2.1.2.7 Late Loaded Property Atom Element respectively).

Figure 83: Null Shape LOD Element data collection
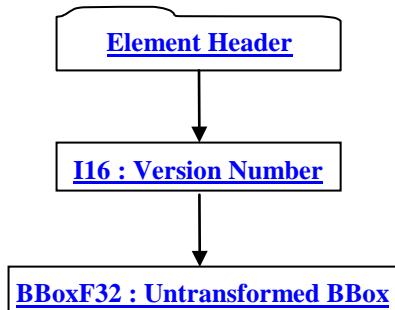
```
┌─────────────────────┐
│   Element Header     │
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│ I16 : Version Number │
└─────────────────────┘
           │
           ▼
┌─────────────────────────┐
│ BBoxF32 : Untransformed BBox │
└─────────────────────────┘
```

Complete description for Element Header can be found in 6.1.3.2.1Element Header.

I16 : Version Number
Version Number is the version identifier for this Null Shape LOD Element. Version number "0x0001" is currently the only valid value.

BBoxF32 : Untransformed BBox
The Untransformed BBox is an axis-aligned LCS bounding box and represents the untransformed extents for this Null Shape LOD Element.

6.2.2.2 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.). "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 center and radius) from which LODs can be generated.

Figure 84: Primitive Set Shape Element data collection
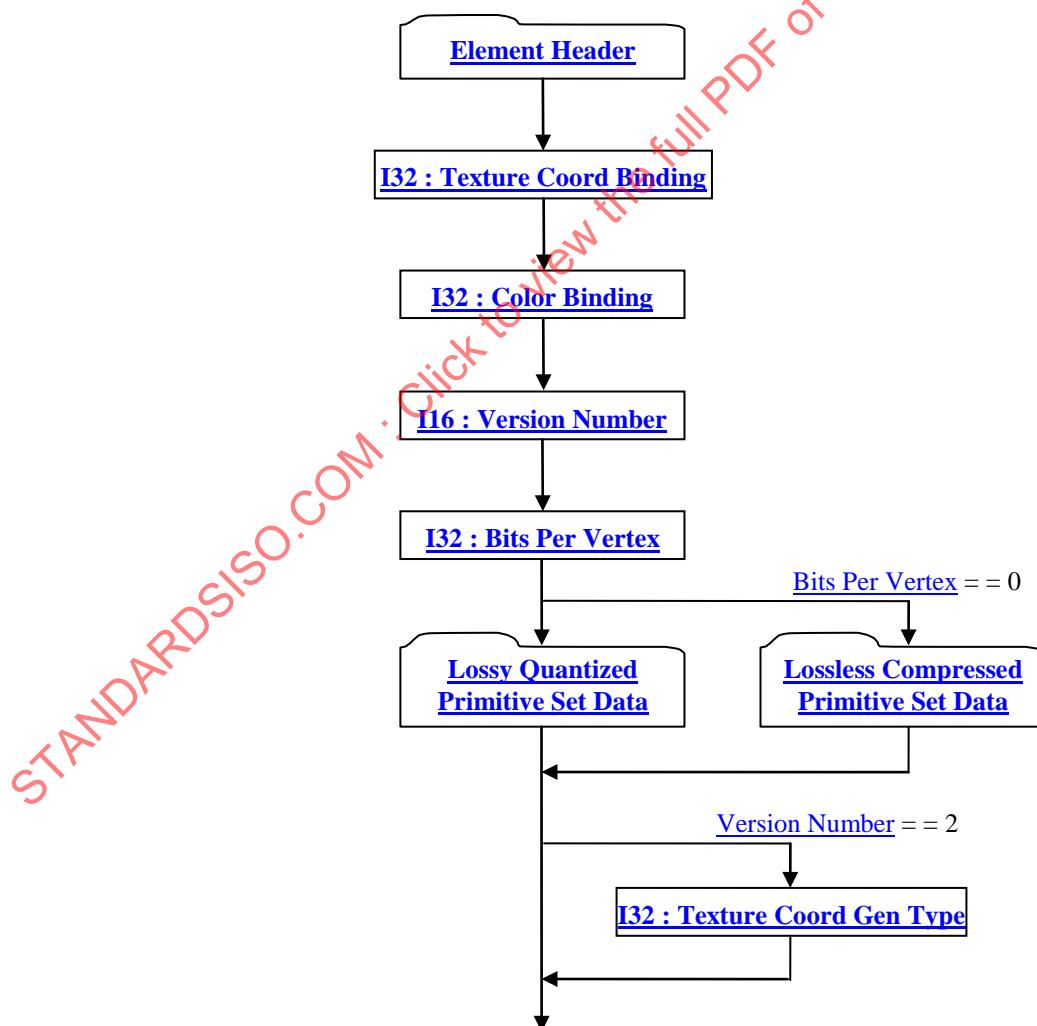
Complete description for Element Header can be found in 6.1.3.2.1Element Header.

I32 : Texture Coord Binding

Texture Coord Binding specifies how (at what granularity) texture coordinate data is supplied ("bound") for the shape. Valid values are as follows:

= 0	– None. Shape has no texture coordinate data.
= 1	– Per Vertex. Shape has texture coordinates for every vertex.

I32 : Color Binding

Color Binding specifies how (at what granularity) color data is supplied ("bound") for the shape. Valid values are the same as documented for Texture Coord Binding data field.

I16 : 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.

= 1	– Version-1 Format
= 2	– Version-2 Format

I32 : Bits Per Vertex

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

I32 : Texture Coord Gen Type

Texture Coord Gen Type specifies how texture coordinates are to be generated.

= 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.

6.2.2.2.1 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 6.2.2.2 Primitive Set Shape Element for complete description).

Figure 85: Lossless Compressed Primitive Set Data data collection
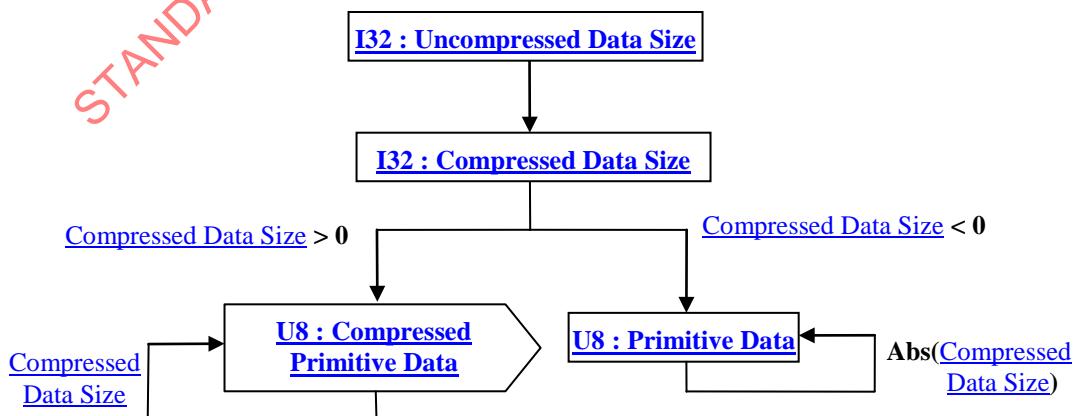
I32 : Uncompressed Data Size

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, color, 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], [color], [type]}, …, **for all primitives**

Where the data elements have the following size and meaning:

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)
color	RGB	- Red, Green, Blue color component values
type	I32	- Primitive Type = 0 – Box = 1 – Cylinder = 2 – Pyramid = 3 – Sphere = 4 – Tri-Prism

Table 5: Primitive Set Primitive Data Elements

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:

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 center X	base center Y	base center Z	spine X	spine Y	spine Z	radius 1	radius 2	N/A	N/A
Pyramid	base center X	base center Y	base center Z	length X	length Y	length Z	orientation in Quaternion form			
Sphere	center X	center Y	center 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			

Table 6: Primitive Set "params#" Data Fields Interpretation

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 "ZLIB deflation compression" method. The Compressed Primitive Data field is only present if Compressed Data Size value is greater than zero. See 7 Data Compression and Encoding for more details on ZLIB compression and ZLIB library version used.

6.2.2.2.2 Lossy Quantized Primitive Set Data

The Lossy Quantized Primitive Set Data collection contains all the per-primitive information (i.e. reserved, params1, params2, params3, color, 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 6.2.2.2 Primitive Set Shape Element for compete description).

The interpretation of the three per-primitive "params#" data fields is primitive type dependent. See Table 6: Primitive Set "params#" Data Fields Interpretation in 6.2.2.2.1 Lossless Compressed Primitive Set Data for per-primitive type description of the "params#" data fields.

Figure 86: Lossy Quantized Primitive Set Data data collection
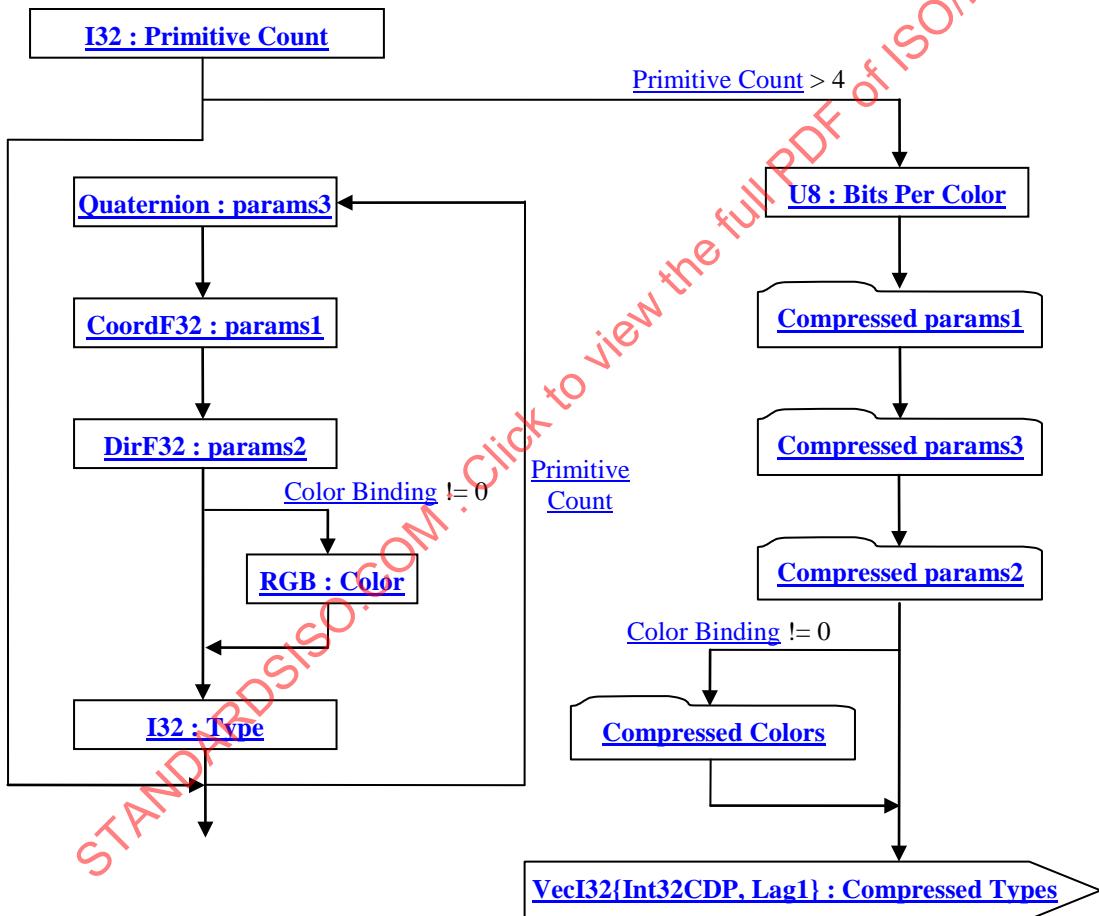
I32 : Primitive Count

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

Quaternion : params3

Interpretation of params3 data field is primitive Type dependent. See Table 6: Primitive Set "params#" Data Fields Interpretation in 6.2.2.2.1 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 6: Primitive Set "params#" Data Fields Interpretation in 6.2.2.2.1 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 6: Primitive Set "params#" Data Fields Interpretation in 6.2.2.2.1 Lossless Compressed Primitive Set Data for per-primitive type description of the params1 data fields.

RGB : Color

Color specifies the Red, Green Blue color components for the primitive. This data field is only present if previously read Color Binding (see 6.2.2.2 Primitive Set Shape Element) is not equal to "0".

I32 : Type

Type specifies the primitive type. See Table 5: Primitive Set Primitive Data Elements in 6.2.2.2.1 Lossless Compressed Primitive Set Data for valid primitive Type values.

U8 : Bits Per Color

Bits Per Color specifies the number of quantization bits per color component. Value must 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 5: Primitive Set Primitive Data Elements in 6.2.2.2.1 Lossless Compressed Primitive Set Data.

6.2.2.2.2.1 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 6: Primitive Set "params#" Data Fields Interpretation in 6.2.2.2.1 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 7 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.

Figure 87: Compressed params1 data collection

VecF32 : Quantization Range Min/Max Pairs

↓

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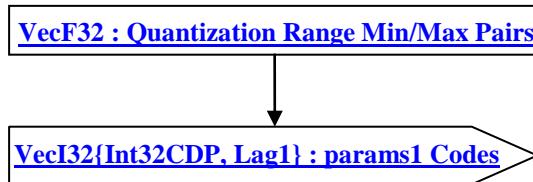
VecF32 : Quantization Range Min/Max Pairs

Quantization Range Min/Max Pairs is a vector of Uniform Quantizer range min/max value pairs. There must 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.

6.2.2.2.2.2 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 6: Primitive Set "params#" Data Fields Interpretation in 6.2.2.2.1 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 7 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 87: Compressed params1 data collection.

6.2.2.2.2.3 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 6: Primitive Set "params#" Data Fields Interpretation in 6.2.2.2.1 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 "DirF32", 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 7 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 87: Compressed params1 data collection.

6.2.2.2.2.4 Compressed Colors

Compressed Colors is the compressed representation of the *color* data for all the primitives in the Primitive Set. This data collection is only present if previously read Color Binding (see 6.2.2.2 Primitive Set Shape Element) is not equal to "0".

The *color* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Color number of quantization bits) for each collection of ordinate values. Since *color* 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 7 Data Compression and Encoding for more complete description of Uniform Quantizer.

The JT Format packs all the *color* 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 color[0], prim2 color[0],…primN color[0],
prim1 color[1], prim2 color[1],…primN color[1],
prim1 color[2], prim2 color[2],…primN color[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 *color* quantization codes that corresponds to the above described "ordinate dependent order" packed array of *color* data.

The storage format of Compressed Colors is exactly the same as that documented in Figure 87: Compressed params1 data collection.

6.2.2.3 Wire Harness Set Shape Element

Object Type ID: 0x4cc7a523, 0x728, 0x11d3, 0x9d, 0x8b, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2

A Wire Harness Set Shape Element defines the data necessary to procedurally generate LODs for a list of wire harnesses. "Procedurally generate" means that the raw geometric shape definition data (e.g. vertices, polygons, normals, etc) for LODs is not directly stored; instead some descriptive shape information is stored from which LODs can be generated at load time.

Figure 88: Wire Harness Set Shape Element data collection
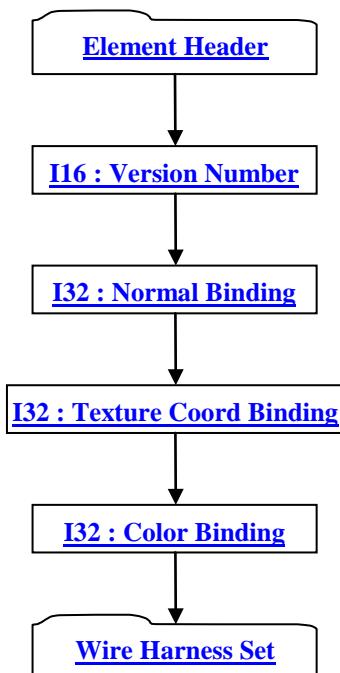
Complete description for Element Header can be found in 6.1.3.2.1Element Header.

I16 : Version Number

Version Number is the version identifier for this Wire Harness Set Shape Element. Version number "0x0001" is currently the only valid value.

I32 : Normal Binding

Normal Binding specifies how (at what granularity) normal vector data is supplied ("bound") for the shape. Valid values include the following:

= 0	– None. Shape has no normal data.
= 1	– Per Vertex. Shape has a normal vector for every vertex.

I32 : Texture Coord Binding

Texture Coord Binding specifies how (at what granularity) texture coordinate data is supplied ("bound") for the shape. Valid values are the same as documented for Normal Binding data field.

I32 : Color Binding

Color Binding specifies how (at what granularity) color data is supplied ("bound") for the shape. Valid values are the same as documented for Normal Binding data field.

6.2.2.3.1 Wire Harness Set

A Wire Harness Set defines a topological and geometric representation of a set/list of wire harnesses. Each wire harness in the set is a single manufactured wire unit consisting of several physical electrical wires all bound together into a branching structure of wire bundles that terminate at connectors. Note that only the wires are modeled by the Wire Harness Set, not any of the physical representation of the connectors or wrapping material used to secure the harness.

The topology of a wire Harness can be viewed as a non-directed acyclic graph. Each node of the graph is represented in 3D space by a Branch Node. Each edge of the graph is represented in 3D space by a Bundle. Physical Wires are defined to trace a path from one leaf Branch Node in the graph to another leaf Branch Node in the graph using an ordered list of Wire Segments. So essentially there are two topologies modeled, the graph topology of the Bundles, and the topology of the Wires.

The geometry of the wire Harness defines the 3D location of the Branch Nodes, the 3D spine curve (path) of the Bundles, and the physical radius and multiplicity of the Wires. These geometry definitions are referenced though the topology structure.

Figure 89: Wire Harness Set data collection
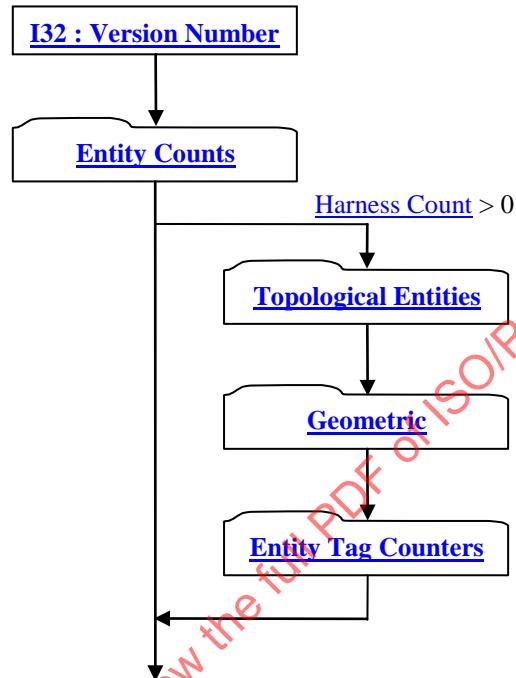
I32 : Version Number
Version Number is the version identifier for this Wire Harness Set. Version number "1" is currently the only valid value.

6.2.2.3.1.1 Entity Counts
Entity Counts Specifies the counts for (number of) each of the entity types which exists within a set of wire harnesses.

Figure 90: Entity Counts data collection

I32 : Harness Count

I32 : Bundle Count

I32 : Wire Count

I32 : Wire Segment Count

I32 : Branch Node Count

I32 : Bundle Spine Curve Count

I32 : Branch Point Count
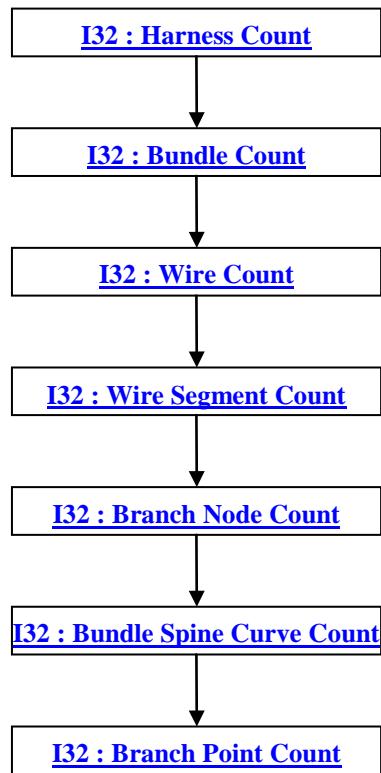
I32 : Harness Count
Harness Count specifies the number of Wire Harness entities in the list.

I32 : Bundle Count
Bundle Count specifies the number of Bundle entities in the list.

I32 : Wire Count
Wire Count specifies the number of Wire entities in the list.

I32 : Wire Segment Count
Wire Segment Count specifies the number of Wire Segment entities in the list.

I32 : Branch Node Count
Branch Node Count specifies the number of Branch Node entities in the list.

I32 : Bundle Spine Curve Count
Bundle Spine Curve Count specifies the number of Bundle Spine Curve entities in the list.

I32 : Branch Point Count
Branch Point Count specifies the number of Branch Point entities in the list.

6.2.2.3.1.2 Topological Entities

Figure 91: Topological Entities data collection
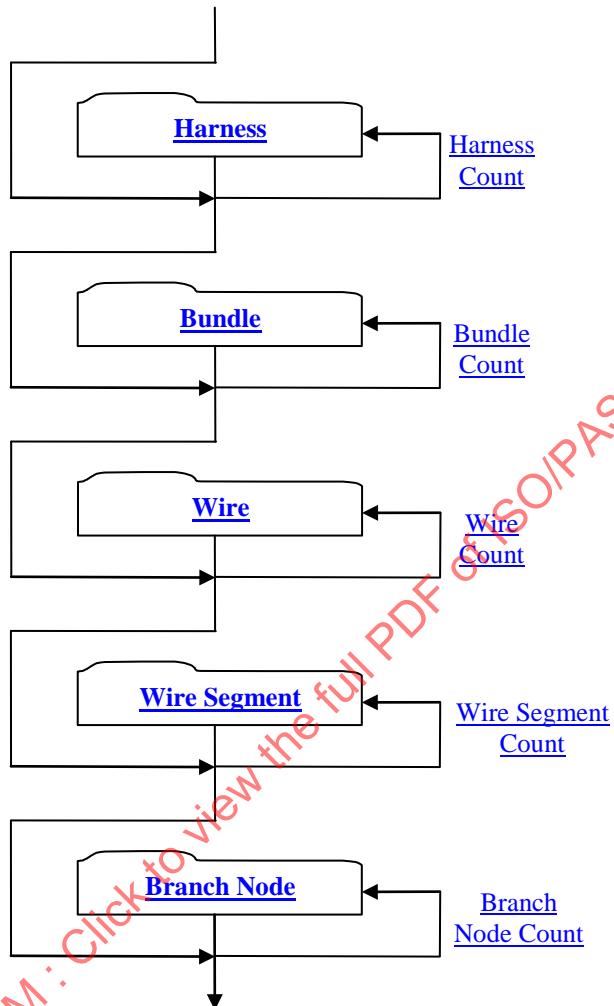
6.2.2.3.1.2.1 Harness

A Harness is single manufactured wire unit consisting of several physical electrical wires all bound together into a branching structure of wire bundles that terminate at connectors. A Harness is made up of an ordered set of Bundles, an ordered set of Branch Nodes, and a set of Wires.

Figure 92: Harness data collection

```
┌──────────────────────────────┐
│   I32 : Harness Data1         │
└──────────────────────────────┘
              │
              ▼
┌──────────────────────────────┐
│   I32 : Harness Data2         │
└──────────────────────────────┘
              │
              ▼
┌──────────────────────────────┐
│   I32 : First Branch Node Index │
└──────────────────────────────┘
              │
              ▼
┌──────────────────────────────┐
│   I32 : Last Branch Node Index  │
└──────────────────────────────┘
              │
              ▼
┌──────────────────────────────┐
│   I32 : First Wire Index      │
└──────────────────────────────┘
              │
              ▼
┌──────────────────────────────┐
│   I32 : Last Wire Index       │
└──────────────────────────────┘
```
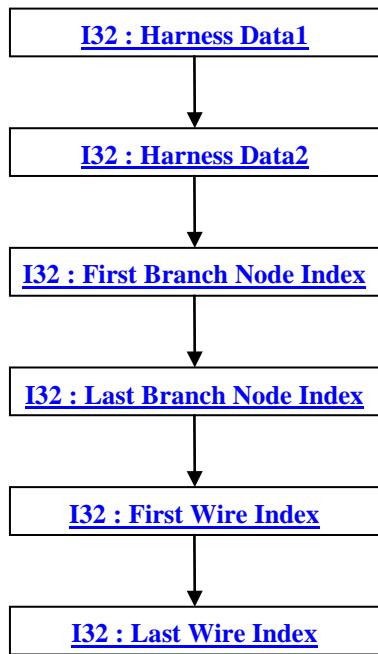
I32 : Harness Data1

Harness Data1 is a collection of Harness information encoded/packed within a single I32 using the following bit allocation. All undocumented bits are reserved.

Bits 0 - 23	Specifies the index into the list of Bundles of the first Bundle in the Harness. All Bundles inclusive between first Bundle index and last Bundle index (see Harness Data2) are part of this Harness
Bits 24 - 30	Represents bits 8-14 of the I16 Harness tag identifier. Note that bits 0-7 of the I16 Harness tag identifier can be found in Harness Data2 data field. Using C-language syntax the complete Harness tag identifier can be built as follows: Tag = ((Harness Data1 & 0x7f000000) >> 16) | ((Harness Data2 & 0xff000000) >> 24)

I32 : Harness Data2

Harness Data2 is a collection of Harness information encoded/packed within a single I32 using the following bit allocation. All undocumented bits are reserved.

Bits 0 - 23	Specifies the index into the list of Bundles of the last Bundle in the Harness. All Bundles inclusive between first Bundle index (see Harness Data1) and last Bundle index are part of this Harness
Bits 24 - 31	Represents bits 0-7 of the I16 Harness tag identifier. Note that bits 8-14 of the I16 Harness tag identifier can be found in Harness Data1 data field. Using C-language syntax the complete Harness tag identifier can be built as follows: Tag = ((Harness Data1 & 0x7f000000) >> 16) | ((Harness Data2 & 0xff000000) >> 24)

I32 : First Branch Node Index

First Branch Node Index specifies the index into the list of Branch Nodes of the first Branch Node in the Harness. All Branch Nodes inclusive between First Branch Node Index and Last Branch Node Index are part of this Harness.

I32 : Last Branch Node Index

Last Branch Node Index specifies the index into the list of Branch Nodes of the Last Branch Node in the Harness. All Branch Nodes inclusive between First Branch Node Index and Last Branch Node Index are part of this Harness.

I32 : First Wire Index

First Wire Index specifies the index into the list of Wires of the first Wire in the Harness. All Wires inclusive between First Wire Index and Last Wire Index are part of this Harness.

I32 : Last Wire Index

Last Wire Index specifies the index into the list of Wires of the last Wire in the Harness. All Wires inclusive between First Wire Index and Last Wire Index are part of this Harness.

6.2.2.3.1.2.2 Bundle

A Bundle models the group of wires that span between two Branch Nodes in a Harness. A Bundle is composed of a start and end Branch Node which models the topological points at which a Harness branches into two or more Bundles. A Bundle also refers to a Bundle Spine Curve which represents the 3D path that the Bundle centerline makes in space. Note that only the Bundle centerline is modeled; the routing of each individual wire in the Bundle is considered immaterial (i.e. left to a system processing this data to arrange how it sees fit). The Bundle Spine Curve ends must coincide with the Bundle's start and end Branch Nodes.

Figure 93: Bundle data collection
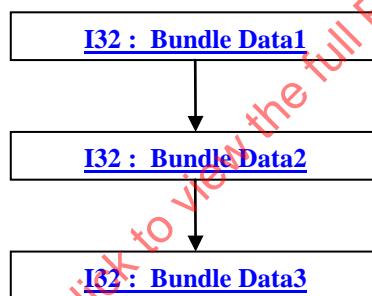
I32 : Bundle Data1

Bundle Data1 is a collection of Bundle information encoded/packed within a single I32 using the following bit allocation. All undocumented bits are reserved.

Bits 0 - 23	Specifies the index into the list of Branch Nodes of the start Branch Node for the Bundle.
Bits 24 - 30	Represents bits 16-22 of the 23 bit Bundle tag identifier. Note that bits 8-15 and bits 0-7 of the 23 bit Bundle tag identifier can be found in Bundle Data2 and Bundle Data3 data fields respectively. Using C-language syntax the complete Bundle tag identifier can be built as follows: Tag = ((Bundle Data1 & 0x7f000000) >> 8) | ((Bundle Data2 & 0xff000000) >> 16) | ((Bundle Data3 & 0xff000000) >> 24)

I32 : Bundle Data2

Bundle Data2 is a collection of Bundle information encoded/packed within a single I32 using the following bit allocation. All undocumented bits are reserved.

Bits 0 - 23	Specifies the index into the list of Branch Nodes of the end Branch Node for the Bundle.
Bits 24 - 31	Represents bits 8-15 of the 23 bit Bundle tag identifier. Note that bits 16-22 and bits 0-7 of the 23 bit Bundle tag identifier can be found in Bundle Data1 and Bundle Data3 data fields respectively. Using C-language syntax the complete Bundle tag identifier can be built as follows: Tag = ((Bundle Data1 & 0x7f000000) >> 8) | ((Bundle Data2 & 0xff000000) >> 16) | ((Bundle Data3 & 0xff000000) >> 24)

I32 : Bundle Data3

Bundle Data3 is a collection of Bundle information encoded/packed within a single I32 using the following bit allocation. All undocumented bits are reserved.

Bits 0 - 23	Specifies the index into the list of Bundle Spine Curves of the spine curve associated with the Bundle.
Bits 24 - 31	Represents bits 0-7 of the 23 bit Bundle tag identifier. Note that bits 16-22 and bits 8-15 of the 23 bit Bundle tag identifier can be found in Bundle Data1 and Bundle Data2 data fields respectively. Using C-language syntax the complete Bundle tag identifier can be built as follows: Tag = ((Bundle Data1 & 0x7f000000) >> 8) | ((Bundle Data2 & 0xff000000) >> 16) | ((Bundle Data3 & 0xff000000) >> 24)

6.2.2.3.1.2.3 Wire

A Wire models the contiguous nature of a single strand of wire from beginning to end. A Wire passes through an ordered list of Bundles as identified by the ordered list of Wire Segments composing a Wire. In order to reduce data explosion, and leverage the fact that many Wires in a Harness have the same physical representation, semantic meaning, and start/end branch nodes, the Wire entity supports a multiplicity factor. This multiplicity factor allows a single Wire entity to represent more than one discrete physical wire in a Harness. Only physical wires of circular cross-section are modeled by a Wire entity.

Figure 94: Wire data collection
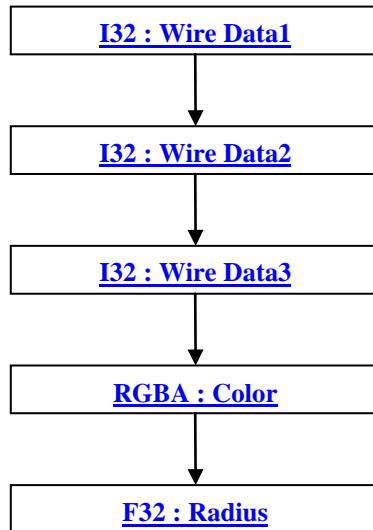
I32 : Wire Data1

Wire Data1 is a collection of Wire information encoded/packed within a single I32 using the following bit allocation. All undocumented bits are reserved.

Bits 0 - 23	Specifies the index into the list of Wire Segments of the first Wire Segment in the Wire. All Wire Segments inclusive between first Wire Segment index and last Wire Segment index (see Wire Data2) are part of this Wire
Bits 24 - 30	Represents bits 16-22 of the 23 bit Wire tag identifier. Note that bits 8-15 and bits 0-7 of the 23 bit Wiretag identifier can be found in Wire Data2 and Wire Data3 data fields respectively. Using C-language syntax the complete Wire tag identifier can be built as follows: Tag = ((Wire Data1 & 0x7f000000) >> 8) | ((Wire Data2 & 0xff000000) >> 16) | ((Wire Data3 & 0xff000000) >> 24)

I32 : Wire Data2

Wire Data2 is a collection of Wire information encoded/packed within a single I32 using the following bit allocation. All undocumented bits are reserved.

Bits 0 - 23	Specifies the index into the list of Wire Segments of the last Wire Segment in the Wire. All Wire Segments inclusive between first Wire Segment index (see Wire Data1) and last Wire Segment index are part of this Wire
Bits 24 - 31	Represents bits 8-15 of the 23 bit Wire tag identifier. Note that bits 16-22 and bits 0-7 of the 23 bit Wire tag identifier can be found in Wire Data1 and Wire Data3 data fields respectively. Using C-language syntax the complete Wire tag identifier can be built as follows: Tag = ((Wire Data1 & 0x7f000000) >> 8) | ((Wire Data2 & 0xff000000) >> 16) | ((Wire Data3 & 0xff000000) >> 24)

I32 : Wire Data3

Wire Data3 is a collection of Wire information encoded/packed within a single I32 using the following bit allocation. All undocumented bits are reserved.

Bits 0 - 23	Specifies the Wire multiplicity factor. The multiplicity factor is used to allow a single Wire to represent multiple (more than one) discrete physical wires that have the same semantic meaning, attributes and starting/ending Wire Segment.
Bits 24 - 31	Represents bits 0-7 of the 23 bit Wire tag identifier. Note that bits 16-22 and bits 8-15 of the 23 bit Wire tag identifier can be found in Wire Data1 and Wire Data2 data fields respectively. Using C-language syntax the complete Wire tag identifier can be built as follows: Tag = ((Wire Data1 & 0x7f000000) >> 8) | ((Wire Data2 & 0xff000000) >> 16) | ((Wire Data3 & 0xff000000) >> 24)

RGBA : Color

Color specifies the Red, Green, Blue, and Alpha components of the Wire color.

F32 : Radius

Radius specifies the physical radius of the Wire (thus only physical Wires of circular cross-section can be modeled).

6.2.2.3.1.2.4 Wire Segment

A Wire Segment indirectly models the inclusion of a Wire in a particular Bundle (i.e. that a Wire passes through a particular Bundle).

Figure 95: Wire Segment data collection

I32 : Bundle Index

Bundle Index specifies the index into the list of Bundles of the Bundle this Wire Segment is associated with.

I32 : Tag

Tag specifies the tag identifier for the Wire Segment entity.

6.2.2.3.1.2.5 Branch Node

Leaf Branch Nodes model the topological terminating points of a Harness (i.e. where a connector would be) while internal Branch Nodes model the topological points at which a Harness branches into two or more Bundles.

Figure 96: Branch Node data collection

```
┌─────────────────────────────────┐
│    I32 : Branch Point Index      │
└─────────────────────────────────┘
                 │
                 ▼
┌─────────────────────────────────┐
│            I32 : Tag             │
└─────────────────────────────────┘
```
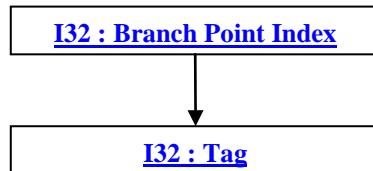
I32 : Branch Point Index

Branch Point Index specifies the index into the list of Branch Points of the Branch Point associated with the Branch Node.

I32 : Tag

Tag specifies the tag identifier for the Branch Node entity.

6.2.2.3.1.3 Geometric Entities

Figure 97: Geometric data collection

```
┌──────────────────────────┐
│    Bundle Spine Curve     │◄────  Bundle
└──────────────────────────┘        Spine Curve
                                    Count

┌──────────────────────────┐
│ CoordF32 : Branch Point Coord │◄──  Branch Point
└──────────────────────────┘        Count
```
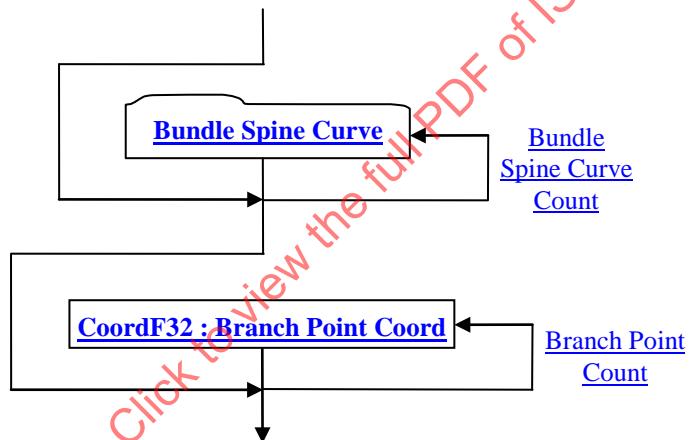
CoordF32 : Branch Point Coord

Branch Point Coord specifies the XYZ local coordinate system point coordinates for the Branch Point.

6.2.2.3.1.3.1 Bundle Spine Curve

A Bundle Spine Curve is referenced by a Bundle and represents the precise 3D geometric path that the referencing Bundle's centerline makes in space

Figure 98: Bundle Spine Curve data collection
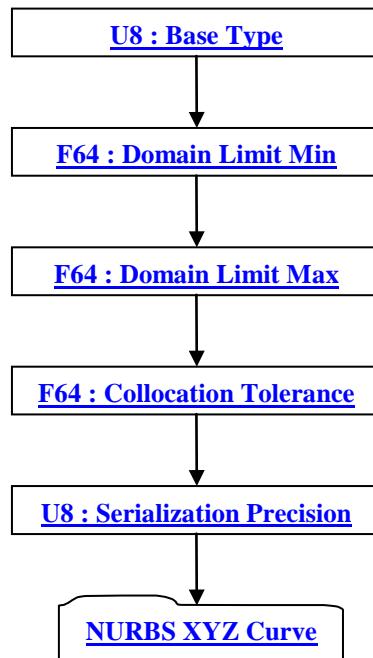
U8 : Base Type

Base Type specifies the curve base type identifier. 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. Valid Base Type values include the following:

= 1	– Curve is a NURBS curve

F64 : Domain Limit Min

Domain Limit Min specifies the minimum value placed on the parametric domain. A value of "-1.0" indicates that no additional domain limit has been set on the real parametric domain.

F64 : Domain Limit Max

Domain Limit Max specifies the maximum value placed on the parametric domain. A value of "-1.0" indicates that no additional domain limit has been set on the real parametric domain.

F64 : Collocation Tolerance

Collocation Tolerance specifies the tolerance to be used for determining whether two points are collocated. A value of "-1" indicates that this tolerance has not been set

U8 : Serialization Precision

Serialization Precision specifies whether the curve data is serialized in single or double precision. Currently only double precision serialization is supported for Bundle Spine Curves, but a Serialization Precision value is still included in the specification to allow for future expansion of the JT Format to support single precision Bundle Spine Curves. Valid Serialization Precision values include the following:

= 0	– Double precision serialization

6.2.2.3.1.3.1.1 NURBS XYZ Curve

NURBS XYZ Curve data collection defines a single model space NURBS curve. This format for storing NURBS XYZ curves is only used within the Wire Harness Set Shape Element.

