
**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*

STANDARDSISO.COM : Click to view the full PDF of ISO 14306:2017



STANDARDSISO.COM : Click to view the full PDF of ISO 14306:2017



COPYRIGHT PROTECTED DOCUMENT

© ISO 2017, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Content

Content	iii
Foreword	xvi
Introduction.....	xvii
1 Scope	1
2 Normative references.....	1
3 Terms, definitions and abbreviated terms	1
3.1 Terms and definitions	1
3.2 Abbreviated terms	5
4 Notational conventions.....	6
5 File Format	14
5.1 File Structure	14
5.1.1 File Header	14
5.1.2 TOC Segment.....	16
5.1.3 Data Segment.....	17
5.2 Data Segments.....	22
6 LSG Segment	24
6.1 Graph Elements	24
6.1.1 Node Elements.....	25
6.1.2 Attribute Elements.....	48
6.2 Property Atom Elements.....	92
6.2.1 Base Property Atom Element	92
6.2.2 String Property Atom Element	93
6.2.3 Integer Property Atom Element	94
6.2.4 Floating Point Property Atom Element	94
6.2.5 JT Object Reference Property Atom Element.....	95
6.2.6 Date Property Atom Element.....	96
6.2.7 Late Loaded Property Atom Element	97
6.2.8 Vector4f Property Atom Element	99
6.3 Property Table	99
6.3.1 Element Property Table	100
7 Shape LOD Segment.....	102
7.1 Shape LOD Element	102
7.1.1 Base Shape LOD Element.....	102
7.1.2 Vertex Shape LOD Element	103
7.1.3 Tri-Strip Set Shape LOD Element	118
7.1.4 Polyline Set Shape LOD Element.....	118
7.1.5 Point Set Shape LOD Element	119
7.1.6 Null Shape LOD Element	120
7.2 Primitive Set Shape Element.....	120
7.2.1 Lossless Compressed Primitive Set Data.....	122
7.2.2 Lossy Quantized Primitive Set Data	124
8 Geometry Segments.....	129
8.1 STEP B-Rep Segment	129
8.2 XT B-Rep Segment	129
8.3 JT ULP Segment	129
8.4 JT LWPA Segment.....	129
8.5 Wireframe Segment.....	129
8.6 JT B-Rep Element (deprecated)	129
9 Meta Data Segment	131

9.1	Property Proxy Meta Data Element.....	131
9.1.1	Date Property Value	133
9.2	PMI Manager Meta Data Element	134
9.2.1	PMI Entities	137
9.2.2	PMI Associations	158
9.2.3	PMI User Attributes	160
9.2.4	PMI String Table	161
9.2.5	PMI Model Views.....	161
9.2.6	Generic PMI Entities	163
9.2.7	PMI CAD Tag Data	169
9.2.8	PMI Polygon Data	170
10	Data Compression and Encoding	174
10.1	Common Compression Data Collection Formats	174
10.1.1	Int32 Compressed Data Packet.....	174
10.1.2	Int32 Compressed Data Packet Mk. 2	179
10.1.3	Float64 Compressed Data Packet.....	185
10.1.4	Compressed Vertex Coordinate Array	189
10.1.5	Compressed Vertex Normal Array	190
10.1.6	Compressed Vertex Texture Coordinate Array	192
10.1.7	Compressed Vertex Colour Array.....	194
10.1.8	Compressed Vertex Flag Array.....	196
10.1.9	Point Quantizer Data	197
10.1.10	Texture Quantizer Data	197
10.1.11	Colour Quantizer Data.....	198
10.1.12	Uniform Quantizer Data	200
10.1.13	Compressed Entity List for Non-Trivial Knot Vector	200
10.1.14	Compressed Control Point Weights Data.....	204
10.1.15	Compressed Curve Data.....	205
10.1.16	Compressed CAD Tag Data.....	208
10.2	Encoding Algorithms	211
10.2.1	Uniform Data Quantization	211
10.2.2	Bitlength CODEC	211
10.2.3	Arithmetic CODEC	212
10.2.4	Deering Normal CODEC	217
10.3	zlib compression	219
11	Common Data Conventions and Constructs.....	220
11.1	Late-Loading Data	220
11.2	TOC Segment Location.....	220
11.3	Bit Fields.....	220
11.4	Empty Field	220
11.5	Local version numbers	220
11.5.1	Version numbers	221
11.6	Hash Value	223
11.7	Scene graph construction	223
11.8	Metadata Conventions	224
11.8.1	CAD Properties	224
11.8.2	PMI Properties	228
11.8.3	Tessellation Properties	324
11.8.4	Miscellaneous Properties	325
11.9	LSG Attribute Accumulation Semantics	326
11.10	LSG Part Structure	327
11.11	Range LOD Node Alternative Rep Selection	327
11.12	Brep Face Group Associations.....	327
11.13	Handling of different states / variants in a JT file	328
11.13.1	Reference Sets.....	329
11.14	Watermarks	329
11.14.1	Background.....	329
11.14.2	Issue – Missing specification in the ISO	330

11.14.3 Recommended Solution	330
11.14.4 Issue – Protection of data.....	332
12 Conformance requirements.....	333
Annex A (informative) Object Type Identifiers	334
Annex B (normative) STEP B-Rep	338
B.1 STEP B-Rep Element.....	338
B.1.1 STEP B-Rep Data.....	338
B.1.2 STEP schema.....	339
B.1.3 STEP format	339
B.1.4 STEP external element reference in ISO 10303-21.....	339
Annex C (normative) STEP schema	341
Annex D (informative) Coding Algorithms – An Implementation	414
D.1 Common classes	414
D.1.1 CntxEntry class	414
D.1.2 CntxEntryBase2 class.....	414
D.1.3 ProbabilityContext class.....	415
D.1.4 ProbContex2 class	416
D.1.5 CodecDriver class	419
D.1.6 CodecDriver2 class	422
D.2 Bitlength decoding classes	426
D.2.1 BitLengthCodec class.....	426
D.2.2 BitLengthCodec2 class.....	430
D.3 Arithmetic decoding classes.....	438
D.3.1 ArithmeticProbabilityRange class	438
D.3.2 ArithmeticCodec class.....	438
D.3.3 ArithmeticCodec2 class.....	442
D.4 Deering Normal decoding classes.....	446
D.4.1 DeeringNormalLookupTable class	446
D.4.2 DeeringNormalCodec class.....	448
Annex E (informative) Hashing – An Implementation	451
Annex F (informative) Polygon Mesh Topology Coder	454
F.1 DualVFMesh	455
F.2 Topology Decoder	461
F.2.1 MeshCoderDriver class.....	461
F.2.2 MeshCodec class.....	464
F.2.3 MeshDecoder class.....	470
Annex G (informative) XT B- Rep segment.....	473
G.1 XT B-Rep Element	473
G.1.1 XT B-Rep Data.....	474
G.2 Procedural Geometry	474
G.2.1 Background.....	474
G.2.2 Issue.....	474
G.2.3 Recommended Solution	476
G.3 Non-Manifold Bodies.....	476
G.3.1 Background.....	476
G.3.2 Issue.....	477
G.3.3 Recommended Solution	477
Annex H (informative) XT B- Rep data segment.....	478
H.1 Introduction to the XT B- Rep data segment	478
H.2 Logical Layout	478
H.2.1 Schema.....	479
H.2.2 Embedded schemas.....	479
H.2.3 Field types	480
H.2.4 Variable-length nodes	481
H.2.5 Unresolved indices.....	481

H.3	Physical Layout	481
H.3.1	Binary.....	481
H.4	Model Structure	482
H.4.1	Topology.....	482
H.4.2	General points	483
H.4.3	Entity definitions	483
H.4.4	Entity matrix.....	487
H.4.5	Representation of manifold bodies	487
H.5	Schema Definition	489
H.5.1	Underlying types	489
H.5.2	Geometry	489
H.5.3	Topology.....	521
H.5.4	Associated Data.....	534
H.6	Node Types	545
H.7	Node Classes	547
H.8	System Attribute Definitions	547
H.8.1	Hatching Attributes	547
H.8.2	Density Attributes.....	549
H.8.3	Region.....	551
H.8.4	Colour	551
H.8.5	Reflectivity	551
H.8.6	Translucency	552
H.8.7	Name	552
H.8.8	Incremental faceting.....	552
H.8.9	Transparency	552
H.8.10	Non-mergeable edges	553
H.8.11	Group merge behaviour.....	553
H.8.12	Unicode name	553
H.8.13	Monikers	553
Annex I (informative)	PMI Data Segment	559
Annex J (informative)	Mapping table from ISO 10303-42 to XT B-Rep	560
J.1	Wireframe Geometry	560
J.2	Surfaces without topology	560
J.3	Shapes represented by wireframe models with topology.....	561
J.4	Manifold Surfaces with Topology	561
J.5	Advanced boundary representation solids	563
J.5.1	Color	564
J.5.2	Name	565
J.5.3	Wireframe style.....	565
J.5.4	Visibility	566
Annex K (informative)	JT B-rep Segment	567
K.1	JT B-Rep Element.....	567
K.1.1	Topological Entity Counts	569
K.1.2	Geometric Entity Counts	570
K.1.3	Topology Data.....	571
K.1.4	Geometric Data	580
K.1.5	Topological Entity Tag Counters	589
K.1.6	B-Rep CAD Tag Data	591
Annex L (informative)	Wireframe Segment.....	592
L.1	Wireframe Rep Element	592
L.1.1	Wireframe MCS Curves Geometric Data	594
L.1.2	Wireframe Rep CAD Tag Data	594
Annex M (informative)	JT ULP Segment	596
M.1	JT ULP Element	596
M.1.1	Topology Data.....	598
M.1.2	Geometric Data	615
M.1.3	Material Attribute Element Properties	637

M.1.4	Information Recovery.....	638
M.1.5	Non-Manifold Bodies.....	643
M.2	JT LWPA Segment.....	643
M.2.1	JT LWPA Element.....	644
M.2.2	Analytic Surface Geometry.....	644
Annex N	(informative) Change History	647
N.1	General	647
N.2	Changes made in edition 2	647
N.2.1	Summary of the changes.....	647
N.2.2	Additions made to the document	647
N.2.3	Addition of a new STEP B-rep segment definition	647
N.2.4	Updates of the documentation from the Prostep industrialization guidelines	648
N.2.5	Addition of a conformance clause.....	648
N.2.6	Addition of a mapping table JT ISO to STEP.....	648
N.2.7	Content modified from edition 1	648
N.2.8	Elements deprecated from edition 1	648
Bibliography		649

Figures

Figure 1	— rectangle box diagram	6
Figure 2	— folder diagram	6
Figure 3	— rectangle box with lines at left and right sides diagram	7
Figure 4	— rectangle box with clipped right side corners	7
Figure 5	— compressed data packet diagram.....	7
Figure 6	— data type : field name diagram	8
Figure 7	— data filed dependency example.....	9
Figure 8	— loop construct example	10
Figure 9	— loop construct with iterations example	10
Figure 10	— JT File Structure.....	14
Figure 11	— File Header data collection	15
Figure 12	— TOC Segment data collection.....	16
Figure 13	— TOC Entry data collection.....	17
Figure 14	— Data Segment data collection.....	18
Figure 15	— Segment Header data collection	18
Figure 16	— Data collection	20
Figure 17	— Logical Element Header data collection.....	20
Figure 18	— Element Header data collection	20
Figure 19	— Logical Element Header ZLIB data collection.....	22
Figure 20	— LSG Segment data collection	24
Figure 21	— Base Node Element data collection	25
Figure 22	— Base Node Data collection.....	25
Figure 23	— Partition Node Element data collection	27
Figure 24	— Vertex Count Range data collection	28
Figure 25	— Group Node Element data collection.....	29
Figure 26	— Group Node Data collection	30
Figure 27	— Instance Node Element data collection	31
Figure 28	— Part Node Element data collection.....	32
Figure 29	— Assembly node with SUBNODE	32
Figure 30	— Assembly node without SUBNODE	33
Figure 31	— Displaying Nodes that have SUBNODE properties	33
Figure 32	— Meta Data Node Element data collection	34
Figure 33	— Meta Data Node Data collection	34
Figure 34	— LOD Node Element data collection	34
Figure 35	— LOD Node Data collection.....	35
Figure 36	— Range LOD Node Element data collection	36
Figure 37	— Switch Node Element data collection	37