Figure 99: NURBS XYZ Curve data collection
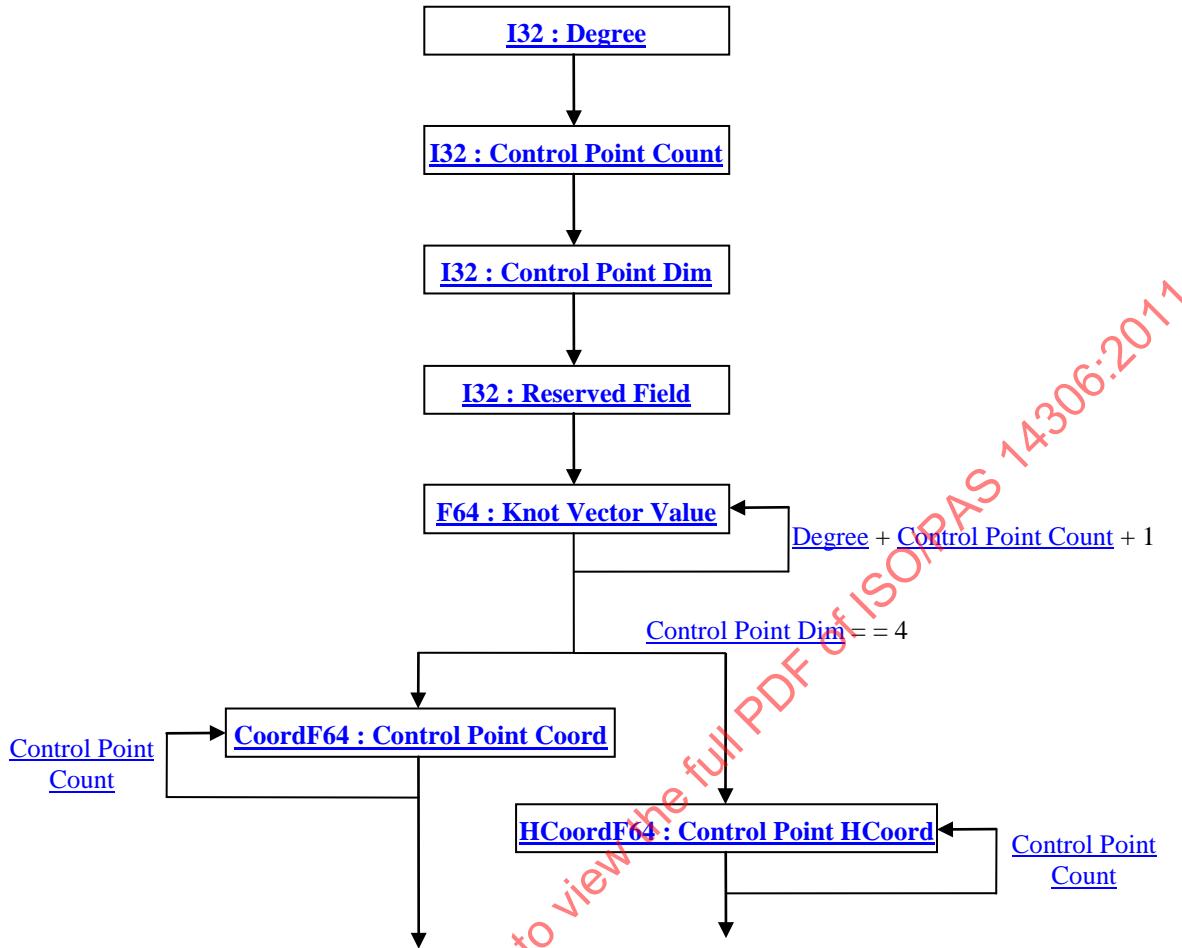
I32 : Degree

Degree specifies the NURBS curve degree. The degree value must be within the range [1:16] inclusive.

I32 : Control Point Count

Control Point Count specifies the number of control points for the NURBS curve.

I32 : Control Point Dim

Control Point Dim specifies the dimensionality of the control points. Valid dimensionality values include the following:

= 3	− Non-Rational (each control point has 3 coordinates)
= 4	− Rational (each control point has 4 coordinates)

I32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

F64 : Knot Vector Value

Knot Vector Value specifies a single value within the NURBS curve knot vector. There should be a total of "Degree + Control Point Count + 1" of these values. The list of these values forms the total NURBS curve knot vector which must be *clamped* and *non-decreasing*; where *clamped* means knot multiplicity of "degree + 1" at both the beginning and end of the knot vector.

CoordF64 : Control Point Coord
Control Point Coord specifies the XYZ coordinates for a single control point

HCoordF64 : Control Point HCoord
Control Point HCoord specifies the XYZW homogeneous coordinates for a single control point.

6.2.2.3.1.4 Entity Tag Counters

Entity Tag Counters data collection specifies the next available "unique" tag value for each topological entity type in a Wire Harness Set. These are rolling tag counters that are meant to be used for assigning a unique tag when a new entity is added to a Wire Harness Set.

Figure 100: Entity Tag Counters data collection
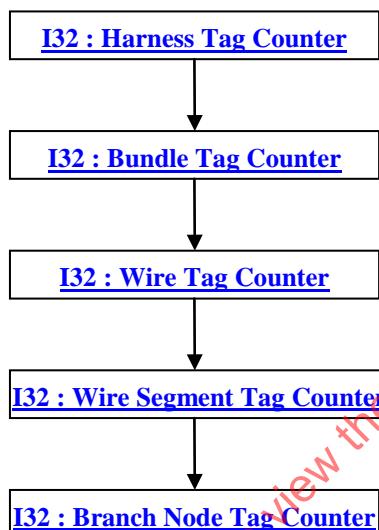
I32 : Harness Tag Counter
Harness Tag Counter specifies the next available "unique" tag value for Harness entity.

I32 : Bundle Tag Counter
Bundle Tag Counter specifies the next available "unique" tag value for Bundle entity.

I32 : Wire Tag Counter
Wire Tag Counter specifies the next available "unique" tag value for Wire entity.

I32 : Wire Segment Tag Counter
Wire Segment Tag Counter specifies the next available "unique" tag value for Wire Segment entity.

I32 : Branch Node Tag Counter
Branch Node Tag Counter specifies the next available "unique" tag value for Branch Node entity.

6.2.3 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 6.2.4 XT B-Rep Segment.

JT B-Rep Segments are typically referenced by Part Node Elements (see 6.2.1.1.1.5Part Node Element) using Late Loaded Property Atom Elements (see 6.2.1.2.7Late Loaded Property Atom Element). The JT B-Rep Segment type supports ZLIB compression on all element data, so all elements in JT B-Rep Segment use the Element Header ZLIB form of element header data.

Figure 101: JT B-Rep Segment data collection

```
┌─────────────────────┐
│  Segment Header     │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│  JT B-Rep Element   │
└─────────────────────┘
```
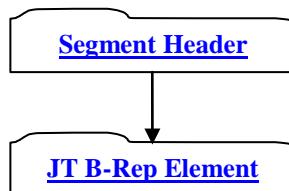
Complete description for Segment Header can be found in 6.1.3.1Segment Header.

6.2.3.1 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 [10] and [11] listed in 2 References and Additional Information section of this document.

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 102: JT B-Rep Element data collection
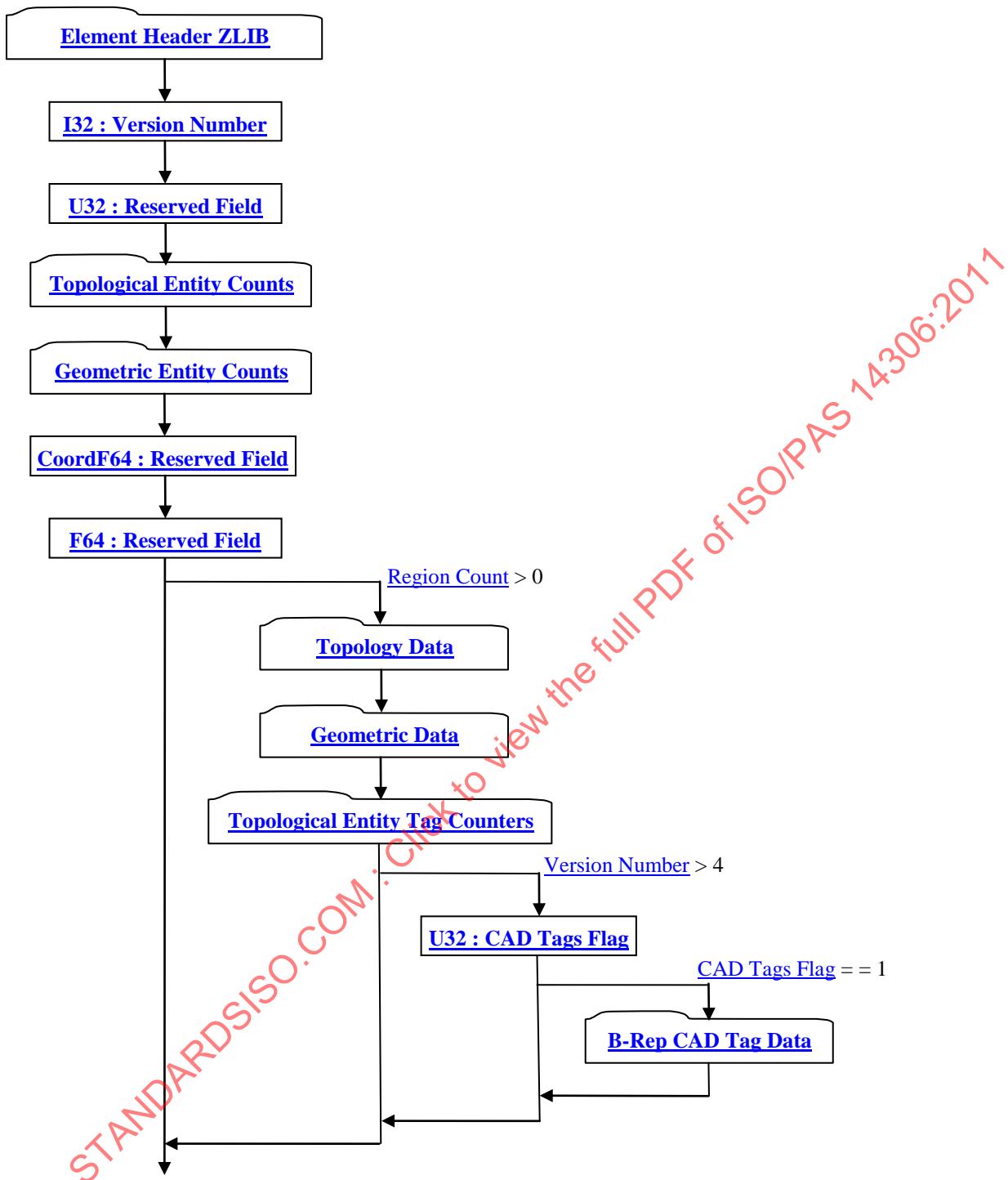
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

I32 : Version Number

Version Number is the version identifier for this JT B-Rep Element. Version numbers "4" and "5" are currently supported.

U32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

CoordF64 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

F64 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

U32 : CAD Tags Flag

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

6.2.3.1.1 Topological Entity Counts

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

Figure 103: Topological Entity Counts data collection

| I32 : Region Count |
| I32 : Shell Count |
| I32 : Face Count |
| I32 : Loop Count |
| I32 : CoEdge Count |
| I32 : Edge Count |
| I32 : Vertex Count |
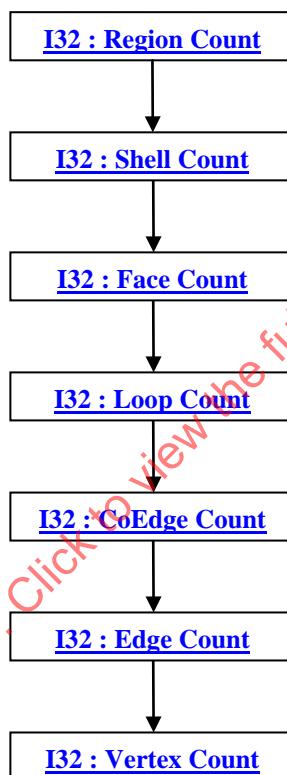
I32 : Region Count

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

6.2.3.1.2 Geometric Entity Counts

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

Figure 104: Geometric Entity Counts data collection

```
┌─────────────────────────┐
│   I32 : Surface Count    │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│  I32 : PCS Curve Count   │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│  I32 : MCS Curve Count   │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│    I32 : Point Count     │
└─────────────────────────┘
```
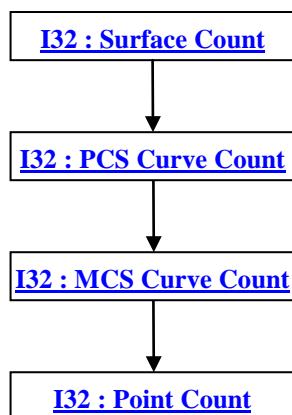
I32 : Surface Count

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.

6.2.3.1.3 Topology Data

Figure 105: Topology Data data collection
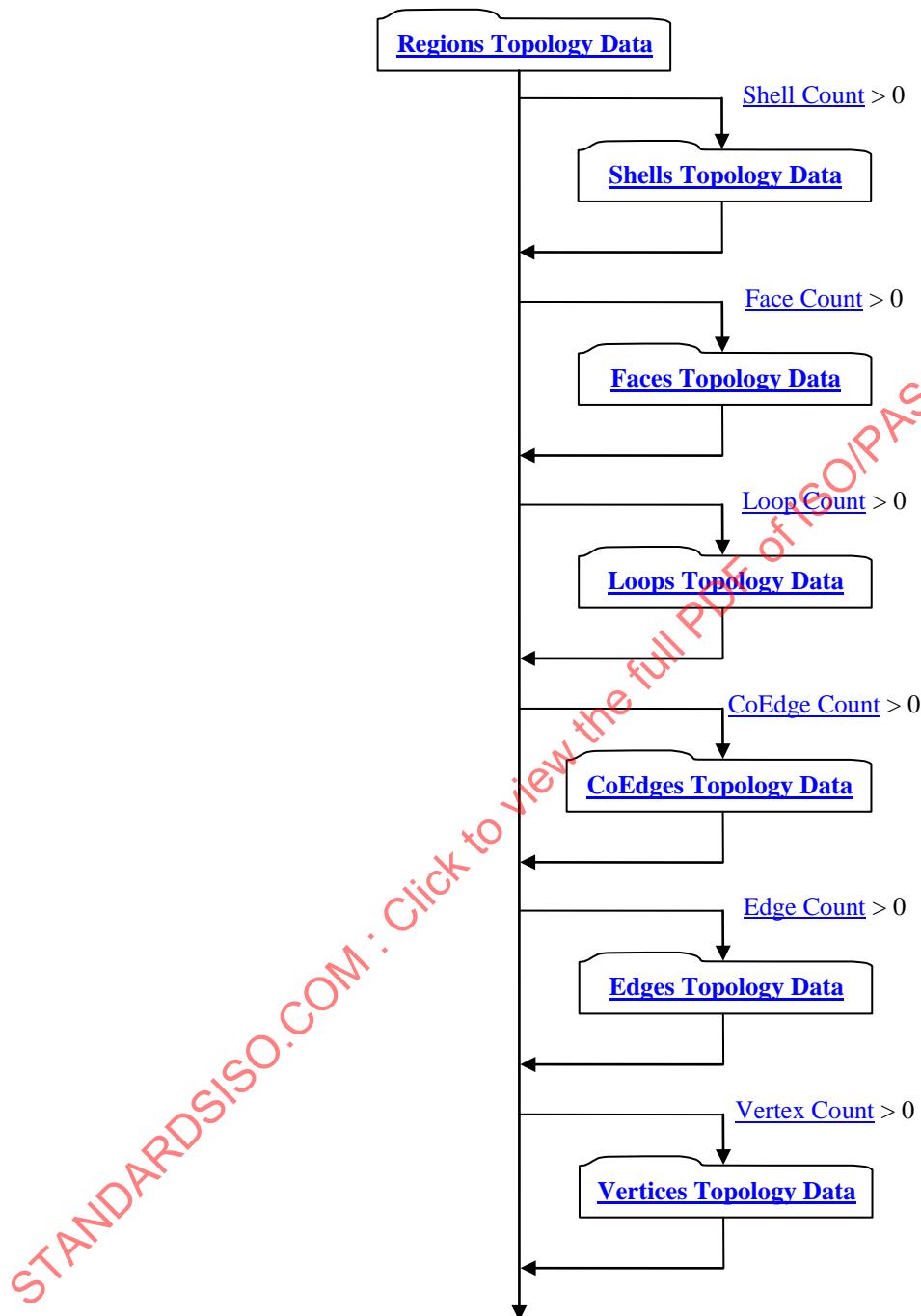
6.2.3.1.3.1 Regions Topology Data

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).

Figure 106: Regions Topology Data data collection

VecI32{Int32CDP, Lag1} : First Shell Indices

VecI32{Int32CDP, Lag1} : Last Shell Indices

VecI32{Int32CDP, Lag1} : Region Tags
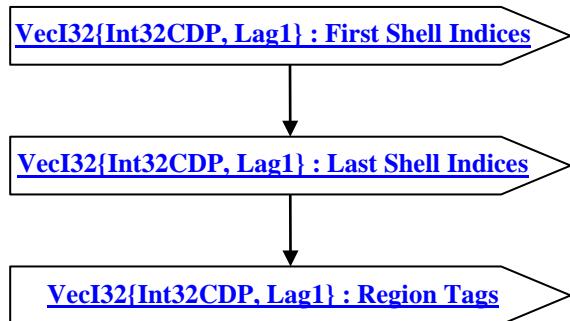
VecI32{Int32CDP, Lag1} : First Shell Indices

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.

6.2.3.1.3.2 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).

Figure 107: Shells Topology Data data collection

VecI32{Int32CDP, Lag1} : First Face Indices

VecI32{Int32CDP, Lag1} : Last Face Indices

VecI32{Int32CDP, Lag1} : Shell Tags

VecI32{Int32CDP, Xor1} : Shell Anti-Hole Flags
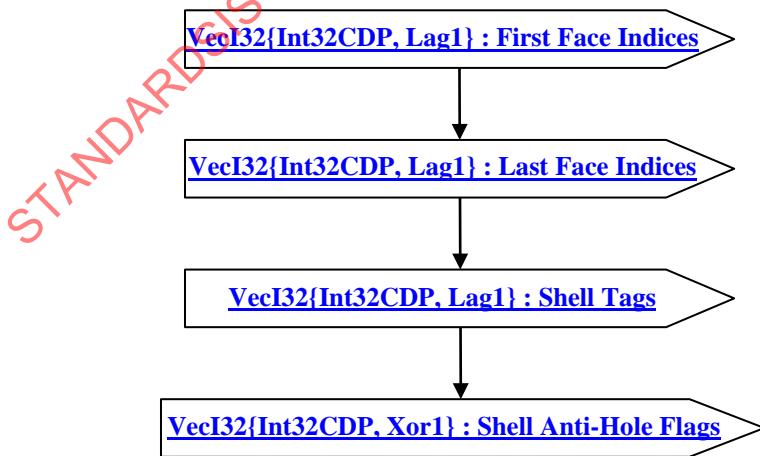
VecI32{Int32CDP, Lag1} : First Face Indices

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:

= 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.

6.2.3.1.3.3 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 must be trimmed with at least one "anti-hole" Trim Loop and may be trimmed with one 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 must be defined with a counter-clockwise orientation whereas "hole" Trim Loops must 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 108 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 108: Trim Loop example in parameter Space - One Face with 2 Holes
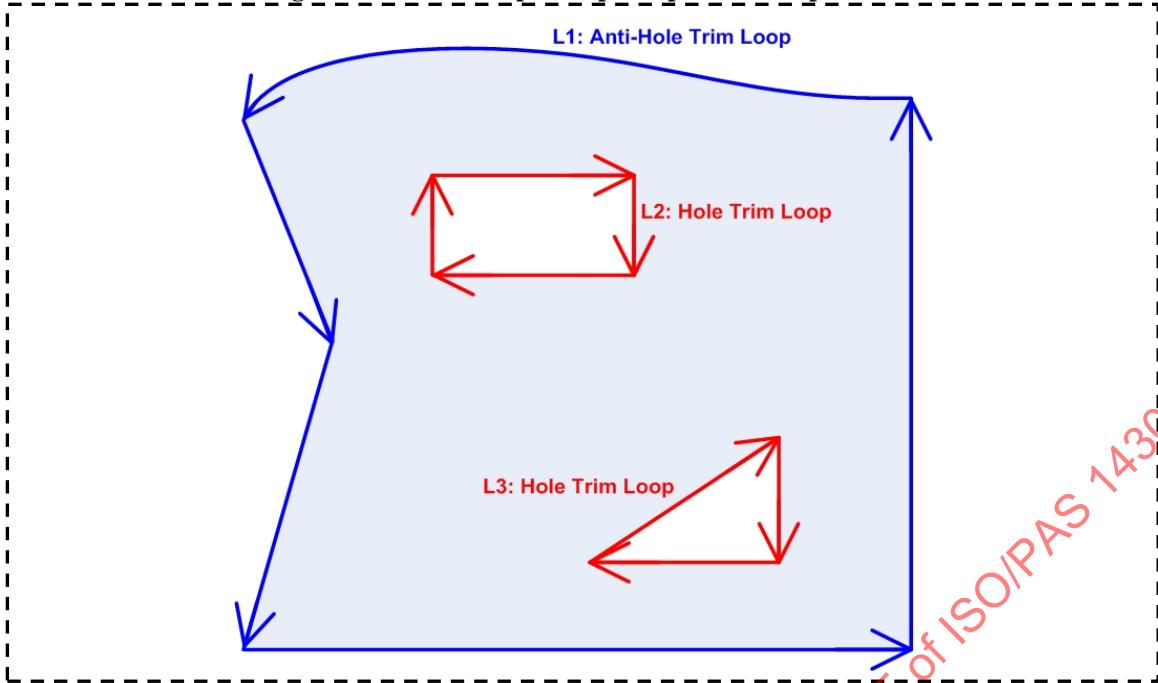
Each Face's underlying Geometric Surface is identified by an index into a list of Geometric Surfaces. Each Face's defining Trim Loops are identified in a list of trim Loops by an index for both the first Trim Loop and the last Trim Loop in each Face (i.e. all Trim Loops inclusive between the specified first and last Trim Loop list index define the particular Face).

Figure 109: Faces Topology Data data collection
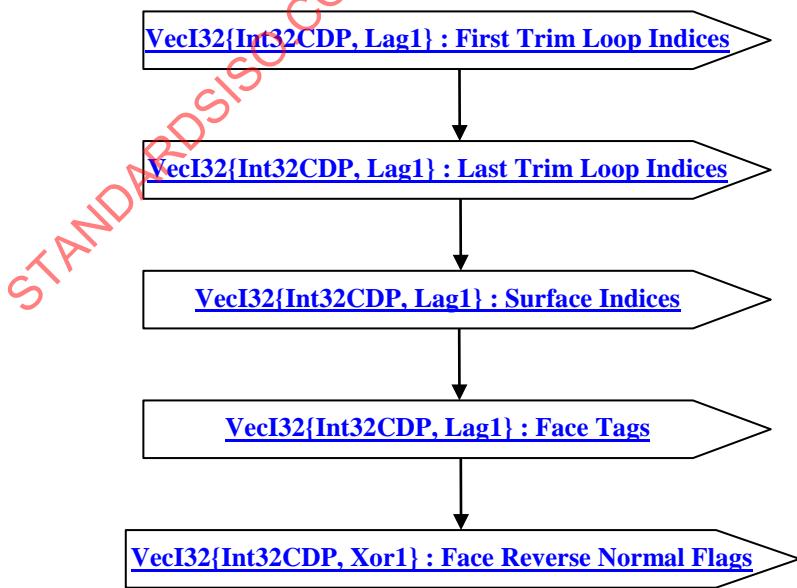
VecI32{Int32CDP, Lag1} : First Trim Loop Indices

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:

= 0	– Face normal is not reversed
= 1	– Shell normal is reversed.

Face Reverse Normal Flags uses the Int32 version of the CODEC to compress and encode data.

6.2.3.1.3.4 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 must 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).

Figure 110: Loops Topology Data data collection

VecI32{Int32CDP, Lag1} : First CoEdge Indices

VecI32{Int32CDP, Lag1} : Last CoEdge Indices

VecI32{I32CDP, Lag1} : Loop Tags

VecI32{I32CDP, Xor1} : Anti-Hole Flags
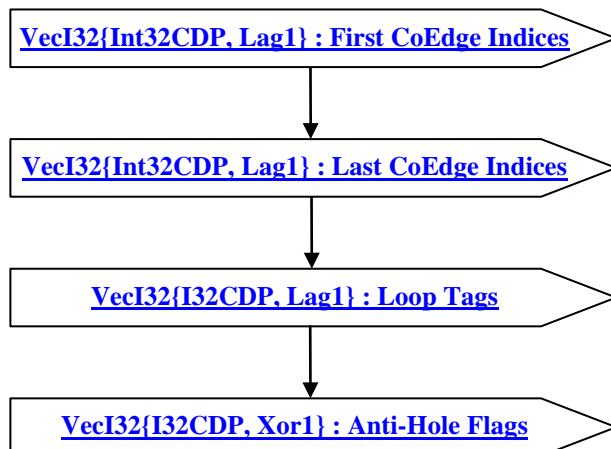
VecI32{Int32CDP, Lag1} : First CoEdge Indices

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:

= 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.

6.2.3.1.3.5 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 ("use") for the same underlying Edge. This sharing of the same underlying Edge by two adjoining Faces necessitates the need for a "MCS Curve Reversed Flag" on each CoEdge to indicate the edge traversal direction (i.e. for a proper manifold topology definition each CoEdge must traverse the Edge in opposite directions).

Figure 111: CoEdges Topology Data data collection

```
┌──────────────────────────────────────────────┐
│ VecI32{Int32CDP, Lag1} : Edge Indices          ╲
└──────────────────────────────────────────────┘
                        │
                        ▼
┌──────────────────────────────────────────────┐
│ VecI32{Int32CDP, Lag1} : PCS Curve Indices      ╲
└──────────────────────────────────────────────┘
                        │
                        ▼
┌──────────────────────────────────────────────┐
│ VecI32{Int32CDP, Lag1} : CoEdge Tags            ╲
└──────────────────────────────────────────────┘
                        │
                        ▼
┌──────────────────────────────────────────────┐
│ VecI32{Int32CDP, Xor1} : MCS Curve Reversed Flags ╲
└──────────────────────────────────────────────┘
```
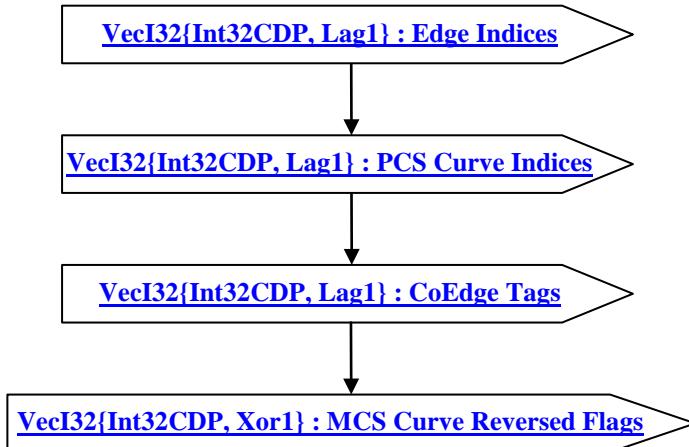
VecI32{Int32CDP, Lag1} : Edge Indices

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:

= 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.
.

6.2.3.1.3.6 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).

Figure 112: Edges Topology Data data collection

VecI32{Int32CDP, Lag1} : Start Vertex Indices

VecI32{Int32CDP, Lag1} : End Vertex Indices

VecI32{Int32CDP, Lag1} : MCS Curve Indices

VecI32{Int32CDP, Lag1} : Edge Tags
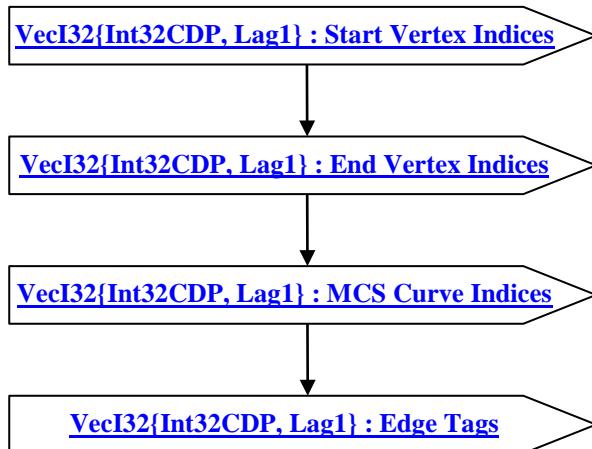
VecI32{Int32CDP, Lag1} : Start Vertex Indices

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.

6.2.3.1.3.7 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).

Figure 113: Vertices Topology Data data collection

VecI32{Int32CDP, Lag1} : Point Indices

VecI32{Int32CDP, Lag1} : Vertex Tags
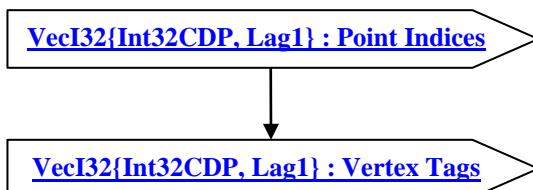
VecI32{Int32CDP, Lag1} : Point Indices

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.

6.2.3.1.4 Geometric Data

Figure 114: Geometric Data data collection
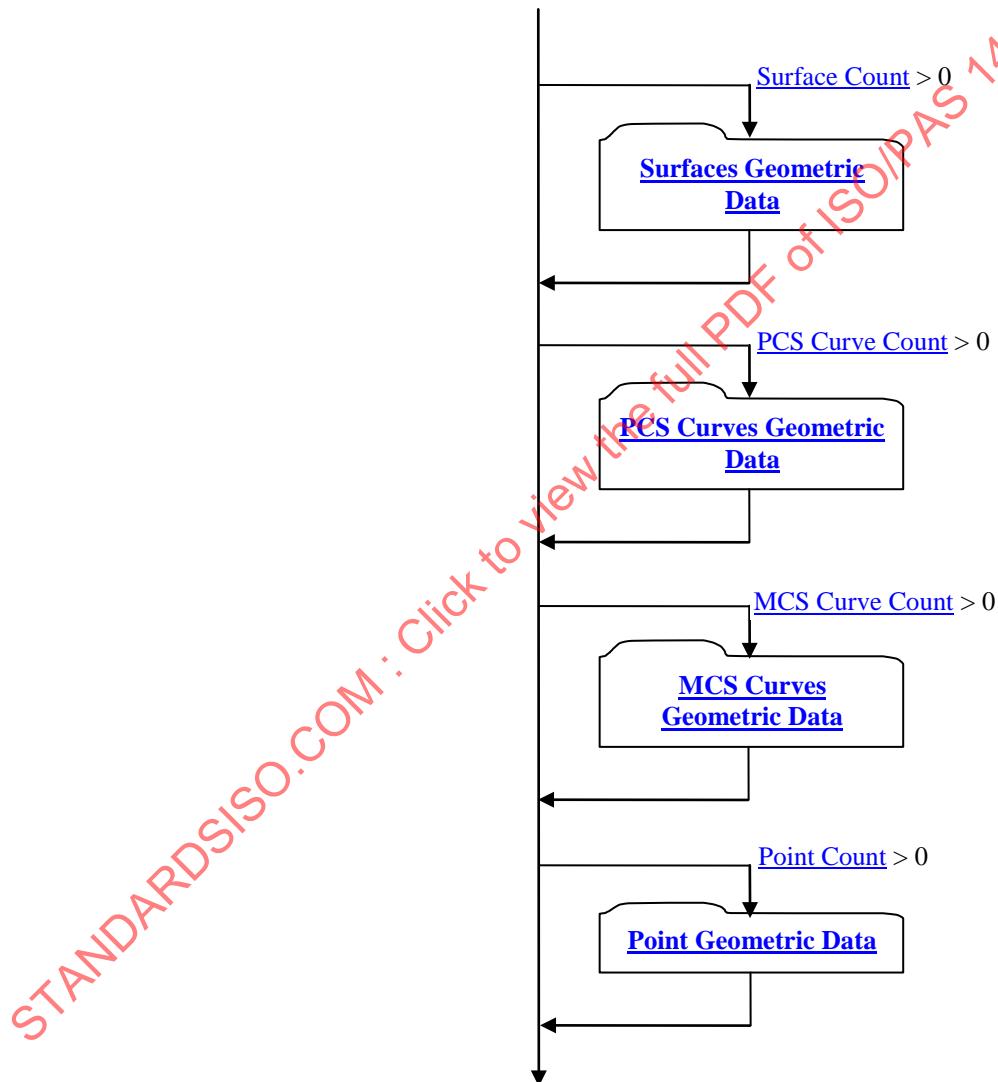
6.2.3.1.4.1 Surfaces Geometric Data

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 6.2.3.1.2 Geometric Entity Counts.

Figure 115: Surfaces Geometric Data data collection
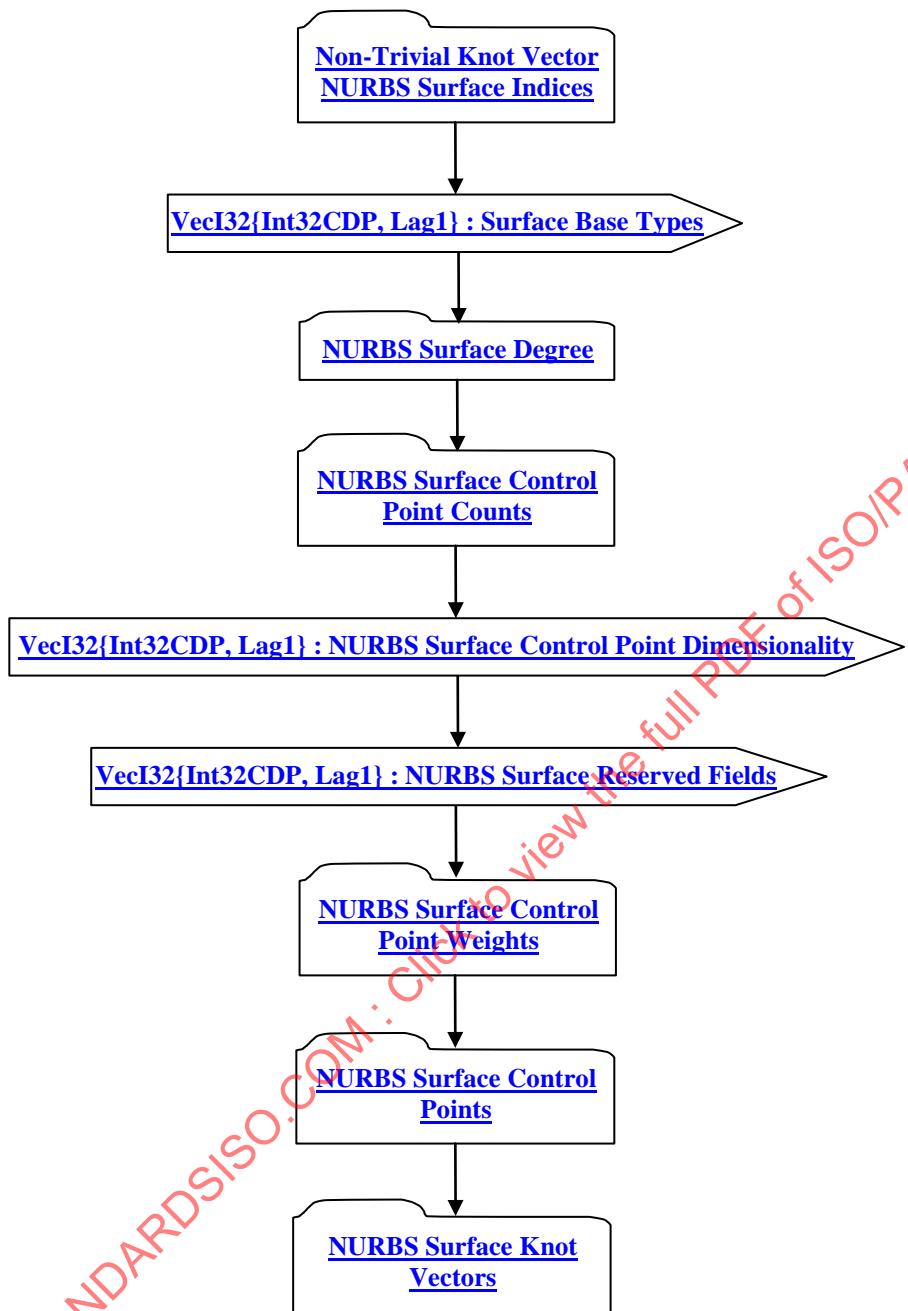
VecI32{Int32CDP, Lag1} : Surface Base Types

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:

= 1	– Surface is a NURBS surface

Surface Base Types uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : 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:

= 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, Lag1} : NURBS Surface Reserved Fields

NURBS Surface Reserved Fields is a vector of data reserved for future expansion of the JT format. Each NURBS Surface in a list of Surfaces has one reserved data field entry in this NURBS Surface Reserved Fields vector. NURBS Surface Reserved Fields uses the Int32 version of the CODEC to compress and encode data

6.2.3.1.4.1.1 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 7.1.8 Compressed Entity List for Non-Trivial Knot Vector.

This Surface index data is stored in a compressed format.

Figure 116: Non-Trivial Knot Vector NURBS Surface Indices data collection
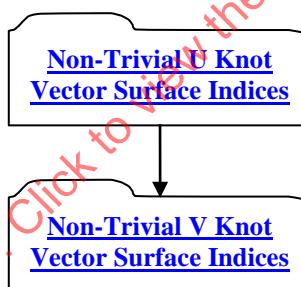
Both Non-Trivial U Knot Vector Surface Indices and Non-Trivial V Knot Vector Surface Indices have the same data format as that documented for data collection 7.1.8 Compressed Entity List for Non-Trivial Knot Vector.

6.2.3.1.4.1.2 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.

Figure 117: NURBS Surface Degree data collection
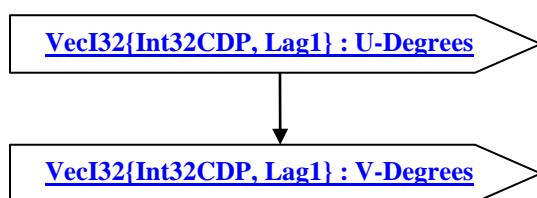
VecI32{Int32CDP, Lag1} : U-Degrees

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, Lag1} : 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.

6.2.3.1.4.1.3 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.

Figure 118: NURBS Surface Control Point Counts data collection
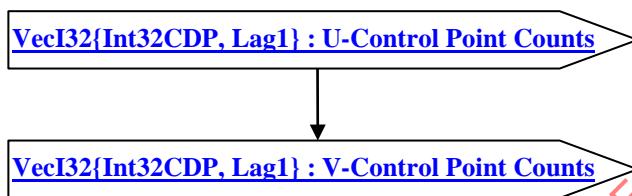
VecI32{Int32CDP, Lag1} : U-Control Point Counts

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, Lag1} : 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.

6.2.3.1.4.1.4 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 119: NURBS Surface Control Point Weights data collection

Complete description for Compressed Control Point Weights Data can be found in 7.1.9 Compressed Control Point Weights Data.

6.2.3.1.4.1.5 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).

Figure 120: NURBS Surface Control Points data collection

> **VecF64{Float64CDP, NULL} : Control Points**

VecF64{Float64CDP, 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 Float64 version of the CODEC to compress and encode data in a "lossless" manner.

6.2.3.1.4.1.6 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 6.2.3.1.4.1.1 Non-Trivial Knot Vector NURBS Surface Indices.

The knot vector data for the list of Surfaces is stored in a compressed format.

Figure 121: NURBS Surface Knot Vectors data collection
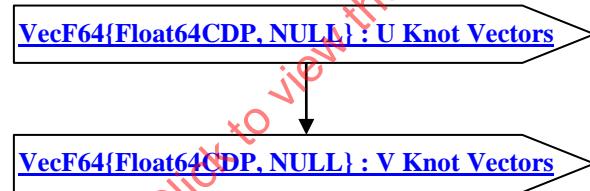
> **VecF64{Float64CDP, NULL} : U Knot Vectors**

> **VecF64{Float64CDP, NULL} : V Knot Vectors**

VecF64{Float64CDP, 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 Float64 version of the CODEC to compress and encode data.

VecF64{Float64CDP, 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 Float64 version of the CODEC to compress and encode data.

6.2.3.1.4.2 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 6.2.3.1.2 Geometric Entity Counts.

Figure 122: PCS Curves Geometric Data data collection
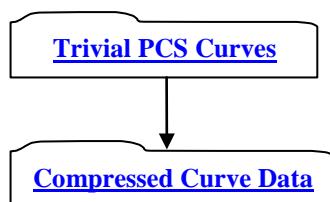
```
┌─────────────────────────┐
│   Trivial PCS Curves     │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│  Compressed Curve Data   │
└─────────────────────────┘
```

Complete description for Compressed Curve Data can be found in 7.1.10 Compressed Curve Data.

6.2.3.1.4.2.1 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.

Figure 123: Trivial PCS Curves data collection

```
┌─────────────────────────────────────────┐
│ 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{Float64CDP, 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

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 must always be made up of four Trivial Domain UV curves.

= 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 must always be made up of four UV curves

= 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 as follows:

$$P0[0] - P5[0] = 0$$
$$P1[0] - P4[0] = 0$$
$$P2[1] - P7[1] = 0$$
$$P3[1] - P6[1] = 0$$
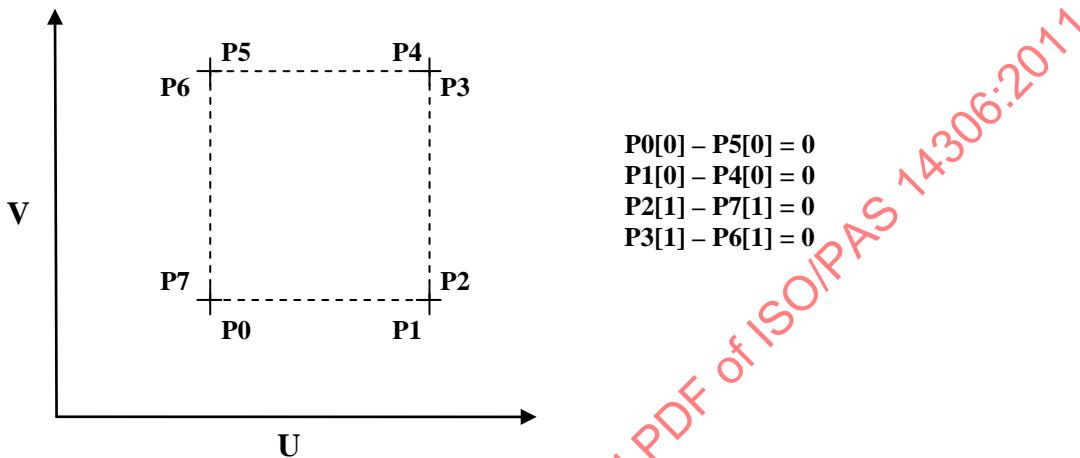
I32 : Trivial Domain UV Curves Exist Flag

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).

= 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{Float64CDP, 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 Float64 version of the CODEC to compress and encode data.

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:

= 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.

6.2.3.1.4.3 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 6.2.3.1.2 Geometric Entity Counts.

Figure 124: MCS Curves Geometric Data data collection

Compressed Curve Data

Complete description for Compressed Curve Data can be found in 7.1.10 Compressed Curve Data.

6.2.3.1.4.4 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 6.2.3.1.2 Geometric Entity Counts.

Figure 125: Point Geometric Data data collection

CoordF32 : Point Coordinates Point Count
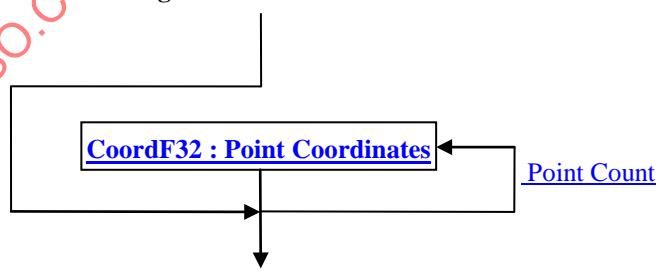
CoordF32 : Point Coordinates

Point Coordinates specifies the XYZ coordinate components for a Point.

6.2.3.1.5 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 126: Topological Entity Tag Counters data collection

```
┌─────────────────────────────┐
│  I32 : Region Tag Counter   │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│   I32 : Shell Tag Counter   │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│   I32 : Face Tag Counter    │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│   I32 : Loop Tag Counter    │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│  I32 : CoEdge Tag Counter   │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│   I32 : Edge Tag Counter    │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│  I32 : Vertex Tag Counter   │
└─────────────────────────────┘
```
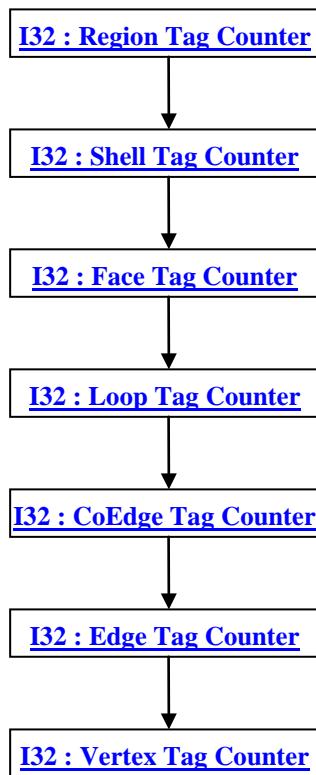
I32 : Region Tag Counter
Region tag Counter specifies the next available "unique' tag value for Region entity.

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.

6.2.3.1.6 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 6.2.3.1 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 6.2.3.1.1 Topological Entity Counts.

Figure 127: B-Rep CAD Tag Data data collection

Compressed CAD
Tag Data

Complete description for Compressed CAD Tag Data can be found in 7.1.11 Compressed CAD Tag Data.

6.2.4 XT B-Rep Segment

XT B-Rep Segment contains an Element that defines the precise geometric Boundary Representation data for a particular Part in Parasolid boundary representation (XT) 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 6.2.3 JT B-Rep Segment.

XT B-Rep Segments are typically referenced by Part Node Elements (see 6.2.1.1.1.5Part Node Element) using Late Loaded Property Atom Elements (see 6.2.1.2.7Late Loaded Property Atom Element). The XT B-Rep Segment type supports ZLIB compression on all element data, so all elements in XT B-Rep Segment use the Element Header ZLIB form of element header data.

6.2.4.1 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 Parasolid boundary representations (XT) format.

Figure 128: XT B-Rep Element data collection
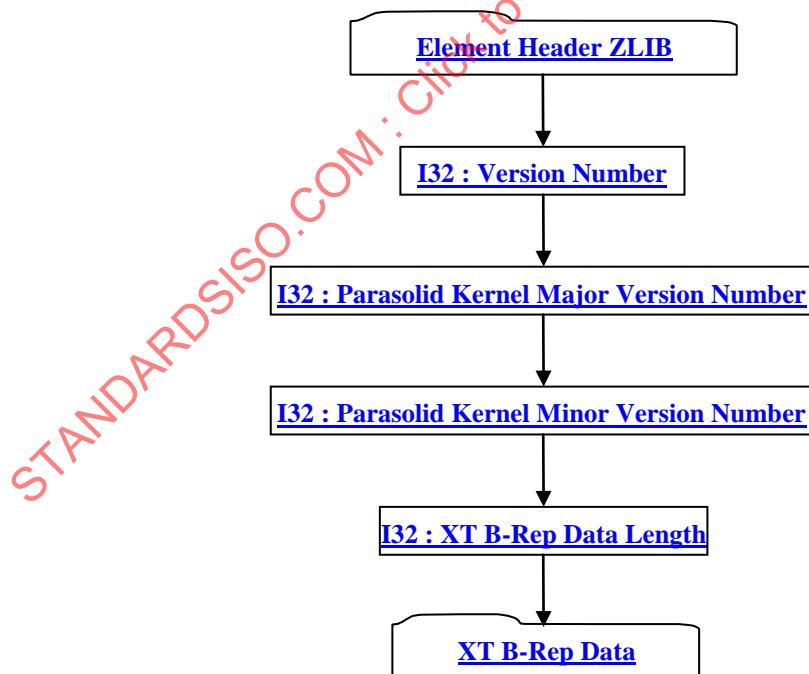
Element Header ZLIB

I32 : Version Number

I32 : Parasolid Kernel Major Version Number

I32 : Parasolid Kernel Minor Version Number

I32 : XT B-Rep Data Length

XT B-Rep Data

Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

I32 : Version Number

Version Number is the version identifier for this XT B-Rep Element. Version number "1" is currently the only valid value.

I32 : Parasolid Kernel Major Version Number

Parasolid Kernel Major Version Number specifies the major version number for the revision of Parasolid that wrote the XT B-Rep data into JT File.

I32 : Parasolid Kernel Minor Version Number

Parasolid Kernel Minor Version Number specifies the minor version number for the revision of Parasolid that wrote the XT B-Rep data into JT File.

I32 : XT B-Rep Data Length

XT B-Rep Data Length specifies the length in bytes of the XT B-Rep Data 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.

6.2.4.1.1 XT B-Rep Data

The XT B-Rep Data collection specifies the raw stream of bytes that Parasolid uses to represent a Part's B-Rep Body(s) in an external file. The XT B-Rep Data collection format in the JT file is exactly equivalent to the Parasolid XT "Neutral Binary" encoding format as written by the Parasolid "PK_PART_transmit" interface routine.

Complete documentation for the Parasolid XT "Neutral Binary" encoding format as written by "PK_PART_transmit" can be found in "*Appendix D Parasolid XT Format Reference*" of this document.

6.2.5 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 6.2.1.1.1.5 Part Node Element) using a Late Loaded Property Atom Element (see 6.2.1.2.7 Late Loaded Property Atom Element). The Wireframe Segment type supports ZLIB compression on all element data, so all elements in Wireframe Segment use the Element Header ZLIB form of element header data.

Figure 129: Wireframe Segment data collection
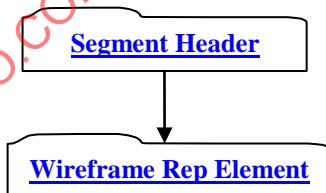
Complete description for Segment Header can be found in 6.1.3.1Segment Header.

6.2.5.1 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.

Figure 130: Wireframe Rep Element data collection
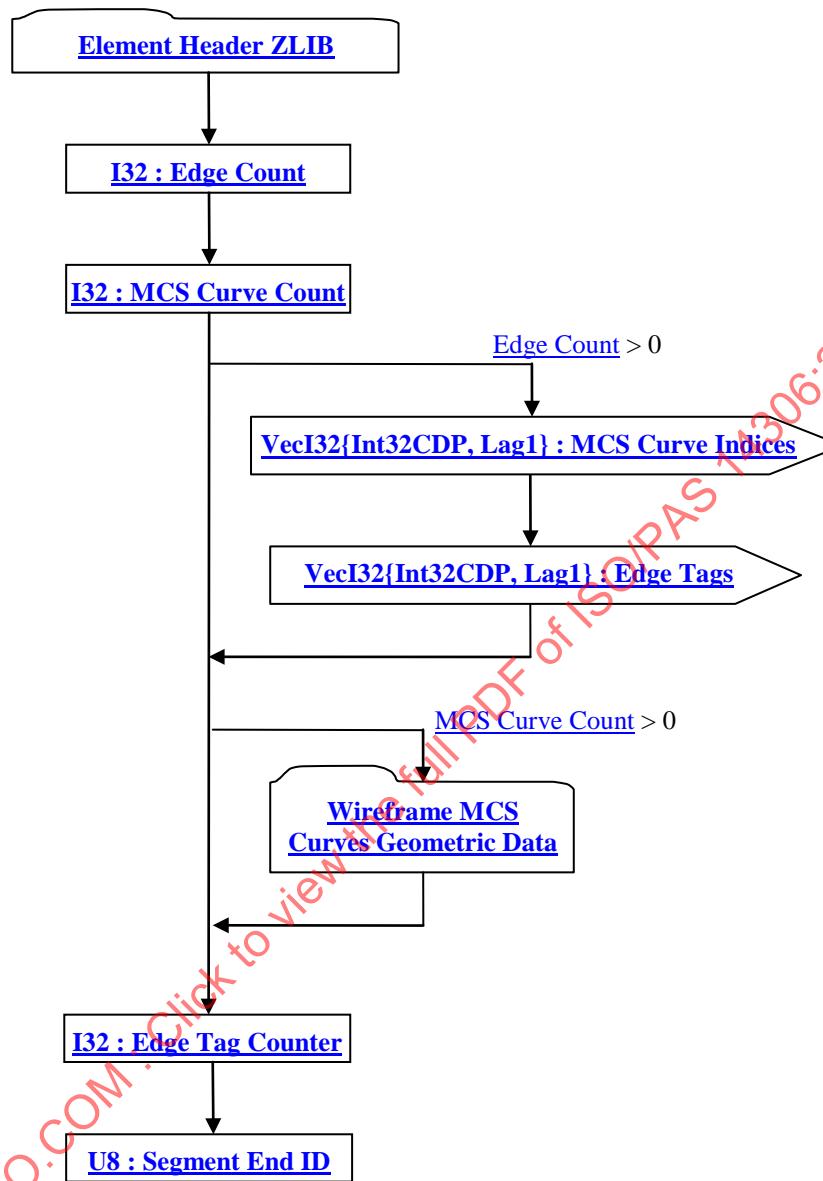
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

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, 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.

I32 : Edge Tag Counter

Edge Tag Counter specifies the next available "unique" tag value for Edge entity.

U8 : Segment End ID

Segment End ID defines the segment end identifier. This field should always have a value of "114".

6.2.5.1.1 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 6.2.5.1 Wireframe Rep Element.

Figure 131: Wireframe MCS Curves Geometric Data data collection

Compressed Curve Data

Complete description for Compressed Curve Data can be found in 7.1.10 Compressed Curve Data.

6.2.6 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 6.2.1.1.1.5Part Node Element) using Late Loaded Property Atom Elements (see 6.2.1.2.7Late Loaded Property Atom Element).

The Meta Data Segment type supports ZLIB compression on all element data, so all elements in Meta Data Segment use the Element Header ZLIB form of element header data.

Figure 132: Meta Data Segment data collection

Segment Header
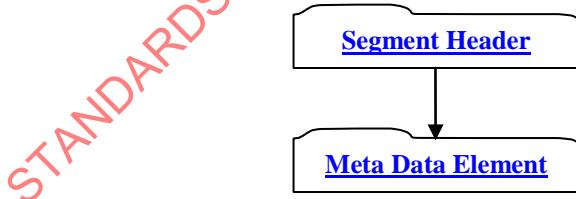
Meta Data Element

Complete description for Segment Header can be found in 6.1.3.1 Segment Header.

The following sub-sections document the various Meta Data Element types.

6.2.6.1 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 6.2.1.1.1.6 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 many several data types.

Figure 133: Property Proxy Meta Data Element data collection
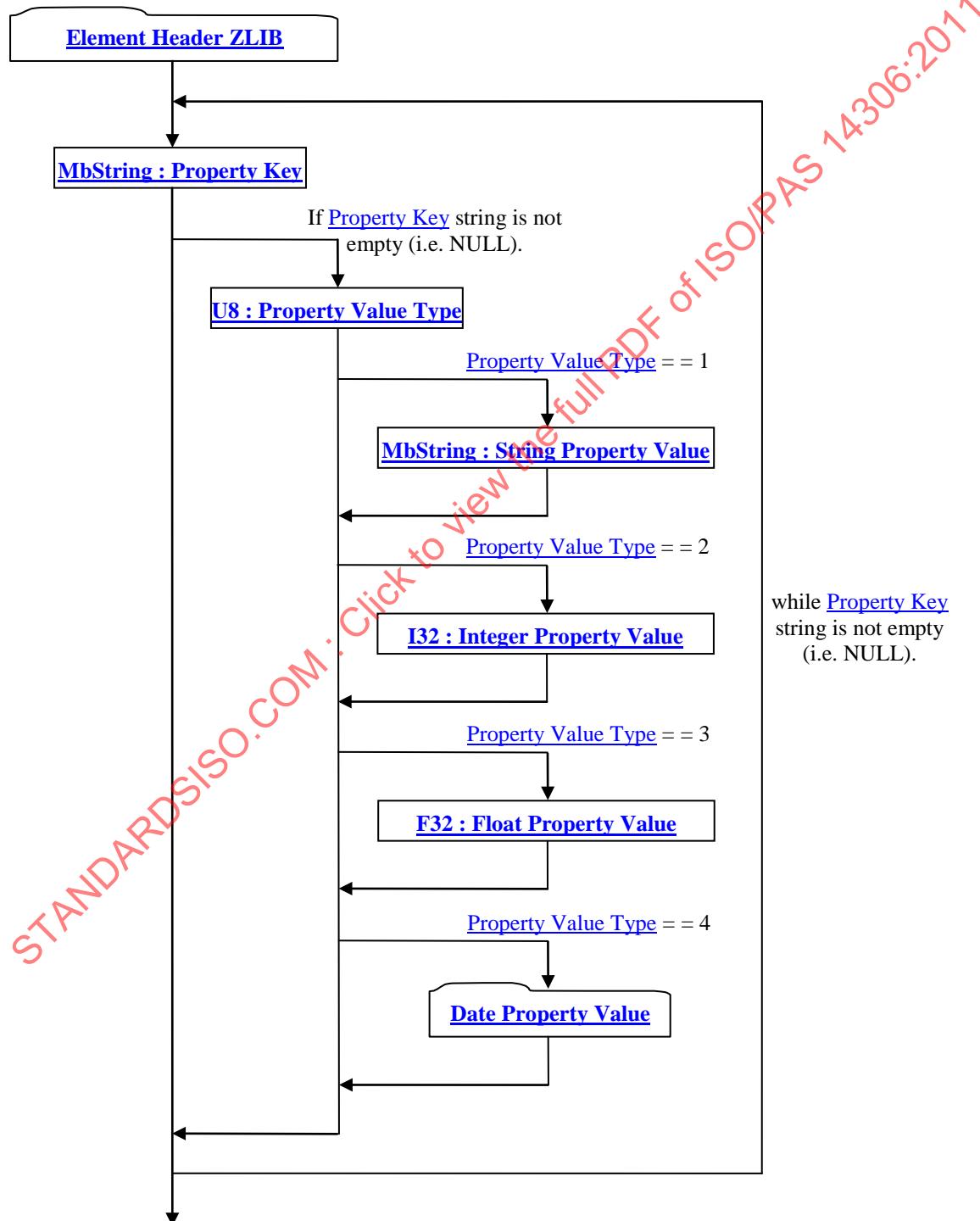
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

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:

= 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.

6.2.6.1.1 Date Property Value

Date Property Value data collection represents a date as a combination of year, month, day, hour, minute, and second data fields.

Figure 134: Date Property Value data collection
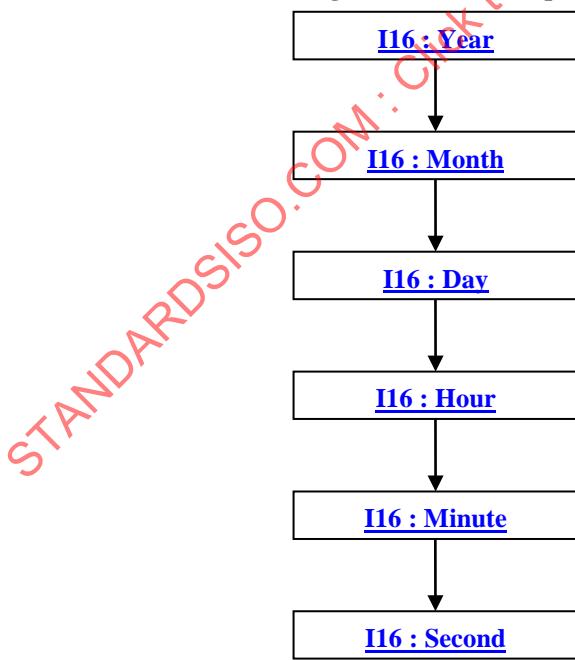
I16 : Year

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.

6.2.6.2 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 135: PMI Manager Meta Data Element data collection
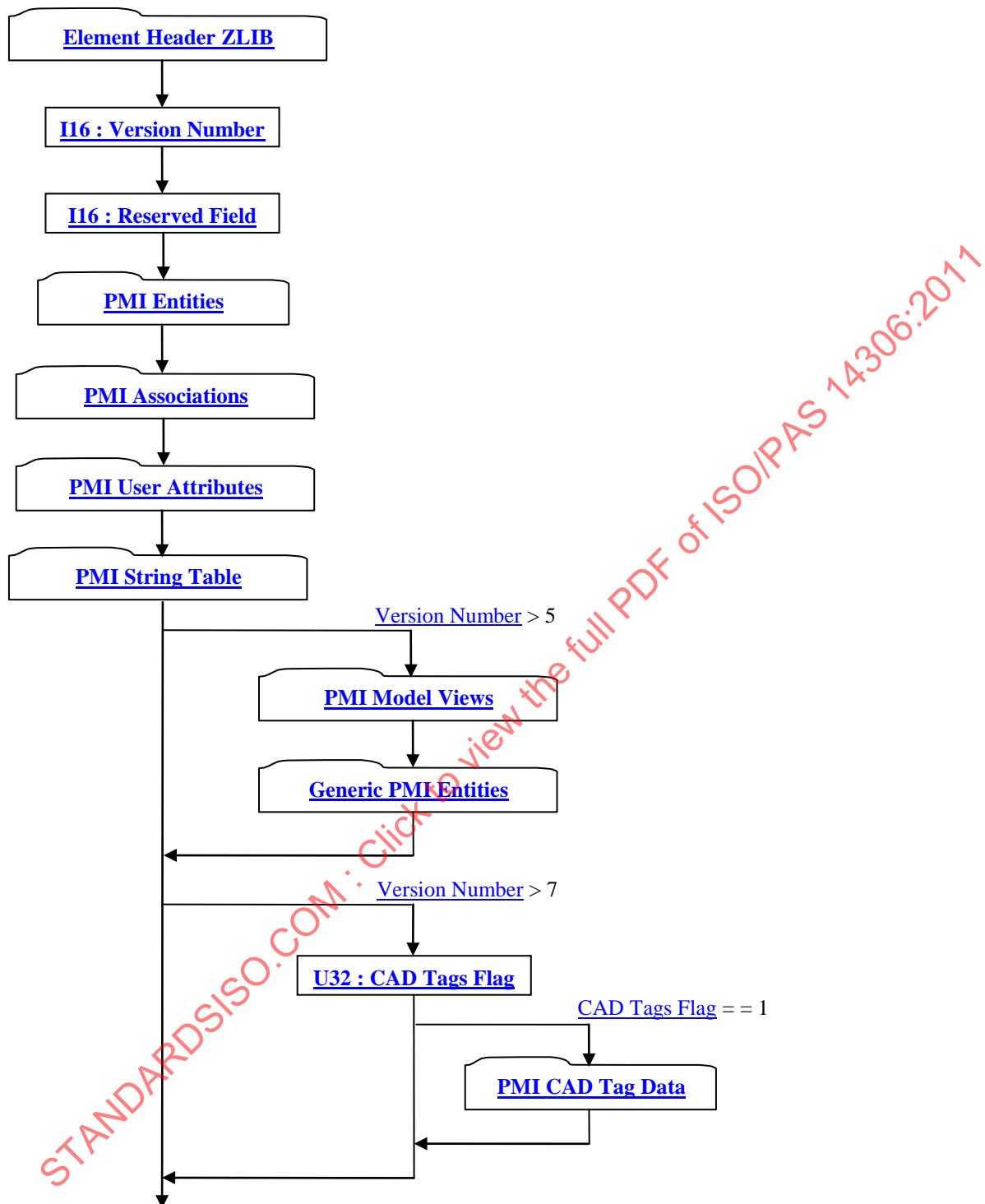
Complete description for Element Header ZLIB can be found in 6.1.3.2.2 Element Header ZLIB.

I16 : Version Number

Version Number is the version identifier for the PMI. There are several PMI versions that must be supported for JT File format 8.1. This is because if an older JT File format containing PMI is read and then re-exported to JT File Format 8.1, the exported PMI data must be maintained in the version format originally read from the initial JT file (i.e. PMI data read from a JT File is not migrated to new version format when re-exported to another JT File format).

The valid PMI version numbers are as follows:

= 3	– Version-3
= 4	– Version-4
= 5	– Version-5
= 6	– Version-6
= 7	– Version-7
= 8	– Version-8

I16 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

U32 : CAD Tags Flag

CAD Tags Flag is a flag indicating whether CAD Tag data exist for the PMI.

6.2.6.2.1 PMI Entities

Figure 136: PMI Entities data collection
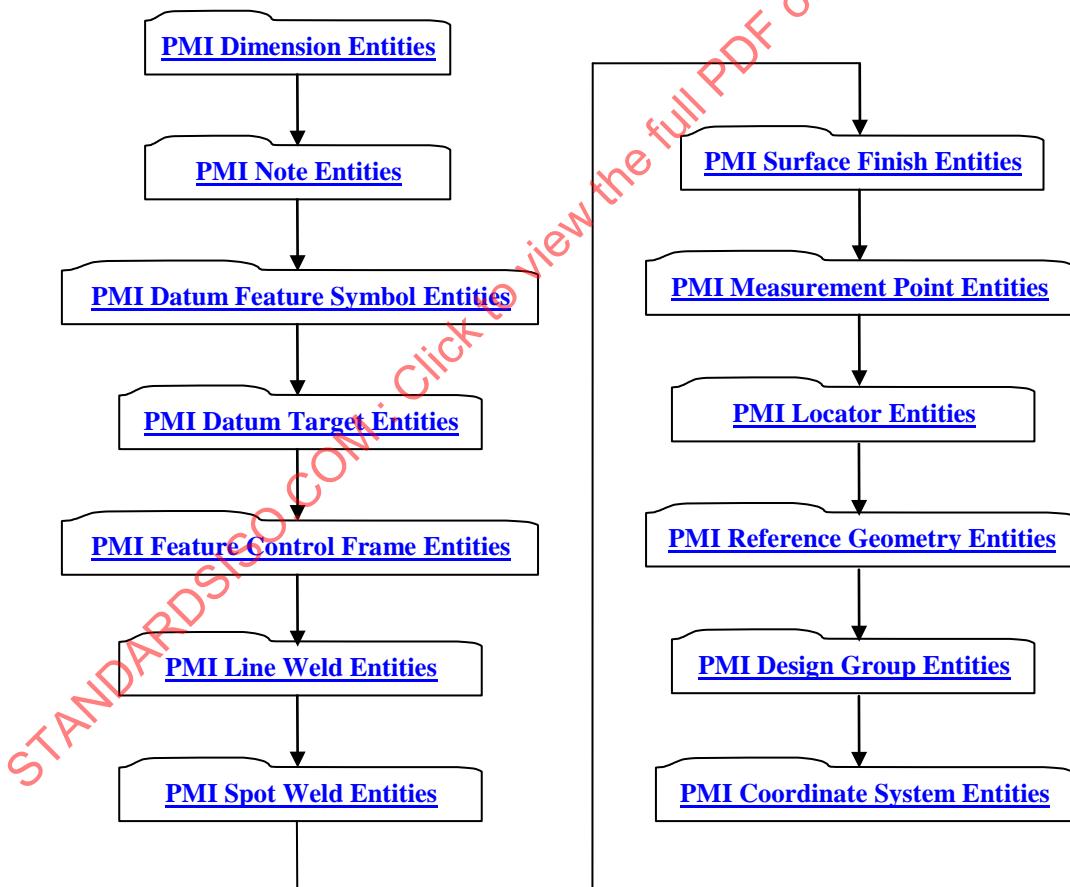
6.2.6.2.1.1 PMI Dimension Entities

The PMI Dimension Entities data collection defines data for a list of Dimensions.

Figure 137: PMI Dimension Entities data collection
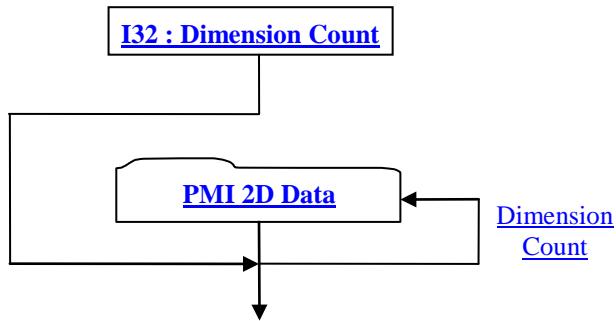
I32 : Dimension Count

Dimension Count specifies the number of Dimension entities.

6.2.6.2.1.1.1 PMI 2D Data

The PMI 2D Data collection defines a data format common to all 2D based PMI entities.

Figure 138: PMI 2D Data data collection
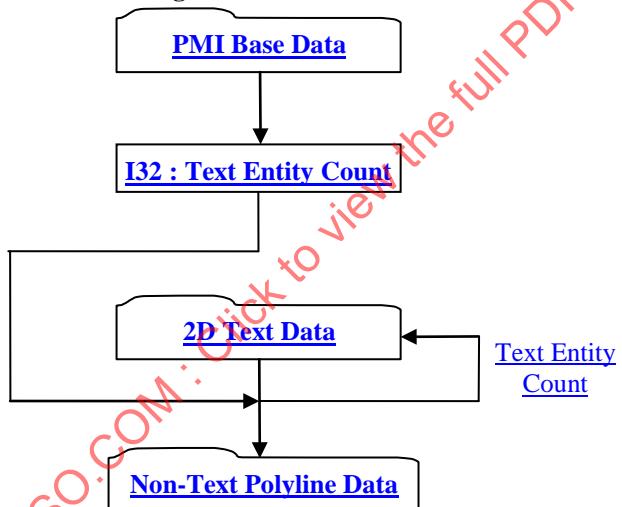
I32 : Text Entity Count

Text Entity Count specifies the number of Text entities in the particular PMI entity.

6.2.6.2.1.1.1.1 PMI Base Data

The PMI Base Data collection defines the basic/common data that every 2D and 3D PMI entity contains

Figure 139: PMI Base Data data collection
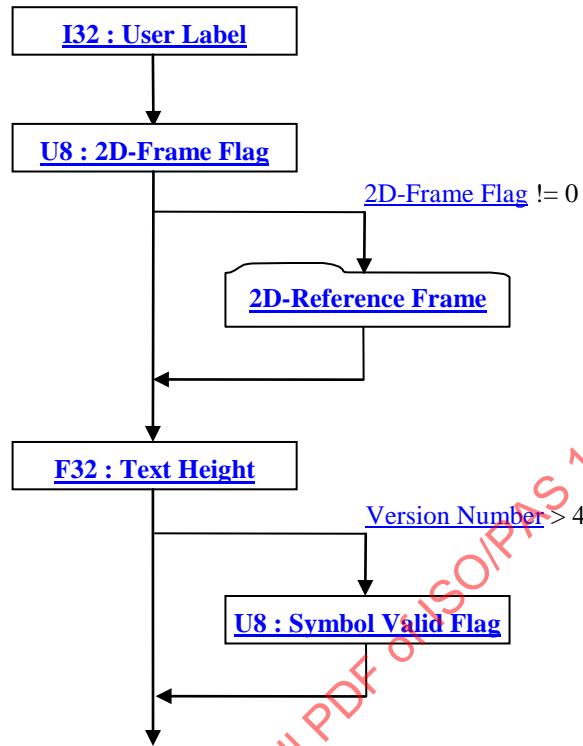
I32 : User Label
User Label specifies the particular PMI entity identifier.

U8 : 2D-Frame Flag
2D-Frame Flag is a flag specifying whether 6.2.6.2.1.1.1.1.1 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 6.2.6.2.6 Generic PMI Entities because for Generic PMI Entities all the 6.2.6.2.1.1.1.3 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 6.2.6.2 PMI Manager Meta Data Element is greater than "4."

6.2.6.2.1.1.1.1.1 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 140: 2D-Reference Frame data collection
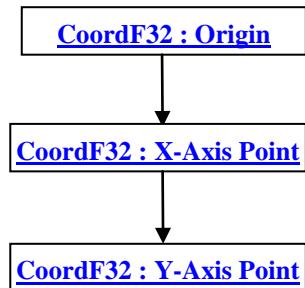
CoordF32 : Origin

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.

6.2.6.2.1.1.1.2   2D Text Data

The 2D Text Data collection defines a 2D text entity/primitive.

Figure 141: 2D Text Data data collection
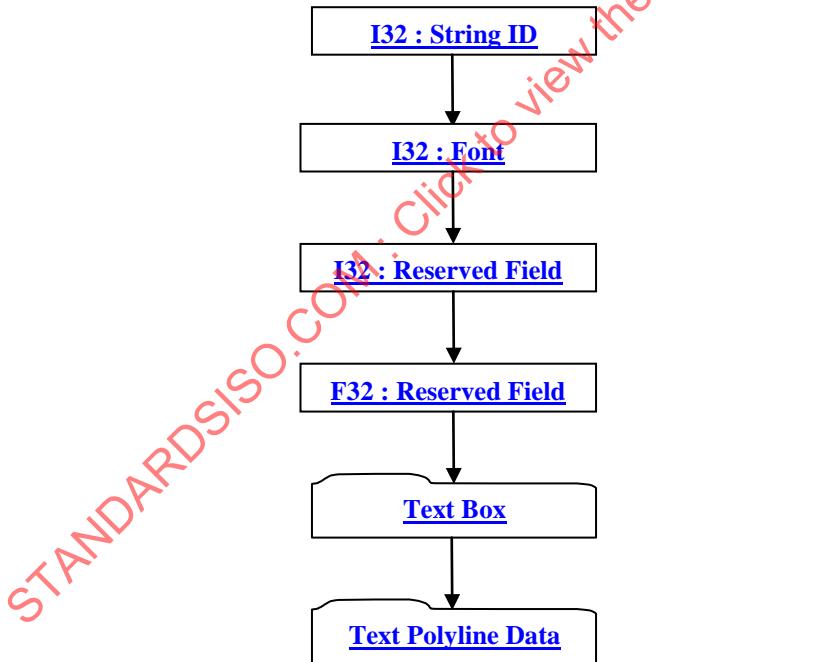
I32 : String ID

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 6.2.6.2.4 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:

= 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 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

F32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

6.2.6.2.1.1.1.2.1 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 6.2.6.2.1.1.1.1 2D-Reference Frame.

Figure 142: Text Box data collection