Figure 38 — Base Shape Node Element data collection	38
Figure 39 — Base Shape Data collection	38
Figure 40 — Vertex Count Range data collection	40
Figure 41 — Vertex Shape Node Element data collection	41
Figure 42 — Vertex Shape Data collection	41
Figure 43 — Quantization Parameters data collection	42
Figure 44 — Tri-Strip Set Shape Node Element data collection	42
Figure 45 — Polyline Set Shape Node Element data collection	43
Figure 46 — Point Set Shape Node Element data collection	44
Figure 47 — Polygon Set Shape Node Element data collection	45
Figure 48 — NULL Shape Node Element data collection	45
Figure 49 — Primitive Set Shape Node Element data collection	46
Figure 50 — Primitive Set Quantization Parameters data collection	48
Figure 51 — Base Attribute Data collection	49
Figure 52 — Material Attribute Element data collection	51
Figure 53 — Texture Image Attribute Element data collection	54
Figure 54 — Texture Vers-1 Data collection	56
Figure 55 — Texture Environment data collection	58
Figure 56 — Texture Coord Generation Parameters data collection	61
Figure 57 — Inline Texture Image Data collection	62
Figure 58 — Image Format Description data collection	63
Figure 59 — Texture Vers-2 Data collection	66
Figure 60 — Texture Vers-3 Data collection	69
Figure 61 — Draw Style Attribute Element data collection	72
Figure 62 — Light Set Attribute Element data collection	73
Figure 63 — Infinite Light Attribute Element data collection	74
Figure 64 — Base Light Data collection	75
Figure 65 — Shadow Parameters data collection	76
Figure 66 — Point Light Attribute Element data collection	78
Figure 67 — Spread Angle value with respect to the light cone	79
Figure 68 — Attenuation Coefficients data collection	80
Figure 69 — Linestyle Attribute Element data collection	81
Figure 70 — Pointstyle Attribute Element data collection	82
Figure 71 — Geometric Transform Attribute Element data collection	83
Figure 72 — Shader Effects Attribute Element data collection	85
Figure 73 — Texture Coordinate Generator Attribute Element data collection	87
Figure 74 — Mapping Plane Element data collection	88
Figure 75 — Mapping Cylinder Element data collection	89
Figure 76 — Mapping Sphere Element data collection	90
Figure 77 — Mapping TriPlanar Element data collection	91
Figure 78 — Base Property Atom Element data collection	92
Figure 79 — Base Property Atom Data collection	93
Figure 80 — String Property Atom Element data collection	93
Figure 81 — Integer Property Atom Element data collection	94
Figure 82 — Floating Point Property Atom Element data collection	95
Figure 83 — JT Object Reference Property Atom Element data collection	95
Figure 84 — Date Property Atom Element data collection	96
Figure 85 — Late Loaded Property Atom Element data collection	98
Figure 86 — Vector4f Property Atom Element data collection	99
Figure 87 — Property Table data collection	100
Figure 88 — Element Property Table data collection	101
Figure 89 — Shape LOD Segment data collection	102
Figure 90 — Base Shape LOD Element data collection	102
Figure 91 — Base Shape LOD Data collection	103
Figure 92 — Vertex Shape LOD Element data collection	103
Figure 93 — Vertex Shape LOD Data collection	104
Figure 94 — TopoMesh LOD Data collection	106
Figure 95 — TopoMesh LOD Data collection	106
Figure 96 — TopoMesh Topologically Compressed LOD Data collection	107

Figure 97 — Topologically Compressed Rep Data Collection.....	108
Figure 98 — Topologically Compressed Vertex Records data collection.....	111
Figure 99 — TopoMesh Compressed Rep Data V1 data collection.....	112
Figure 100 — TopoMesh Compressed Rep Data V2 data collection.....	115
Figure 101 — Tri-Strip Set Shape LOD Element data collection	118
Figure 102 — Polyline Set Shape LOD Element data collection.....	119
Figure 103 — Point Set Shape LOD Element data collection	119
Figure 104 — Null Shape LOD Element data collection	120
Figure 105 — Primitive Set Shape Element data collection.....	121
Figure 106 — Lossless Compressed Primitive Set Data collection	123
Figure 107 — Lossy Quantized Primitive Set Data collection	125
Figure 108 — Compressed params1 data collection.....	127
Figure 109 — Meta Data Segment data collection	131
Figure 110 — Meta Data Segment data collection	132
Figure 111 — Date Property Value data collection	134
Figure 112 — PMI Manager Meta Data Element data collection	135
Figure 113 — PMI Entities data collection	137
Figure 114 — PMI Dimension Entities data collection.....	137
Figure 115 — PMI 2D Data collection	138
Figure 116 — PMI Base Data collection	139
Figure 117 — 2D-Reference Frame data collection	140
Figure 118 — 2D Text Data collection	140
Figure 119 — Text Box data collection	142
Figure 120 — Constructing Text Polylines from data arrays.....	143
Figure 121 — Text Polyline Data collection.....	144
Figure 122 — Constructing Non-Text Polylines from packed 2D data arrays	145
Figure 123 — Non-Text Polyline Data collection.....	146
Figure 124 — PMI Note Entities data collection	147
Figure 125 — PMI Datum Feature Symbol Entities data collection.....	148
Figure 126 — PMI Datum Target Entities data collection.....	148
Figure 127 — PMI Feature Control Frame Entities data collection	149
Figure 128 — PMI Line Weld Entities data collection	149
Figure 129 — PMI Spot Weld Entities data collection	150
Figure 130 — PMI 3D Data collection	151
Figure 131 — PMI Surface Finish Entities data collection	152
Figure 132 — PMI Measurement Point Entities data collection.....	153
Figure 133 — PMI Locator Entities data collection.....	154
Figure 134 — PMI Reference Geometry Entities data collection.....	154
Figure 135 — PMI Design Group Entities data collection	155
Figure 136 — Design Group Attribute data collection.....	156
Figure 137 — PMI Coordinate System Entities data collection	157
Figure 138 — PMI Associations data collection.....	158
Figure 139 — PMI User Attributes data collection	160
Figure 140 — PMI String Table data collection	161
Figure 141 — PMI Model Views data collection.....	162
Figure 142 — Generic PMI Entities data collection.....	164
Figure 143 — PMI Property data collection	166
Figure 144 — PMI Property Atom data collection	168
Figure 145 — PMI CAD Tag Data collection.....	169
Figure 146 — PMI Polygon Data	171
Figure 147 — Int32 Compressed Data Packet data collection.....	175
Figure 148 — Int32 Probability Contexts data collection	177
Figure 149 — Int32 Probability Context Table Entry data collection	178
Figure 150 — Int32 Compressed Data Packet Mk. 2 data collection.....	181
Figure 151 — Int32 Probability Contexts Mk. 2 data collection	183
Figure 152 — Int32 Probability Context Table Entry Mk. 2 data collection	184
Figure 153 — Float64 Compressed Data Packet data collection.....	186
Figure 154 — Float64 Probability Contexts data collection	188
Figure 155 — Float64 Probability Context Table Entry data collection	188

Figure 156 — Compressed Vertex Coordinate Array data collection	189
Figure 157 — Compressed Vertex Normal Array data collection	191
Figure 158 — Compressed Vertex Texture Coordinate Array data collection	193
Figure 159 — Compressed Vertex Colour Array data collection.....	195
Figure 160 — Compressed Vertex Flag Array data collection	197
Figure 161 — Point Quantizer Data collection.....	197
Figure 162 — Texture Quantizer Data collection.....	198
Figure 163 — Colour Quantizer Data collection	199
Figure 164 — Uniform Quantizer Data collection.....	200
Figure 165 — Compressed Entity List for Non-Trivial Knot Vector data collection	202
Figure 166 — Compressed Control Point Weights Data collection	204
Figure 167 — Compressed Curve Data collection	205
Figure 168 — Non-Trivial Knot Vector NURBS Curve Indices data collection.....	207
Figure 169 — NURBS Curve Control Point Weights data collection	207
Figure 170 — NURBS Curve Control Points data collection.....	207
Figure 171 — Compressed CAD Tag Data collection	209
Figure 172 — Compressed CAD Tag Type-2 Data collection	210
Figure 173 — Sextant Coding on the Sphere	218
Figure 174 — ISO 14306 format Convention for Modelling each Part in LSG	327
Figure 175 — CAD Component with Reference sets	329
Figure 176 — Watermark Example	330
Figure 177 — JT Data with Watermark in JT2GO	331
Figure 178 — JT Data with Watermark in LiteBox3D - ISO based JT Viewer	332
Figure 179 — STEP B-Rep Element data collection	338
Figure 180 — XT B-Rep Element data collection	473
Figure 181 — Non-manifold body	477
Figure 182 — Split a face	556
Figure 183 — Merge faces	557
Figure 184 — JT B-Rep Segment data collection.....	567
Figure 185 — JT B-Rep Segment data collection.....	568
Figure 186 — Topological Entity Counts data collection	569
Figure 187 — Geometric Entity Counts data collection	570
Figure 188 — Topology Data collection.....	571
Figure 189 — Regions Topology Data collection.....	572
Figure 190 — Shells Topology Data collection	573
Figure 191 — Trim Loop example in parameter Space - One Face with 2 Holes.....	574
Figure 192 — Faces Topology Data collection.....	575
Figure 193 — Loops Topology Data collection	576
Figure 194 — CoEdges Topology Data collection	577
Figure 195 — Edges Topology Data collection	578
Figure 196 — Vertices Topology Data collection	579
Figure 197 — Geometric Data collection	580
Figure 198 — Surfaces Geometric Data collection	581
Figure 199 — Non-Trivial Knot Vector NURBS Surface Indices data collection.....	582
Figure 200 — NURBS Surface Degree data collection	583
Figure 201 — NURBS Surface Control Point Counts data collection	583
Figure 202 — NURBS Surface Control Point Weights data collection	584
Figure 203 — NURBS Surface Control Points data collection.....	584
Figure 204 — NURBS Surface Knot Vectors data collection	585
Figure 205 — NURBS Surface Knot Vectors collection	585
Figure 206 — Trivial PCS Curves data collection	586
Figure 207 — Equality of corresponding curve end coordinates of opposite sides of the box	587
Figure 208 — MCS Curves Geometric Data collection	589
Figure 209 — Point Geometric Data collection	589
Figure 210 — Topological Entity Tag Counters data collection	590
Figure 211 — B-Rep CAD Tag Data collection	591
Figure 212 — wireframe segment data collection.....	592
Figure 213 — Wireframe Rep Element data collection	593
Figure 214 — Wireframe MCS Curves Geometric Data collection	594

Figure 215 — Wireframe Rep CAD Tag Data collection	595
Figure 216 — JT ULP Segment data collection	596
Figure 217 — JT ULP Element data collection	597
Figure 218 — Topology Data collection	598
Figure 219 — Topological Entity Counts data collection	599
Figure 220 — Combined Predictor Type data collection	600
Figure 221 — Regions Topology Data collection	601
Figure 222 — Shells Topology Data collection	602
Figure 223 — Faces Topology Data collection	603
Figure 224 — Loops Topology Data collection	606
Figure 225 — CoEdges Topology Data collection	608
Figure 226 — Surface Domain Classification	610
Figure 227 — Edges Topology Data collection	612
Figure 228 — Vertices Topology Data collection	614
Figure 229 — Geometric Data collection	615
Figure 230 — Geometric Entity Counts data collection	616
Figure 231 — Degree Table data collection	617
Figure 232 — Recover Nurbs Degree	618
Figure 233 — Number of Control Points Table data collection	619
Figure 234 — Recover Number of Control Points	620
Figure 235 — Dimension Table data collection	621
Figure 236 — Recover Dimension	622
Figure 237 — 3D Unit Vector Table data collection	623
Figure 238 — Recover Dimension	624
Figure 239 — 2D Unit Vector Table data collection	625
Figure 240 — Recover 2D Unit Vector	625
Figure 241 — 3D MCS Point Table data collection	626
Figure 242 — Recover 3D MCS Points	627
Figure 243 — Knot Vector Table data collection	628
Figure 244 — Recover Knot Vectors	629
Figure 245 — 1D MCS Table data collection	631
Figure 246 — Recover 1D MCS Table	632
Figure 247 — PCS Value Table data collection	633
Figure 248 — Recover PCS Value Table	634
Figure 249 — Radian Table data collection	634
Figure 250 — Recover Radian Table	635
Figure 251 — Weight Table data collection	636
Figure 252 — Recover Weight Table	637
Figure 253 — Material Attribute Element Properties	638
Figure 254 — Material Attribute Element Properties	639
Figure 255 — PCS Curve Recovery from Surface Domain	640
Figure 256 — MCS Curve Recovery	641
Figure 257 — MCS Curve Recovery from Surface Geometry	642
Figure 258 — PCS Curve Recovery from MCS Curve and Surface Geometry	643
Figure 259 — JT LWPA Segment data collection	643
Figure 260 — JT LWPA Element data collection	644
Figure 261 — Analytic Surface Geometry data collection	645
Figure 262 — Analytic Surface Creation	646

Tables

Table 1 — Symbols	6
Table 2 — Predictor Type	8
Table 3 — Basic Data Types	10
Table 4 — Composite Data Types	11
Table 5 — Segment attributes	17
Table 6 — Segment Types	18