```
         ┌─────────────────────────────┐
         │    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       │
    └───────────────────────────────────────┘
```
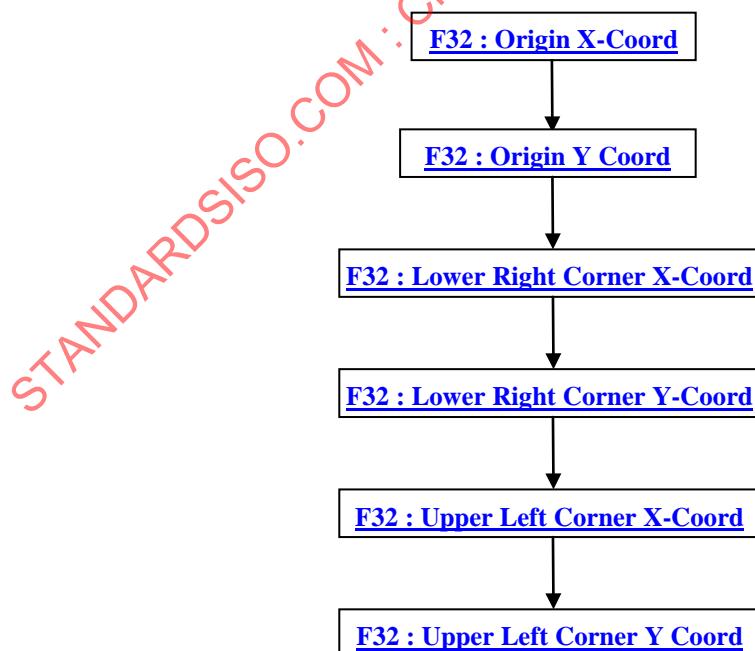
F32 : Origin X-Coord

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.

6.2.6.2.1.1.1.2.2 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:
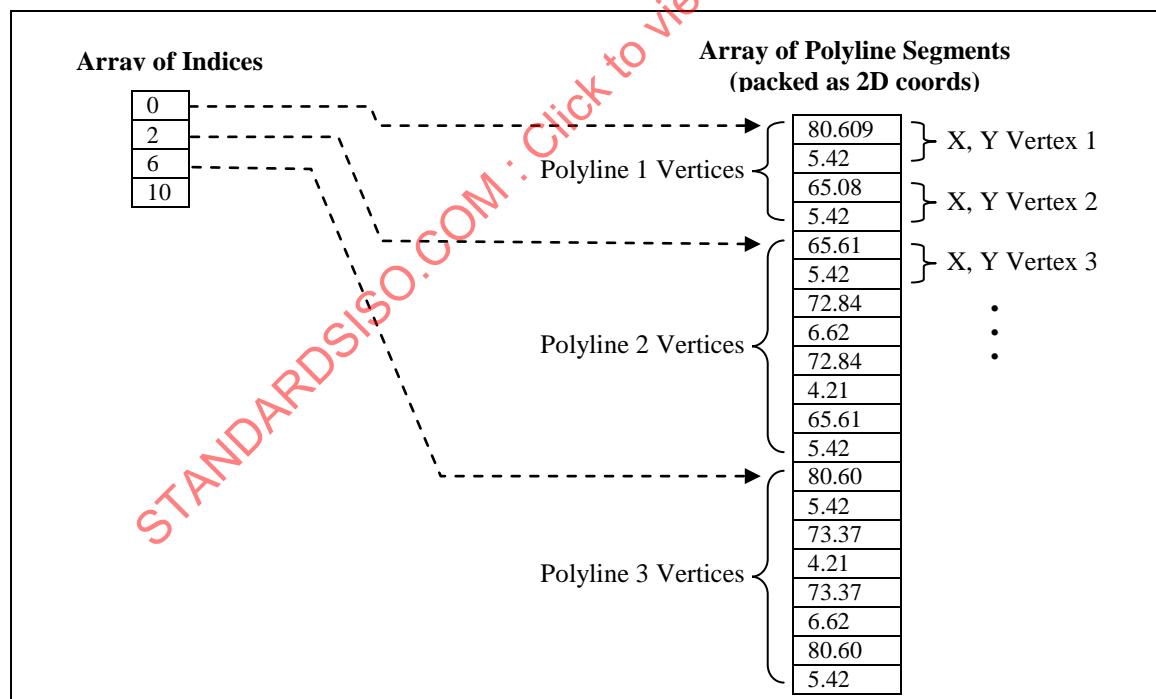
Figure 143: Constructing Text Polylines from data arrays

This data is represented in JT file in the following format:

Figure 144: Text Polyline Data data collection
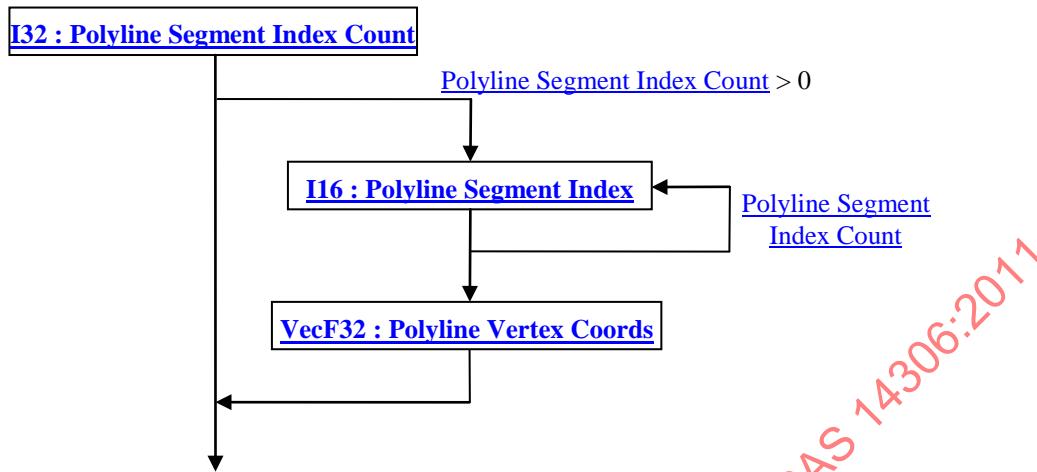
I32 : Polyline Segment Index Count

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 6.2.6.2.1.1.1.1 2D-Reference Frame.

6.2.6.2.1.1.1.3   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 6.2.6.2.6 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 6.2.6.2 PMI Manager Meta Data Element, greater than "4" an array of values that sequentially specify the polyline type in the polyline segments array is included.

Figure 145 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 "[XYZ][XYZ][XYZ]…"
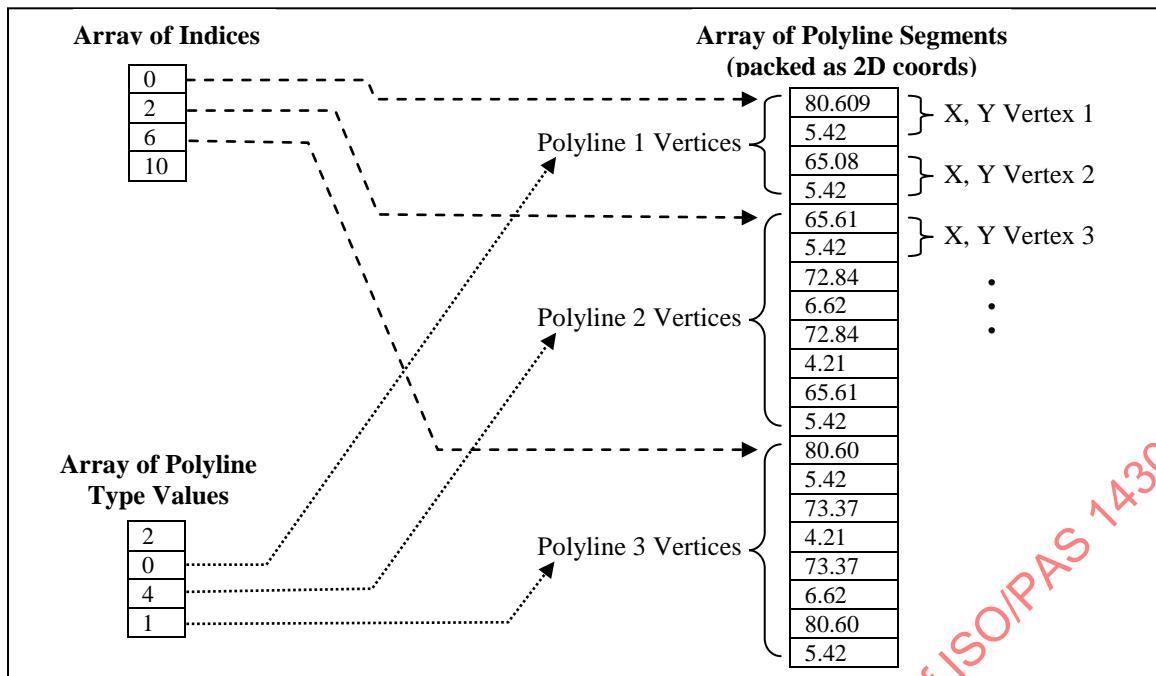
Figure 145: Constructing Non-Text Polylines from packed 2D data arrays

This data is represented in the JT format as follows:

Figure 146: Non-Text Polyline Data data collection
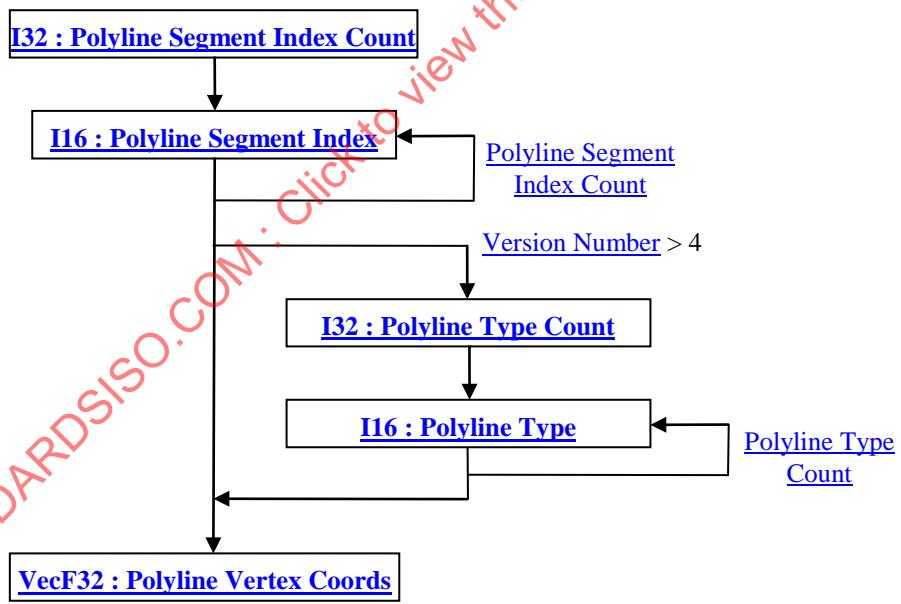
I32 : Polyline Segment Index Count
Polyline Segment Index Count specifies the number of polyline segment indices.

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 145: 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:

= 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 6.2.6.2.1.1.1.1.1 2D-Reference Frame.

6.2.6.2.1.2 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 147: PMI Note Entities data collection
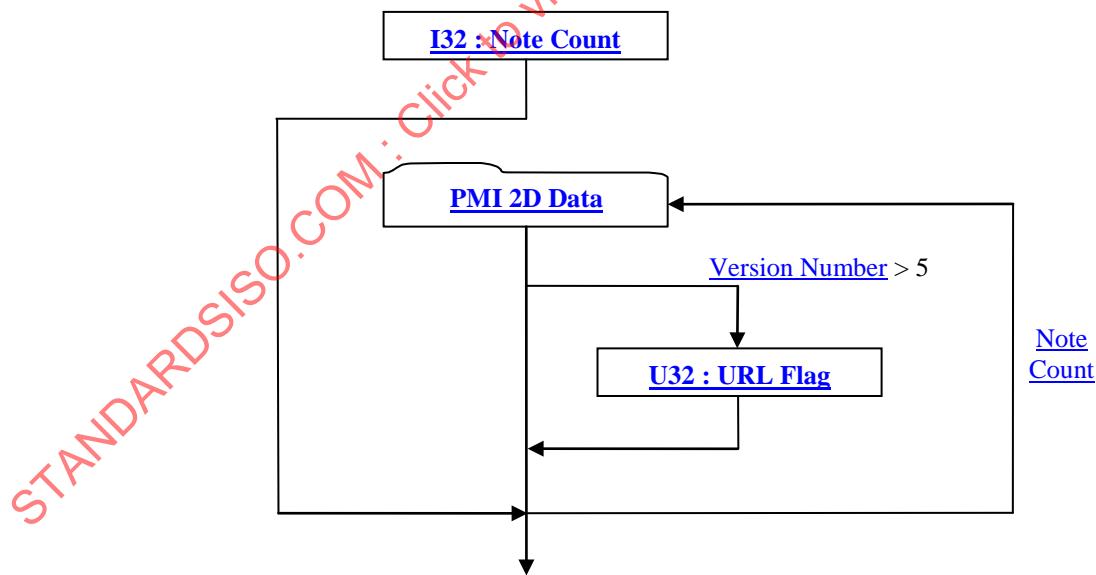
Complete description for PMI 2D Data can be found in 6.2.6.2.1.1.1 PMI 2D Data.

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 6.2.6.2 PMI Manager Meta Data Element, is greater than "5". The URL is the actual text of the note as specified in PMI 2D Data.

6.2.6.2.1.3 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 148: PMI Datum Feature Symbol Entities data collection
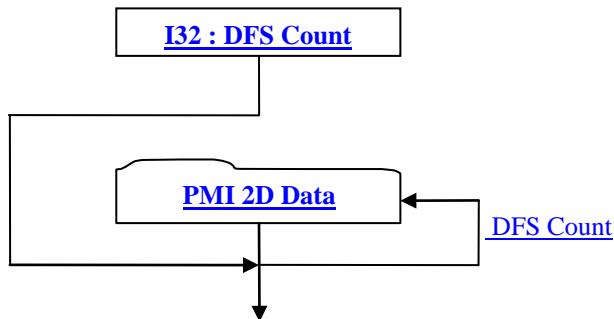
Complete description for PMI 2D Data can be found in 6.2.6.2.1.1.1 PMI 2D Data.

I32 : DFS Count

DFS Count specifies the number of Datum Feature Symbol entities.

6.2.6.2.1.4 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 149: PMI Datum Target Entities data collection
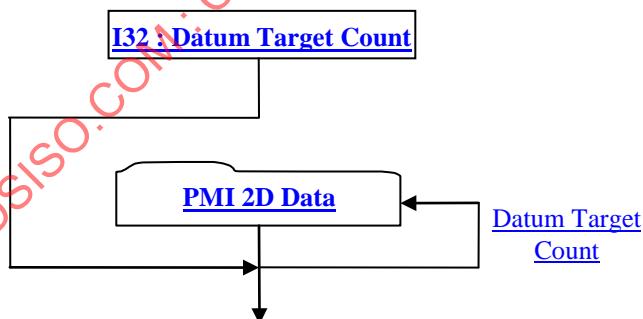
Complete description for PMI 2D Data can be found in 6.2.6.2.1.1.1 PMI 2D Data.

I32 : Datum Target Count

Datum Target Count specifies the number of Datum Target entities.

6.2.6.2.1.5 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 150: PMI Feature Control Frame Entities data collection
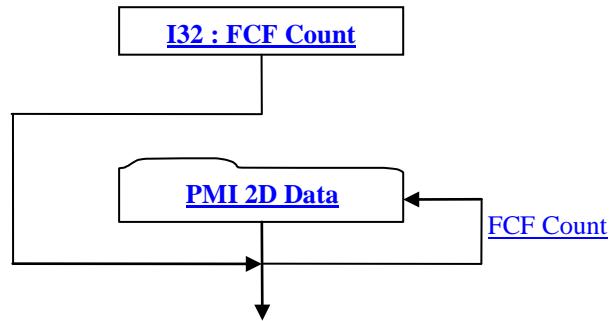
Complete description for PMI 2D Data can be found in 6.2.6.2.1.1.1 PMI 2D Data.

I32 : FCF Count

FCF Count specifies the number of Feature Control Frame entities.

6.2.6.2.1.6 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 151: PMI Line Weld Entities data collection
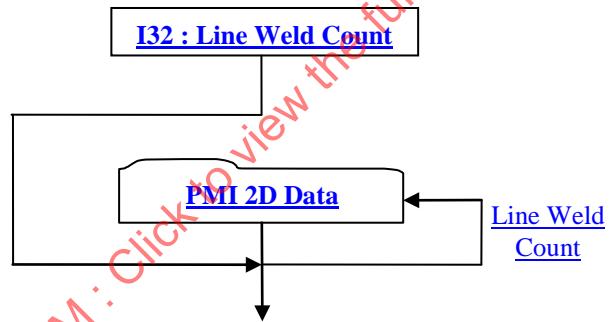
Complete description for PMI 2D Data can be found in 6.2.6.2.1.1.1 PMI 2D Data.

I32 : Line Weld Count

Line Weld Count specifies the number of Line Weld entities.

6.2.6.2.1.7 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 6.2.6.2PMI Manager Meta Data Element, is greater than or equal to "4".

Figure 152: PMI Spot Weld Entities data collection
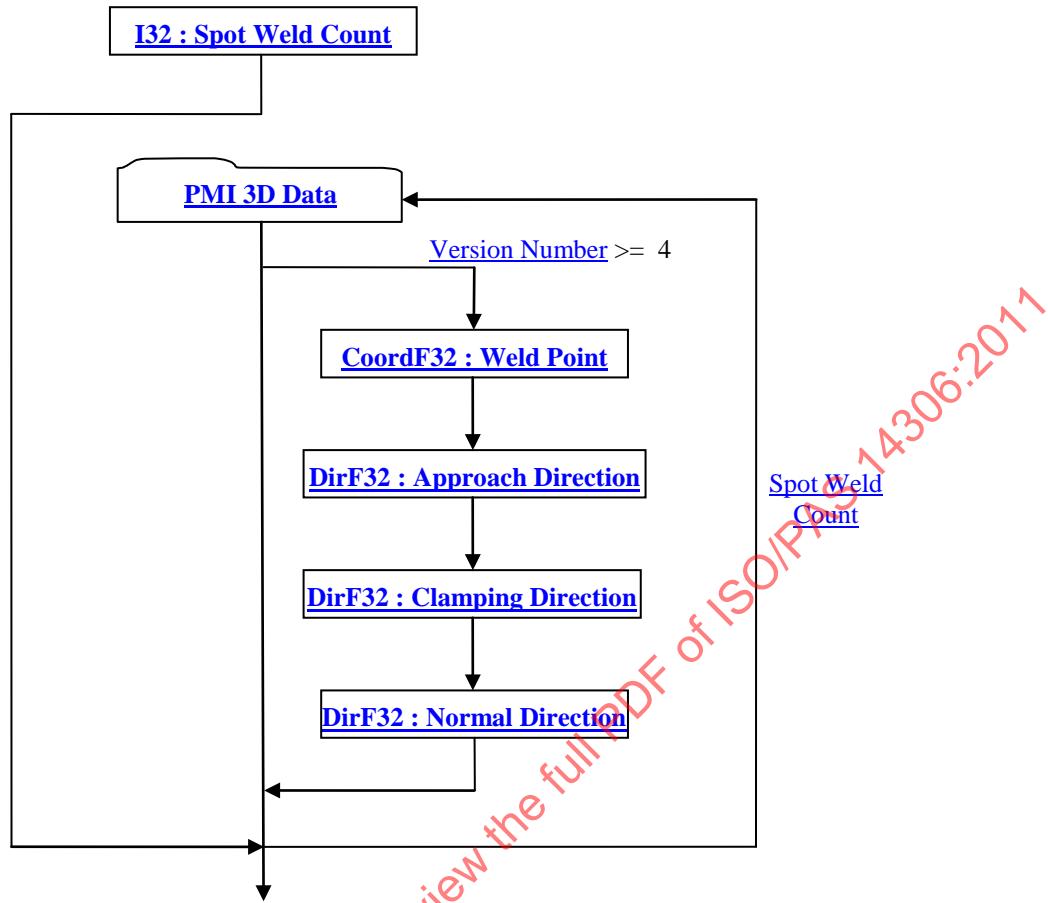
I32 : Spot Weld Count

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.

6.2.6.2.1.7.1 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 143 of 6.2.6.2.1.1.1.2.2 Text Polyline Data.

Figure 153: PMI 3D Data data collection
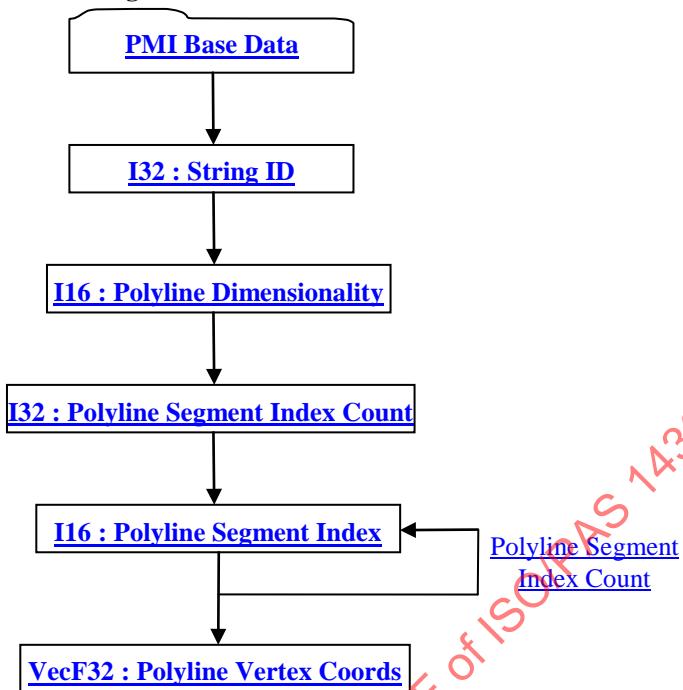
Complete description for PMI Base Data can be found in 6.2.6.2.1.1.1 PMI Base Data.

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 6.2.6.2.4 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:

= 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.

6.2.6.2.1.8 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 154: PMI Surface Finish Entities data collection
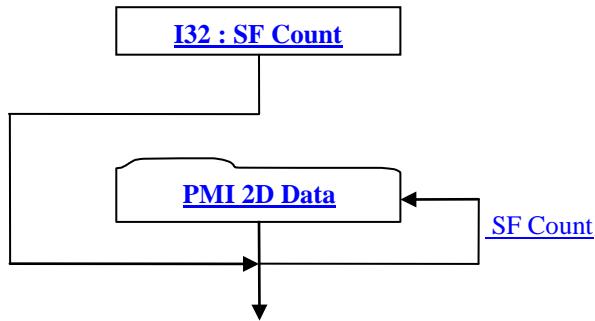
Complete description for PMI 2D Data can be found in 6.2.6.2.1.1.1 PMI 2D Data.

I32 : SF Count
SF Count specifies the number of Surface Finish symbol entities.

6.2.6.2.1.9 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 6.2.6.2PMI Manager Meta Data Element, is greater than or equal to "4".

Figure 155: PMI Measurement Point Entities data collection
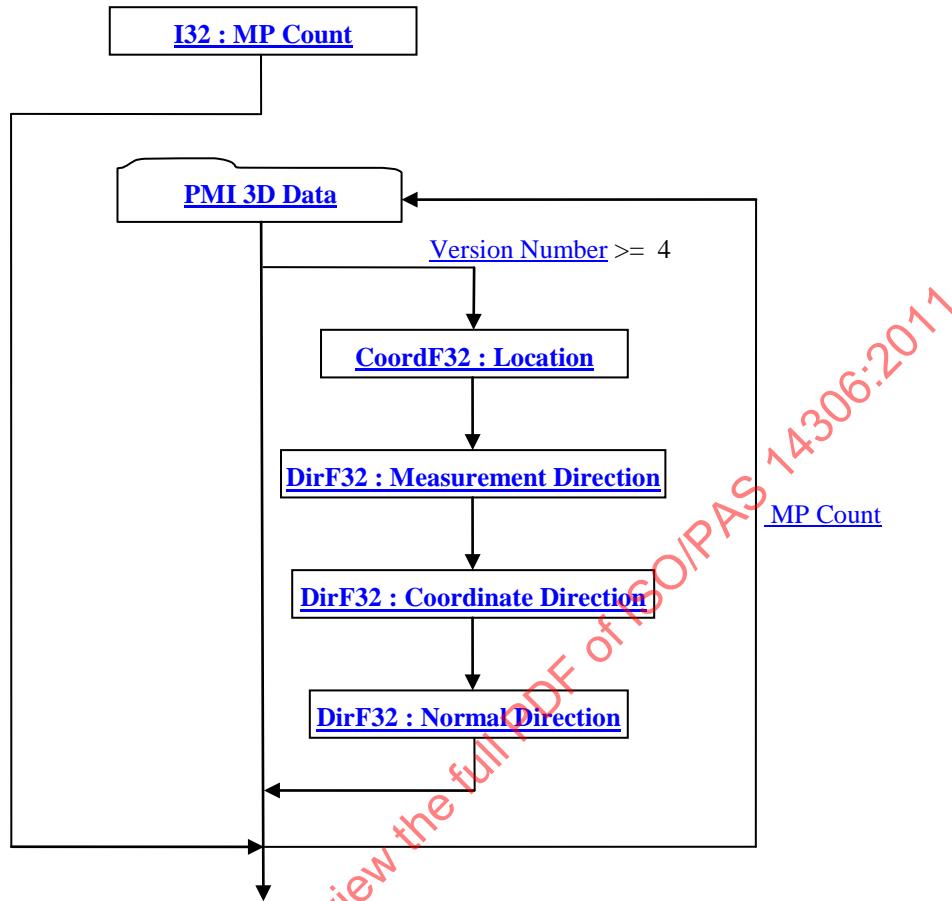
Complete description for PMI 3D Data can be found in 6.2.6.2.1.7.1 PMI 3D Data.

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 lye 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.

6.2.6.2.1.10 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 156: PMI Locator Entities data collection

I32 : Locator Count

PMI 2D Data
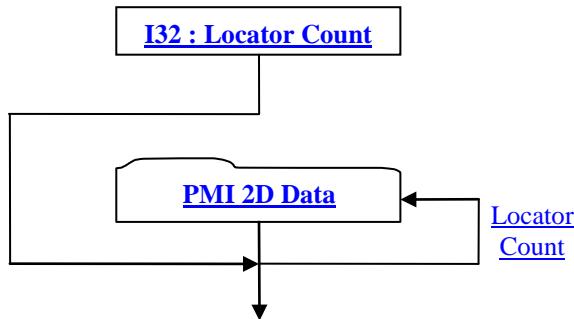
Locator Count

Complete description for PMI 2D Data can be found in 6.2.6.2.1.1.1 PMI 2D Data.

I32 : Locator Count

Locator Count specifies the number of Locator symbol entities.

6.2.6.2.1.11 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 6.2.6.2.1.7.1 PMI 3D Data) as follows:

Polyline Segment Index[1]	Implied Reference Geometry Type
= = 1	Point
= = 2	Polyline
> 2	Polygon

Figure 157: PMI Reference Geometry Entities data collection

I32 : Reference Geometry Count

PMI 3D Data

Reference Geometry Count
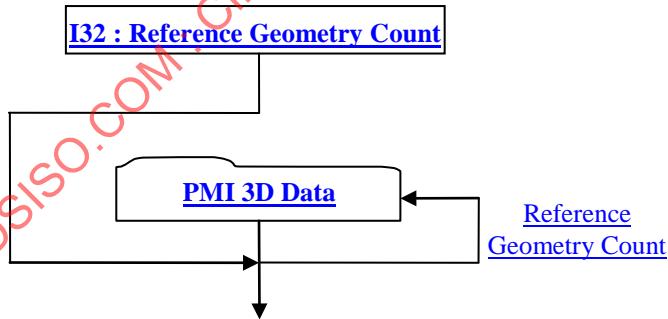
Complete description for PMI 3D Data can be found in 6.2.6.2.1.7.1 PMI 3D Data.

I32 : Reference Geometry Count

Reference Geometry Count specifies the number of Reference Geometry entities.

6.2.6.2.1.12 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 6.2.6.2.2 PMI Associations), where specific PMI entities are associated as "destinations" to a "source" PMI Design Group.

Figure 158: PMI Design Group Entities data collection
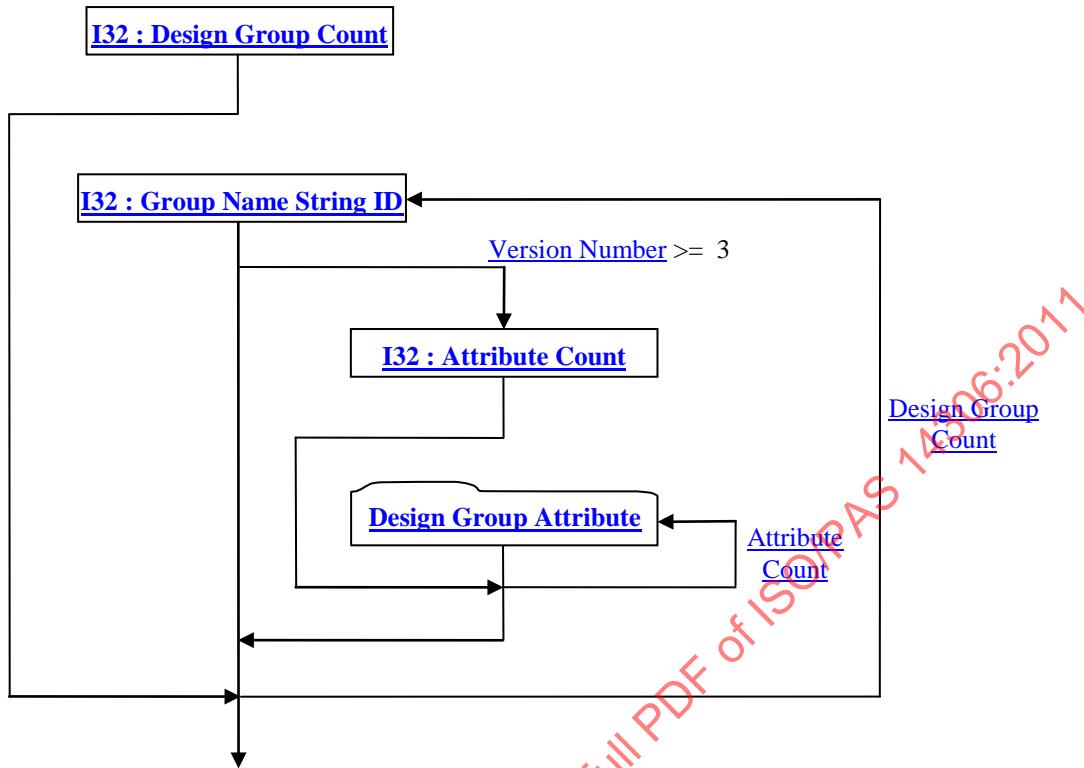
I32 : Design Group Count
Design Group Count specifies the number of Design Group entities.

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 6.2.6.2.4 PMI String Table. An identifier value of "-1" indicates no string.

I32 : Attribute Count
Attribute Count specifies the number of Design Group Attribute data collections

6.2.6.2.1.12.1 Design Group Attribute

The Design Group Attribute data collection defines a group property/attribute.

Figure 159: Design Group Attribute data collection
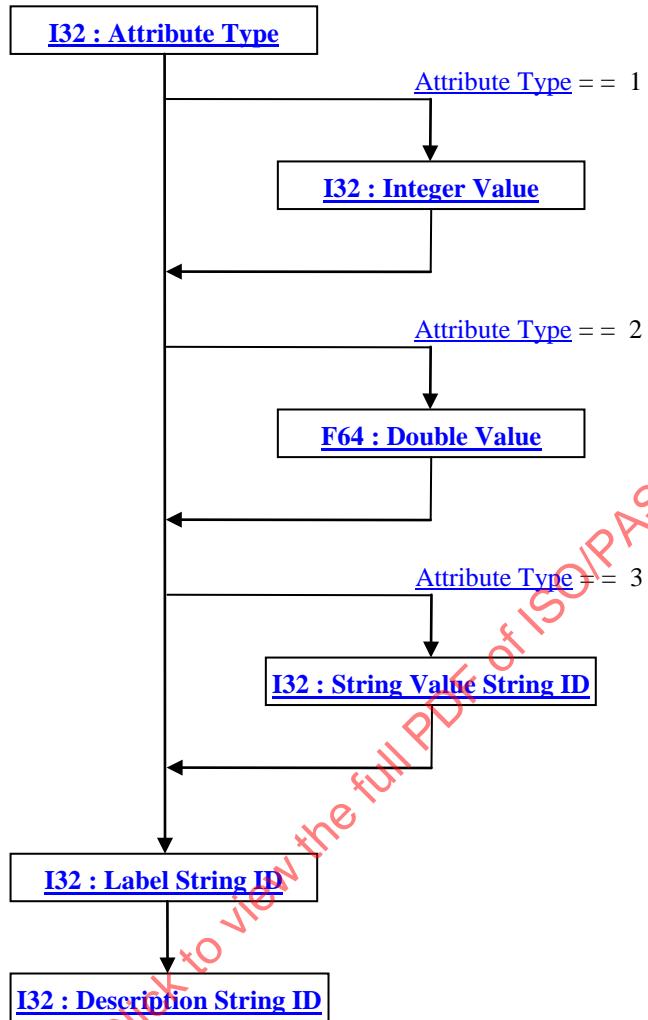
I32 : Attribute Type
Attribute Type specifies the attribute type. Valid types include the following:

= 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 6.2.6.2.4 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 6.2.6.2.4 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 6.2.6.2.4 PMI String Table. An identifier value of "-1" indicates no string.

6.2.6.2.1.13 PMI Coordinate System Entities

The PMI Coordinate System Entities data collection defines data for a list of Coordinate Systems.

Figure 160: PMI Coordinate System Entities data collection
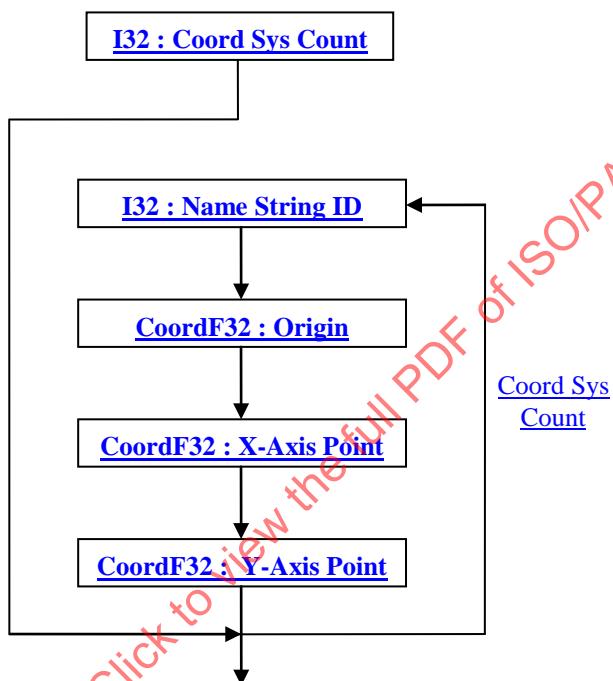
I32 : Coord Sys Count

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 6.2.6.2.4 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.

6.2.6.2.2 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 161: PMI Associations data collection
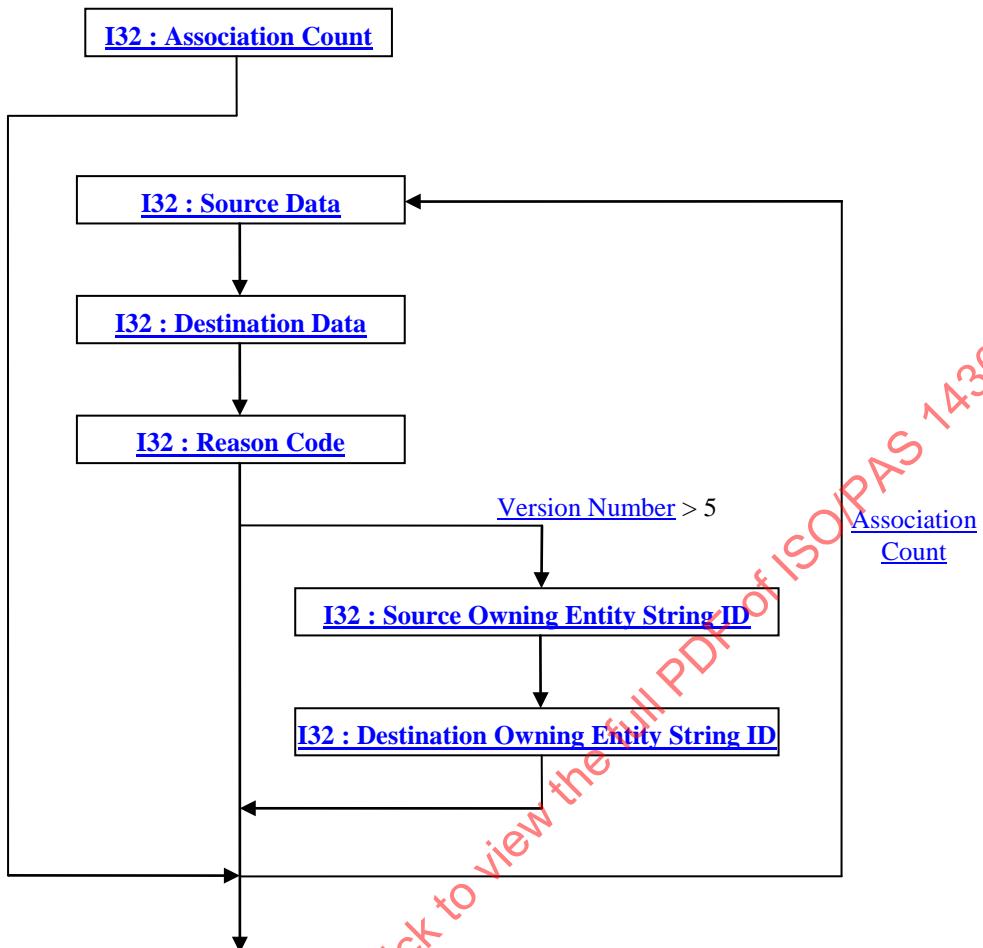
I32 : Association Count

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 undocumented bits are reserved.

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 6.2.6.2.7 PMI CAD Tag Data) identifying the CAD Tag belonging to the particular source entity.

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 : Reason Code

Reason Code specifies the "reason" for the association. Valid Reason Codes include the following:

= 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 : 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 6.2.6.2.4 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). This data field is only present if Version Number, as defined in 6.2.6.2 PMI Manager Meta Data Element, is greater than "5".

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 6.2.6.2.4 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). This data field is only present if Version Number, as defined in 6.2.6.2 PMI Manager Meta Data Element, is greater than "5".

6.2.6.2.3 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 162: PMI User Attributes data collection
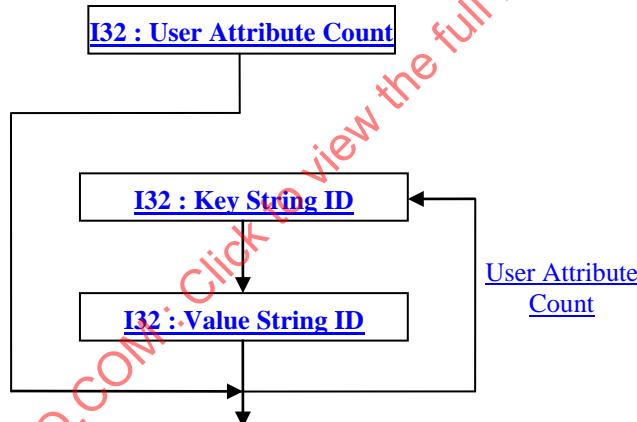
I32 : User Attribute Count

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 6.2.6.2.4 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 6.2.6.2.4 PMI String Table. An identifier value of "-1" indicates no string.

6.2.6.2.4 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 163: PMI String Table data collection
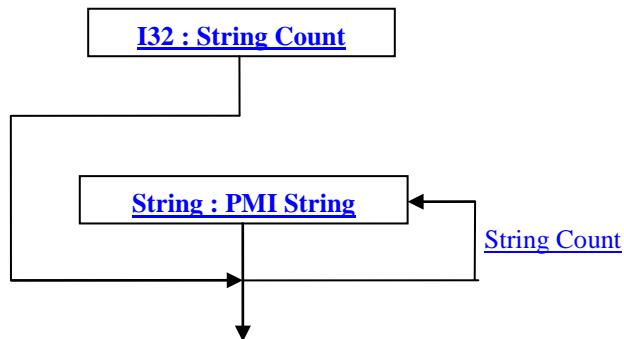
I32 : String Count

String Count specifies the number of character strings in the string table.

String : PMI String

PMI String specifies the character string.

6.2.6.2.5 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 6.2.6.2.2 PMI Associations), where specific PMI entities are associated as "destinations" to a "source" PMI Model View.

Figure 164: PMI Model Views data collection
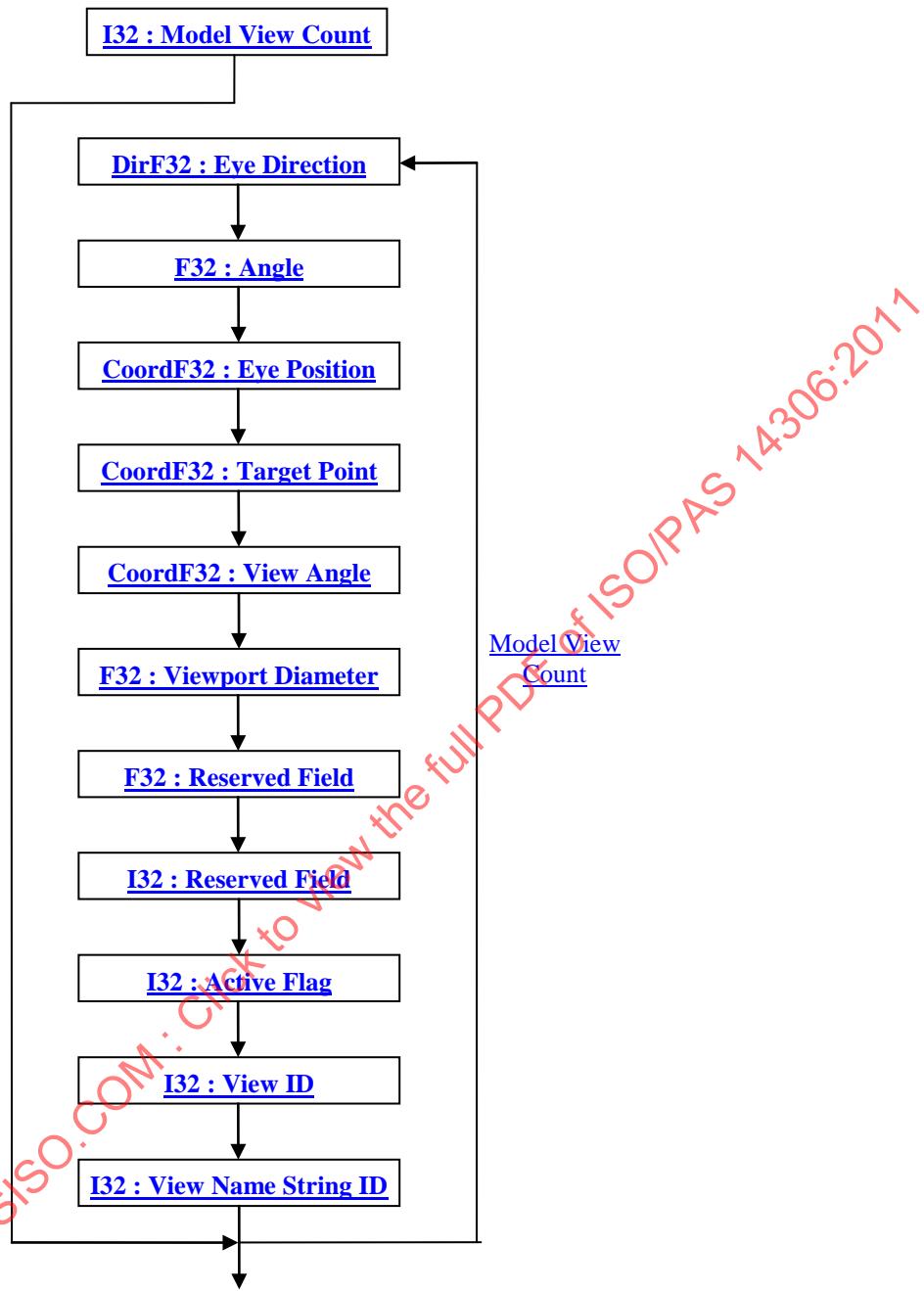
I32 : Model View Count
Model View Count specifies the number of Model Views.

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 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

I32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion

I32 : Active Flag

Active Flag is a flag specifying whether this Model View is the "active" view. Valid values include the following:

= 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 6.2.6.2.4 PMI String Table. An identifier value of "-1" indicates no string.

6.2.6.2.6 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 165: Generic PMI Entities data collection
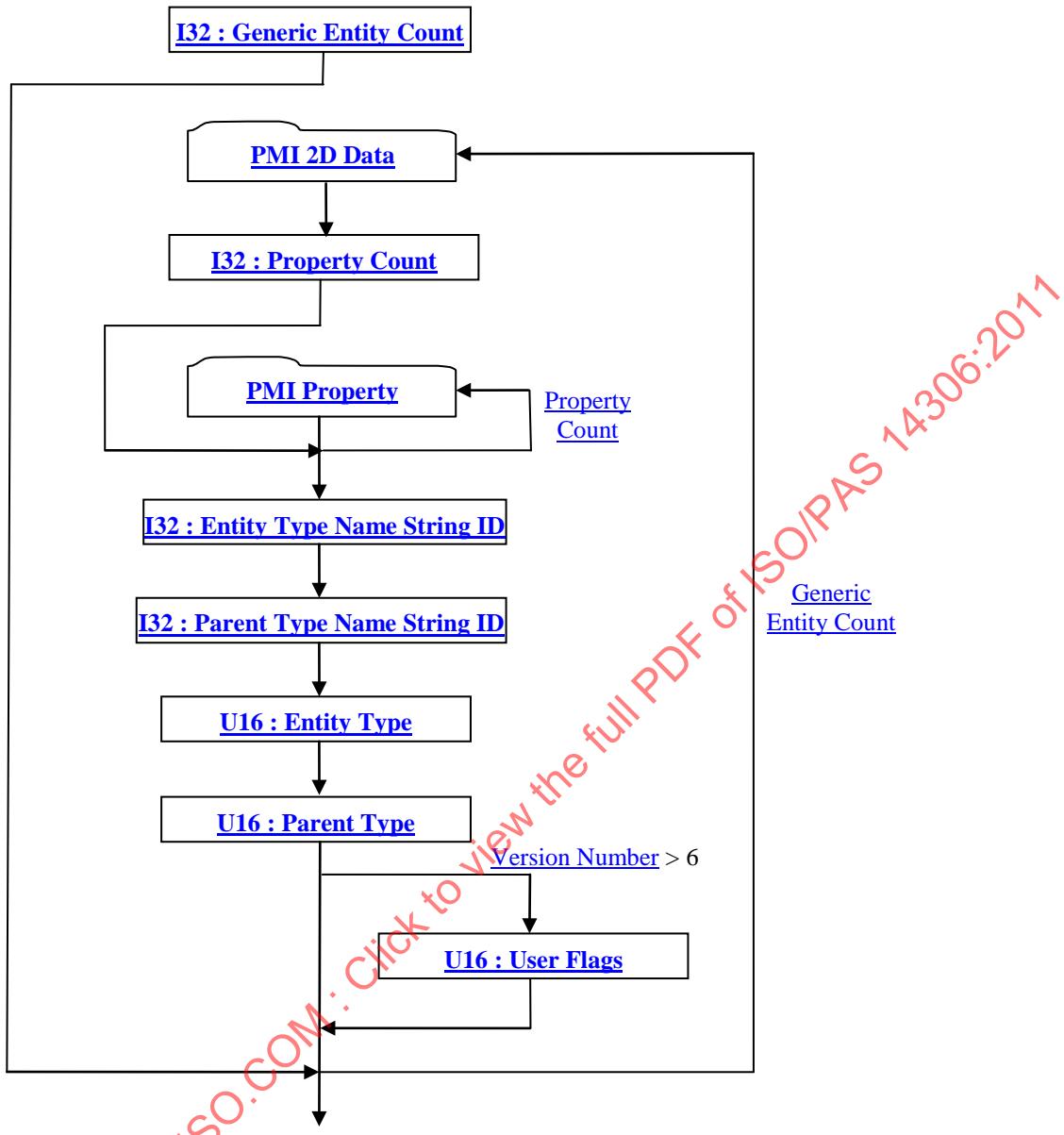
Complete description for PMI 2D Data can be found in 6.2.6.2.1.1.1 PMI 2D Data.

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.

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 6.2.6.2.4 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 6.2.6.2.4 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.

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

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 undocumented bits are reserved.

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.

6.2.6.2.6.1 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 166: PMI Property data collection

Key PMI Property Atom

Value PMI Property Atom
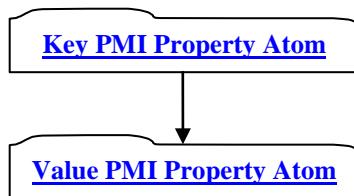
Both Key PMI Property Atom and Value PMI Property Atom have the same format as that documented in 6.2.6.2.6.1.1 PMI Property Atom.

Although there is no reference compliant requirements for what the PMI Property key/value pairs must 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 6.2.6.2.6.1.1 PMI Property Atom for an explanation of what is meant by "encoded string value" for the "key" and "value" data).

Table 7: 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
"PMITextForegroundColor"	"#"	Hexadecimal integer representing RGB color 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 must be zero. The maximum value for a single byte is 0xFF (i.e. intensity value is in the range [0:255]).
"PMITextBackgroundColor"	"#"	Same as "PMITextForegroundColor"
"PMITextBackgroundOpacity"	"#"	Unsigned decimal integer representing opacity percentage. Actual opacity is: decoded# / 100.0
"PMITextShowBorder"	"#"	Unsigned decimal integer: 0 = = False; 1 = = True

"Key" Property Atom Value String	"Value" Property Atom Value String Encoding Format	Decoding Notes
"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 displayed in the plane of the entity so that it rotates with view.
"PMIGeometryColor"	"#"	Same as "PMITextForegroundColor"
"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.

6.2.6.2.6.1.1 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 167: PMI Property Atom data collection
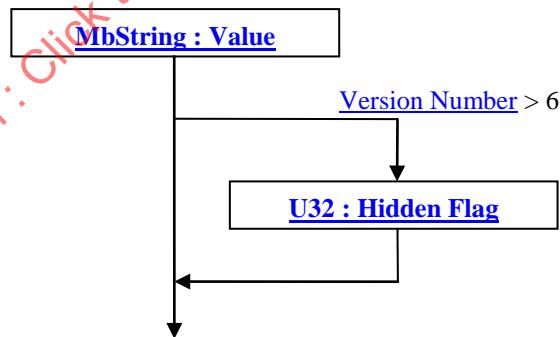
MbString : Value

Value specifies the property atom value encoded into a String. See Table 7: 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:

= 0	− Property is not hidden.
= 1	− Property is hidden.

6.2.6.2.7 PMI CAD Tag Data

The PMI CAD Tag Data collection contains the list of persistent IDs, as defined in the CAD System, to uniquely identify individual PMI entities. The existence of this PMI CAD Tag Data collection is dependent upon the value of previously read data field CAD Tags Flag as documented in 6.2.6.2 PMI Manager Meta Data Element.

If PMI CAD Tag Data 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 168: PMI CAD Tag Data data collection
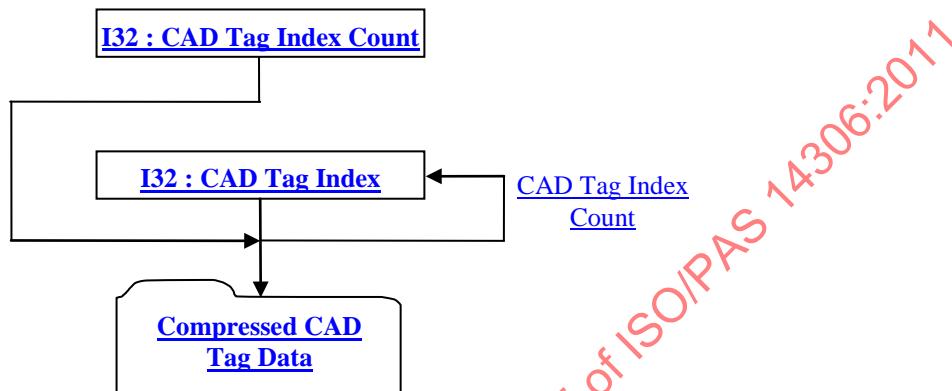
Complete description for Compressed CAD Tag Data can be found in 7.1.11 Compressed CAD Tag Data.

I32 : CAD Tag Index Count

CAD Tag Index Count specifies the total number of CAD Tag indices. This value must be equal to the summation of the previously read count values for all the PMI entities supporting CAD Tags. The formula is as follows:

CAD Tag Index Count = 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.)

6.2.7 PMI Data Segment

The PMI Manager Meta Data Element (as documented in 6.2.6.2 PMI Manager Meta Data Element) can sometimes also be represented in a PMI Data Segment. This can occur when a pre JT 8 version file is migrated to JT 8.1 version file. So from a parsing point of view a PMI Data Segment should be treated exactly the same as a 6.2.6 Meta Data Segment.

7 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 ZLIB deflation to advanced arithmetic 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 trade off 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 a Huffman or Arithmetic CODEC (see 7.2 Encoding Algorithms).

Beyond the data collection specific compression/encoding, some JT format Data Segment types (see 6.1.3 Data Segment) also support having a ZLIB 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 additional compressed using a ZLIB deflation algorithm.

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 C:Decoding Algorithms – An Implementation.

7.1   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 6.2 Data Segments section of the document.

7.1.1   Int32 Compressed Data Packet

The Int32 Compressed Data Packet collection represents the format used to encode/compress a collection of data into a series of Int32 based symbols. Note that the Int32 Compressed Data Packet collection can in itself contain another Int32 Compressed Data Packet collection if there are any "Out-Of-Band data." In the context of the JT format data compression algorithms and Int32 Compressed Data Packet, "out-of-band data" has the following meaning.

CODECs like arithmetic and Huffman (see 7.2 Encoding Algorithms for technical description) exploit the statistics present in the relative frequencies of the values being encoded. Values that occur frequently enough allow both of these methods to encode each of the values as a "symbol" in fewer bits that it would take to encode the value itself. 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 7.1.1.1.1 Int32 Probability Context Table Entry for futher details on the representation of "escape" symbol).

Essentially the "out-of-band data" is the high-entropy junk/residue/slag left over after the CODECs have squeezed all the advantage out that they can. However, this "out-of-band data" is sent back around because sometimes there are *different* statistics to be taken advantage of. When you're down to the *really* icky "out-of-band" slag, 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.

Figure 169: Int32 Compressed Data Packet data collection
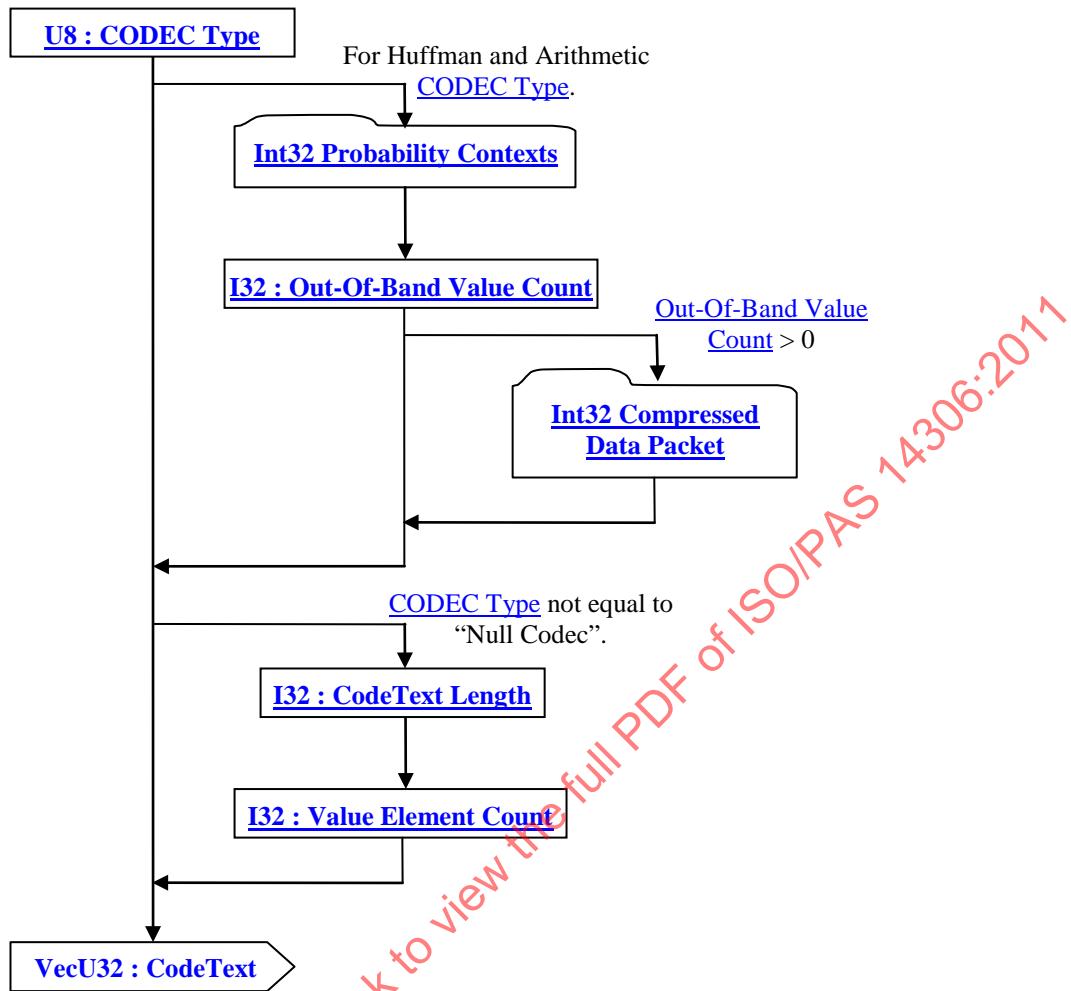
U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See 7.2 Encoding Algorithms for complete explanation of each of the encoding algorithms.

= 0	– Null CODEC
= 1	– Bitlength CODEC
= 2	– Huffman CODEC
= 3	– Arithmetic CODEC

I32 : Out-Of-Band Value Count

Out-Of-Band Value Count specifies the number of values that are "Out-Of-Band." This data field is only present for Huffman and Arithmetic CODEC Types.

I32 : CodeText Length

CodeText Length specifies the total number of bits of CodeText data (CodeText data field is described below). This data field is only present if CODEC Type is not equal to "Null CODEC."

I32 : Value Element Count

Value Element 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). This data field is only present if CODEC Type is not equal to "Null CODEC." Upon completion of decoding the CodeText data field below, the number of decoded symbol values should be equal to Value Element Count.

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.

7.1.1.1 Int32 Probability Contexts

Int32 Probability Contexts data collection is a list of Probability Context Tables. The Int32 Probability Contexts data collection is only present for Huffman and Arithmetic CODEC Types. 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, and gives all the information necessary to reconstruct the Huffman codes for the Huffman CODEC.

Figure 170: Int32 Probability Contexts data collection
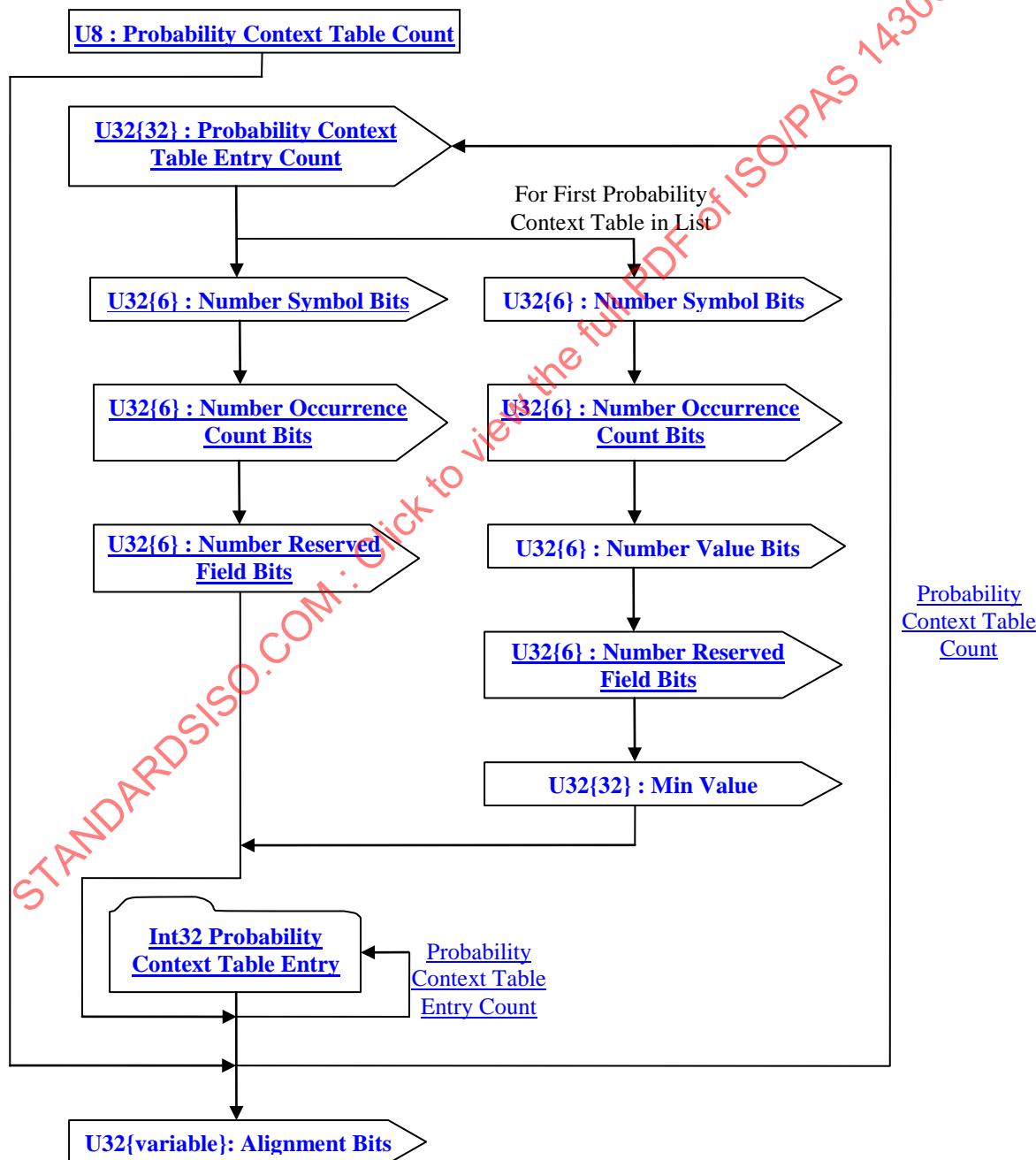
U8 : Probability Context Table Count

Probability Context Table Count specifies the number of Probability Context Tables to follow and will always have a value of either "1" or "2".

U32{32} : Probability Context Table Entry Count

Probability Context Table Entry Count specifies the number of entries in this Probability Context Table.

U32{6} : Number Symbol Bits

Number Symbol Bits specifies the number of bits used to encode the Symbol range.

U32{6} : Number Occurrence Count Bits

Number Occurrence Count Bits specifies the number of bits used to encode the Occurrence Count range.

U32{6} : Number Value Bits

Number Value Bits specifies the number of bits used to encode the Associated Value range.

U32{6} : Number Reserved Field Bits

Number Reserved Field Bits specifies the number of bits used for the Reserved Field in 7.1.1.1.1 Int32 Probability Context Table Entry.

U32{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 7.1.1.1.1 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 the JtFile 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.

7.1.1.1.1 Int32 Probability Context Table Entry

Figure 171: Int32 Probability Context Table Entry data collection

U32{Number Symbol Bits} : Symbol

U32{Number Occurrence Count Bits} : Occurrence Count

U32{Number Value Bits} : Associated Value

U32{Number Reserved Field Bits} : Reserved Field
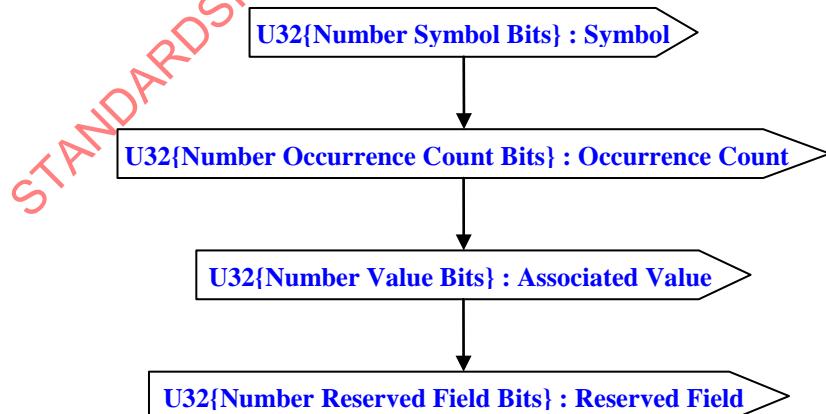
U32{Number Symbol Bits} : Symbol

Symbol is a small integer number associated with a specific value in the context table. It serves only to impose an order on the entries in the Probability Context Table. The symbol is stored with a "+2" added to the value and thus a reader must subtract "2" from the read value to get the true symbol value. Complete description for Number Symbol Bits can be found in 7.1.1.1 Int32 Probability Contexts.

Note: Even though the symbol is written as a U32{Number Symbol Bits} it is possible to end up with a negative number after subtracting "2" from the read in value. One example that will occur frequently is the escape symbol used for out-of-band data which will have the value "0" in the file, however it will become "-2", its true symbol value, after subtracting "2" from the read in "0" value.

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 7.1.1.1 Int32 Probability Contexts.

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. 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 Value Element Count in section 7.1.1 for number of symbols to be decoded).

- During Arithmetic decoding as described in C.4.2.

  o *pDriver->numSymbolsToRead()* – Refers to the total number of symbols to be decoded (i.e. Value Element Count in section 7.1.1).

  o *pCurrContext->totalCount()* – Refers to the sum of the "Occurrence Count" values for all the symbols associated with a Probability Context.

U32{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 7.1.1.1 Int32 Probability Contexts.

Note: The associated value for an escape symbol is undefined and therefore can be any valid U32 number.

U32{Number Reserved Field Bits} : Reserved Field

Reserved Field is a data field reserved for future JT format expansion. Complete description for Number Reserved Field Bits can be found in 7.1.1.1 Int32 Probability Contexts.

7.1.2 Float64 Compressed Data Packet

The Float64 Compressed Data Packet collection represents the format used to encode/compress a collection of data into a series of Float64 based symbols. This compression format also uses the concept of "out-of-band data" in its data contents definition. In the context of the JT format data compression algorithms and Float64 Compressed Data Packet, "out-of-band data" has the following meaning.

CODECs like arithmetic and Huffman (see 7.2 Encoding Algorithms for technical description) exploit the statistics present in the relative frequencies of the values being encoded. Values that occur frequently enough allow both of these methods to encode each of the values as a "symbol" in fewer bits that it would take to encode the value itself. Values that occur too infrequently to take advantage of this property are written *aside* into the "out-of-band data" array. An "escape" symbol (i.e. value of "-2") is encoded in their place as a placeholder in the primal CODEC. Essentially the "out-of-band data" is the high-entropy junk/residue/slag left over after the CODECs have squeezed all the advantage out that they can.

Whereas the Int32 Compressed Data Packet (see 7.1.1 Int32 Compressed Data Packet) then sends this "out-of-band data" back around through a new CODEC looking for *different* statistics to be taken advantage of, the Float64 Compressed Data Packet simply writes out the "out-of-band data" array with no additional encoding attempted.

Figure 172: Float64 Compressed Data Packet data collection
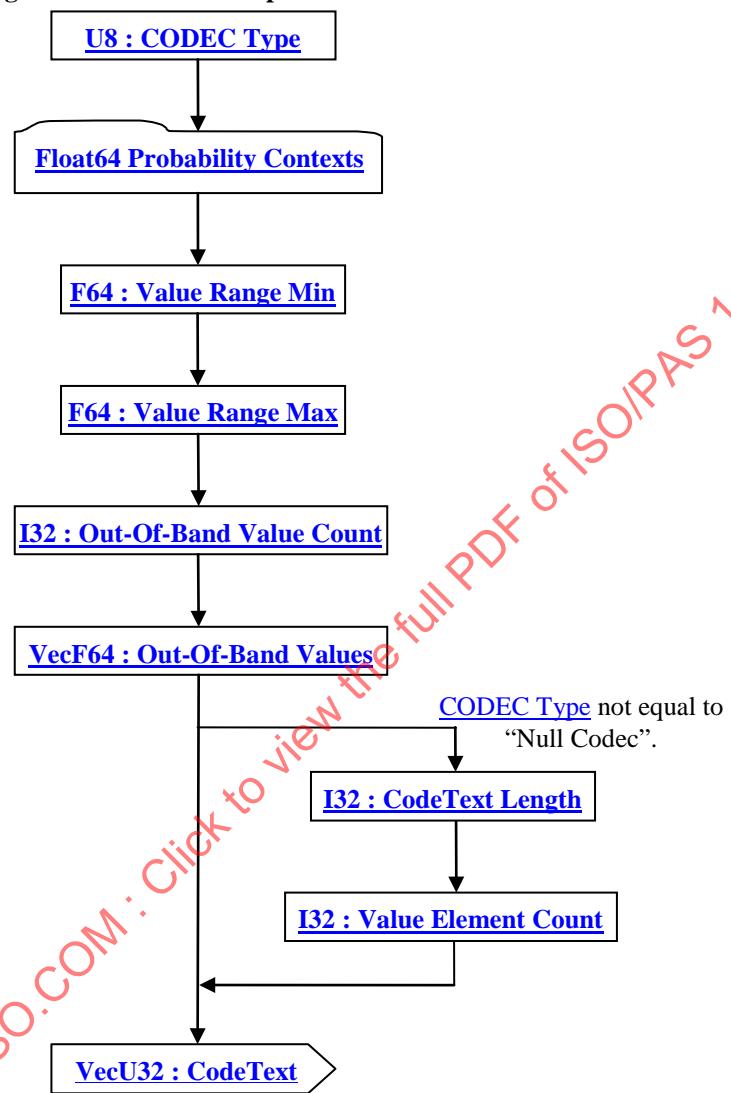
U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See 7.2 Encoding Algorithms for complete explanation of each of the encoding algorithms.

= 0	− Null CODEC
= 1	− Bitlength CODEC
= 2	− Huffman CODEC
= 3	− Arithmetic CODEC

F64 : Value Range Min

Value Range Min specifies the minimum of the value range used to encode the values.

F64 : Value Range Max

Value Range Max specifies the maximum of the value range used to encode the values.

I32 : Out-Of-Band Value Count

Out-Of-Band Value Count specifies the number of values that are "Out-Of-Band."

VecF64 : Out-Of-Band Values

Out-Of-Band Values specifies the vector/list of "Out-Of-Band" values.

I32 : CodeText Length

CodeText Length specifies the total number of bits of CodeText data (described below). This data field is only present if CODEC Type is not equal to "Null CODEC."

I32 : Value Element Count

Value Element 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). This data field is only present if CODEC Type is not equal to "Null CODEC." Upon completion of decoding the CodeText data field below, the number of decoded symbol values should be equal to Value Element Count.

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.

7.1.2.1 Float64 Probability Contexts

Float64 Probability Contexts data collection is a list of Probability Context Tables. 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, and gives all the information necessary to reconstruct the Huffman codes for the Huffman CODEC.

Figure 173: Float64 Probability Contexts data collection
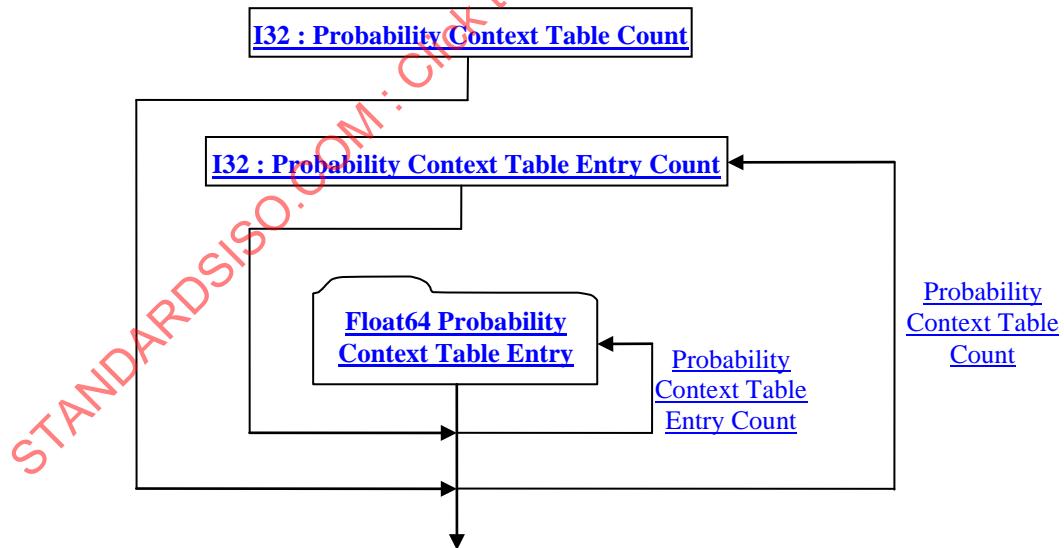
I32 : Probability Context Table Count

Probability Context Table Count specifies the number of Probability Context Tables to follow and will always have a value of either "1" or "2".

I32 : Probability Context Table Entry Count

Probability Context Table Entry Count specifies the number of entries in this Probability Context Table.

7.1.2.1.1 Float64 Probability Context Table Entry

Figure 174: Float64 Probability Context Table Entry data collection

```
┌─────────────────────────┐
│      I32 : Symbol        │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│  I32 : Occurrence Count  │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│   F64 : Associated Value │
└─────────────────────────┘
            │
            ▼
┌─────────────────────────┐
│   I32 : Reserved Field   │
└─────────────────────────┘
```
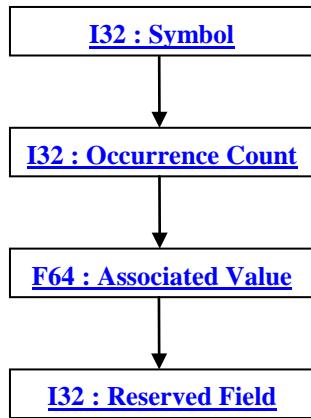
I32 : Symbol

Symbol is a small integer number associated with a specific value in the context table. It serves only to impose an order on the entries in the Probability Context Table. Note that a value of "-2" represents the "escape" symbol placeholder encoded for "out-of-band data" (see 7.1.2 Float64 Compressed Data Packet for additional details).

I32 : Occurrence Count

Occurrence Count specifies the relative frequency of the value.

F64 : 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.

I32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

7.1.3 Vertex Based Shape Compressed Rep Data

The Vertex Based Shape Compressed Rep Data collection is the compressed and/or encoded representation of the vertex coordinates, normal, texture coordinate, and color data for a vertex based shape. All vertex based shape elements (e.g. Tri-Strip Set Shape LOD Element, Polyline Set Shape LOD Element) use this data collection format to compress/encode their geometric data.

Figure 175: Vertex Based Shape Compressed Rep Data data collection
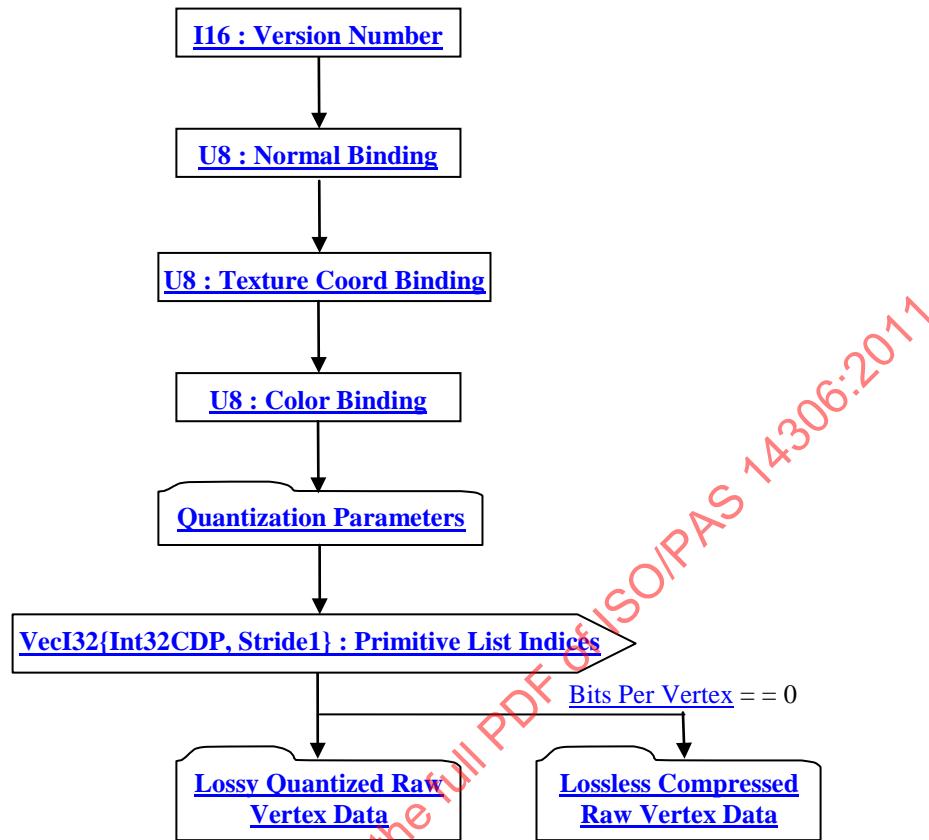
Complete description for Quantization Parameters can be found in 6.2.1.1.1.10.2.1.1Quantization Parameters.

I16 : Version Number

Version Number is the version identifier for this Vertex Based Shape Rep Data. Version number "0x0001" is currently the only valid value.

U8 : Normal Binding

Normal Binding specifies how (at what granularity) normal vector data is supplied ("bound") for the Shape Rep in either the Lossless Compressed Raw Vertex Data or Lossy Quantized Raw Vertex Data collections.

= 0	– None. No normal data.
= 1	– Per Vertex. Normal vector for every vertex.
= 2	– Per Facet. Normal vector for every face/polygon.
= 3	– Per Primitive. Shape has a normal vector for each shape primitive (e.g. a 6.2.1.1.1.10.3 Tri-Strip Set Shape Node Element is made up of a collection of independent and unconnected triangle strips; where each strip constitutes one primitive of the shape and thus there would be a normal per triangle strip).

U8 : Texture Coord Binding

Texture Coord Binding specifies how (at what granularity) texture coordinate data is supplied ("bound") for the Shape Rep in either the Lossless Compressed Raw Vertex Data or Lossy Quantized Raw Vertex Data collections. Valid values are the same as documented for Normal Binding data field.

U8 : Color Binding

Color Binding specifies how (at what granularity) color data is supplied ("bound") for the Shape Rep in either the Lossless Compressed Raw Vertex Data or Lossy Quantized Raw Vertex Data collections. Valid values are the same as documented for Normal Binding data field..

Vecl32{Int32CDP, Stride1} : 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.

7.1.3.1 Lossless Compressed Raw Vertex Data

The Lossless Compressed Raw Vertex Data collection contains all the per-vertex information (i.e. UV texture coordinates, color, normal vector, XYZ coordinate) stored in a "lossless" compression format for all primitives of the shape. The Lossless Compressed Raw Vertex Data collection is only present when the Quantization Parameters Bits Per Vertex data field equals "0" (See 6.2.1.1.1.10.2.1.1 Quantization Parameters for complete description).

Figure 176: Lossless Compressed Raw Vertex Data data collection
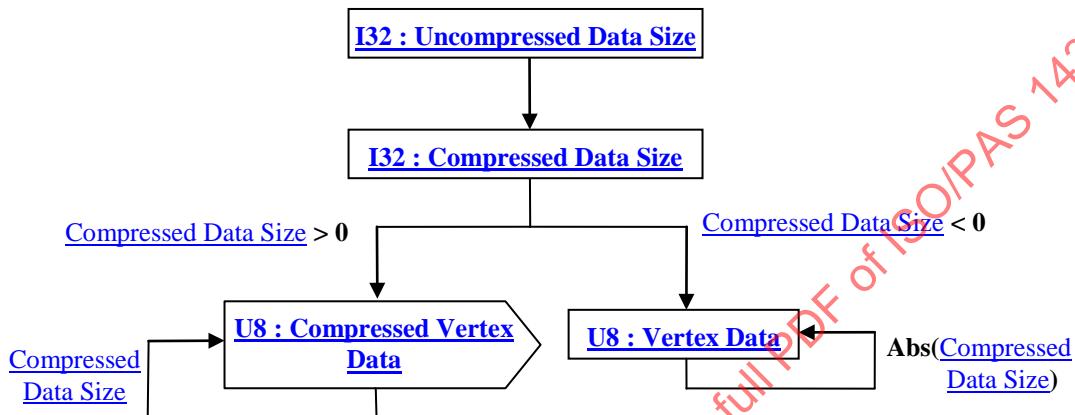
I32 : Uncompressed Data Size

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

I32 : Compressed Data Size

Compressed Data Size specifies the compressed size of Vertex Data or Compressed Vertex Data in bytes. If the Compressed Data Size is negative, then the Compressed Vertex 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 : Vertex Data

The Vertex Data field is a packed array of the raw per vertex data (i.e. UV texture coordinates, color, normal vector, XYZ coordinate). The Vertex Data field is only present if Compressed Data Size value is less than zero.

The existence of texture coordinate, color, and normal vector data within Vertex Data array is dependent upon the Normal Binding, Texture Coord Binding, and Color Binding values previously read for this shape (see 7.1.3Vertex Based Shape Compressed Rep Data). Note that XYZ coordinate data is always present.

The per vertex data is packed in Vertex Data array as F32 types using an interleaved data format/order as follows:

{[u,v] [r,g,b] [nx,ny,nz] x,y,z}, {[u,v] [r,g,b] [nx,ny,nz] x,y,z}, …, for all vertices.

Where the data elements have the following meaning:

[u, v]	- Texture Coordinates for Vertex
[r, g, b]	- Red, Green, Blue color components for Vertex
[nx, ny, nz]	- X, Y, Z Normal Vector components for Vertex
x, y, z	- X, Y, Z Position Coordinate for Vertex

Given this format of the Vertex Data, the previously read vertex binding information, and the previously read Primitive List Indices, a reader can then implicitly compute the data stride (length of one vertex entry in Vertex Data), number of primitives, and number of vertices for the shape.

U8 : Compressed Vertex Data

The Compressed Vertex Data field represents the same data as documented in Vertex Data field above except that the data is compressed using the general "ZLIB deflation compression" method. The Compressed Vertex Data field is only present if Compressed Data Size value is greater than zero. See 7 Data Compression and Encoding for more details on ZLIB compression and ZLIB library version used.

7.1.3.2 Lossy Quantized Raw Vertex Data

The Lossy Quantized Raw Vertex Data collection contains all the per-vertex information (i.e. UV texture coordinates, color, normal vector, XYZ coordinate) stored in a "lossy" encoding/compression format for all primitives of the shape. The Lossy Quantized Raw Vertex Data collection is only present when the Quantization Parameters Bits Per Vertex data field is NOT equal to "0" (See 6.2.1.1.1.10.2.1.1 Quantization Parameters for compete description).

Figure 177: Lossy Quantized Raw Vertex Data data collection
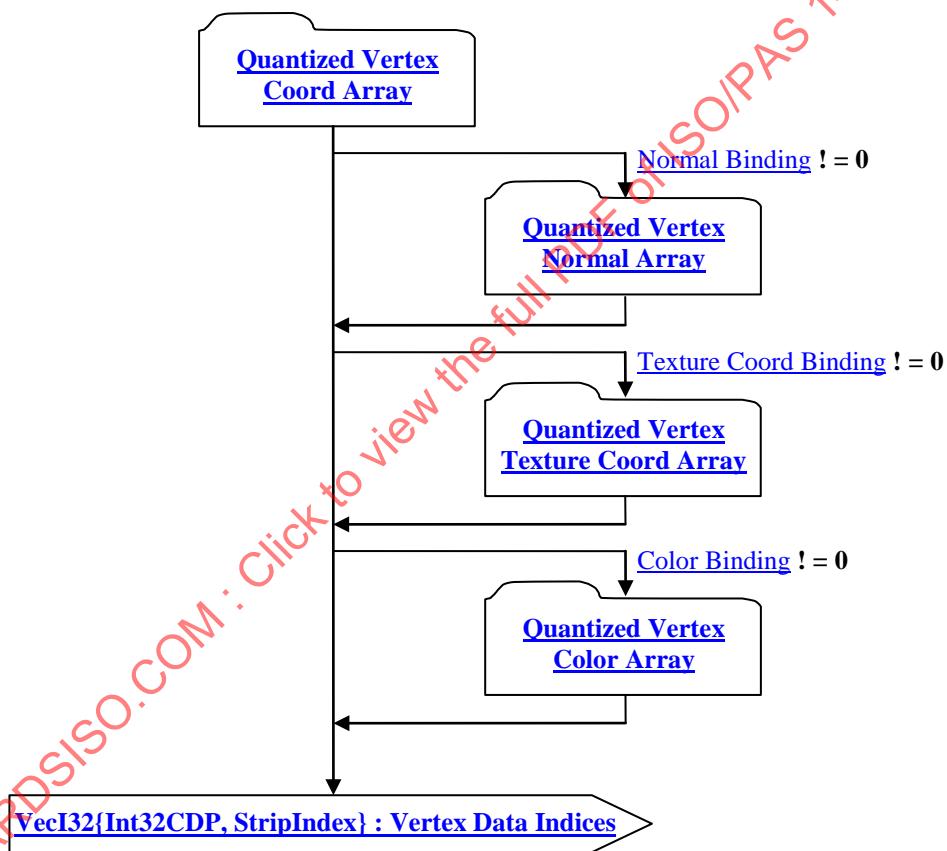
VecI32{Int32CDP, StripIndex} : Vertex Data Indices

Vertex Data 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 Colors) 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.

7.1.3.2.1 Quantized Vertex Coord Array

The Quantized Vertex Coord Array data collection contains the quantization data/representation for a set of vertex coordinates.

Figure 178: Quantized Vertex Coord Array data collection
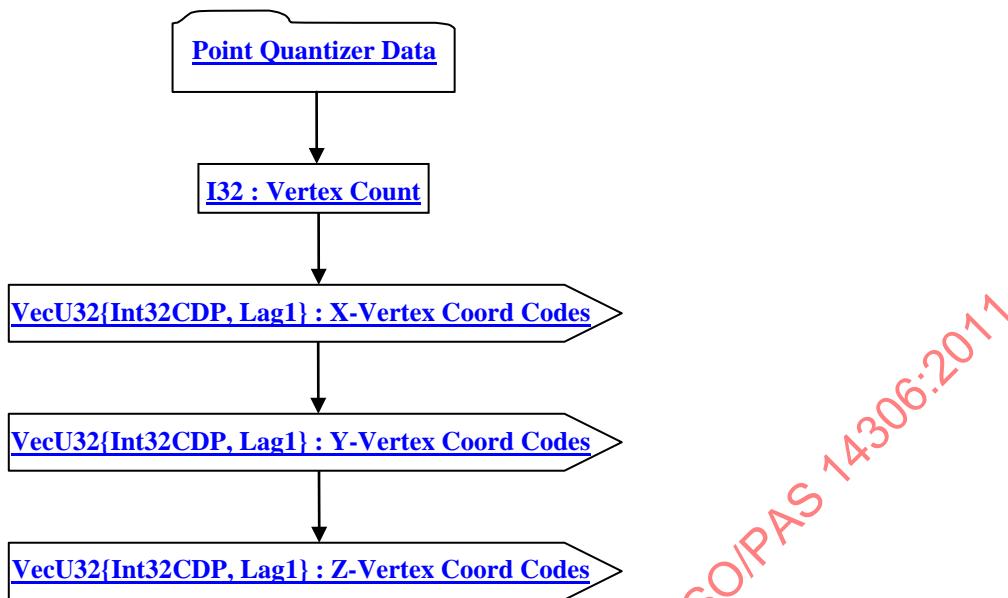
Complete description for Point Quantizer Data can be found in 7.1.4 Point Quantizer Data.

I32 : Vertex Count

Vertex Count specifies the count (number of unique) vertices in the Vertex Codes arrays.

VecU32{Int32CDP, Lag1} : X-Vertex Coord Codes

X Vertex Coord Codes is a vector of quantizer "codes" for all the X-components of a set of vertex coordinates. X-Vertex Coord Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : Y-Vertex Coord Codes

Y Vertex Coord Codes is a vector of quantizer "codes" for all the Y-components of a set of vertex coordinates. Y-Vertex Coord Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : Z-Vertex Coord Codes

Z Vertex Coord Codes is a vector of quantizer "codes" for all the Z-components of a set of vertex coordinates. Z-Vertex Coord Codes uses the Int32 version of the CODEC to compress and encode data.

7.1.3.2.2 Quantized Vertex Normal Array

The Quantized Vertex Normal Array data collection contains the quantization data/representation for a set of vertex normals. Quantized Vertex Normal Array data collection is only present if previously read Normal Binding value is not equal to zero (See 7.1.3Vertex Based Shape Compressed Rep Data for complete explanation of Normal Binding data field).

A variation of the CODEC developed by Michael Deering at Sun Microsystems is used to encode the normals. The variation being that the "Sextants" are arranged differently than in Deering's scheme [6], for better delta encoding. See 7.2.5 Deering Normal CODEC for a complete explanation on the Deering CODEC used.

Figure 179: Quantized Vertex Normal Array data collection
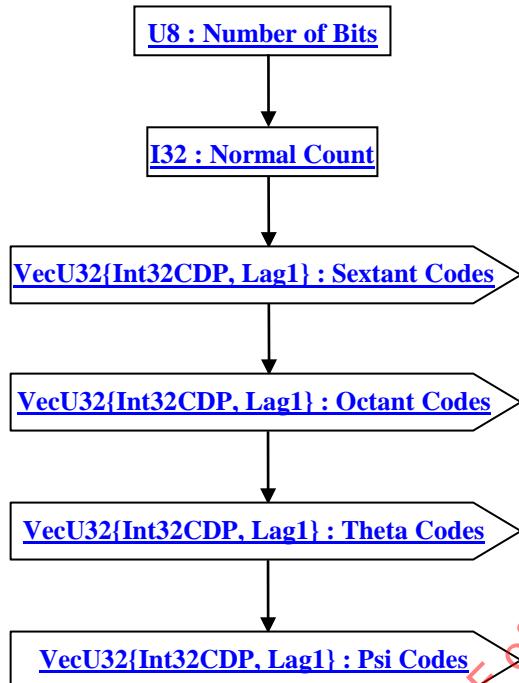
U8 : Number of Bits

I32 : Normal Count

VecU32{Int32CDP, Lag1} : Sextant Codes

VecU32{Int32CDP, Lag1} : Octant Codes

VecU32{Int32CDP, Lag1} : Theta Codes

VecU32{Int32CDP, Lag1} : Psi Codes

U8 : Number of Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision) for the Theta and PSI angles. This value must satisfy the following condition: "0 <= Number of Bits <= 13".

I32 : Normal Count

Normal Count specifies the count (number of unique) Normal Codes.

VecU32{Int32CDP, Lag1} : Sextant Codes

Sextant Codes is a vector of "codes" (one per normal) for a set of normals identifying which Sextant of the corresponding sphere Octant each normal is located in. Sextant Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : Octant Codes

Octant Codes is a vector of "codes" (one per normal) for a set of normals identifying which sphere Octant each normal is located in. Octant Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : Theta Codes

Theta Codes is a vector of "codes" (one per normal) for a set of normals representing in Sextant coordinates the quantized theta angle for each normal's location on the unit radius sphere; where theta angle is defined as the angle in spherical coordinates about the Y-axis on a unit radius sphere. Theta Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : Psi Codes

Psi Codes is a vector of "codes" (one per normal) for a set of normals representing in Sextant coordinates the quantized Psi angle for each normal's location on the unit radius sphere; where Psi angle is defined as the longitudinal angle in spherical coordinates from the y = 0 plane on the unit radius sphere. Psi Codes uses the Int32 version of the CODEC to compress and encode data

7.1.3.2.3 Quantized Vertex Texture Coord Array

The Quantized Vertex Texture Coord Array data collection contains the quantization data/representation for a set of vertex texture coordinates. Quantized Vertex Texture Coord Array data collection is only present if previously read Texture Coord Binding value is not equal to zero (See 7.1.3Vertex Based Shape Compressed Rep Data for complete explanation of Texture Coord Binding data field).

Figure 180: Quantized Vertex Texture Coord Array data collection
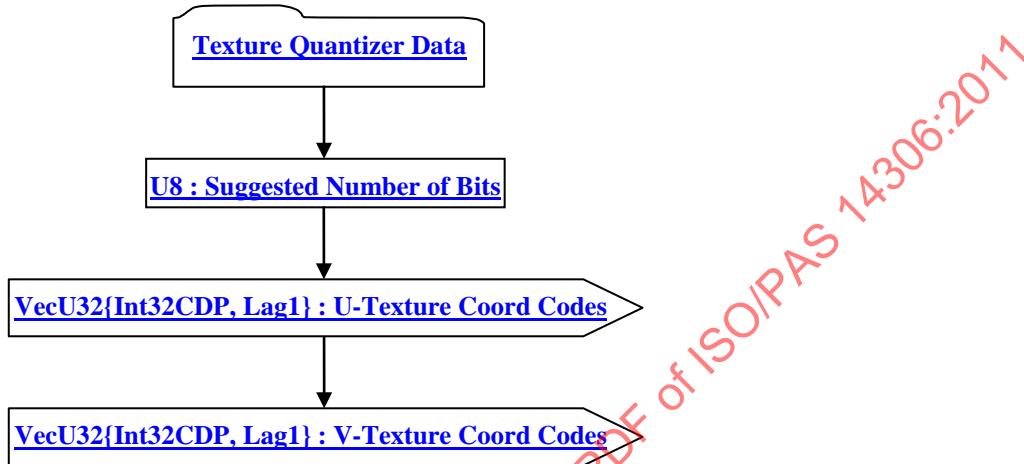
 Complete description for Texture Quantizer Data can be found in 7.1.5 Texture Quantizer Data.

U8 : Suggested Number of Bits

Suggested Number of Bits specifies the suggested number of quantization bits per texture coordinate U and V components. It is only a suggested value (and has no real value for a JT file loader/reader) because the actual number of bits used may differ (increased or decreased) depending on the range of values for texture coordinates. The actual number of quantization bits used is specified within Texture Quantizer Data. Value must be within range [0:24] inclusive.

VecU32{Int32CDP, Lag1} : U-Texture Coord Codes

U-Texture Coord Codes is a vector of quantizer "codes" for all the U-components of a set of vertex texture coordinates. U-Texture Coord Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : V-Texture Coord Codes

V-Texture Coord Codes is a vector of quantizer "codes" for all the V-components of a set of vertex texture coordinates. V-Texture Coord Codes uses the Int32 version of the CODEC to compress and encode data.

7.1.3.2.4 Quantized Vertex Color Array

The Quantized Vertex Color Array data collection contains the quantization data/representation for a set of vertex colors. Quantized Vertex Color Array data collection is only present if previously read Color Binding value is not equal to zero (See 7.1.3Vertex Based Shape Compressed Rep Data for complete explanation of Color Binding data field).

Figure 181: Quantized Vertex Color Array data collection
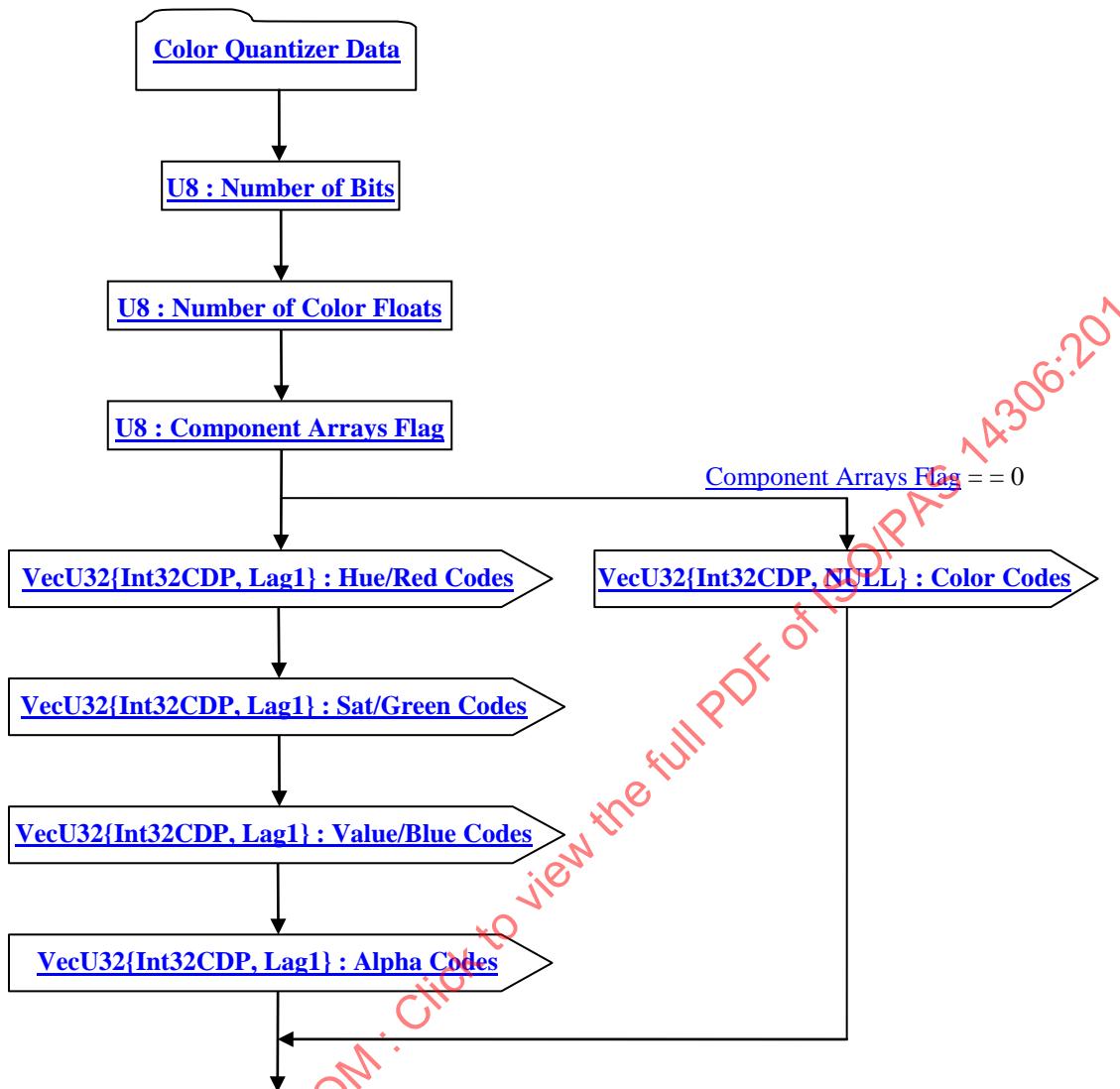
Complete description for Color Quantizer Data can be found in 7.1.6 Color Quantizer Data.

U8 : Number of Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision) for each of the 3 or 4 color components. This value must satisfy the following condition: "0 <= Number Of Bits <= 8".

U8 : Number of Color Floats

Number of Color Floats specifies the number of floating point values used to represent the color components. Valid values include the following:

= 1	– All components packed into a single 32 bit value…8 bits per component.
= 3	– Each RGB/HSV color component representing in its own floating point value. Alpha always assumed to be 1.
= 4	– Each RGBA/HSVA color component representing in its own floating point value.

U8 : Component Arrays Flag

Component Arrays Flag is a flag indicating whether color components are encoded as a single integer or if the encoding is broken up (separated) into an array for each color component.

= 0	– Encoded as single integer…thus one compressed collection of codes
= 1	– Encoding is broken up (separated) into an array of codes for each color component…thus a compressed collection of codes for each color component.

VecU32{Int32CDP, Lag1} : Hue/Red Codes

Hue/Red Codes is a vector of quantizer "codes" for all the Hue/Red color components of a set of vertex colors. Whether HSV or RGB color model is being used (i.e. Hue or Red) is indicated by a flag stored in the Color Quantizer Data. Hue/Red Codes is only present when data field Component Arrays Flag == 1. Hue/Red Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : Sat/Green Codes

Sat/Green Codes is a vector of quantizer "codes" for all the Saturation/Green color components of a set of vertex colors. Whether HSV or RGB color model is being used (i.e. Saturation or Green) is indicated by a flag stored in the Color Quantizer Data. Sat/Green Codes is only present when data field Component Arrays Flag == 1. Sat/Green Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : Value/Blue Codes

Value/Blue Codes is a vector of quantizer "codes" for all the Value/Blue color components of a set of vertex colors. Whether HSV or RGB color model is being used (i.e. Value or Blue) is indicated by a flag stored in the Color Quantizer Data. Value/Blue Codes is only present when data field Component Arrays Flag == 1. Value/Blue Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, Lag1} : Alpha Codes

Alpha Codes is a vector of quantizer "codes" for all the Alpha color components of a set of vertex colors. Alpha Codes is only present when data field Component Arrays Flag == 1. Alpha Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP, NULL} : Color Codes

Color Codes is a vector of quantizer "codes" for a set of vertex colors. Color Codes is only present when data field Component Arrays Flag == 0. Color Codes uses the Int32 version of the CODEC to compress and encode data.

7.1.4 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 182: Point Quantizer Data data collection
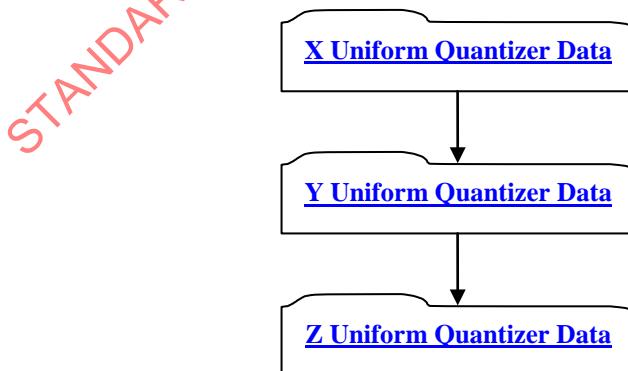
Complete description for X Uniform Quantizer Data, Y Uniform Quantizer Data and Z Uniform Quantizer Data can be found in 7.1.7 Uniform Quantizer Data.

7.1.5 Texture Quantizer Data

A Texture Quantizer Data collection is made up of two Uniform Quantizer Data collections; there is a separate Uniform Quantizer Data collection for the U, and V values of texture coordinates.

Figure 183: Texture Quantizer Data data collection

U Uniform Quantizer Data

V Uniform Quantizer Data
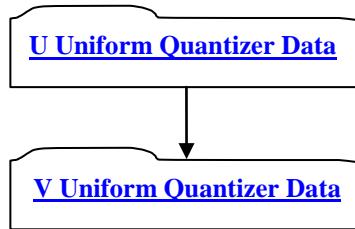
Complete description for U Uniform Quantizer Data, and V Uniform Quantizer Data can be found in 7.1.7 Uniform Quantizer Data.

7.1.6 Color Quantizer Data

A Color Quantizer Data collection contains the quantizer information for each of the color components. The Color Quantizer utilizes a separate Uniform Quantizer Data collection for each of the 4 color components, but if the HSV color 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 color component are constant, only the Number of Bits of precision for each color component's Uniform Quantizer is stored. The Uniform Quantizer range values for the HSV color components should always be assumed to be the following:

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

Figure 184: Color Quantizer Data data collection
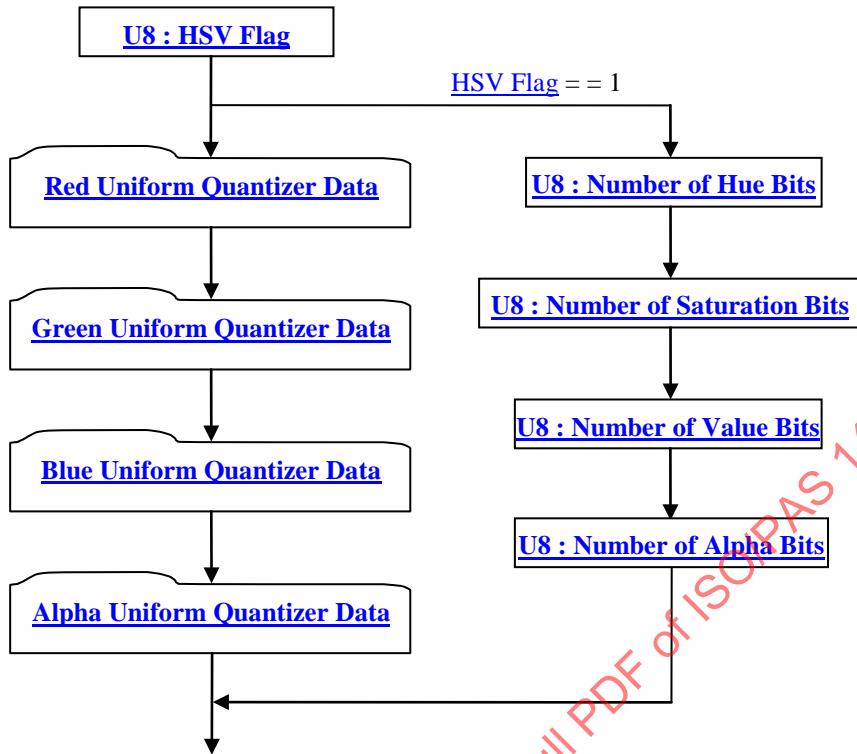
Complete descriptions for Red Uniform Quantizer Data, Green Uniform Quantizer Data, Blue Uniform Quantizer Data, and Alpha Uniform Quantizer Data can be found in 7.1.7 Uniform Quantizer Data. These four Uniform Quantizer Data collections are only present when data field HSV Flag = = 0.

U8 : HSV Flag

HSV Flag is a flag indicating whether color component data is stored in HSV color model form.

= 0	– Color component data stored in RGB color model form.
= 1	– Color component data stored in HSV color 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 color. 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 color. 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 color. 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 color. Number of Alpha Bits data is only present when data field HSV Flag = = 1.

7.1.7 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 185: Uniform Quantizer Data data collection
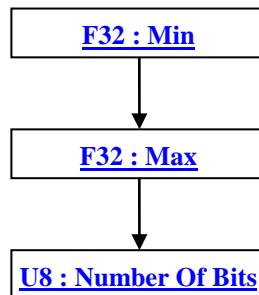
F32 : Min

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 must satisfy the following condition: "0 <= Number Of Bits <= 32".

7.1.8 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:

1. Case-1 for trivial knot vector
   a. Number of knots is an even number
   b. Knot vector has a [0:1] knot range
   c. There are no interior knots (i.e. NumberKnots = = 2 * (NurbsEntityDegree + 1)

2. Case-2 for trivial knot vector
   a. Number of knots is an even number.
   b. Knot vector has a [0:1] knot range
   c. NurbsEntityDegree < 3
   d. Difference between successive non-repeating knots (i.e. KnotDelta) is:
      i. 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."

Figure 186: Compressed Entity List for Non-Trivial Knot Vector data collection
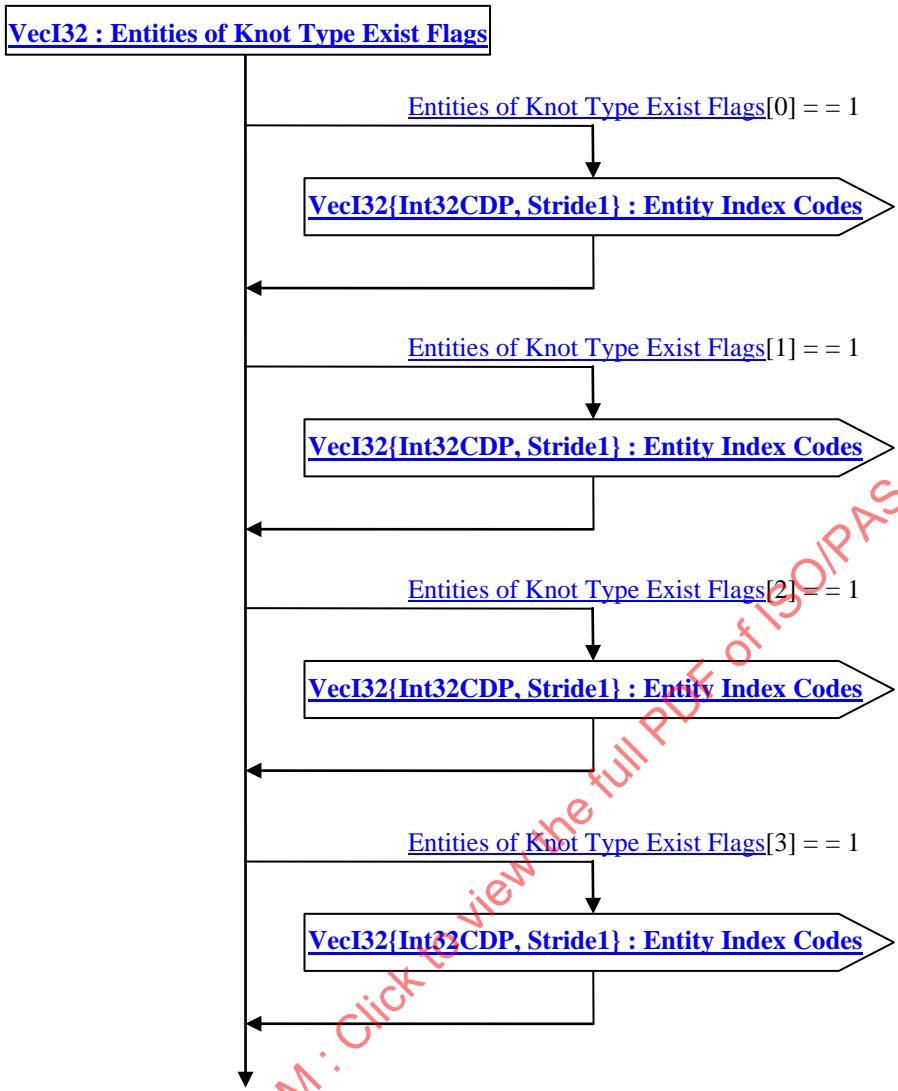
VecI32 : Entities of Knot Type Exist Flags

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 the knot count is even or odd and whether 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:

[0]	– Flag indicating whether Entity IDs data collection exists for "Even Count [0:1] Range" knot type. = 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. = 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. = 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.
	= 0 – No Entity IDs data collection exists.
	= 1 – Entity IDs data collection exists.

VecI32{Int32CDP, Stride1} : 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.

7.1.9 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.

Figure 187: Compressed Control Point Weights Data data collection
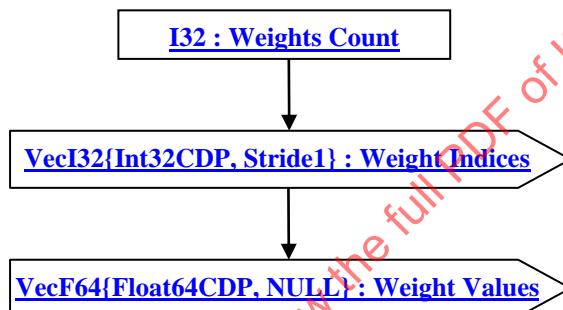
I32 : Weights Count

Weights Count specifies the total number of Weights. This count can differ from the Control Point count (see 6.2.3.1.4.1.3 NURBS Surface Control Point Counts) because if the Control Point Dimensionality is non-rational (see data field NURBS Surface Control Point Dimensionality in 6.2.3.1.4.1 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 "1" for Control Points that don't have a Weight value stored).

VecI32{Int32CDP, Stride1} : 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{Float64CDP, NULL} : Weight Values

Weight Values is a vector of weight values for the conditional set of weights. Weight Values uses the Float64 version of the CODEC to compress and encode data.

7.1.10 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 6.2.3.1 JT B-Rep Element and 6.2.5.1 Wireframe Rep Element respectively.

Figure 188: Compressed Curve Data data collection