Table 7 — Object Base Types	21
Table 8 — Compression flag values	22
Table 9 — Compression algorithm values	22
Table 10 — Node Flag values	26
Table 11 — Partition flag bits	27
Table 12 — SUBNODE Property	33
Table 13 — Compression level values	39
Table 14 — Texture Coord Binding values	47
Table 15 — Version Number values	47
Table 16 — Texture Coord Gen Type values	47
Table 17 — State Flag values	49
Table 18 — Material Attribute data field inhibit values	50
Table 19 — Material Attribute Version number value	52
Table 20 — Material Attribute Data Flag values	52
Table 21 — Texture Image Attribute data field inhibit values	53
Table 22 — Texture Image Version Number values	54
Table 23 — Texture Vers-1 type values	56
Table 24 — Texture Vers-1 Inline Image Storage Flag values	57
Table 25 — Texture Vers-1 Texture Environment Border Mode values	58
Table 26 — Texture Vers-1 Texture Environment Mipmap Magnification Filter values	59
Table 27 — Texture Vers-1 Texture Environment Mipmap Minification Filter values	59
Table 28 — Texture Vers-1 Texture Environment S-Dimen Wrap Mode values	60
Table 29 — Texture Vers-1 Texture Environment Blend Type values	60
Table 30 — Texture Vers-1 Texture Environment Internal Compression Level values	60
Table 31 — Texture Vers-1 Texture Coord Generation Gen Mode values	61
Table 32 — Texture Vers-1 Image Format Description Pixel Format values	63
Table 33 — Texture Vers-1 Image Format Description Pixel Data values	64
Table 34 — Texture Vers-1 Image Format Description Dimensionality values	64
Table 35 — Texture Vers-1 Image Format Description Shared Image Flag values	65
Table 36 — Texture Vers-2 Texture Type values	67
Table 37 — Texture Vers-2 Inline Image Storage Flag values	68
Table 38 — Texture Vers-3 Texture Type values	70
Table 39 — Texture Vers-3 Inline Image Storage Flag values	71
Table 40 — Draw Style Attribute Field Inhibit flag values	71
Table 41 — Draw Style Attribute Data Flag values	72
Table 42 — Light Set Attribute Version Number values	75
Table 43 — Base Light Data Cood System values	76
Table 44 — Base Light Data Shadow Caster Flag values	76
Table 45 — Point Light Attribute Version Number values	79
Table 46 — Point Light Attribute Spread Angle values	79
Table 47 — Linestyle Attribute Data Flag values	81
Table 48 — Pointstyle Attribute Data Flag values	82
Table 49 — Geometric Transform Attribute Stored Value Mask individual bit-flag values	84
Table 50 — Shader Effects Attribute Enable Flag values	85
Table 51 — Shader Effects Attribute Phong Shading Flag values	86
Table 52 — Mapping Plane Matrix Coordinate System values	88
Table 53 — Mapping Cylinder Matrix Coordinate System values	89
Table 54 — Mapping Sphere Matrix Coordinate System values	91
Table 55 — Mapping TriPlanar Matrix Coordinate System values	92
Table 56 — Vertex Shape LOD Bindings values	104
Table 57 — TopoMesh Compressed Rep Data V2 Field Type values	116
Table 58 — Primitive Set Shape Texture Coord Binding values	121
Table 59 — Primitive Set Shape Version Number values	122
Table 60 — Primitive Set Shape Texture Coord Gen Type values	122
Table 61 — Lossless Compressed Primitive Set Data Field values	123
Table 62 — Primitive Set “params#” Data Fields Interpretation	124
Table 63 — Property Proxy Meta Data Property Value Type values	133
Table 64 — PMI Manager Meta Data Version Number values	136
Table 65 — PMI 2D Base Data Font values	141

Table 66 — PMI 2D Non-Text Polyline Type values	146
Table 67 — PMI 3D Data Polyline Dimensionality values	151
Table 68 — PMI Reference Geometry Entity values	154
Table 69 — PMI Design Group Attribute Type values.....	156
Table 70 — PMI Associations Source Data values	158
Table 71 — PMI Associations Reason Code values	159
Table 72 — PMI Model Views Active Flag values	163
Table 73 — Generic PMI Entity Type values	165
Table 74 — Generic PMI User Flag values	166
Table 75 — Common Property Keys and Their Value Encoding formats.....	167
Table 76 — PMI Property Atom Hidden Flag values	168
Table 77 — Common Compression CODEC Type values	176
Table 78 — Int32 Probability Contexts CODEC Type values	182
Table 79 — Float64 Compressed Data Packet CODEC Type values.....	186
Table 80 — Colour Quantizer values	198
Table 81 — Colour Quantizer HSV Flag values	199
Table 82 — Knot Type Exist Flag values.....	202
Table 83 — Compressed Curve Base Type values	206
Table 84 — NURB UV Curve entity dimensionality values.....	206
Table 85 — NURB XYZ Curve entity dimensionality values.....	206
Table 86 — Compressed CAD Tag Type values.....	210
Table 87 — Example assigned probability values	214
Table 88 — Example “probability line” values	214
Table 89 — Example input integer sequence values	214
Table 90 — Example integer number sequence values	215
Table 91 — CAD Property Conventions	224
Table 92 — CAD Optional Property Units	228
Table 93 — PMI Properties	229
Table 94 — Tessellation Property values.....	324
Table 95 — Miscellaneous Property values	325
Table 96 — Texture watermark properties.....	330
Table 97 — Object Type Identifiers.....	334
Table 98 — Object Nodes	479
Table 99 — Field types in order one by one	480
Table 100 — Entity Matrix relations.....	487
Table 101 — Curve node common fields	490
Table 102 — Line Fields.....	491
Table 103 — Circle fields	492
Table 104 — Ellipse fields.....	493
Table 105 — NURB curve fields	496
Table 106 — Curve intersection fields	499
Table 107 — Trimmed curve fields	502
Table 108 — SP curve fields.....	503
Table 109 — Surface node fields	504
Table 110 — Plane fields.....	504
Table 111 — Cylinder fields.....	505
Table 112 — Cone fields	506
Table 113 — Sphere fields	507
Table 114 — Torus fields	508
Table 115 — Blended edge fields	509
Table 116 — Blend boundary surface fields.....	511
Table 117 — Offset surface fields	511
Table 118 — B-Surface fields	512
Table 119 — NURB Surface fields	513
Table 120 — Swept surface fields.....	517
Table 121 — Spun surface fields	518
Table 122 — Point fields	519
Table 123 — Transform fields	519
Table 124 — Transform action fields.....	520

Table 125 — Geometry owner fields.....	521
Table 126 — World topology fields.....	521
Table 127 — Assembly fields	522
Table 128 — Instance fields.....	524
Table 129 — Body fields	525
Table 130 — Geometry to Topology attachment.....	528
Table 131 — Region fields	528
Table 132 — Shell fields.....	529
Table 133 — Face fields	529
Table 134 — Loop fields	530
Table 135 — Fin fields.....	531
Table 136 — Vertex fields	532
Table 137 — Edge fields	532
Table 138 — Associated List.....	534
Table 139 — Pointer List Block.....	535
Table 140 — Attribute Definition ID	535
Table 141 — Field Names	536
Table 142 — Attribute definition	536
Table 143 — Attribute definition action fields	537
Table 144 — Corresponding attribute classes	537
Table 145 — Attribute fields	539
Table 146 — Integer values	541
Table 147 — Real values.....	541
Table 148 — Character values.....	541
Table 149 — Unicode values	541
Table 150 — Point values	542
Table 151 — Vector values	542
Table 152 — Direction values.....	542
Table 153 — Axis values.....	542
Table 154 — Tag values	543
Table 155 — Group fields	543
Table 156 — Group member fields	544
Table 157 — Node types	545
Table 158 — Node classes.....	547
Table 159 — Hatching	547
Table 160 — Planar Hatch.....	548
Table 161 — Radial Hatch.....	548
Table 162 — Parametric Hatch.....	548
Table 163 — Body Density.....	549
Table 164 — Region Density	549
Table 165 — Face Density	550
Table 166 — Edge Density	550
Table 167 — Vertex Density.....	550
Table 168 — Region	551
Table 169 — Colour	551
Table 170 — Reflectivity	551
Table 171 — Translucency	552
Table 172 — Name.....	552
Table 173 — Incremental faceting	552
Table 174 — Transparency	552
Table 175 — Non-mergable edges	553
Table 176 — Group merge behaviour.....	553
Table 177 — Unicode name	553
Table 178 — Moniker attribute use example.....	555
Table 179 — MONIKER/GUID_TABLE_ATTRIB and MONIKER/MONIKER_DATA_ATTRIB	557
Table 180 — MONIKER/BODY_ID_ATTRIB	558
Table 181 — JT B-Rep Shell Topology Anti-Hole Flag values	573
Table 182 — JT B-Rep Face Reverse Normal Flag values	576
Table 183 — JT B-Rep Loops Topology Data Anti-Hole Flag values	577

Table 184 — JT B-Rep MCS Curve Reversed Flag values	578
Table 185 — JT B-Rep Surface Base Type value	582
Table 186 — JT B-Rep NURBS Surface Control Point Dimensionality values.....	582
Table 187 — Trivial Domain Loops Exist Flag values	587
Table 188 — Trivial Box Loops Exist Flag values	587
Table 189 — Trivial Domain UV Curves Exist Flag values	588
Table 190 — Trivial UV Curve Para Domain Side Codes values.....	588
Table 191 — JT ULP Shell Anti-Hole Flag values.....	602
Table 192 — JT ULP Flag Bit Array Look Index values	604
Table 193 — JT ULP Supported Surface Type values	605
Table 194 — JT ULP Supported Knot Type Values.....	605
Table 195 — JT ULP Face Reverse Normal Flag values.....	605
Table 196 — JT ULP Loops Topology Flag Bit Array values	606
Table 197 — JT ULP Loops Topology Reverse Normal Flag values.....	607
Table 198 — JT ULP Recover Edge Indices Flag Bit Array values.....	609
Table 199 — JT ULP Recover Edge Indices PCS curve type values.....	610
Table 200 — JT ULP PCS Curve Type values.....	610
Table 201 — JT ULP PCS Curve Type XYZ Reversed Flag values	611
Table 202 — JT ULP PCS Curve Type isUVInc Flag values	611
Table 203 — JT ULP Edges Topology Recover MCS Curve Indices Flag Bit Array values	613
Table 204 — JT ULP Edges Topology Recover MCS Curve Type values.....	613
Table 205 — Parameter Domain.....	631

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

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

This second edition cancels and replaces the first edition (ISO 14306:2012), which has been technically revised.

The main changes compared to the previous edition are as follows:

- addition of a STEP B-Rep Segment based on the ISO 10303 STEP suite of standards;
- addition of a new geometry chapter listing all geometry segments in the annexes;
- updates of the documentation from the ProSTEP iViP industrialization guidelines;
- addition of a conformance clause;
- addition of a mapping table JT ISO to STEP;
- deprecation of the JT Brep segment;
- deprecation of the PMI data segment which is superseded by the meta data segment.

Introduction

The ISO 14306 format is an industry focused, high-performance, lightweight, flexible file format for capturing and repurposing 3D product definition data for visualization to enable collaboration and validation throughout the extended enterprise. The ISO 14306 format is streamable and contains compression for compact and efficient representation.

Some of the highlights of the ISO 14306 format include:

- built-in support for assemblies, sub-assemblies and part constructs;
- a flexible partitioning scheme, supporting single or multiple files;
- b-rep solid shape representations to provide precision to the light-weight viewing processes;
- product manufacturing information in support of paperless manufacturing initiatives;
- precise and imprecise wireframe shape representations;
- discrete purpose-built levels of detail;
- triangle sets, polygon sets, point sets, line sets and implicit primitive sets (such as cylinder, cone and sphere);
- a full array of visual attributes such as for materials, textures, lights;
- hierarchical bounding box and bounding spheres;
- 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 ISO 14306 format, the overall physical structure/organization of the format is also designed to support operations such as:

- offline optimizations of the data contents, i.e. file granularity and flexibility optimized to meet the needs of enterprise data translation solutions;
- asynchronous streaming of content, i.e. viewing optimizations such as view frustum and occlusion culling and fixed-framerate display modes;
- layers, and layer filters.

NOTE This document is based on the JT Open version 9.5 specification, which defines a scene graph with geometry specific node and attribute support, extended to support ISO 10303.

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 patents USA 20110199382, 8019788.

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.

STANDARDSISO.COM : Click to view the full PDF of ISO 14306:2017

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

1 Scope

This document defines the syntax and semantics of a file format for the 3D visualization and interrogation of lightweight geometry and product manufacturing information derived from CAD systems, using visualization software tools that do not need the full capability of a CAD system.

This document has been adopted as a 3D visualization capability in addition to the ISO 10303 series.

The ISO 10303 series are the ISO standards adopted for the engineering data exchange, sharing and long term archiving of product definition information throughout the product lifecycle.

In this document 3D visualization is defined as the visual presentation on a screen or another media of graphical and textual 3 dimensional representations of a set of data representing an object, information or results of a computational process in order to enable decision process by a human looking at the data visualized in a medium.