```
        ┌─────────────────────────┐
        │   Non-Trivial Knot Vector │
        │   NURBS Curve Indices     │
        └─────────────────────────┘
                    │
                    ▼
    ┌──────────────────────────────────────┐
    │ VecI32{Int32CDP, Lag1} : Curve Base Types │
    └──────────────────────────────────────┘
                    │
                    ▼
    ┌──────────────────────────────────────┐
    │ VecI32{Int32CDP, Lag1} : NURBS Curve Degrees │
    └──────────────────────────────────────┘
                    │
                    ▼
    ┌──────────────────────────────────────────────┐
    │ VecI32{Int32CDP, Lag1} : NURBS Curve Control Point Counts │
    └──────────────────────────────────────────────┘
                    │
                    ▼
    ┌──────────────────────────────────────────────────────┐
    │ VecI32{Int32CDP, Lag1} : NURBS Curve Control Point Dimensionality │
    └──────────────────────────────────────────────────────┘
                    │
                    ▼
    ┌──────────────────────────────────────────────┐
    │ VecI32{Int32CDP, Lag1} : NURBS Curve Reserved Fields │
    └──────────────────────────────────────────────┘
                    │
                    ▼
        ┌─────────────────────────┐
        │   NURBS Curve Control     │
        │   Point Weights           │
        └─────────────────────────┘
                    │
                    ▼
        ┌─────────────────────────┐
        │   NURBS Curve Control     │
        │   Points                  │
        └─────────────────────────┘
                    │
                    ▼
    ┌──────────────────────────────────────────────┐
    │ VecF64{Float64CDP, NULL} : NURBS Curve Knot Vectors │
    └──────────────────────────────────────────────┘
```

VecI32{Int32CDP, Lag1} : 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:

= 1	– Curve is a NURBS curve

Curve Base Types uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : 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, Lag1} : 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, Lag1} : 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:

| = 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:

| = 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, Lag1} : NURBS Curve Reserved Fields

NURBS Curve Reserved Fields is a vector of data reserved for future expansion of the JT format. Each NURBS Curve in a list of Curves has one reserved data field entry in this NURBS Curve Reserved Fields vector. NURBS Curve Reserved Fields uses the Int32 version of the CODEC to compress and encode data

VecF64{Float64CDP, 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 cumulated 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 7.1.10.1 Non-Trivial Knot Vector NURBS Curve Indices. NURBS Curve Knot Vectors uses the Float64 version of the CODEC to compress and encode data.

7.1.10.1 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 7.1.8 Compressed Entity List for Non-Trivial Knot Vector.

This Curve index data is stored in a compressed format.

Figure 189: Non-Trivial Knot Vector NURBS Curve Indices data collection

Compressed Entity List
for Non-Trivial Knot
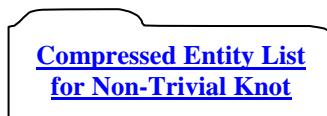
Complete description for Compressed Entity List for Non-Trivial Knot Vector can be found in 7.1.8 Compressed Entity List for Non-Trivial Knot Vector.

7.1.10.2 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 190: NURBS Curve Control Point Weights data collection
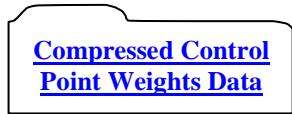
Compressed Control
Point Weights Data

Complete description for Compressed Control Point Weights Data can be found in 7.1.9 Compressed Control Point Weights Data.

7.1.10.3 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 191: NURBS Curve Control Points data collection

VecF64{Float64CDP, NULL} : Control Points

VecF64{Float64CDP, 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 cumulated 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 Float64 version of the CODEC to compress and encode data in a "lossless" manner.

7.1.11 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. 6.2.3.1.6 B-Rep CAD Tag Data, 6.2.6.2.7 PMI CAD Tag Data).

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.

Figure 192: Compressed CAD Tag Data data collection
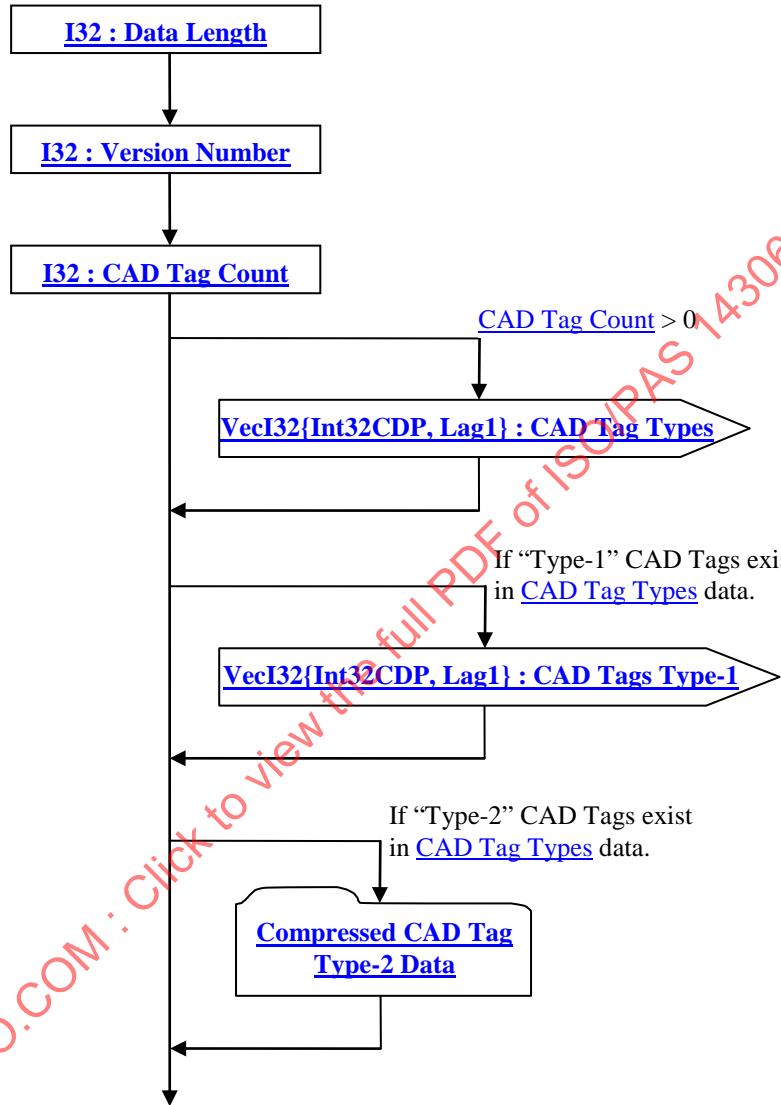
I32 : Data Length

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 version identifier for the Compressed CAD Tag Data. Version number "1" is currently the only valid value.

I32 : CAD Tag Count

CAD Tag Count specifies the number of CAD Tags

VecI32{Int32CDP, Lag1} : 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:

= 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, Lag1} : 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. Thus a loader/reader of JT file must first uncompress/decode and evaluate the previously read CAD Tag Types to determine if there are any Type-1 CAD Tags and if so, then the CAD Tags Type-1 data vector is present.

7.1.11.1 Compressed CAD Tag Type-2 Data

Compressed CAD Tag Type-2 Data collection contains the Type-2 (i.e. 64 Bit integer Type) CAD Tag data for a list of CAD Tags.

The Compressed CAD Tag Type-2 Data collection is only present if there are Type-2 CAD Tags in the CAD Tag Types vector. Thus a loader/reader of JT file must first uncompress/decode and evaluate the previously read CAD Tag Types vector to determine if there are any Type-2 CAD Tags and if so, then the Compressed CAD Tag Type-2 Data collection is present.

Figure 193: Compressed CAD Tag Type-2 Data data collection

VecI32{Int32CDP, Lag1} : First I32 of Type-2 CAD Tags

VecI32{Int32CDP, Lag1} : Second I32 of Type-2 CAD Tags
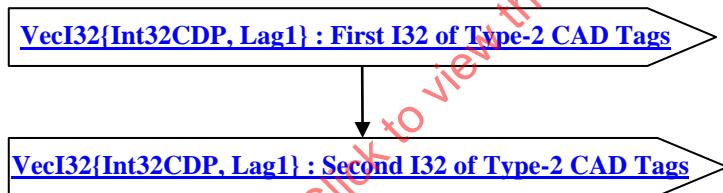
VecI32{Int32CDP, Lag1} : First I32 of Type-2 CAD Tags

First I32 of Type-2 CAD Tags is a vector of the first 32 bits of each Type-2 CAD Tag in the list of CAD Tags. First I32 Of Type-2 CAD Tags uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : Second I32 of Type-2 CAD Tags

Second I32 of Type-2 CAD Tags is a vector of the second 32 bits of each Type-2 CAD Tag in the list of CAD Tags. Second I32 Of Type-2 CAD Tags uses the Int32 version of the CODEC to compress and encode data.

7.2 Encoding Algorithms

The following sections give a brief technical overview/descriptions of the various encoding algorithms used in the JT format. Additional information on each of the algorithms can be found within references listed in 2 References and Additional Information section of this document. Also, a sample implementation of the decoding portion of each algorithm can be found in Annex C:Decoding Algorithms – An Implementation.

7.2.1 Uniform Data Quantization

Uniform Data Quantization is a lossy encoding algorithm in which a continuous set of input values (floating point data) is approximated with integral multipliers (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" must 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.

7.2.2 Bitlength CODEC

This is a very simple compression algorithm that runs an adaptive-width bit field encoding for each value. As each input value is seen, 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.

$$0$$

Indicates no bit field width change ⌐

Example 2: Prefix code to increment field width four times.

111110

Indicates bit field width change
Indicates increment width by step_size
Indicates increment width by step_size
Indicates increment width by step_size
Indicates increment width by step_size
Termination bit
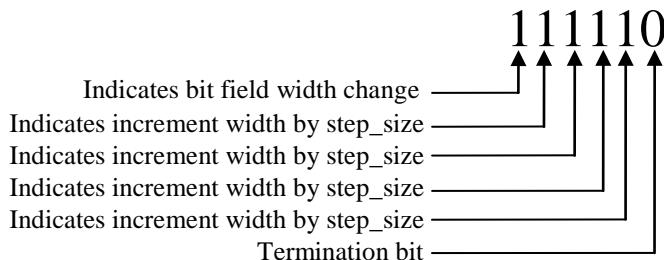
Example 3: Prefix code to decrement field width two times.

1001

Indicates bit field width change
Indicates decrement width by step_size
Indicates decrement width by step_size
Termination bit
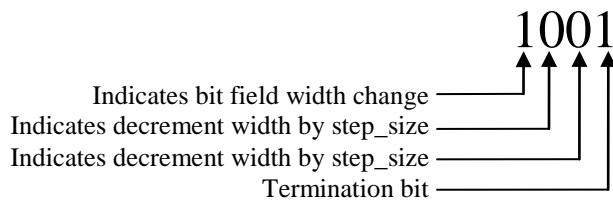
A pseudo-code sample implementation of bit length decoding is available in Annex C:Decoding Algorithms – An Implementation.

7.2.3 Huffman CODEC

The Huffman compression algorithm is named after its inventor, David Huffman, and was developed in 1948 while Mr. Huffman was a Ph.D. student at the Massachusetts Institute of Technology (MIT); the same year as Claude Shannon of Bell Laboratories published his seminal paper "A mathematical theory of communication" that launched the new field of Information Theory. In that same class with David 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.

Huffman compression is a lossless compression algorithm that uses a variable length codeword to encode a source symbol; that is individual symbols (e.g. characters in a text file) are replaced by bit sequence codes that have a distinct length. The codes are assigned to the symbols based on the probabilities of the symbol occurring, such that symbols occurring frequently in a file are given a short sequence while others that are seldom used get a longer bit sequence.

Huffman coding is dependent upon Huffman's algorithm, which takes a list of weights, and builds an extended and complete binary tree of minimum weighted path length (as shown in Figure 194: Huffman Tree). For the JT format usage of Huffman Coding, the list of weights consists of the frequency of symbol occurrence. Although the assignment of a binary value to the edges which leads you to the left or right child of a tree node may be arbitrary, it must be consistent in the tree construction, and typically "0" is associated with an edge leading to a left child and "1" is associated with an edge leading to a right child.

Using this Huffman Tree, variable length codewords are defined for each symbol by concatenating the value associated with the edges as you follow the path from the root of the tree to the leaf associated with a symbol's frequency. An important characteristic of this encoding is that the inverse mapping (i.e. decoding) must be unambiguous, such that there is only one possible way to decompose a string of codewords into individual symbols. It can be proven that "unambiguous decoding" is indeed a property of a Huffman Tree due to the fact that by definition, a Huffman Tree is a complete binary tree.

7.2.3.1 Example

Following is an example to demonstrate in practice the basic principles of Huffman coding.

Suppose you want to compress, using Huffman coding, the following sequence/array of integer data:

$$\{1, 3, 4, 1, 2, 1\}$$

There are 6 integer numbers, so in non compressed form (i.e. fixed size integer coding) this sequence requires 24 bytes or 192 bits.

The frequency count of the numbers in the sequence is as follows:

Number	Frequency
1	3
2	1
3	1
4	1
Total	6

Letting, as in the JT format usage of Huffman coding, the frequency counts be the weights. If we pair each symbol (integer) with its weight, and pass this list of weights to the Huffman algorithm we will get a Huffman Tree (with included edge labels) that looks something like the following:

Figure 194: Huffman Tree
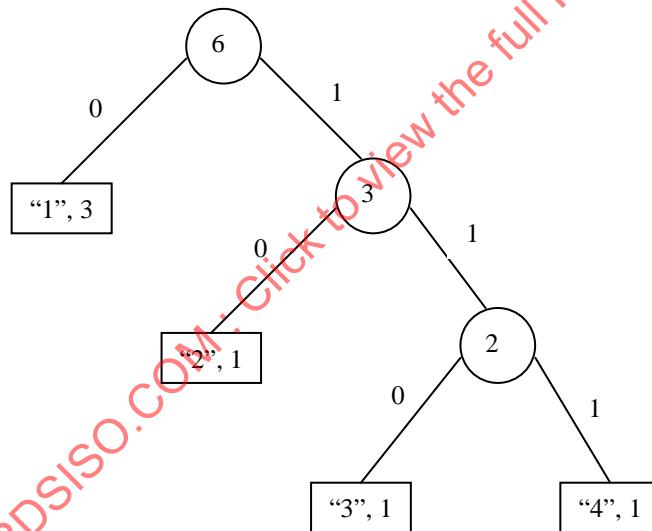
From this tree we get the following Huffman codes for each of the integer numbers:

Number	Frequency	Huffman code	Weighted Code Length (Frequency * #code_bits)
1	3	0	3
2	1	10	2
3	1	110	3
4	1	111	3
Total	6	-	11

Using the Huffman codes the array/sequence of integers can be compressed into the following bit sequence:

01101110100

So the number of bits required to represent the array/sequence of integers in Huffman codeword form is 11 bits (i.e. total of "Weighted Code length), versus 192 bits in standard fixed-size integer encoding.

A pseudo-code sample implementation of Huffman decoding is available in <u>Annex C:Decoding Algorithms – An Implementation</u>.

7.2.4 Arithmetic CODEC

Arithmetic encoding is a lossless compression 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/complexity 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 must 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:

1. 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.
2. Sequentially for each symbol of the input stream, perform two steps
   a. Subdivide the current interval into subintervals based on the input stream symbol probability values as described above.
   b. Select the subinterval corresponding to the current input stream symbol being sequentially processed and make it the new "current interval".
3. 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
```