The file format supports the following information:

- facet information (triangles), stored with geometry compression techniques
- visual attributes such as lights, textures and materials
- product manufacturing information, such as dimensions, tolerances and other attributes
- boundary representation (b-rep) solid model shape representations. Several alternatives are available, including a representation based on the geometry standard defined in ISO 10303
- configuration representations
- delivery methods such as asynchronous streaming of content

The document does not specify the implementation of, or definition of a run-time architecture for viewing or processing of the file format.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 10303-21:2014, *Industrial automation systems and integration — Product data representation and exchange — Part 21: Implementation methods: Clear text encoding of the exchange structure*

ISO 10303-42:2014, *Industrial automation systems and integration – Product data representation and exchange – Part 42: Integrated generic resource: geometric and topological representation*

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

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

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <http://www.electropedia.org/>

3.1.1

3D visualization

visual presentation on a screen or another media of graphical and textual three dimensional representations of a set of data representing an object, information or results of a computational process in order to facilitate capture of the understanding of the object, for visual information sharing with users and sometimes to promote decision process by a human looking at the data visualized in a medium

3.1.2

assembly

related collection of model parts, represented in a ISO 14306 format logical scene graph as a logical graph branch

3.1.3

attribute

objects associated with nodes in a logical scene graph and specifying one of several appearances, positioning, or rendering characteristics of a shape

3.1.4

code text

collection of data in encoded form

3.1.5

coordinate system

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

Note 1 to entry: If not otherwise specified in a data field's description, it is assumed that the data is defined in Local Coordinate System.

3.1.6

degenerated surface

place where a region of the parameter space of the surface is mapped to a single point of Cartesian space

Note 1 to entry: This means that there are several (u,v) parameters defining the same 3D point of the surface. In this region, one or both first partial derivatives are zero. The tip of a cone or the poles of a sphere are common examples.

3.1.7

directed acyclic graph

graph that consists of a set of nodes and a set of edges that connect the nodes in a tree like structure

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

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

Note 3 to entry: 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.

3.1.8

JT enabled application

application which supports reading and/or writing reference compliant ISO 14306 format files

3.1.9

level of detail

LOD

alternative graphical representation for some model component such as part

3.1.10**local coordinate system****LCS**

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

3.1.11**logical scene graph****LSG**

scene graph representing the logical organization of a model

Note 1 to entry: A scene graph contains shapes and attributes representing the model's physical components, properties identifying arbitrary metadata (e.g. names, semantic roles) of those components, and a hierarchical structure expressing the component relationships.

3.1.12**mipmap**

reduced resolution version of a texture map

Note 1 to entry: 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.

3.1.13**model**

representation, in ISO 14306 format, of a physical or virtual product, part, assembly; or collections of such objects

3.1.14**model coordinate system****MCS**

local coordinates transformed by any transforms specified as attributes at or above the node

3.1.15**product and manufacturing information****PMI**

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

Note 1 to entry: This can include data such as:

- dimensions (tolerances for each dimension);
- geometric tolerances of feature (datums, feature control frames);
- manufacturing information (surface finish, welding notations);
- inspection information (key locations points);
- assembly instructions;
- product information (materials, suppliers, part numbers).

3.1.16**property**

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

3.1.17

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

3.1.18

scene graph

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

3.1.19

shader

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

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

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

3.1.20

streaming

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

Note 1 to entry: The motivation for streaming is to more efficiently manage system memory.

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

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

Note 4 to entry: The desired end result of streaming is to deliver only the ISO 14306 data that the user needs, where the user needs it, when the user needs it.

3.1.21

shape

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

3.1.22

texture channel

texture unit plus the texture environment

Note 1 to entry: The ISO 14306 format meaning for texture channel is the same as in OpenGL [1].

3.1.23

texture object

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

Note 1 to entry: The ISO 14306 format meaning for texture object is the same as in OpenGL [1].

3.1.24

texture unit

piece of hardware that takes a sample of a texture

Note 1 to entry: The ISO 14306 format meaning for texture unit is the same as in OpenGL [1].

3.1.25

view coordinate system

world coordinates transformed by a view matrix

3.1.26

world coordinate system

WCS

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

3.2 Abbreviated terms

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

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

4 Notational conventions

Diagrams and field descriptions

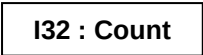
Symbolic diagrams are used to describe the structure of the ISO 14306 file. The symbols used in these diagrams have the following meaning:

Table 1 — Symbols

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

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

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



I32 : Count

Figure 1 — rectangle box diagram

Diagram “folder” (i.e. logical data collections) symbols are simply titled with a collection name. In the example

below the collection name is “Graph Elements.”



Graph Elements

Figure 2 — folder diagram

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



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

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

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

Figure 4 — rectangle box with clipped right side corners

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

The two supported compressed data packet types are:

Int32CDP – The Int32CDP (i.e. Int32 Compressed Data Packet) represents 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 [10.1.1 Int32 Compressed Data Packet](#).

Int32CDP2 – The Int32CDP2 (i.e. Int32 Compressed Data Packet Mk. 2) represents a second-generation version of the above compressed data packet, and sports a simplified and more compact file layout, and the ability to more efficiently encode clustered data and bitfields. A complete description for Int32 Compressed Data Packet Mk. 2 can be found in [10.1.2 Int32 Compressed Data Packet Mk. 2](#).

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 [10.1.3 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.”

Figure 5 — compressed data packet diagram

As mentioned above (with Predictor Type algorithm), the *primal* input data values are NOT always what is encoded/compressed. This is because the *primal* input data is first run through a Predictor Type

algorithm, which produces an output array of residual values (i.e. difference from the predicted value), and this resulting output array of *residual* values is the data which is actually encoded/compressed. The ISO 14306 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 ISO 14306 format supported Predictor Type algorithms are as follows (note that a sample implementation of decoding the predictor *residual* values back into the primal values can be found in Annex C).

Table 2 — Predictor Type

Predictor Type	Description
Lag1	Predicts as last value
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 <i>noninclusive</i> , else it predicts the value as the one two back plus two. In pseudo-code form the predicted values is computed as follows: <pre>if(val2back - val4back < 8 && val2back - val4back > -8) iPredicted = val2back + (val2back - val4back); else iPredicted = val2back + 2;</pre>
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

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

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

VecU32 : CodeText

Figure 6 — data type : field name diagram

Note that for some JT file [Segment Types](#) there is 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 shall 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 ISO 14306 format data compression and encoding can be found in 5.1.3 Data Segment and 10 Data Compression and Encoding.

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

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

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

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

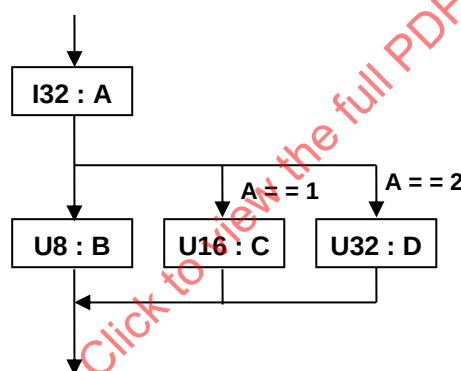


Figure 7 — data filed dependency example

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

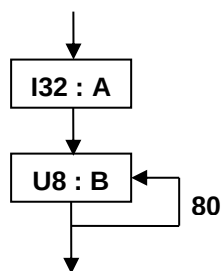


Figure 8 — loop construct example

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

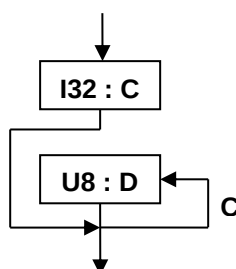


Figure 9 — loop construct with iterations example

Data Types

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

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

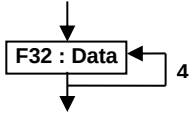
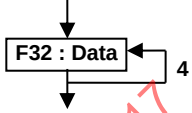
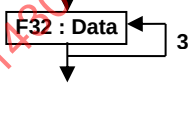
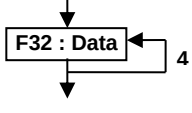
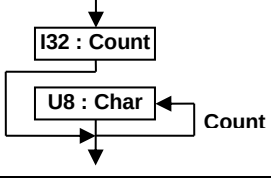
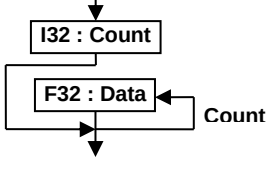
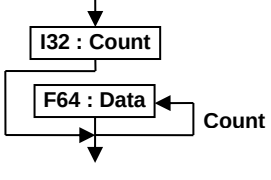
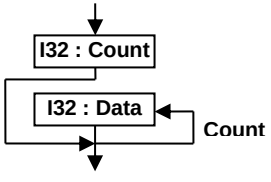
Table 3 — Basic Data Types

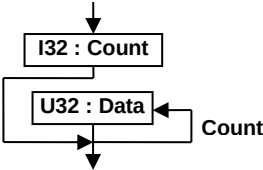
Type	Description
UChar	An unsigned 8-bit byte.
U8	An unsigned 8-bit integer value.
U16	An unsigned 16-bit integer value.
U32	An unsigned 32-bit integer value.
U64	An unsigned 64-bit integer value.
I16	A signed two's complement 16-bit integer value.
I32	A signed two's complement 32-bit integer value.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

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

Table 4 — Composite Data Types

Type	Description	Symbolic Diagram
BBoxF32	The BBoxF32 type defines a bounding box using two CoordF32 types to store the XYZ coordinates for the bounding box minimum and maximum corner points.	
CoordF32	The CoordF32 type defines X, Y, Z coordinate values. So a CoordF32 is made up of three F32 base types.	
CoordF64	The CoordF64 type defines X, Y, Z coordinate values. So a CoordF64 is made up of three F64 base types.	
DirF32	The DirF32 type defines X, Y, Z components of a direction vector. So a DirF32 is made up of three F32 base types.	
GUID	The GUID type is a 16 byte (128-bit) number. GUID is stored/written to the JT file using a four-byte word (U32), 2 two-byte words (U16), and 8 one-byte words (U8) such as: {3F2504E0-4F89-11D3-9A-0C-03-05-E8-2C-33-01} In the ISO 14306 format GUIDs are used as unique identifiers (e.g. Data Segment ID, Object Type ID, etc.)	
HCoordF32	The HCoordF32 type defines X, Y, Z, W homogeneous coordinate values. So an HCoordF32 is made up of four F32 base types.	
HCoordF64	The HCoordF64 type defines X, Y, Z, W homogeneous coordinate values. So an HCoordF64 is made up of four F64 base types	
MbString	The MbString type starts with an I32 that defines the number of characters (NumChar) the string contains. The number of bytes of character data is "2 * NumChar" (i.e. the strings are written out as multi-byte characters where each character is U16 size).	
Mx4F32	Defines a 4-by-4 matrix of F32 values for a total of 16 F32 values. The values are stored in row major order (right most subscript, column varies fastest), that is, the first 4 elements form the first row of the matrix.	

PlaneF32	The PlaneF32 type defines a geometric Plane using the General Form of the plane equation ($Ax + By + Cz + D = 0$). The PlaneF32 type is made up of four F32 base types where the first three F32 define the plane unit normal vector (A, B, C) and the last F32 defines the negated perpendicular distance (D), along normal vector, from the origin to the plane.	
Quaternion	The Quaternion type defines a 3-dimensional orientation (no translation) in quaternion linear combination form ($a + bi + cj + dk$) 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.	
RGB	The RGB type defines a colour composed of Red, Green, Blue components, each of which is a F32. So a RGB type is made up of three F32 base types. The Red, Green, Blue colour values typically range from 0.0 to 1.0.	
RGBA	The RGBA type defines a colour composed of Red, Green, Blue, Alpha components, each of which is a F32. So a RGBA type is made up of four F32 base types. The Red, Green, Blue colour values typically range from 0.0 to 1.0. The Alpha value ranges from 0.0 to 1.0 where 1.0 indicates completely opaque.	
String	The String type starts with an I32 that defines the number of characters (NumChar) the string contains. The number of bytes of character data is "NumChar" (i.e. the strings are written out as single-byte characters where each character is U8 size).	
VecF32	The VecF32 type defines a vector/array of F32 base type. The type starts with an I32 that defines the count of following F32 base type data. So a VecF32 is made up of one I32 followed by that number of F32. Note that it is valid for the I32 count number to be equal to "0", indicating no following F32.	
VecF64	The VecF64 type defines a vector/array of F64 base type. The type starts with an I32 that defines the count of following F64 base type data. So a VecF64 is made up of one I32 followed by that number of F64. Note that it is valid for the I32 count number to be equal to "0", indicating no following F64.	
VecI32	The VecI32 type defines a vector/array of I32 base type. The type starts with an I32 that defines the count of following I32 base type data. So a VecI32 is made up of one I32 followed by that number of I32. Note that it is valid for the I32 count number to be equal to "0", indicating no following I32.	

VecU32	The VecU32 type defines a vector/array of U32 base type. The type starts with an I32 that defines the count of following U32 base type data. So a VecU32 is made up of one I32 followed by that number of U32. Note that it is valid for the I32 count number to be equal to “0”, indicating no following U32.	

Empty field

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

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

5 File Format

All objects represented in the ISO 14306 format are assigned an “object identifier” (e.g. see [6.1.1.1.1 Base Node Data](#), or [6.1.2.1.1 Base Attribute Data](#)) and all references from one object to another object are represented in the ISO 14306 format using the referenced object’s “object identifier”. It is the responsibility of ISO 14306 format readers/writers to maintain the integrity of these object references by doing appropriate pointer unswizzling/swizzling as ISO 14306 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).

All objects stored in a JT file are classified by type and thus include an object type identifier as part of their persisted data.

Object Type Identifiers specifications are defined in Annex A.