7.2.4.1 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:

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:

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:

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 must 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 must 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}:

	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
Encode "12" [0.6, 0.8)	79999	60000	20000	.25
Encode "12" [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 "20" [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 "19" [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 must 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 most significant digit of High is "0" and the 2nd most significant digit of low is "9")  )
{
        underflow = TRUE
}
```

When/If it is identified that "underflow" is occurring, the encoding algorithm must 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 must 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:Decoding Algorithms – An Implementation.

7.2.5 Deering Normal CODEC

M. Deering first published his work on geometry compression in 1995 [5] and later helped present a course on the subject at SIGGRAPH'99 [6]. Although Deering's approach to geometric compression involves compression of vertices, colors 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 195). 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 195). 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.

Figure 195: Sphere divided into eight octants and octant divided into six sextants with each sextant assigned an identifying three bit code.
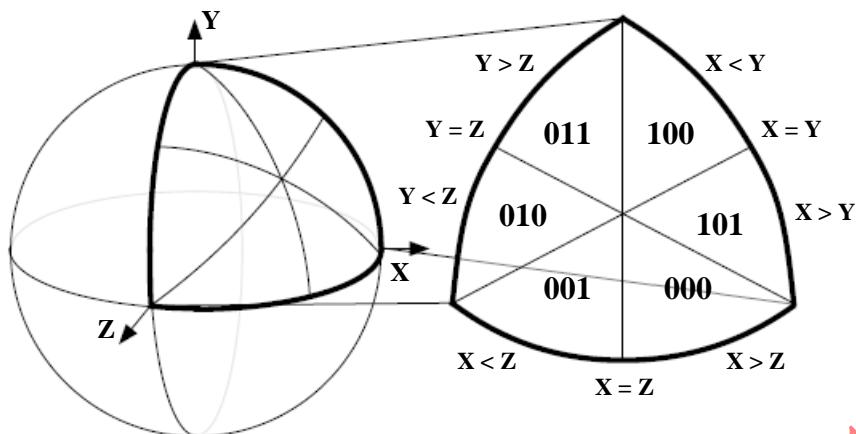
Note that the sextant three bit code assignments used by the JT format (as seen in Figure 195) are slightly modified from the original assignments as specified by Deering.

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 "n" is in the range of 0 to 6) and the relationship between these n-bit integers and angles θ and φ for a given "n" is:

$$\theta(\theta'_n) = \text{asin tan}(\varphi_{max} * (n - \theta'_n) / 2^n)$$

$$\varphi(\varphi'_n) = \varphi_{max} * \varphi'_n / 2^n$$

Thus to encode (i.e. quantize) a given normal **N** into θ'_n and φ'_n:

- **N** must be first represented (see Figure 195) in the positive octant and appropriate sextant within that octant, resulting in **N'**.
- Then **N'** must 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 195.
- Next two n-bit fields specify θ'_n and φ'_n values respectively.

7.3 ZLIB Compression

ZLIB compression is a lossless data compression algorithm and is essentially the same as that in gzip and Zip. Zlib's compression method, called deflation, creates compressed data as a sequence of blocks. The JT format uses *Version 1.1.2* of the ZLIB compression library.

8   Usage Guide

The proceeding sections of this document specify the mandatory clauses for creating a reference compliant Version 8.1 JT file. This "Usage Guide" 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.

8.1   Late-Loading Data

From its inception, the JT format was designed to scale from representing data necessary for lightweight web-based viewing, to representing data necessary for full product digital mockup and 3D product definition. This ability of the JT format to represent such a robust 3D product definition allows a single JT format based 3D digital asset to be leveraged across the extended enterprise by many dissimilar applications with varying data needs/requirements.

With this sharing of the single JT format based 3D digital asset by many dissimilar applications, comes the need to be "performance sensitive" (both in runtime memory footprint and actual data load time) to exactly what, how much, and when certain JT format data must be loaded. To that end the JT format was designed/structured to support not requiring all segments of data to be sequentially loaded/read in one pass. This concept of delaying the loading of segments of data until actually needed is referred to within this JT Format Reference document as "late-loading data". The JT format has many structures in support of this concept of late-loading data and it is recommended as a best practice that writers/loaders of JT data leverage these constructs accordingly. Examples of these JT format constructs in support of (but not necessarily late-loading data include the following (note that "in support of" does not necessarily mean that the construct (e.g. TOC Segment) is only used for purposes of late loading data):

- TOC Segment
- Partition Node Element
- Meta Data Node Element
- Late Loaded Property Atom Element

8.2   Bit Fields

In the 6 File Format section of this reference many bit field data descriptions (e.g. 6.2.1.1.1.1.1 Base Node Data "Node Flags" field) contain the words "*All undocumented bits are reserved*." These words should be interpreted to mean that these undocumented bits should be set to "0" when writing the bit field data to a JT file.

8.3   Reserved Field

In the 6 File Format section of this reference some data fields may be named/documented "Reserved Field" (e.g. 6.2.1.1.1.7.1 LOD Node Data "Reserved Field" field). A "Reserved Field" exists for potential future expansion of the Format and best practices 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 Reserved 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 "Reserved Fields" should be written with the same value that was read from the originating JT file.

8.4   Metadata Conventions

Although there are really no restrictions/limits/requirements on what metadata (i.e. properties) can/must 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 6.2.1.2 Property Atom Elements section of this document for complete description of the file Elements used to attach this property information to nodes.

8.4.1 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.

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_CENTER_OF_GRAVITY	Center 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 prorgam ther Part originated from	MbString	MbString	<string>	Optional

Table 8: CAD Property Conventions

8.4.1.1 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.

8.4.1.2 Optional Properties

Optional properties can be provided, but if the property is a units based value, then the value must 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 must be as follows:

Optional Property	Units
CAD_SURFACE_AREA	(JT_PROP_MEASUREMENT_UNITS)^2
CAD_VOLUME	(JT_PROP_MEASUREMENT_UNITS)^3
CAD_DENSITY	CAD_MASS_UNITS/(JT_PROP_MEASUREMENT_UNITS)^3
CAD_MASS	CAD_MASS_UNITS
CAD_CENTER_OF_GRAVITY	JT_PROP_MEASUREMENT_UNITS
CAD_PROP_MATERIAL_THICKNESS	JT_PROP_MEASUREMENT_UNITS

Table 9: CAD Optional Property 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 8.4.1CAD Properties) implies that all solids within a single JT part are of a uniform density, which may not be true in all cases.
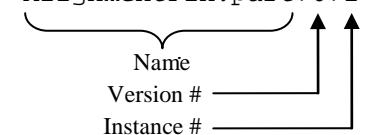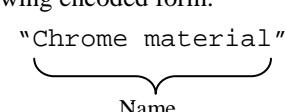
8.4.2 Tessellation Properties

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

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: "30.000000 40.000000"	MbString	space separated F32 values	Numeric

8.4.3 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.

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 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"	MbString	*\<string\>*	
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	May be attached to Tri-Strip Set Shape Node Element to indicate that the Set's tri-strip primitives are sorted such that strips of length 1 (i.e. triangles) come first and then strips of length 2 (i.e. quads) next and then all other strips of length greater than 2 follow in no particular order.	MbString	*\<string\>*	"TriStripsSorted"
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: "AlignmentPin.part;0;1:" Name Version # Instance # For attribute names this string has the following encoded form: "Chrome material" Name	MbString	*\<string\>*	

8.5 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 6.2.1.1.2 Attribute Elements), there are some general rules which apply:

1. Attributes at lower level in the LSG take precedence and replace or accumulate with attributes set at higher levels
2. Nodes without associated attributes inherit those of their parents.
3. Attributes inherit only from their parents, thus a node's attributes do not affect that node's siblings.
4. The root of a partition inherits the attributes in effect at the referring partition node.
5. Attributes can be declared "final", which terminates accumulation of that attribute type at that attribute and propagates the accumulated values there to all descendants of the associated node. Descendants can explicitly do a one-shot override of "final" using the attribute "force" flag, 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." 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!

8.6 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 196: JT Format Convention for Modeling each Part in LSG
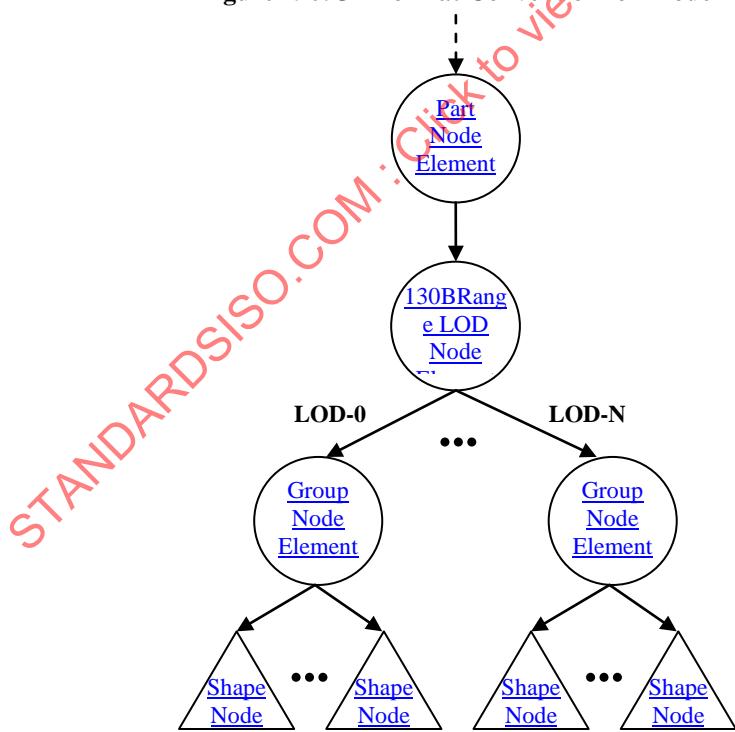
8.7   Range LOD Node Alternative Rep Selection

Best practices suggest that LSG traversers apply the following strategy, at Range LOD Nodes (see 6.2.1.1.1.8 Range LOD Node Element), when making alternative representation selection decisions based on Range Limits: The first alternate representation is valid when the distance between the center 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

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 10: Object Type Identifiers lists the assigned identifier for each Object Type that can exist in a Version 8.1 JT file.

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, 0x59, 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
0x4cc7a521, 0x728, 0x11d3, 0x9d, 0x8b, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2	Wire Harness Set Shape Node Element
0x10dd1001, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Attribute Element
0x10dd1014, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Draw Style Attribute Element
0xad8dccc2, 0x7a80, 0x456d, 0xb0, 0xd5, 0xdd, 0x3a, 0xb, 0x8d, 0x21, 0xe7	Fragment Shader 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
0xaa1b831d, 0x6e47, 0x4fee, 0xa8, 0x65, 0xcd, 0x7e, 0x1f, 0x2f, 0x39, 0xdb	Shader Effects Attribute Element
0x10dd1073, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Texture Image Attribute Element
0x2798bcad, 0xe409, 0x47ad, 0xbd, 0x46, 0xb, 0x37, 0x1f, 0xd7, 0x5d, 0x61	Vertex Shader Attribute 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

GUID	Object Type
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 PMI Segment (Segment Type = 3)	
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
0x10dd109f, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Polygon 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
0x10dd10b0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Vertex Shape LOD Element
0x4cc7a523, 0x728, 0x11d3, 0x9d, 0x8b, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2	Wire Harness Set Shape 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

Table 10: Object Type Identifiers

Annex B: **Semantic Value Class Shader Parameter Values**

6.2.1.1.2.12 Vertex Shader Attribute Element and 6.2.1.1.2.13 Fragment Shader Attribute Element contain shader parameters. These shader parameters can be of a "Semantic" Value Class which indicates that the shader parameter is implicitly tied/bound to a piece of either OpenGL or JT graphics system state. Table 11 below documents all the possible "Semantic" Value Class shader parameter Values (i.e. the graphics system state the parameter is bound to).

Table 11: Semantic Value Class Shader Parameter Values

Value	Description of Semantically Bound Graphics State	Notes
= 0	– Unknown	
Related to Current OpenGL State		
= 30	– View Transform Matrix	Cg only
= 31	– Combined Model-View Transform Matrix	Cg only
= 32	– Projection Transform Matrix	Cg only
= 33	– Texture Transform Matrix	Cg only
= 34	– Combined Model-View-Projection Transform Matrix	Cg only
= 35	– View Matrix Transposed	Cg only
= 36	– Combined Model-View Transform Matrix Transposed	Cg only
= 37	– Projection Transform Matrix Transposed	Cg only
= 38	– Texture Transform Matrix Transposed	Cg only
= 39	– Combined Model-View-Projection Transform Matrix Transposed	Cg only
= 40	– View Transform Matrix Inverse	Cg only
= 41	– Combined Model-View Transform Matrix Inverse	Cg only
= 42	– Projection Transform Matrix Inverse	Cg only
= 43	– Texture Transform Matrix Inverse	Cg only
= 44	– Combined Model-View-Projection Transform Matrix Inverse	Cg only
= 45	– View Transform Matrix Inverse Transposed	Cg only
= 46	– Combined Model-View Transform Matrix Inverse Transposed	Cg only
= 47	– Projection Transform Matrix Inverse Transposed	Cg only
= 48	– Texture Transform Matrix Inverse Transposed	Cg only
= 49	– Combined Model-View-Projection Transform Matrix Inverse Transposed	Cg only
Related to Current JT State		
= 70	– Current Model Transform	
= 71	– Current Model Transform Transposed	
= 72	– Current Model Transform Inverse	
= 73	– Current Model Transform Inverse Transposed	
= 75	– Current Material Emissive Color	
= 76	– Current Material Diffuse Color	
= 77	– Current Material Specular Color	
= 78	– Current Material Ambient Color	
= 79	– Current Material Shininess	
= 80	– Current Fog Color	
= 81	– Separate Specular Color Flag	
= 82	– Global Ambient Light Color	
= 99	– Number of VPCS Lights	
= 100	– VPCS Light-0 Diffuse Color	
= 101	– VPCS Light-0 Specular Color	
= 102	– VPCS Light-0 Ambient Color	
= 103	– VPCS Light-0 Attenuation	

Value	Description of Semantically Bound Graphics State	Notes
= 104	– VPCS Light-0 Position	
= 105	– VPCS Light-0 Direction	
= 106	– VPCS Light-0 Spot Direction	
= 107	– VPCS Light-0 Spot Cone Angle	
= 108	– VPCS Light-0 Cosine of Spot Cone Angle	
= 109	– VPCS Light-0 Spot Exponent	
= 110	– VPCS Light-0 Shadow Opacity	
= 120 → 130	– Same as values 100 → 110 except for VPCS Light-1	
= 140 → 150	– Same as values 100 → 110 except for VPCS Light-2	
= 160 → 170	– Same as values 100 → 110 except for VPCS Light-3	
= 180 → 190	– Same as values 100 → 110 except for VPCS Light-4	
= 200 → 210	– Same as values 100 → 110 except for VPCS Light-5	
= 220 → 230	– Same as values 100 → 110 except for VPCS Light-6	
= 240 → 250	– Same as values 100 → 110 except for VPCS Light-7	
= 499	– Number of MCS Lights	
= 500 → 510	– Same as values 100 → 110 except for MCS Light-0	
= 520 → 530	– Same as values 100 → 110 except for MCS Light-1	
= 540 → 550	– Same as values 100 → 110 except for MCS Light-2	
= 560 → 570	– Same as values 100 → 110 except for MCS Light-3	
= 580 → 590	– Same as values 100 → 110 except for MCS Light-4	
= 600 → 610	– Same as values 100 → 110 except for MCS Light-5	
= 620 → 630	– Same as values 100 → 110 except for MCS Light-6	
= 640 → 650	– Same as values 100 → 110 except for MCS Light-7	
= 899	– Number of WCS Lights	
= 900 → 910	– Same as values 100 → 110 except for WCS Light-0	
= 920 → 930	– Same as values 100 → 110 except for WCS Light-1	
= 940 → 950	– Same as values 100 → 110 except for WCS Light-2	
= 960 → 970	– Same as values 100 → 110 except for WCS Light-3	
= 980 → 990	– Same as values 100 → 110 except for WCS Light-4	
= 1000 → 1010	– Same as values 100 → 110 except for WCS Light-5	
= 1020 → 1030	– Same as values 100 → 110 except for WCS Light-6	
= 1040 → 1050	– Same as values 100 → 110 except for WCS Light-7	
= 1500	– Current Texture Object-0	Cg only
= 1501	– Current Texture Object-1	Cg only
= 1502	– Current Texture Object-2	Cg only
= 1503	– Current Texture Object-3	Cg only
= 1504	– Current Texture Object-4	Cg only
= 1505	– Current Texture Object-5	Cg only
= 1506	– Current Texture Object-6	Cg only
= 1507	– Current Texture Object-7	Cg only
= 1600	– Current Texture Unit-0	GLSL only
= 1601	– Current Texture Unit-1	GLSL only
= 1602	– Current Texture Unit-2	GLSL only
= 1603	– Current Texture Unit-3	GLSL only
= 1604	– Current Texture Unit-4	GLSL only
= 1605	– Current Texture Unit-5	GLSL only
= 1606	– Current Texture Unit-6	GLSL only
= 1607	– Current Texture Unit-7	GLSL only

Value	Description of Semantically Bound Graphics State	Notes
= 1700	– Texture Channel-0 VCS Texture Coordinate Generation S-Plane	
= 1701	– Texture Channel-0 VCS Texture Coordinate Generation T-Plane	
= 1702	– Texture Channel-0 VCS Texture Coordinate Generation R-Plane	
= 1703	– Texture Channel-0 VCS Texture Coordinate Generation Q-Plane	
= 1710 → 1713	– Same as 1700 → 1703 except for Chanel-1 VCS	
= 1720 → 1723	– Same as 1700 → 1703 except for Chanel-2 VCS	
= 1730 → 1733	– Same as 1700 → 1703 except for Chanel-3 VCS	
= 1740 → 1743	– Same as 1700 → 1703 except for Chanel-4 VCS	
= 1750 → 1753	– Same as 1700 → 1703 except for Chanel-5 VCS	
= 1760 → 1763	– Same as 1700 → 1703 except for Chanel-6 VCS	
= 1770 → 1773	– Same as 1700 → 1703 except for Chanel-7 VCS	
= 2000 → 2003	– Same as 1700 → 1703 except for Chanel-0 MCS	
= 2010 → 2013	– Same as 1700 → 1703 except for Chanel-1 MCS	
= 2020 → 2023	– Same as 1700 → 1703 except for Chanel-2 MCS	
= 2030 → 2033	– Same as 1700 → 1703 except for Chanel-3 MCS	
= 2040 → 2043	– Same as 1700 → 1703 except for Chanel-4 MCS	
= 2050 → 2053	– Same as 1700 → 1703 except for Chanel-5 MCS	
= 2060 → 2063	– Same as 1700 → 1703 except for Chanel-6 MCS	
= 2070 → 2073	– Same as 1700 → 1703 except for Chanel-7 MCS	
= 3000	– Texture Channel-0 Matrix	
= 3001	– Texture Channel-1 Matrix	
= 3002	– Texture Channel-2 Matrix	
= 3003	– Texture Channel-3 Matrix	
= 3004	– Texture Channel-4 Matrix	
= 3005	– Texture Channel-5 Matrix	
= 3006	– Texture Channel-6 Matrix	
= 3007	– Texture Channel-7 Matrix	
= 3100	– Texture Channel-0 Map Resolution	
= 3101	– Texture Channel-1 Map Resolution	
= 3102	– Texture Channel-2 Map Resolution	
= 3103	– Texture Channel-3 Map Resolution	
= 3104	– Texture Channel-4 Map Resolution	
= 3105	– Texture Channel-5 Map Resolution	
= 3106	– Texture Channel-6 Map Resolution	
= 3107	– Texture Channel-7 Map Resolution	
= 3200	– Texture Channel-0 Map Resolution Inverses (i.e. 1.0 /"Map Resolution")	
= 3201	– Texture Channel-1 Map Resolution Inverses	
= 3202	– Texture Channel-2 Map Resolution Inverses	
= 3203	– Texture Channel-3 Map Resolution Inverses	
= 3204	– Texture Channel-4 Map Resolution Inverses	
= 3205	– Texture Channel-5 Map Resolution Inverses	
= 3206	– Texture Channel-6 Map Resolution Inverses	
= 3207	– Texture Channel-7 Map Resolution Inverses	
= 3300	– Texture Channel-0 Blend Color	
= 3301	– Texture Channel-1 Blend Color	
= 3302	– Texture Channel-2 Blend Color	

Value	Description of Semantically Bound Graphics State	Notes
= 3303	– Texture Channel-3 Blend Color	
= 3304	– Texture Channel-4 Blend Color	
= 3305	– Texture Channel-5 Blend Color	
= 3306	– Texture Channel-6 Blend Color	
= 3307	– Texture Channel-7 Blend Color	

Annex C: **Decoding Algorithms – An Implementation**

This Appendix provides a sample C++ implementation for the decoding portion of the various compression CODECs (as detailed in 7.2 Encoding Algorithms) used in the JT format. This sample code is not intended to be fully functional decoder class implementations, but is instead intended to demonstrate the fundamentals of implementing the decoding portion of the CODEC algorithms used in the JT format.

C.1 Common classes
The following sub-sections define some general classes used by all the decoding algorithms.

C.1.1 CntxEntry class

```
//
// Type used to build probability context tables.
// Used by ProbabilityContext class.
//
class CntxEntry
{
public:

   Int32 iSym;          // Symbol
   Int32 cCount;        // Number of occurrences
   Int32 cCumCount;     // Cumulative number of occurrences
   Int32 iNextCntx = 0; // Next context if this symbol seen
};
```

C.1.2 ProbabilityContext class

```
//
// Type used to build probability context tables.
// Used by CodecDriver class.
//
class ProbabilityContext
{
public:

   // Returns total cumulative count for all context entries
   Int32 totalCount();

   // Returns number of context entries
   Int32 numEntries();

   // Returns context entry of index iEntry
   Bool getEntry(Int32 iEntry, CntxEntry& rpEntry);

   // Looks up the next context field given by the context entry
   // with input symbol 'iSymbol'
   Bool lookupNextContext(Int32 iSymbol, Int32& iNextContext);

   // Looks up the index of the context entry for the given
   // input symbol 'iSymbol'
   Bool lookupSymbol(Int32 iSymbol, Int32& iOutEntry);

   // Looks up the index of the context entry that falls just above
   // the accumulated count.
   Bool lookupEntryByCumCount(Int32 iCount, Int32& iOutEntry);
};
```

C.1.3 CodecDriver class

```
//
// A class that deals with the conversions from SYMBOL to VALUE and
// provides end-consumer APIs for using the codecs.
//
class CodecDriver
{
public:
    // ---------- Codec Decoding Interface ----------
    // Returns the number of symbols to be read
    Int32 numSymbolsToRead();

    // Returns index of the first context entry and total number of bits
    Bool getDecodeData(Int32& iFirstContext, Int32& nTotalBits);

    // Returns the next encoded symbol and its number of bits
    Bool getNextCodeText(UInt32& uCodeText, Int32& nBits);

    // Adds the decoded symbol back to the driver
    Bool addOutputSymbol(Int32 iSymbol, Int32& iNextContext) ;

    // ---------- Symbol Probability Context Interface ----------
    Bool clearAllContexts();

    // Retrieves a new probability context
    Bool getNewContext(ProbabilityContext& rpCntx);

    // Returns the total number of contexts
    Int32 numContexts();

    // Returns the probability context for a given index
    Bool getContext(Int32 iSymContext, ProbabilityContext& rpCntx);

    // ---------- Predictor Type Residual Unpacking ----------

    typedef enum
    {
        PredLag1      = 0,
        PredLag2      = 1,
        PredStride1   = 2,
        PredStride2   = 3,
        PredStripIndex = 4,
        PredRamp      = 5,
        PredXor1      = 6,
        PredXor2      = 7,
        PredNULL      = 8
    } PredictorType;

    // Returns the original values from the predicted residual values.
    static Bool unpackResiduals(Vector<Int32>& rvResidual,
                                Vector<Int32>& rvVals,
                                PredictorType  ePredType);

    static Bool unpackResiduals(Vector<Float64>& rvResidual,
                                Vector<Float64>& rvVals,
                                PredictorType  ePredType);

    // Predict values
```

217

```
   static Int32 predictValue(Vector<Int32>& vVal,
                             Int32 iIndex,
                             PredictorType ePredType);

   static Float64 predictValue(Vector<Float64>& vVal,
                               Int32 iIndex,
                               PredictorType ePredType);
}

Bool CodecDriver::unpackResiduals(Vector<Int32>& rvResidual,
                                  Vector<Int32>& rvVals,
                                  PredictorType  ePredType)
{
   Int32 iPredicted;

   Int32 len = rvResidual.length();
   rvVals.setLength(len);
   Int32* aVals = (Int32 *) rvVals;
   Int32* aResidual = (Int32 *) rvResidual;

   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, i, ePredType);

         if( ePredType == PredXor1 || ePredType == PredXor2 )
         {
            // Decode the residual as the current value XOR predicted
            aVals[i] = aResidual[i] ^ iPredicted;
         }
         else
         {
            // Decode the residual as the current value plus predicted
            aVals[i] = aResidual[i] + iPredicted;
         }
      }
   }

   return True;
}

Bool CodecDriver::unpackResiduals(Vector<Float64>& rvResidual,
                                  Vector<Float64>& rvVals,
                                  PredictorType  ePredType)
{
   if( ePredType == PredXor1 || ePredType == PredXor2 )
      return False;

   if( ePredType == PredNULL )
   {
      rvVals = rvResidual;
      return True;
   }
```

```
    Float64 iPredicted;
    Int32 len = rvResidual.length();
    rvVals.setLength(len);

    for( Int32 i = 0; i < len; i++ )
    {
       if( i < 4 )
       {
          // The first four values are just primers
          rvVals[i] = rvResidual[i];
       }
       else
       {
          // Get a predicted value
          iPredicted = predictValue(rvVals, i, ePredType);

          // Decode the value as the residual plus predicted
          rvVals[i]  = rvResidual[i] + iPredicted;
       }
    }

    return True;
}

Int32 CodecDriver::predictValue(Vector<Int32>& vVal,
                                Int32 iIndex,
                                PredictorType ePredType)
{
    Int32* aVals = (Int32 *) rvVals;
    JtInt32 iPredicted,
            v1 = aVals[iIndex-1],
            v2 = aVals[iIndex-2],
            v3 = aVals[iIndex-3],
            v4 = aVals[iIndex-4];

    switch( ePredType )
    {
       default:
       case PredLag1:
       case PredXor1:
          iPredicted = v1;
          break;

       case PredLag2:
       case PredXor2:
          iPredicted = v2;
          break;

       case PredStride1:
          iPredicted = v1 + (v1 - v2);
          break;

       case PredStride2:
          iPredicted = v2 + (v2 - v4);
          break;

       case PredStripIndex:
          if( v2 - v4 < 8 && v2 - v4 > -8 )
             iPredicted = v2 + (v2 - v4);
          else
             iPredicted = v2 + 2;
```

```
            break;

        case PredRamp:
            iPredicted = iIndex;
            break;
    }

    return iPredicted;
}


Float64 CodecDriverBase::predictValue(Vector<Float64>& vVal,
                                      Int32 iIndex,
                                      PredictorType ePredType)
{
    Float64* aVals = (Float64 *) rvVals;
    Float64 iPredicted,
            v1 = aVals[iIndex-1],
            v2 = aVals[iIndex-2],
            v3 = aVals[iIndex-3],
            v4 = aVals[iIndex-4];

    switch( ePredType )
    {
        default:
        case PredLag1:
            iPredicted = v1;
            break;

        case PredLag2:
            iPredicted = v2;
            break;

        case PredStride1:
            iPredicted = v1 + (v1 - v2);
            break;

        case PredStride2:
            iPredicted = v2 + (v2 - v4);
            break;

        case PredStripIndex:
            if( v2 - v4 < 8 && v2 - v4 > -8 )
                iPredicted = v2 + (v2 - v4);
            else
                iPredicted = v2 + 2;
            break;

        case PredRamp:
            iPredicted = iIndex;
            break;
    }

    return iPredicted;
}
```

C.2 Bitlength decoding classes

The following sub-sections contain a sample implementation of the decoding portion of the Bitlength CODEC algorithm. A summary technical explnation of the Bitlength CODEC can be found in 7.2.2 Bitlength CODEC.

C.2.1 BitLengthCodec class

```
class BitLengthCodec
{
public:
   // This method decodes a given stream of symbols into their values.
   // The stream is described by the codec driver
   Bool decode(CodecDriver* pDriver);

   Int32 cStepBits = 2;
};

Bool BitLengthcodec::decode(CodecDriver* pDriver)
{
   Int32 iBit;             // Codetext bit number
   Int32 nBits = 0;        // Number of codetext bits decoded so far
   Int32 nTotalBits = 0;   // Total number of codetext bits expected
   Int32 nValBits = 0;     // Number of accumulated value bits
   Int32 iContext = 0;     // Probability context number
   Int32 iSymbol;          // Decoded symbol value
   UInt32 uVal = 0;        // Current chunk of codetext bits
   UInt32 uAccVal = 0;     // Number of valid bits returned from
                           // getNextCodeText
   UInt32 uLastIncBit = 0; // Used to calculate whether terminator bit
                           // is 0 or 1
   Int32 cNumCurBits = 0;  // Current field width in bits
   Int32 nAccBits = 0;     // Number of bits accum'ed in uAccVal
   Int32 iDecodeState = 0; // State of decoder; see below

   // Get codetext from the driver and loop over it until it's gone!
   pDriver->getDecodeData(iContext, nTotalBits);

   while( nBits < nTotalBits )
   {
      // Get the next 32 bits from the input stream
      pDriver->getNextCodeText(uVal, nValBits);

      // Scan through each bit either walking the Huffman code
      // tree or accumulating escaped bit values.
      Int32 n = min(32, min(nValBits, nTotalBits - nBits));
      for( iBit = 0; iBit < n ; iBit++ )
      {
         // Code-accumulation mode is handled is this block
         // as many bits at a time as possible.
         if( iDecodeState == 2 )
         {
            // Slice off as many bits as we can all at once.
            Int32 m = min(n - iBit, cNumCurBits - nAccBits);
            if( m < 32 )
            {
               uAccVal <<= m;
               uAccVal |= ((uVal >> (32 - m)) & ((1 << m) - 1));
               nAccBits += m;
               iBit += m - 1;
```

```
            // Advance the bit-marching counters
            uVal <<= m;
            nBits += m;
            nValBits -= m;
         }
         else
         {
            uAccVal = uVal;
            nAccBits += m;
            iBit += m - 1;

            // Advance the bit-marching counters
            uVal = 0;
            nBits += m;
            nValBits -= m;
         }

         if( nAccBits >= cNumCurBits )
         {
            // Convert and sign-extend the symbol
            iSymbol = Int32(uAccVal);
            iSymbol <<= (32 - cNumCurBits);
            iSymbol >>= (32 - cNumCurBits);

            // Output the symbol and restart
            pDriver->addOutputSymbol(iSymbol, iContext);
            iDecodeState = 0;
            uAccVal  = 0;
            nAccBits = 0;
         }
      }
      else
      {
         // All other decode states are handled one bit at a time
         // inside this block.
         // Get the next bit
         uAccVal = (uVal >> 31);

         switch( iDecodeState )
         {

            // DecodeState = 0: Recognize prefix bit (0=Same size
            // code, 1=Different size code).
            case 0:
               // Recognize "same length" prefix code
               if( uAccVal == 0 )
                  iDecodeState = 2;
               else
               {
                  // Recognize "different length" prefix code
                  iDecodeState = 1;
                  uLastIncBit = 2;
               }

               uAccVal  = 0;
               break;

            case 1:  // Length adjustment mode
               // Recognize the terminator bit
               if( uLastIncBit != 2 && (uAccVal ^ uLastIncBit) )
               {
```

```
                          iDecodeState = 2;
                          uLastIncBit = 2;
                      }
                      else
                      {
                          // Recognize the "increment" prefix code
                          if( uAccVal == 1 )
                          {
                              cNumCurBits += cStepBits;
                          }
                          else
                          {
                              // Recognize the "decrement" prefix code
                              cNumCurBits -= cStepBits;
                          }

                          uLastIncBit = uAccVal;
                      }

                      uAccVal  = 0;
                      break;
              }

              // Advance the bit-marching counters that keep track of the
              // "current codetext bit", and how many bits are left.
              uVal <<= 1;
              nBits++;
              nValBits--;
          }
      }
  }

  // If the last symbol was zero and the current bit length
  // is also zero, then the above loop terminated before
  // actually decoding the last zero-valued symbol.  Test
  // for that condition here and decode it if necessary.
  if( iDecodeState == 2 && cNumCurBits == 0 )
  {
      // Output the symbol and restart
      iSymbol = Int32(0);
      pDriver->addOutputSymbol(iSymbol, iContext);
  }

  return True;
}
```

C.3 Huffman decoding classes

The following sub-sections contain a sample implementation of the decoding portion of the Huffman CODEC algorithm. A summary technical explnation of the Huffman CODEC can be found in 7.2.3 Huffman CODEC.

C.3.1 HuffCodeData class

HuffCodeData is a helper class for keeping track of a given symbol and the bits used to describe it.