5.1 File Structure

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

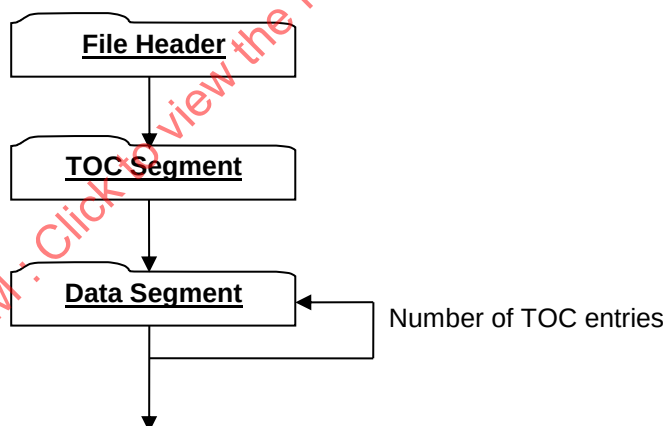


Figure 10 — JT File Structure

5.1.1 File Header

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

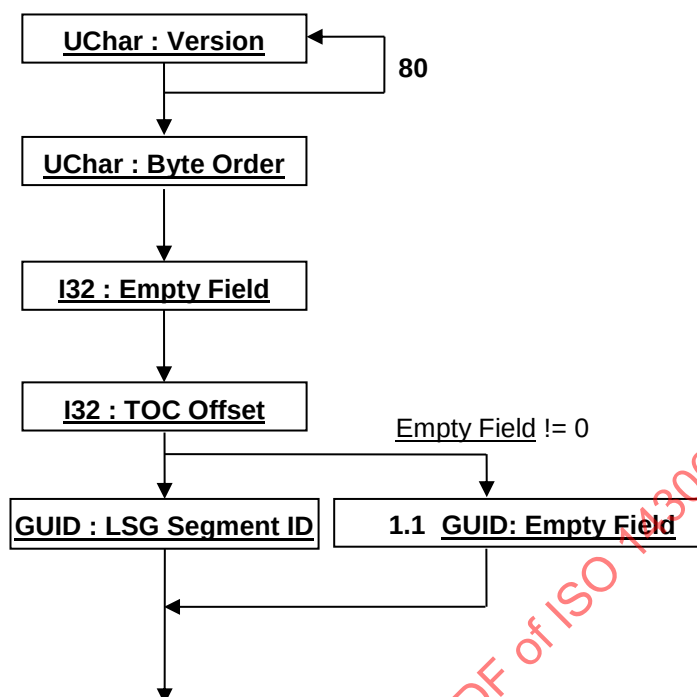


Figure 11 — File Header data collection

UChar : Version

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

Version *M*.*n* *Comment*

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

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

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

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

As an example, for a JT Version 9.5 file written by a Direct Model (DM) library version 7.3.4.0 this string will look as follows:

"Version 9.5 JT DM 7.3.4.0

\n\r\n"

UChar : Byte Order

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

0 – Least Significant byte first (LsbFirst)

1 – Most Significant byte first (MsbFirst)

I32 : Empty Field

Refer to best practice [11.4](#) Empty Field.

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 shall be accessible from the root LSG Data Segment.

GUID: Empty Field

Refer to best practice [11.4](#) Empty Field.

5.1.2 TOC Segment

The TOC Segment contains information identifying and locating all individually addressable Data Segments within the file. A TOC Segment is always required to exist somewhere within a JT file. The actual location of the TOC Segment within the file is specified by the File Header segment's "TOC Offset" field. The TOC Segment contains one TOC Entry for each individually addressable Data Segment in the file.

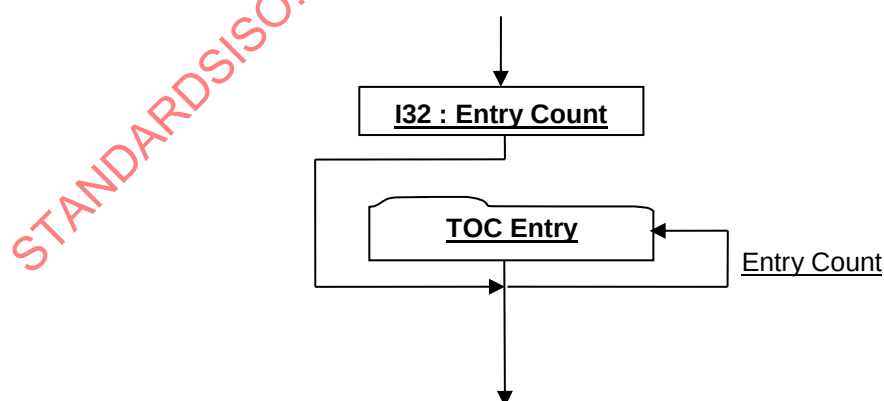


Figure 12 — TOC Segment data collection

I32 : Entry Count

Entry Count is the number of entries in the TOC.

TOC Entry

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

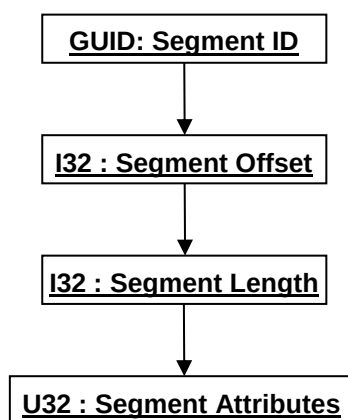


Figure 13 — TOC Entry data collection

GUID : Segment ID

Segment ID is the globally unique identifier for the segment.

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.

Table 5 — Segment attributes

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

5.1.3 Data Segment

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

Beyond specific data field compression/encoding, some Data Segment types also have 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 6 — 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.

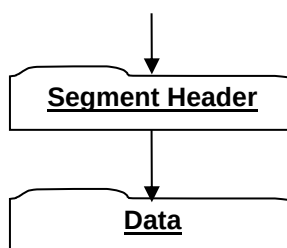


Figure 14 — Data Segment data collection

5.1.3.1 Segment Header

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

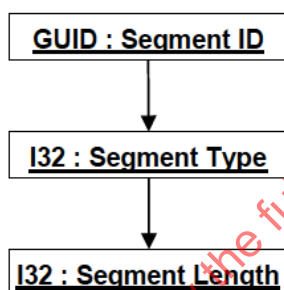


Figure 15 — Segment Header data collection

GUID : Segment ID

Global Unique Identifier for the segment.

I32 : Segment Type

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

The complete list of segment types is as follows. The column labelled “ZLIB Applied?” denotes whether ZLIB compression is conditionally applied to the entirety of the segment's Data payload. The column Mandatory status specifies whether the segment shall be present in the file according to the conformance clause.

Table 6 — Segment Types

Type	Data Contents	ZLIB Applied?	Mandatory status
1	Logical Scene Graph	Yes	Mandatory
2	B-Rep description in JT	Yes	Deprecated
3	PMI Data	Yes	Deprecated
4	Meta Data and PMI	Yes	Optional
6	Shape	No	Mandatory
7	Shape LOD0	No	Optional
8	Shape LOD1	No	Optional
9	Shape LOD2	No	Optional

Type	Data Contents	ZLIB Applied?	Mandatory status
10	Shape LOD3	No	Optional
11	Shape LOD4	No	Optional
12	Shape LOD5	No	Optional
13	Shape LOD6	No	Optional
14	Shape LOD7	No	Optional
15	Shape LOD8	No	Optional
16	Shape LOD9	No	Optional
17	XT B-Rep	Yes	Optional
18	Wireframe Representation	Yes	Optional
20	ULP	Yes	Optional
24	LWPA	Yes	Optional
32	STEP B-rep	Yes	Optional

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

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

- not a descendant of an LOD node;
- is referenced by (i.e. is a child of) more than one LOD node;
- Shape has its own built-in LODs;
- no way to determine what LOD a Shape Segment represents.

132 : Segment Length

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

5.1.3.2 Data

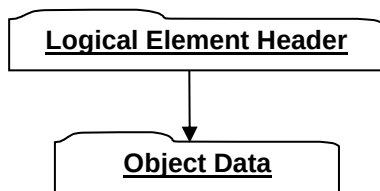
The interpretation of the Data section depends on the Segment Type. See 5.2 Data Segments for complete description for all Data Segment that may be contained in a JT file.

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 Table 8 — 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 Logical 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 [10 Data Compression and Encoding](#).

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



For Segment Types that support ZLIB compression on all Segment Data
(see [Table 6 — Segment Types](#).)

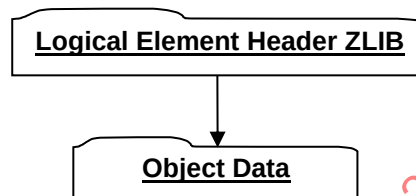


Figure 16 — Data collection

Logical Element Header

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

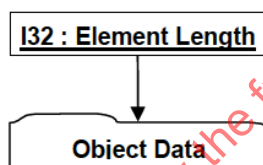


Figure 17 — Logical Element Header data collection

Complete description for Logical Element Header can be found in the [Element Header](#).

I32 : Element Length

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

Element Header

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

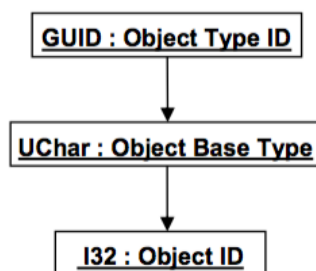


Figure 18 — Element Header data collection

GUID : Object Type ID

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

UChar : Object Base Type

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

Valid Object Base Types include the following:

Table 7 — Object Base Types

Base Type	Description	Base Type's Data Format
255	None	None
0	Base Graph Node Object	<u>6.1.1.1.1 Base Node Data</u>
1	Group Graph Node Object	<u>6.1.1.3.1 Group Node Data</u>
2	Shape Graph Node Object	<u>0 Base Shape Data</u>
3	Base Attribute Object	<u>6.1.2.1.1 Base Attribute Data</u>
4	Shape LOD	None
5	Base Property Object	<u>0 Base Property Atom Data</u>
6	JT Object Reference Object	<u>6.2.5 JT Object Reference Property Atom Element</u> without the Logical Element Header ZLIB data collection.
8	JT Late Loaded Property Object	<u>6.2.7 Late Loaded Property Atom Element</u> without the Logical Element Header ZLIB data collection.
9	JtBase (none)	None

I32 : Object ID

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

Logical Element Header ZLIB

Logical 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 8 — Segment Type for information on whether a particular Segment Type supports ZLIB compression on all data in the Segment.

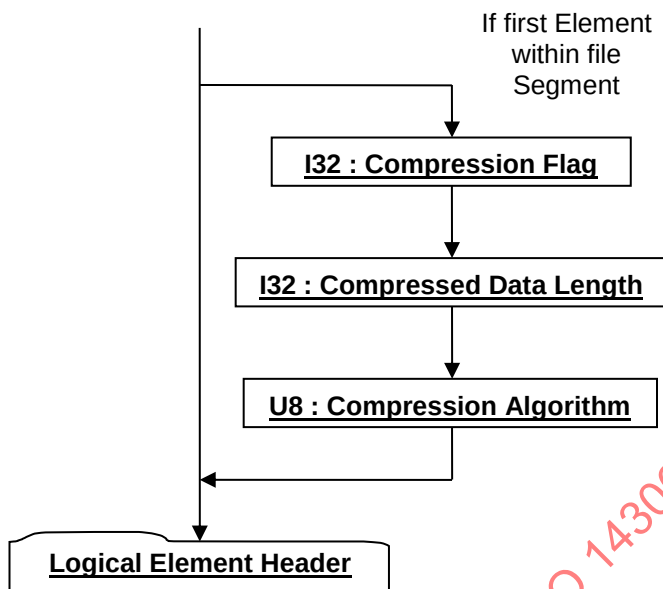


Figure 19 — Logical Element Header ZLIB data collection

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

I32 : 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:

Table 8 — Compression flag values

= 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:

Table 9 — Compression algorithm values

= 1	No compression
= 2	ZLIB compression

Object Data

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

5.2 Data Segments

A JT file consists of the following mandatory segments of data:

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

In addition of these mandatory segments a set of optional segments for geometry representation, PMI and meta data are available:

For geometry representation:

- STEP B-Rep segment contains an element that defines the precise geometric boundary representation data for a particular part in normative STEP Part 42 (ISO 10303-42) boundary representation format as a part 21 compressed file (ISO 10303-21). See ANNEX B.
- JT B-Rep Segment contains an element that defines the precise geometric boundary representation data for a particular part in boundary representation format. See ANNEX K.
- XT B-Rep Segment contains an element that defines the precise geometric boundary representation data for a particular part in boundary representation format. See ANNEX G.
- ULP Segment contains an element that defines the approximate geometric boundary representation data for a particular part in JT ULP format. See ANNEX M.
- LWPA Segment contains an element that defines light weight precise analytic data for a particular part. More specifically LWPA contains the collection of analytic surfaces in the b-rep definition of the part. See ANNEX M.
- WireFrame Segment contains an element that defines the precise 3D wireframe data for a particular part. See ANNEX L.

For PMI and metadata:

- Meta Data and PMI Segment is used to store large collections of meta-data in separate addressable segments of the JT File. Storing meta-data in a separate addressable segment allows references (from within the JT file) to these segments to be constructed such that the meta-data can be late-loaded.
- In pre JT 8 version, a deprecated PMI Segment was used to store PMI manager meta data element. From a parsing point of view, a PMI data segment should be treated exactly the same as a meta data and PMI segment. Reference [PMI Data Segments](#) for best practice.

6 LSG Segment

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

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

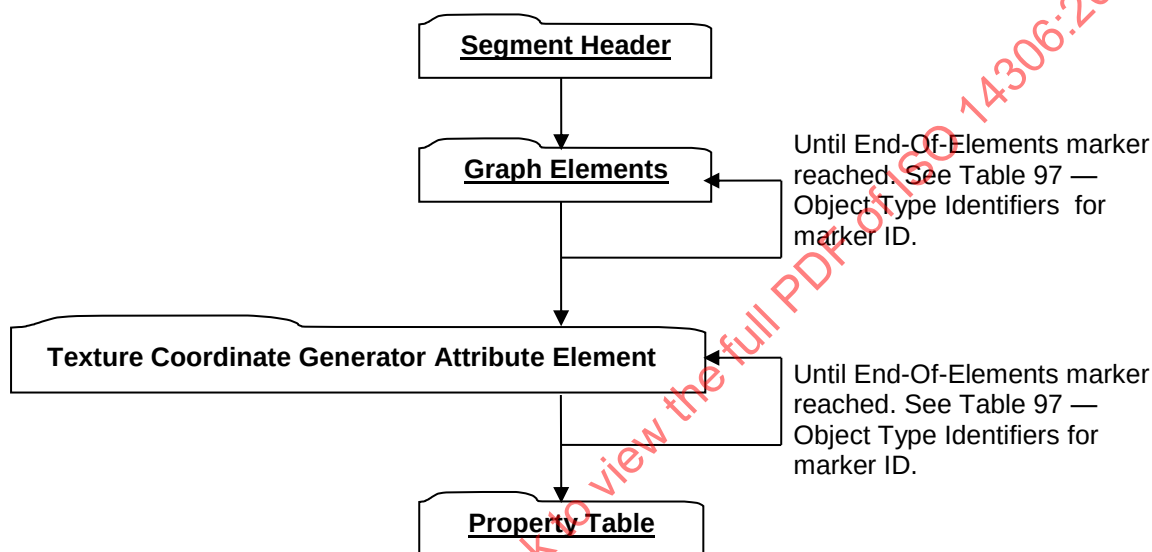


Figure 20 — LSG Segment data collection

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

6.1 Graph Elements

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

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

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

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

6.1.1 Node Elements

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

6.1.1.1 Base Node Element

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

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

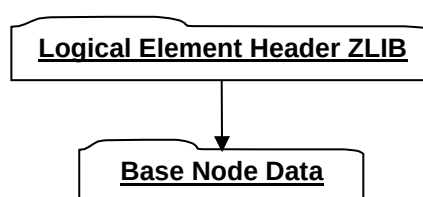


Figure 21 — Base Node Element data collection

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

6.1.1.1.1 Base Node Data

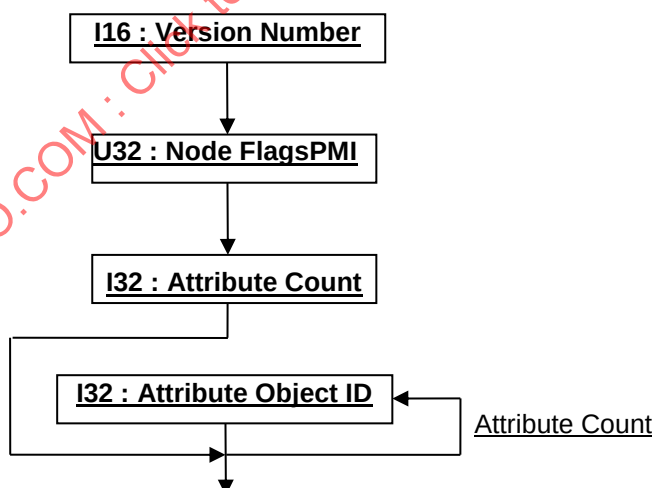


Figure 22 — Base Node Data collection

I16 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice **11.5 Local version numbers**.

U32 : Node FlagsPMI

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

Table 10 — Node Flag values

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

I32 : Attribute Count

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

I32 : Attribute Object ID

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

6.1.1.2 Partition Node Element

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

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

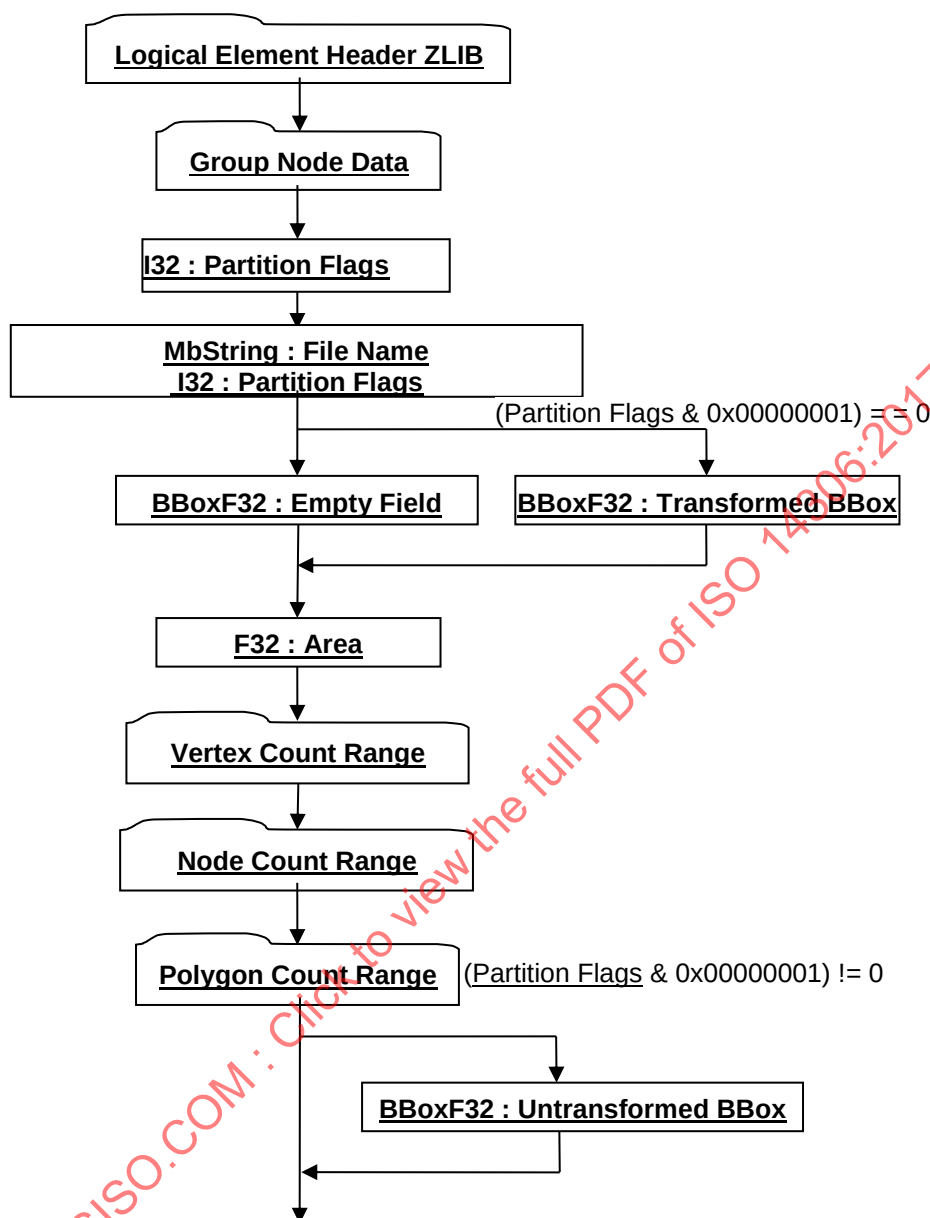


Figure 23 — Partition Node Element data collection

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

Complete description for Group Node Data can be found in **6.1.1.3.1 Group 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 bits fields that are not defined as in use should be set to "0".

Table 11 — Partition flag bits

0x00000001	Untransformed bounding box is written.

MbString : File Name

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

BBoxF32 : Empty Field

Refer to best practice [11.4](#) Empty Field.

BBoxF32 : Transformed BBox

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

F32 : Area

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

BBoxF32 : Untransformed BBox

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

6.1.1.2.1 Vertex Count Range

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

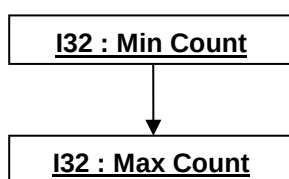


Figure 24 — Vertex Count Range data collection

I32 : Min Count

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

I32 : Max Count

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

6.1.1.2.2 Node Count Range

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

descendants. The maximum value represents the greatest node count that can be achieved by the Partition Node's descendants.

The data format for Node Count Range is the same as that described in [6.1.1.2.1 Vertex Count Range](#).

6.1.1.2.3 Polygon Count Range

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

The data format for Polygon Count Range is the same as that described in [6.1.1.2.1 Vertex Count Range](#).

6.1.1.3 Group Node Element

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

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

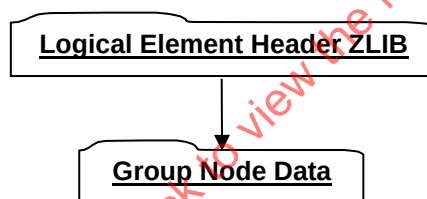


Figure 25 — Group Node Element data collection

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

6.1.1.3.1 Group Node Data

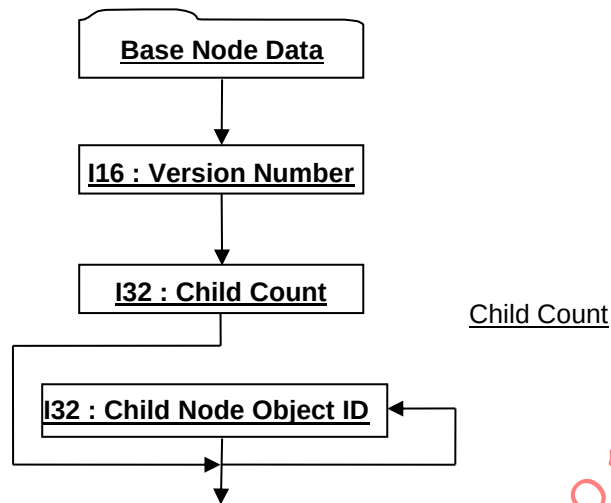


Figure 26 — Group Node Data collection

Complete description for Base Node Data can be found in [6.1.1.1 Base Node Data](#).

I16 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice [11.5 Local version numbers](#).

I32 : Child Count

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

I32 : Child Node Object ID

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

6.1.1.4 Instance Node Element

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

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

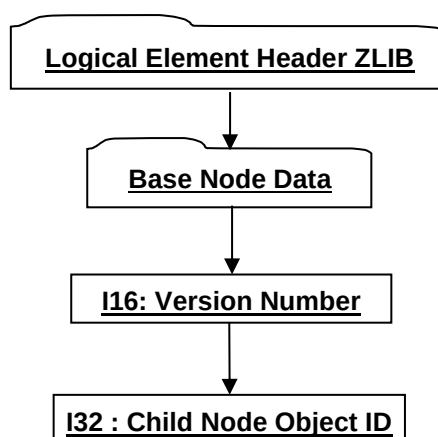


Figure 27 — Instance Node Element data collection

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

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

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 11.5 Local version numbers.

I32 : Child Node Object ID

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

6.1.1.5 Part Node Element

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

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

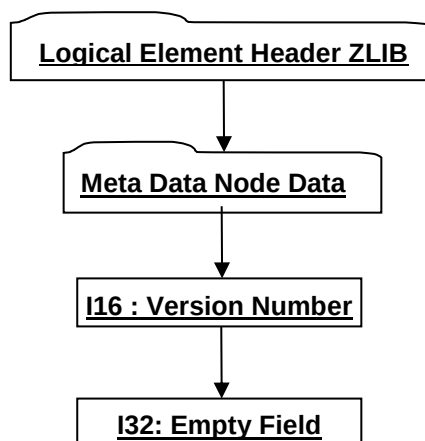


Figure 28 — Part Node Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 11.5 Local version numbers.

I32: Empty Field

Refer to best practice 11.4 Empty Field.

A SUBNODE property can be defined on a part or assembly node in a JT file. By convention, the node with this property is considered to be part of the parent as opposed to being a normal node entity in an assembly. SUBNODE properties can be used to represent a number of CAD constructs in JT files including reference sets (see 11.13.1 for more information about Reference sets).