```
class HuffCodeData
{
public:
   HuffCodeData() :
      iSymbol(0), iBitCode(0), nCodeLen(0)
   {
   }

   HuffCodeData(Int32&  symbol,
                UInt32& bitCode,
                Int32&  codeLen) :
      iSymbol(symbol), iBitCode(bitCode), nCodeLen(codeLen)
   {
   }

   HuffCodeData(Int32& symbol) :
      iSymbol(symbol), iBitCode(0), nCodeLen(0)
   {
   }

   Bool operator < (HuffCodeData& rhs)
   {
      if( this->iSymbol < rhs.iSymbol )
         return True;
      else
         return False;
   }

   Bool operator == (HuffCodeData& rhs)
   {
      if( this->iSymbol == rhs.iSymbol )
         return True;
      else
         return False;
   }

   Int32  iSymbol;
   Int32  nCodeLen;
   UInt32 iBitCode;
};
```

C.3.2 HuffTreeNode class

HuffTreeNode is a helper class used in the construction of the Huffman tree. It contains the symbol, its frequency, the Huffman code and its length, and pointers to the 'left' and 'right' nodes.

```
class HuffTreeNode
{
public:
   HuffTreeNode() :
      cSymcounts(0)
   {
```

```
   }

   Bool operator < (HuffTreeNode& rhs)
   {
      if( this->cSymCounts < rhs.cSymCounts )
         return True;
      else
         return False;
   }

   Int32       cSymCounts;
   HuffTreeNode* pLeft;
   HuffTreeNode* pRight;
   HuffCodeData sData;
};
```

C.3.3 HuffCodecContext class

HuffCodecContext is a class that defines the Huffman context

```
class HuffCodecContext
{
public:
   HuffCodecContext() :
      cLength(0), nCodeLength(0), uCode(0)
   {
   }

   // Used when constructing the Huffman code
   Int32  cLength; // Length of Huffman code currently
                   // under construction.
   UInt32 uCode;   // Code under construction

   // Used to store the final Huffman code table
   OrderedVector<HuffCodeData> vCodes; // Ordered by symbol number

   // Used during encoding
   Int32 nCodeLength; // Used to tally up total encoded code length
};
```

C.3.4 HuffmanCodec class

HuffmanCodec is the class that decodes Huffman encoded data.

```
class HuffmanCodec
{
public:
   // Decodes the Huffman codetext present in the vInCode entries to
   // a list of symbols, placing the symbols onto the driver object.
   // This method must contruct a Huffman tree from the symbol
   // statistics present on driver object.
   Bool decode(CodecDriver* pDriver);

private:
   // Build Huffman tree for each probability context
   Bool buildHuffmanForest(CodecDriver* pDriver);

   // Build Huffman tree from symbol statistics
   Bool buildHuffmanTree(ProbabilityContext* pCntx,
                         HuffTreeNode* pRootNode);

   // Assign Huffman bit-codes to leaves of tree
```

```
   Bool assignCodeToTree(HuffTreeNode* pRoot,
                         HuffCodecContext& rCntxt);

   // Convert codetext vector to symbol vector
   Bool codetextToSymbols(CodecDriver* pDriver);

   Vector<HuffTreeNode*> vpHuffTrees; // Indexed by context number
   Vector<HuffCodecContext> vHuffCntx; // HuffmanCodecContexts
};

Bool HuffmanCodec::decode(CodecDriver* pDriver)
{
   // Build a Huffman tree for each probability context
   buildHuffmanForest(pDriver);

   // Convert codetext to symbols
   codetextToSymbols(pDriver);

   return True;
}

Bool HuffmanCodec::buildHuffmanForest(CodecDriver* pDriver)
{
   HuffTreeNode* pRoot = NULL;
   Int32 nCntx = pDriver->numContexts();
   Int32 i;
   for( i = 0; i < nCntx; i++ )
   {
      // Get the i'th context
      ProbabilityContext* pCntx = NULL;
      pDriver->getContext(i, pCntx);

      // Create Huffman tree from probability context
      buildHuffmanTree(pCntx, pRoot);

      // Assign Huffman codes
      assignCodeToTree(pRoot, vHuffCntx[i]);

      // Store the completed Huffman tree
      vpHuffTrees[i] = pRoot;
   }

   return True;
}

Bool HuffmanCodec::buildHuffmanTree(ProbabilityContext* pCntx,
                                    HuffTreeNode* pRootNode)
{
   HeapVector<HuffTreeNode*> heap;
   HuffTreeNode* pNode = NULL;

   // Initialize all the nodes and add them to the heap.
   Int32 nEntries = pCntx->numEntries();
   for( Int32 i = 0; i < nEntries; i++ )
   {
      CntxEntry* pEntry = NULL;
      pCntx->getEntry(i, pEntry);
      pNode->sData.iSymbol = pEntry->iSym;
      pNode->cSymCounts    = pEntry->cCount;
      pNode->pLeft = NULL;
      pNode->pRight = NULL;
```

```
      heap.add(pNode);
   }

   HuffTreeNode* pNewNode1 = NULL;
   HuffTreeNode* pNewNode2 = NULL;

   while( heap.length() > 1 )
   {
      // Get the two lowest-frequency nodes.
      heap.getMin(pNewNode1);
      heap.getMin(pNewNode2);

      //Combine the low-freq nodes into one node.
      pNode->sData.iSymbol = 0xdeadbeef;
      pNode->pLeft        = pNewNode1;
      pNode->pRight       = pNewNode2;
      pNode.cSymCounts = pNewNode1->cSymCounts + pNewNode2->cSymCounts;

      //Add the new node to the heap.
      heap.add(pNode);
   }

   // Set the root node.
   heap.getMin(pNode);
   pRootNode = pNode;

   return True;
}

Bool HuffmanCodec::assignCodeToTree(HuffTreeNode* pNode,
                                    HuffCodecContext& rCntxt)
{
   if( pRoot->pLeft != 0 )
   {
      rCntxt.uCode <<= 1;
      rCntxt.uCode |=  1;
      rCntxt.cLength++;
      assignCodeToTree(pRoot->pLeft, rCntxt);
      rCntxt.cLength--;
      rCntxt.uCode >>= 1;
   }

   if( pRoot->pRight != 0 )
   {
      rCntxt.uCode <<= 1;
      rCntxt.cLength++;
      assignCodeToTree(pRootpRight, rCntxt);
      rCntxt.cLength--;
      rCntxt.uCode >>= 1;
   }

   if( pRoot->pRight != 0 )
      return True;

   // Set the code and its length for the node.
   pRoot->sData.iBitCode = rCntxt.uCode;
   pRoot->sData.nCodeLen = rCntxt.cLength;

   // Setup the internal symbol look-up table.
   Int32 null = 0;
   rCntxt.vCodes.insert(HuffCodeData(pRoot->sData.iSymbol,
```

227

```
                                        pRoot->sData.iBitCode,
                                        pRoot->sData.nCodeLen), null);


    return True;
}

Bool HuffmanCodec::codetextToSymbols(CodecDriver* pDriver)
{
    HuffTreeNode* pHNode = NULL;
    UInt32 mask = 1 << 31;
    Int32 j,
          nBits = 0,
          nTotalBits = 0,
          nValBits = 0,
          iContext = 0;
    UInt32 uVal;

    pDriver->getDecodeData(iContext, nTotalBits);
    pHNode = vpHuffTrees[iContext];

    while( nBits < nTotalBits )
    {
        // Get the next 32 bits from the input stream
        pDriver->getNextCodeText(uVal, nValBits);

        // Scan through each bit either walking the Huffman code
        // tree or accumulating escaped bit values.
        for( j = 0; j < 32 && nBits < nTotalBits && nValBits > 0; j++ )
        {
            // March to the next node
            pHNode = (uVal & mask) ? pHNode->pLeft : pHNode->pRight;

            // If the node is a leaf, output a symbol and restart
            if( pHNode->pLeft == 0 && pHNode->pRight == 0 )
            {
                pDriver->addOutputSymbol(pHNode->sData.iSymbol, iContext);
                pHNode = vpHuffTrees[iContext];
            }

            uVal <<= 1;
            nBits++;
            nValBits--;
        }
    }

    return True;
}
```

C.4 Arithmetic decoding classes

The following sub-sections contain a sample implementation of the decoding portion of the Arithmetic CODEC algorithm. A summary technical explnation of the Arithmetic CODEC can be found in 7.2.4 Arithmetic CODEC.

C.4.1 ArithmeticProbabilityRange class

```
class ArithmeticProbabilityRange
{
public:
   UInt16 low_count;
   UInt16 high_count;
   UInt16 scale;
}
```

C.4.2 ArithmeticCodec class

ArithmeticCodec class is the class that decodes arithmetic encoded data.

```
class ArithmeticCodec
{
public:
   ArithmeticCodec() :
      code = 0x0000,
      low = 0x0000,
      high = 0xffff,
      nUnderflowBits = 0,
      bitBuffer =0x00000000,
      nBits = 0
   {
   }

   // Decodes a list of symbols. The codecDriver provides the range
   // info for the symbols to decode.  It also stores the symbols as
   // they are decoded.
   Bool decode(CodecDriver* pDriver);

private:
   // Remove the most recently decoded symbol from the front of the
   // list of encoded symbols.
   Bool removeSymbolFromStream(ArithmeticProbabilityRange& sym,
                              CodecDriver* pDriver);

   //State variables used in decoding.
   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

   UInt32 bitBuffer; // Temporary i/o buffer
   Int32  nBits;     // Number of bits in _bitBuffer
};

Bool ArithmeticCodec::decode(CodecDriver* pDriver )
{
   ArithmeticProbabilityRange newSymbolRange;
   Int32 iCurrContext, nDummyTotalBits, cSymbolsCurrCtx, iCurrEntry;

   Int32 nSymbols = pDriver->numSymbolsToRead(),

   ProbabilityContext* pCurrContext = NULL;
   CntxEntry* pCntxEntry = NULL;
```

```
    // Initialize decoding process
    Int32 nBitsRead = -1;
    pDriver->getNextCodeText(bitBuffer, nBitsRead);

    low  = 0;
    high = 0xffff;
    code = (bitBuffer >> 16);

    bitBuffer <<= 16;
    nBits = 16;

    // Begin decoding
    pDriver->getDecodeData(iCurrContext, nDummyTotalBits);
    for( Int32 ii = 0; ii < nSymbols; ii++ )
    {
        pDriver->getContext(iCurrContext, pCurrContext);

        cSymbolsCurrCtx = pCurrContext->totalCount();
        UInt16 rescaledCode =
            ((((UInt32)(code - low) + 1) * (UInt32) cSymbolsCurrCtx - 1) /
              ((UInt32)(high - low) + 1));

        pCurrContext->lookupEntryByCumCount((Int32)rescaledCode,
                                            iCurrEntry);

        pCurrContext->getEntry(iCurrEntry, pCntxEntry);

        newSymbolRange.high_count = pCntxEntry->cCumCount +
                                    pCntxEntry->cCount;
        newSymbolRange.low_count  = pCntxEntry->cCumCount;
        newSymbolRange.scale      = cSymbolsCurrCtx;

        removeSymbolFromStream(newSymbolRange, pDriver);

        pDriver->addOutputSymbol(pCntxEntry);

        iCurrContext = pCntxEntry->iNextCntx;
    }

    return True;
}

Bool ArithmeticCodec::removeSymbolFromStream(
                        ArithmeticProbabilityRange& sym,
                        CodecDriver* pDriver)
{
    // First, the range is expanded to account for the symbol removal.
    UInt32 range = UInt32(high - low)  + 1;
    high = low + (UInt32)((range * sym.high_count) / sym.scale - 1);
    low  = low + (UInt32)((range * sym.low_count ) / sym.scale);

    //Next, any possible bits are shipped out.
    for (;;)
    {
        // If the most signif digits match, the bits will be shifted out.
        if( (~(high^low)) & 0x8000 )
        {
        }
        else if( (low & 0x4000) && !(high & 0x4000) )
        {
            // Underflow is threatening, shift out 2nd most signif digit.
```

```
            code ^= 0x4000;
            low  &= 0x3fff;
            high |= 0x4000;
        }
        else
        {
            // Nothing can be shifted out, so return.
            return True;
        }

        low  <<= 1;
        high <<= 1;
        high |=  1;
        code <<= 1;

        if( nBits == 0 )
        {
            Int32 nBits = -1;
            pDriver->getNextCodeText(bitBuffer, nBits);
            nBits = 32;
        }

        code |= (UInt16)(bitBuffer >> 31);
        bitBuffer <<= 1;
        nBits--;
    }
}
```

C.5 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 explnation of the Deering Normal CODEC can be found in 7.2.5 Deering Normal CODEC.

C.5.1 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;
```

231

```
   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;
}
```

C.5.2 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 must 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 D: **Parasolid XT Format Reference**

November 2008

Table of Contents

Introduction to the Parasolid
XT Format

This Parasolid® Transmit File Format manual describes the formats in which Parasolid represents model information in external files. Parasolid is a geometric modeling kernel that can represent wireframe, surface, solid, cellular and general non-manifold models.

Parasolid stores topological and geometric information defining the shape of models in transmit files. These files have a published format so that applications can have access to Parasolid models without necessarily using the Parasolid kernel.

This manual documents the Parasolid transmit file format. This format will change in subsequent Parasolid releases at which time this manual will be updated. As new versions of Parasolid can read and write older transmit file formats these changes will not invalidate applications written based on the information herein.

Types of File Documented

There are a number of different interface routines in Parasolid for writing transmit files. Each of these routines can write slightly different combinations of Parasolid data, the ones that are documented herein are:

- Individual components (or assemblies) written using SAVMOD

- Individual components written using PK_PART_transmit

- Lists of components written using PK_PART_transmit

- Partitions written using PK_PARTITION_transmit

The basic format used to write data in all the above cases is identical; there are a small number of node types that are specific to each of the above file types.

Text and Binary Formats

Parasolid can encode the data it writes out in four different formats:

1. Text (usually ASCII)

2. Neutral binary

3. Bare binary (this is not recommended)

4. Typed binary

In text format all the data is written out as human readable text, they have the advantage that they are readable but they also have a number of disadvantages. They are relatively slow to read and write, converting to and from text forms of real numbers introduces rounding errors that can (in extreme cases) cause problems and finally there are limitations when dealing with multi-byte character sets. Carriage return or line feed characters can appear anywhere in a text transmit file but other unexpected non-printing characters will cause Parasolid to reject the file as corrupt.

Neutral binary is a machine independent binary format.

Bare binary is a machine dependent binary format. It is not a recommended format since the machine type which wrote it must be known before it can be interpreted.

Typed binary is a machine dependent binary format, but it has a machine independent prefix describing the machine type that wrote it and so can be read on all machine types.

Standard File Names and Extensions

Due to changing operation system restrictions on file names over the years Parasolid has used several different file extensions to denote file contents. The recommended set of file extensions is:

- .X_T and .X_B for part files, .P_T and .P_B for partition files.

Extensions that have been used in the past are:

- xmt_txt, xmp_txt - text format files on VMS or Unix platforms

- xmt_bin, xmp_bin - binary format files on VMS or Unix platforms

Logical Layout

The logical layout of a Parasolid transmit file is:

- A human-oriented text header.

  - The initial text header is read and written by applications' Frustrums and is not accessible to Parasolid. Its detailed format is described in the section `Physical layout'.

- A short flag sequence describing the file 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 modeller version used to write the file, as a text string of the form:

    : TRANSMIT FILE created by modeller version 1200123

    This information is used by routines such as PK_PART_ask_kernel_version.

  the schema version describing the field sequences of the part nodes as a text string of the form:

    SCH_1200123_12006

    This example denotes a file written by Parasolid V12.0.123 using schema number 12006: there will be a corresponding file sch_12006 in the Parasolid schema distribution.

    Note that applications writing XT files should use version 1200000 and schema number 12006.

  - The user field size is a simple integer.

- The objects (known as 'nodes') in the file in an unordered sequence, followed by a terminator.

  - Every node in the file 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. If the file contains user fields, and the node is visible at the PK interface, then the fields are followed by the user field, in integers.

  - 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 '0' in a text file, and as a 2-byte integer with value 1 in a binary file.

  - The node with index 1 is the root node of the transmit file as follows:

  -

Contents of file	Type of root node
Body	BODY
Assembly	ASSEMBLY
Array of parts	POINTER_LIS_BLOCK
Partition	WORLD

243

Schema

Parasolid permanent structures are defined in a special language akin to C which generates the appropriate files for a C compiler, the runtime information used by Parasolid, along with a schema file used during transmit and receive. The schema file for version 12.0 is named sch_12006 and is distributed with Parasolid. It is not necessary to have a copy of this file to understand the XT format.

For each node type, the schema file has a node specifier of the form

<nodetype> <nodename>; <description>; <transmit 1/0> <no. of fields> <variable 1/0>

e.g.

29 POINT; Point; 1 6 0

This is followed by a list of field specifiers which say what fields, and in what order, occur in the transmit file.

Field specifiers have the format:

<fieldname>; <type>; <transmit 1/0> <node class> <n_elements>

e.g.

owner; p; 1 1011 1

Nodes and fields with a transmit flag of zero are ephemeral information not written to a transmit file. Only pointer fields have non-zero node class, in which case it specifies the set of node types to which this field is allowed to point. The element count is interpreted as follows:

0   a scalar, a single value

1   a variable length field (see below)

n > 1   an array of n values

Note that in the schema file, fins are referred to as 'halfedges', and groups are referred to as 'features'. These are internal names not used elsewhere in this document.

Embedded schemas

When reading a part, partition, or delta, Parasolid converts any data that it encounters from older versions of Parasolid to the current format using a mixture of automatic table conversion (driven by the appropriate schemas), and explicit code for more complex algorithms.

However, backwards compatibility of file information – that is, reading data created by a newer version of Parasolid into an application (such as data created by a subcontractor) – can never be guaranteed to work using this method, because the older version does not contain any special-case conversion code.

From Parasolid V14 onwards, parts, partitions and deltas can be transmitted with extra information that is intended to replace the schema normally loaded to describe the data layout. This information contains the **differences** between its schema and a defined base schema (currently V13's SCH_13006).

This enables parts, partitions, and deltas to be successfully read into older versions of Parasolid without loss of information.

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 transmitted. Specifically, a full schema definition can contain fields that are not relevant in the context of the transmitted data (fields relating to snapshots, for example), 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)

Physical layout

Most of the 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 Parasolid data directly. Two important elements are

- **short string**s

    These are transmitted as an integer length (**byte**) followed by the characters (without trailing zero).

- **positive integer**s

    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.

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 string**s)

    - fields one by one as

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.

Space compression

For text data in transmit formats PK_transmit_format_text_c and PK_transmit_format_xml_c, a new escape sequence is defined: the 2-character sequence **\9** denotes a sequence of nine spaces. At V14, this applies to attribute definition names, field names, and attribute strings.

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 a given binary file is specified by the "PS<code>" at the start of the file. See the chapter on "Physical Layout" for more information.

The full list of field types used in transmit files 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 files 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;

Note that the ordering is not the same as presented at Parasolid's external PK or KI interfaces.

h   array [3] of doubles

These represent points of intersection between two surfaces; only the position vector is written to a transmit file, as Parasolid will recalculate other data as required. The structure is documented further in the section on intersection curves.

Point

As an example, consider a POINT; its formal description is

```
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;
```

Its corresponding schema file entry is

29 POINT; Point; 1 6 0

node_id; d; 1 0 0

attributes_groups; p; 1 1019 0

owner; p; 1 1011 0

next; p; 1 29 0

previous; p; 1 29 0

pvec; v; 1 0 0

Pointer classes

In the above example, the attributes_groups field must be of class ATTRIB_GROUP_cl, the owner must be of class POINT_OWNER_cl, and the next and previous fields must refer to POINTs. A full list of node types and node classes is given at the end of the document.

Each node class corresponds to a union of pointers given in the Schema Definition section.

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. In the schema the length is notionally given as 1, e.g.

```
struct REAL_VALUES_s          // Real values
    {
    Double                    values[1];                // $f[]
    };
```

Its schema file entry would be

83 REAL_VALUES;   Real values; 1 1 1

values; f; 1 0 1

The number of entries in each such node is indicated by an integer in the transmit file between its nodetype and index, so an example might be

83 3 15 1 2 3

Unresolved indices

In some cases a node will contain an index field which does not correspond to a node in the transmit file, in this case the index is to be interpreted as zero.

Simple example

Here is a reformatted text example of a sheet circle with a color attribute on its single edge:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz*********

PARASOLID !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~0123456789*********

**PART1;MC=osf65;MC_MODEL=alpha;MC_ID=sdlosf6;OS=OSF1;OS_RELEASE=V4.0;FRU=sdl_paras olid_test_osf64;APPL=unknown;SITE=sdl-cambridge-u.k.;USER=davidj;FORMAT=text;GUISE=transmit;DATE=29-mar-2000;

**PART2;SCH=SCH_1200000_12006;USFLD_SIZE=0;

**PART3;

END_OF_HEADER

T51 : TRANSMIT FILE created by modeller version 120000017 SCH_1200000_120060

12 1 12 0 2 0 0 0 0 1e3 1e-8 0 0 0 1 0 3 1 3 4 5 0 6 7 0	body
70 2 0 1 0 0 4 1 20 8 8 8 1 T	list
13 3 3 0 1 0 9 0 0 6 9	shell
50 4 11 0 9 0 0 0 +0 0 0 0 0 1 1 0 0	plane
31 5 10 0 7 0 0 0 +0 0 0 0 0 1 1 0 0 1	circle
19 6 5 0 1 0 0 3 V	region

16 7 6 0 ?10 0 0 5 0 0 1	edge
17 10 0 11 10 10 0 12 7 0 0 +	fin
15 11 7 0 10 9 0	loop
17 12 0 0 0 0 0 10 7 0 0 -	fin (dummy)
14 9 2 13 ?0 0 11 3 4 +0 0 0 0 3	face
81 1 13 12 14 9 0 0 0 0 15	attribute (variable 1)
80 1 14 0 16 8001 0 0 0 0 3 5 0 0 FFFFTFTFFFFFF2	attrib_def (variable 1)
83 3 15 1 2 3	real_values (variable 3)
79 15 16 SDL/TYSA_COLOUR	att_def_id (variable 15)
74 20 8 1 0 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	pointer_lis_block
1 0	terminator

Note that the tolerance fields of the face and edge are unset, and represented as '?' in the text transmit file and that the annotations in the column 'body' to 'terminator' give the node type of each line and are not part of the actual file. If the above file had no trailing spaces, it would be a valid XT file (the leading spaces on some of the lines are necessary).

Physical Layout

Parasolid transmit files have two headers:

- a textual introduction containing human-directed information about the part, written by the Frustrum and not accessible to Parasolid, and

- an internal prefix to the part data, describing to Parasolid the format of the part data and thus not seen explicitly by an application's Frustrum.

Common header

The Parasolid common header recommended to Frustrum writers consists of:

- A preamble containing all characters in the ASCII printing set. This is used by the KID Frustrum to detect obvious network corruption, but could be used to attempt to translate a text file from one character set to another.

- Part 1 data: a sequence of keyword-value pairs, separated by semicolons, of possibly interesting information. All are optional.

```
MC              =       vax, hppa, sparc, ...

                                // make of computer

        MC_MODEL  =       4090, 9000/780, sun4m, ...

                                //  model of computer

        MC_ID       =       ...

                                // unique machine identifier

        OS          =       vms, HP-UX, SunOS, ...

                                // name of operating system

OS_RELEASE =       V6.2, B.10.20, 5.5.1, ...

                                //  version of operating system

FRU      =    sdl_parasolid_test_vax,

                mdc_ugii_v7.0_djl_can_vrh, ...

// frustrum supplier and implementation name

        APPL          =       kid, unigraphics, ...

// application which is using Parasolid

        SITE          =       ...

// site at which application is running

        USER          =       ...

                                // login name of user

        FORMAT      =       binary, text, applio

                                //  format of file

        GUISE       =       transmit, transmit_partition

                                //  guise of file
```

```
KEY           =       ...
                      // name of key
    FILE      =       ...
                      // name of file
    DATE      =       dd-mmm-yyyy
```

// e.g. 5-apr-1998

The 'part 1' data is 'standard' information which should be accessible to the Frustrum (e.g. by operating system calls). It is the responsibility of the Frustrum to gather the relevant information and to format it as described in this specification.

- part 2 data: a sequence of keyword-value pairs, separated by semicolons.

```
    SCH           =       SCH_m_n
```

// name of schema key e.g.SCH_1200000_12006

USFLD_SIZE =   m

// length of user field (0 - 16 integer words)

Applications writing XT files must use a schema name of SCH_1200000_12006

- part 3 data: non-standard information, which is only comprehensible to the Frustrum which wrote it.

The 'part 3' data is non-standard information, which is only comprehensible to the Frustrum which wrote it. However, other Frustrum implementations must be able to parse it (in order to reach the end of the header), and it should therefore conform to the same keyword/value syntax as for 'part 1' and 'part 2' data. However, the choice and interpretation of keywords for the 'part 3' data is entirely at the discretion of the Frustrum which is writing the header.

- a trailer record.

An example is:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz*********

PARASOLID !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~0123456789****************

**PART1;MC=vax;MC_MODEL=4090;MC_ID=VAX14;OS=vms;OS_RELEASE=V6.2;FRU=sdl_parasoli d_test_vax;APPL=unknown;SITE=sdl-cambridge u.k.;USER=ALANS;FORMAT=text;GUISE=transmit;KEY=temp;FILE=TEMP.XMT_TXT;DATE=8-sep-1997;

**PART2;SCH=SCH_701169_7007;USFLD_SIZE=0;

**PART3;

END_OF_HEADER

Keyword Syntax

All keyword definitions which appear in the three parts of data are written in the form

```
    <name>=<value> e.g. MC=hppa;MC_MODEL=9000/710;
```

where

  <name> consists of 1 to 80 uppercase, digit, or underscore characters

  <value> consists of 1 or more ASCII printing characters (except space)

251

Escape sequences provide a way of being able to use the full (7 bit) set of ASCII printing characters and the new line character within keyword values. Certain characters must be escaped if they are to appear in a keyword value:

• Character	Escape sequence
newline	^n
space	^_
semicolon	^;
uparrow	^^

The two character escape sequences may be split by a new line character as they are written to file. They must not cause any output lines to be longer than 80 characters.

Only those characters which belong to the ASCII (7 bit) printing sequence, plus the new line character, can be included as part of a keyword value.

Text

Parasolid has no knowledge of how files are stored. On writing, Parasolid produces an internal bytestream which is then split into roughly 80-character records separated by newline characters ('\n'). The newlines are not significant.

As operating systems vary in their treatment of text data, on reading all newline and carriage return characters ('\r') are ignored, along with any trailing spaces added to the records. However, leading spaces are not ignored, and the file must not contain adjacent space characters not at the end of a record.

Text XT files written by version 12.1 and later versions use escape sequences to output the following characters, except for '\n' at the end of each line:

null "\0"

carriage return "\n"

line feed "\r"

backslash "\\"

These characters are not escaped by versions 12.0 and earlier.

The flag sequence is the character 'T'. This is followed by the length of the modeler version, separated by a space from the characters of the modeler version, similarly the schema version, finally the userfield size. For example:

T

51 : TRANSMIT FILE created by modeller version 1200000

17 SCH_1200000_12006

0

NB: because of ignored layout, what Parasolid would read is

T51 : TRANSMIT FILE created by modeller version 120000017 SCH_1200000_120060

For partition files, the modeller version string would be given as

63 : TRANSMIT FILE (partition) created by modeller version 1200000

All numbers are followed by a single space to separate them from the next entry. Fields of type c and l are not followed by a space.

Logical values (0,1) are represented as characters F,T.

There are two special numeric values (-32764 for integral values, -3.14158e13 for floating point) which are used inside Parasolid to mark an 'unset' or 'null' value, and they are represented in a text transmit file as the question mark '?'. If a vector has one component null, then all three components must be null, and it will be output in a text file as a single '?'.

Binary

There are three types of binary file: `bare' binary, typed binary, and neutral binary. They are distinguished by a short flag sequence at the beginning of the file. In all cases, 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.

As with text files, there are two special numeric values (-32764 for integral values, -3.14158e13 for floating point) which are used inside Parasolid to mark an 'unset' or 'null' value, and they are represented in a text transmit file as the question mark '?'.

bare binary

In bare binary, data is represented in the natural format of the machine which wrote the data. The flag sequence is the single character 'B' (for ASCII machines, \102'). The data must be read on a machine with the same natural format with respect to character set, endianness and floating point format.

typed binary

In typed binary, data is represented in the natural format of the machine that wrote the data. The flag sequence is the 4-byte sequence "PS" followed by a zero byte and a one byte, i.e., 'P' 'S' '\0' '\1', followed by a 3-byte sequence of machine description.

	Byte order	Double representation	Character representation
0	Big-endian	IEEE	ASCII
1	Little-endian	VAX D-float	EBCDIC

neutral binary

In 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'. At Parasolid V9, 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 + 1) ); // remainder: add 1 so > 0
        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.

Model Structure

Topology

This section describes the Parasolid Topology model, it gives an overview of how the nodes in an XT file are joined together. In this section the word 'entity' means a node which is visible to a PK application – a table of which nodes are visible at the PK interface appears in the section `Node Types'.

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

- Disconnected bodies

Each entity is described, and its properties and links to other entities given.

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.

Entity definitions

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.

- A set of geometry (surfaces, curves and points).

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.

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 must 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 must be infinite; all other regions in the body must be finite.

- A set of geometry (surfaces, curve and/or points).

- A body-type. This may be wire, sheet, solid or general.

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 must have exactly one infinite region. The infinite region of a body must 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).

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 must contain at least one vertex, edge, or face.

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.

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 must 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.

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 must be the same as that of the corresponding face. The curve must not deviate by more than the edge tolerance from curves on other fins of the edge, and its ends must be within vertex tolerance of the corresponding vertices.

Note that fins are referred to as 'halfedges' in the Schema file.

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 must lie within vertex tolerance of this curve, and if it is a Trimmed Curve, they must lie within vertex tolerance of the corresponding ends of the curve. The curve must 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**.

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.

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. An XT document must 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 the transmit file.

- Owner.

- Fields. These are data fields consisting of one or more integers, doubles, vectors etc.

There are a number of system attribute definitions which Parasolid creates on startup. These are documented in the section `System Attribute Definitions'. Parasolid applications can create user attribute definitions during a Parasolid session. These are transmitted along with any attributes that use them.

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.

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 a transmit file.

Entity matrix

Thus the relations between entities can be represented in matrix form as follows. The numbers represent the number of distinct entities connected (either directly or indirectly) to the given one.

	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	-

Representation of manifold bodies

Body types

Parasolid 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 must obey the following constraints:

- An acorn body must consist of a single void region with a single shell consisting of a single vertex.

- A wire body must 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 must 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, and the

  A wire is called open if all its components are open, and closed if all its components are closed.

- Solid and sheet bodies must each contain at least one face; they may not contain any wireframe edges or acorn vertices.

- A solid body must consist of at least two regions; at least one of its regions must be solid. Every face in a solid body must 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 must have exactly two fins, which will have opposite senses. Every vertex in a solid body must either belong to a single isolated loop, or belong to one or more edges; in the latter case, the faces which use those edges must 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 must 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 must have exactly one or two fins; if it has two, these must have opposite senses. In a closed sheet body, all the edges will have exactly two fins. Every vertex in a sheet body must either belong to a single isolated loop, or belong to one or more edges; in the latter case, the faces which use those edges must 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 must 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, say).

Schema Definition

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;
    MEMBER_OF_GROUP_s
    };
typedef union ATTRIB_GROUP_u  ATTRIB_GROUP;
```

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;
```

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:

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;       // $p
    GEOMETRIC_OWNER_s
    char                    sense;                  // $c
    };
typedef struct ANY_CURVE_s *ANY_CURVE;
```

- **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
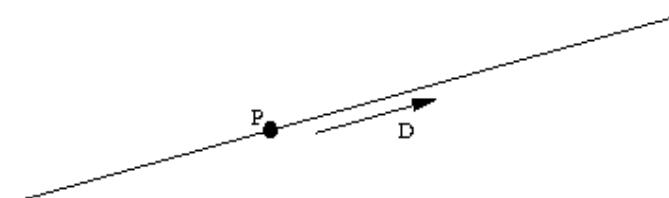
Field name	Data type	Description
pvec	vector	point on the line
direction	vector	direction of the line (a unit vector)

```
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;    // $p
    GEOMETRIC_OWNER_s
    char                         sense;                // $c
    vector                       pvec;                 // $v
    vector                       direction;            // $v
    };
typedef struct LINE_s     *LINE;
```

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.
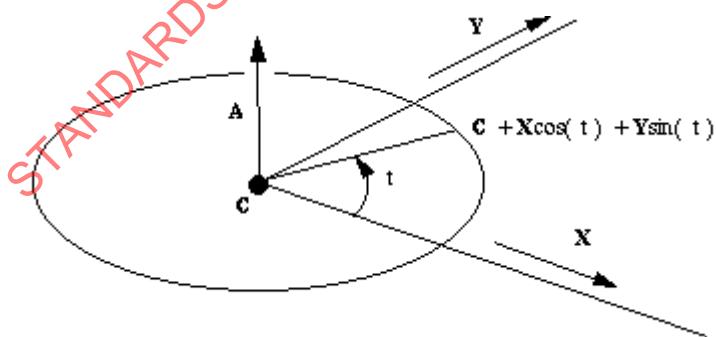
Field name	Data type	Description
centre	vector	Centre of circle
normal	vector	Normal to the plane containing the circle (a unit vector)
x_axis	vector	X axis in the plane of the circle (a unit vector)
radius	double	Radius of circle

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;           // $p
    GEOMETRIC_OWNER_s
    char                       sense;                      // $c
    vector                     centre;                     // $v
    vector                     normal;                     // $v
    vector                     x_axis;                     // $v
    double                     radius;                     // $f
    };
typedef struct CIRCLE_s    *CIRCLE;
```

- **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.
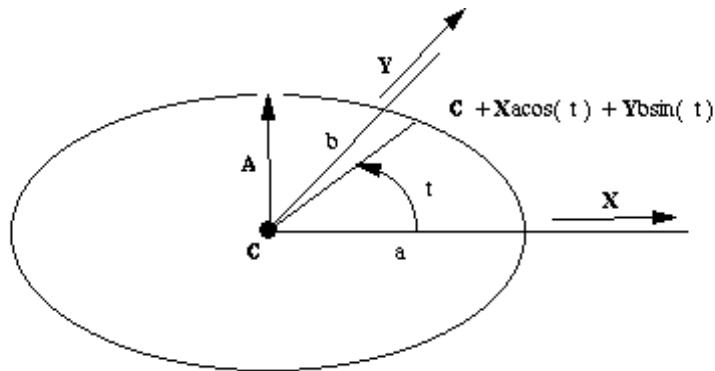
Field name	Data type	Description
centre	Vector	Centre of ellipse
normal	Vector	Normal to the plane containing the ellipse (a unit vector)
x_axis	Vector	major axis in the plane of the ellipse (a unit vector)
major_radius	Double	major radius
minor_radius	Double	minor radius

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;
```

B_CURVE (B-spline curve)

Parasolid supports B spline curves in full NURBS format. The mathematical description of these curves is:

- Non Uniform Rational B-splines as (NURBS)

$$P(t) = \frac{\sum\limits_{i=0}^{n-1} b_i(t) w_i V_i}{\sum\limits_{i=0}^{n-1} b_i(t) w_i}$$

- and the more simple Non Uniform B-spline

$$P(t) = \sum\limits_{i=0}^{n-1} b_i(t) V_i$$

-

- Where:

  n = number of vertices (n_vertices in the PK standard form)

  $V_0 ... V_{n-1}$ are the B-spline vertices

  $w_0 ... w_{n-1}$ are the weights

  $b_i(t), I = 0...n-1$ 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 must 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, Parasolid 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 Parasolid allows it to be defined to have periodic parametrisation. 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 Parasolid 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 must 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 Parasolid 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.

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 above for curves 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 must be continuous.

Parasolid does not support modelling with surfaces that are self-intersecting or contain cusps. Although they can be created they are not permitted to be attached to topology.

Field name	Data type	Description
nurbs	Pointer	Geometric definition
data	Pointer0	Auxiliary information

```
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 an XT file for a NURBS_CURVE is

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