Assembly Node:

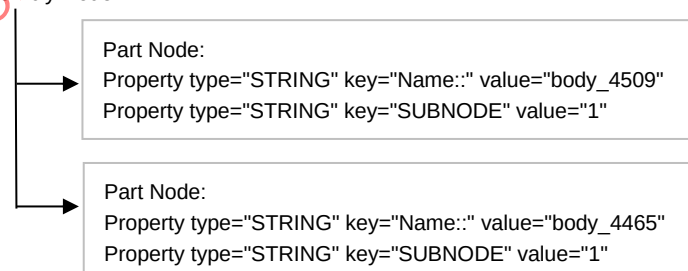


Figure 29 — Assembly node with SUBNODE

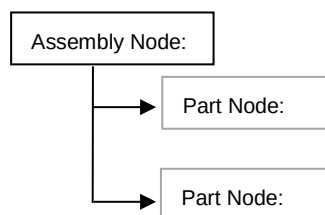


Figure 30 — Assembly node without SUBNODE

The subnode property can be used for viewing applications in order to display a different granularity of the tree.

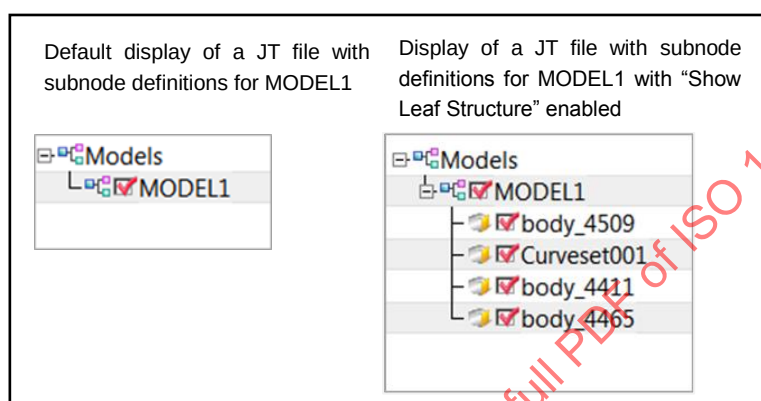


Figure 31 — Displaying Nodes that have SUBNODE properties

The value assigned to the SUBNODE property has no specific meaning. The convention is to set the property to a string value of "1". Parts that contain the SUBNODE property shall contain a string property with a name for the part.

Table 12 — SUBNODE Property

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
SUBNODE	Specifies the node as being a subnode of the parent assembly	String	MbString	0 or 1

6.1.1.6 Meta Data Node Element

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

The Meta Data Node Element is a node type used for storing references to specific "late loaded" meta-data (e.g. properties, PMI). The referenced meta-data is stored in a separate addressable segment of the JT File (see 9 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).

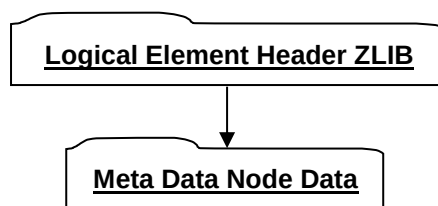


Figure 32 — Meta Data Node Element data collection

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

6.1.1.6.1 Meta Data Node Data

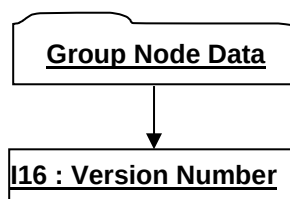


Figure 33 — Meta Data Node Data collection

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

I16 : Version Number

Version Number is the version identifier for this data. For information on local version numbers see best practice 11.5 Local version numbers.

6.1.1.7 LOD Node Element

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

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

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

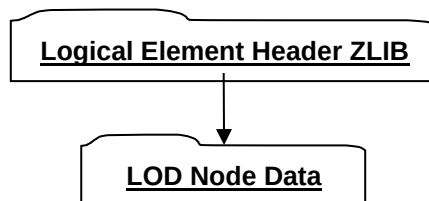


Figure 34 — LOD Node Element data collection

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

6.1.1.7.1 LOD Node Data

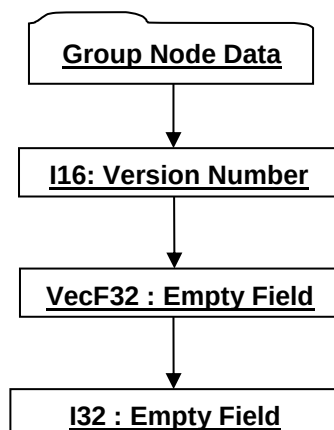


Figure 35 — LOD Node Data collection

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

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 11.5 Local version numbers.

VecF32 : Empty Field

A vector data field. Refer to best practice 11.4 Empty Field.

I32 : Empty Field

Refer to best practice 11.4 Empty Field.

6.1.1.8 Range LOD Node Element

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

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

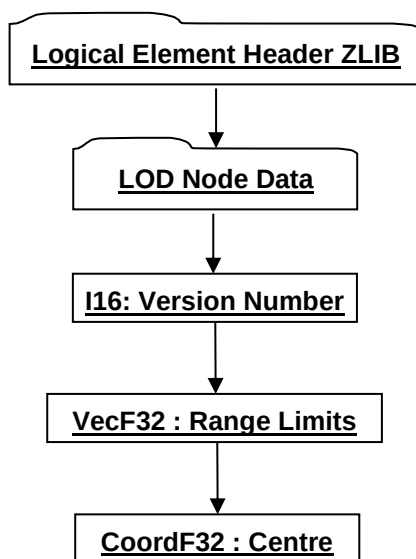


Figure 36 — Range LOD Node Element data collection

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

Complete description for LOD Node Data can be found in [6.1.1.7.1 LOD Node Data](#).

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice [11.5 Local version numbers](#).

VecF32 : Range Limits

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

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

CoordF32 : Centre

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

6.1.1.9 Switch Node Element

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

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

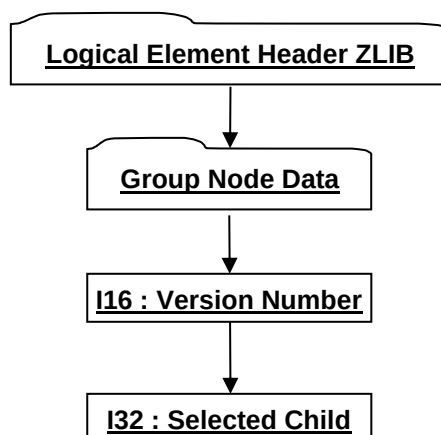


Figure 37 — Switch Node Element data collection

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

Complete description for Group Node Data can be found in [6.1.1.3.1 Group Node Data](#).

I16 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice [11.5 Local version numbers](#).

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 [6.1.1.3.1 Group Node Data](#).

6.1.1.10 Shape Node Elements

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

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

There are several types of Shape Node Elements which the ISO 14306 format supports. The following sub-sections document the various Shape Node Element types.

6.1.1.10.1 Base Shape Node Element

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

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

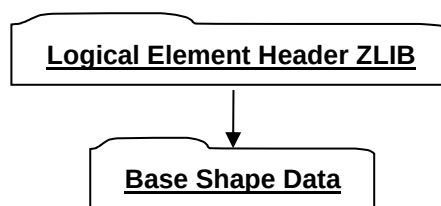


Figure 38 — Base Shape Node Element data collection

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

Base Shape Data

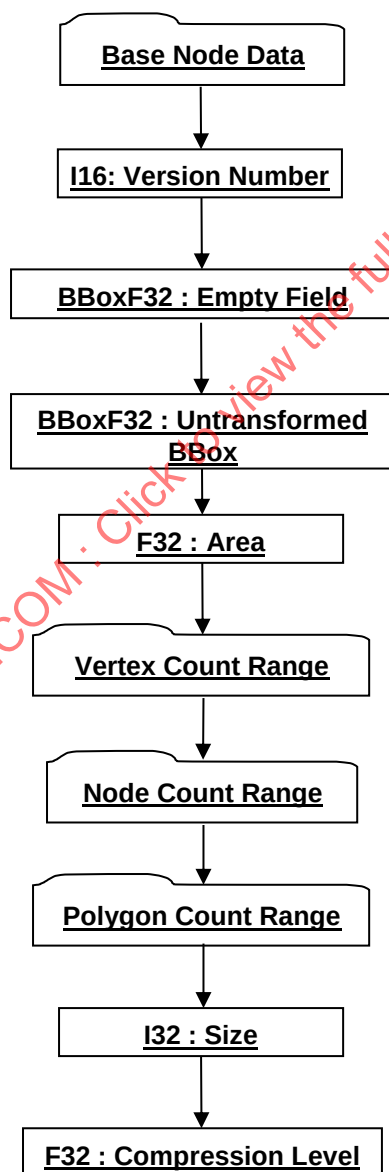


Figure 39 — Base Shape Data collection

Complete description for Base Node Data can be found in [6.1.1.1.1 Base Node Data](#).

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice **11.5 Local version numbers**.

BBoxF32 : Empty Field

Refer to best practice **11.4 Empty Field**.

BBoxF32 : Untransformed BBox

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

F32 : Area

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

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

F32 : Compression Level

Compression Level specifies the qualitative compression level applied to the associated/referenced Shape LOD Element. See **7.1 Shape LOD Element** for complete description of Shape LOD Elements. This compression level value is a qualitative representation of the compression applied to the Shape LOD Element. The absolute compression (derived from this qualitative level) applied to the Shape LOD Element is physically represented in the ISO 14306 format by other data stored with both the Shape Node and the Shape LOD Element (e.g. **Quantization Parameters**), and thus it is not necessary to understand how to map this qualitative value to absolute compression values in order to uncompress/decode the data.

Table 13 — Compression level values

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

Vertex Count Range

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

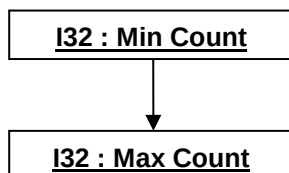


Figure 40 —Vertex Count Range data collection

I32 : Min Count

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

I32 : Max Count

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

Node Count Range

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

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

Polygon Count Range

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

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

6.1.1.10.2 Vertex Shape Node Element

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

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

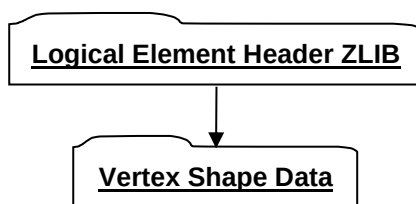


Figure 41 — Vertex Shape Node Element data collection

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

Vertex Shape Data

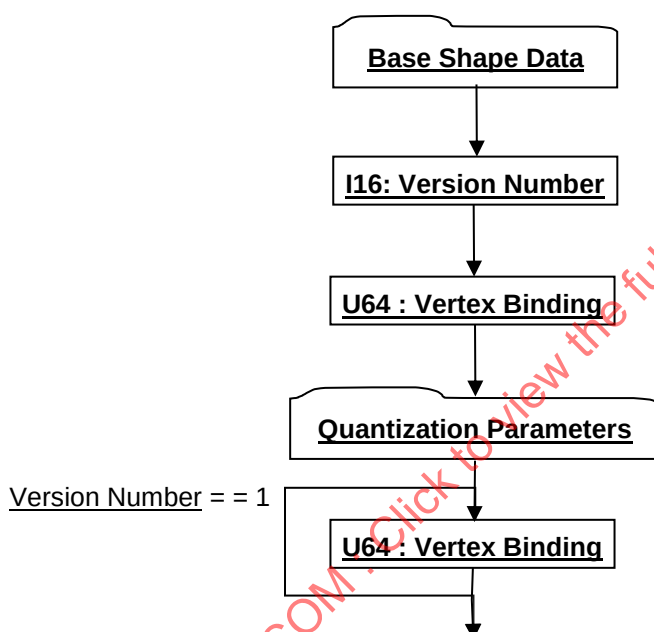


Figure 42 — Vertex Shape Data collection

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

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 11.5 Local version numbers.

U64 : Vertex Binding

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

Quantization Parameters

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

Element in order to determine the Shape quantization level. See 7.1 Shape LOD Element for complete description of Shape LOD Elements.

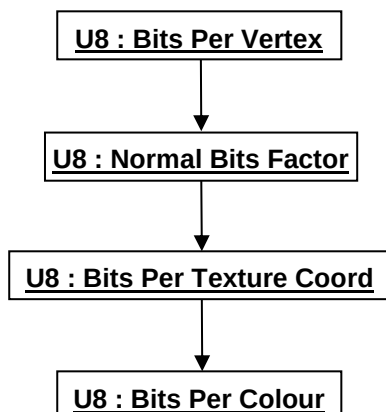


Figure 43 — Quantization Parameters data collection

U8 : Bits Per Vertex

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

U8 : Normal Bits Factor

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

U8 : Bits Per Texture Coord

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

U8 : Bits Per Colour

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

6.1.1.10.3 Tri-Strip Set Shape Node Element

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

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

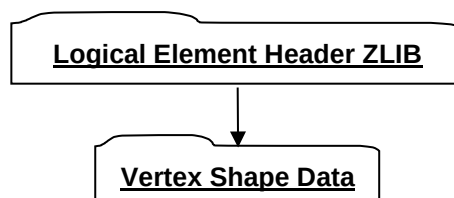


Figure 44 — Tri-Strip Set Shape Node Element data collection

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

Complete description for Vertex Shape Data can be found in [Vertex Shape Data](#).

6.1.1.10.4 Polyline Set Shape Node Element

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

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

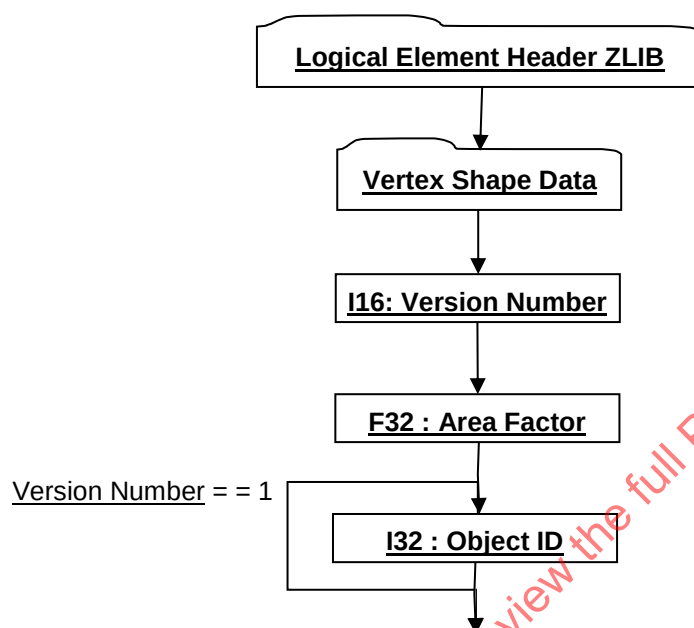


Figure 45 — Polyline Set Shape Node Element data collection

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

Complete description for Vertex Shape Data can be found in [Vertex Shape Data](#).

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice [11.5 Local version numbers](#).

F32: Area Factor

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

U64: Vertex Bindings

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

6.1.1.10.5 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.

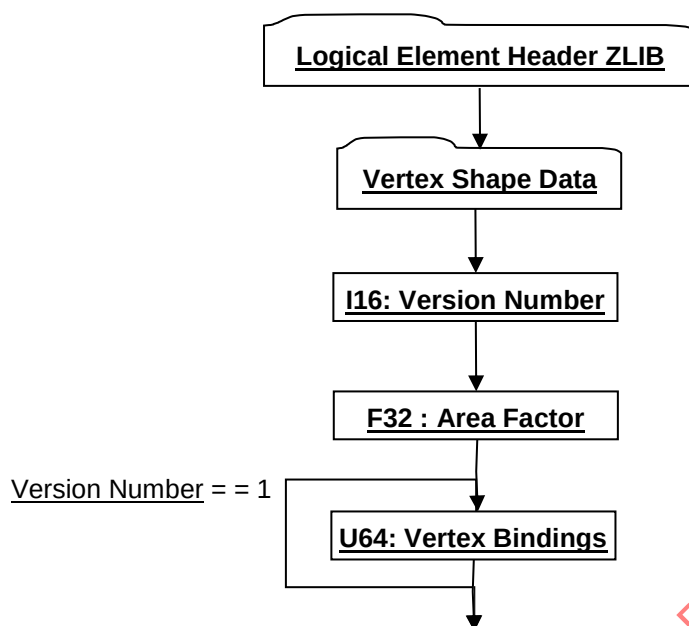


Figure 46 — Point Set Shape Node Element data collection

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

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

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 11.5 Local version numbers.

F32 : Area Factor

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

U64: Vertex Bindings

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

6.1.1.10.6 Polygon Set Shape Node Element

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

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

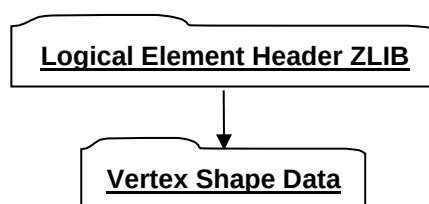


Figure 47 — Polygon Set Shape Node Element data collection

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

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

6.1.1.10.7 NULL Shape Node Element

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

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

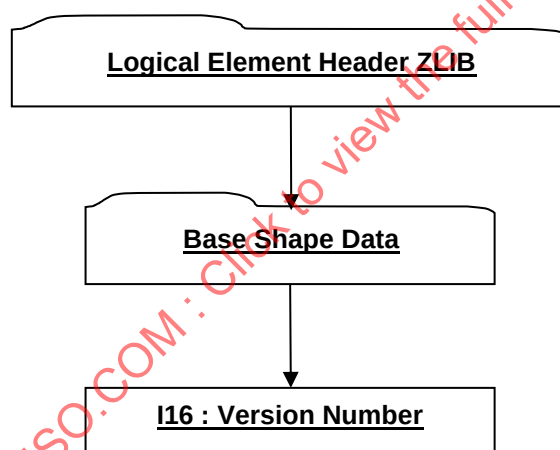


Figure 48 — NULL Shape Node Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 11.5 Local version numbers.

6.1.1.10.8 Primitive Set Shape Node Element

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

A Primitive Set Shape Node Element represents a list/set of primitive shapes (e.g. box, cylinder, sphere, etc.) whose LODs can be procedurally generated. “Procedurally generate” means that the raw geometric

shape definition data (e.g. vertices, polygons, normals, etc.) for LODs is not directly stored; instead some basic shape information is stored (e.g. sphere centre and radius) from which LODs can be generated.

Primitive Set Shape Node Elements actually do not even directly contain this basic shape definition data, but instead reference (using Late Loaded Property Atoms) Primitive Set Shape 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 6.2.7 Late Loaded Property Atom Element and 7.2 Primitive Set Shape Element respectively.

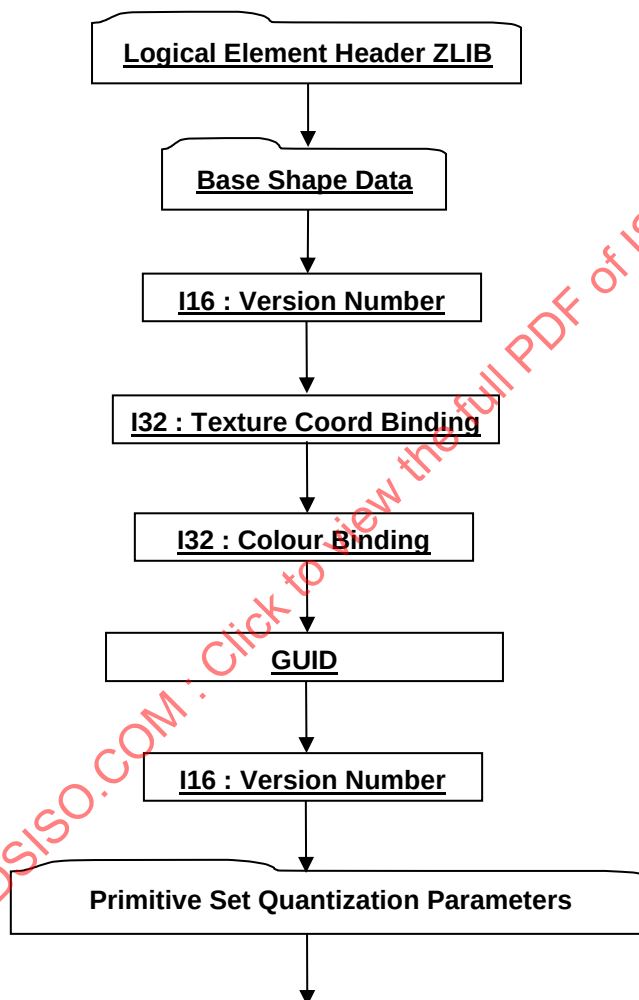


Figure 49 — Primitive Set Shape Node Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 11.5 Local version numbers.

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:

Table 14 — Texture Coord Binding values

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

I32 : Colour Binding

Colour Binding specifies how (at what granularity) colour 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.

Table 15 — Version Number values

= 0	Version 0 Format
= 1	Version 1 Format

I32 : Texture Coord Gen Type

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

Table 16 — Texture Coord Gen Type values

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

Primitive Set Quantization Parameters

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

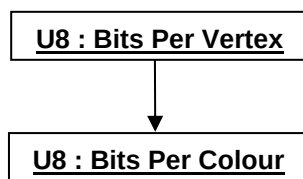


Figure 50 — Primitive Set Quantization Parameters data collection

U8 : Bits Per Vertex

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

U8 : Bits Per Colour

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

6.1.2 Attribute Elements

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

For applications producing or consuming ISO 14306 format data, it is important that the ISO 14306 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 6.1.2.1.1 Base 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 6.1.2.1.1 Base Attribute Data), but do not by default. Note that "force" does not turn OFF "final" – it is simply a one-shot override of "final" for the specific attribute marked as "forcing." An analogy for this "force" and "final" interaction is that "final" is a back-door in the attribute accumulation semantics, and that "force" is a doggy-door in the back-door.

6.1.2.1 Common Attribute Data Containers

6.1.2.1.1 Base Attribute Data

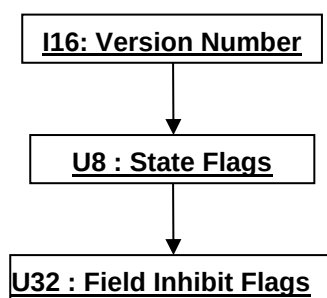


Figure 51 — Base Attribute Data collection

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice **11.5 Local version numbers**.

U8 : State Flags

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

Table 17 — State Flag values

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

U32 : Field Inhibit Flags

Field Inhibit Flags is a collection of flags. 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.

6.1.2.2 Material Attribute Element

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

Material Attribute Element defines the material properties of an object. ISO 14306 format LSG traversal semantics state that material attributes accumulate down the LSG by replacement.

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

Table 18 — Material Attribute data field inhibit values

Field Inhibit Flag Bit	Data Field(s) Bit Applies To
0	<u>Ambient Common RGB Value</u> , <u>Ambient Colour</u>
1	<u>Diffuse Colour and Alpha (Legacy)</u>
2	<u>Specular Common RGB Value</u> , <u>Specular Colour</u>
3	<u>Emission Common RGB Value</u> , <u>Emission Colour</u>
4	<u>Blending Flag</u> , <u>Source Blending Factor</u> , <u>Destination Blending Factor</u>
5	<u>Override Vertex Colour Flag</u>
6	<u>Material Reflectivity</u>
7	<u>Diffuse Colour</u>
8	<u>Diffuse Alpha</u>

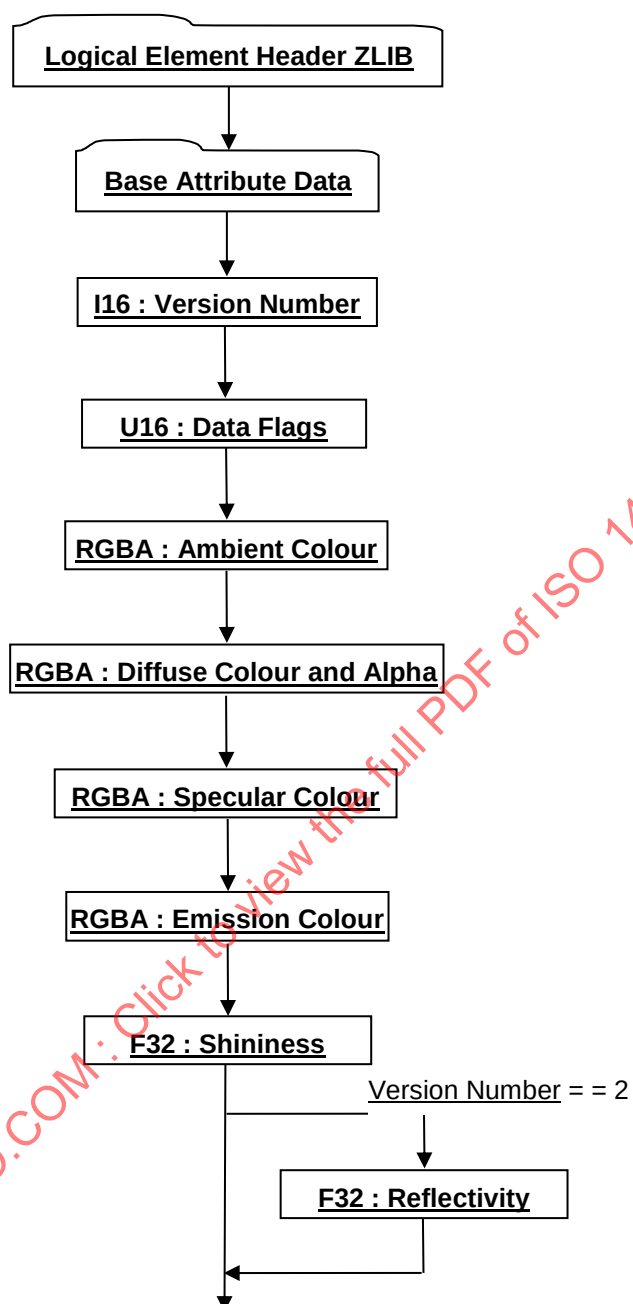


Figure 52 — Material Attribute Element data collection

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

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

Table 19 — Material Attribute Version number value

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

U16 : Data Flags

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

Table 20 — Material Attribute Data Flag values

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

	= 2 – Interpret same as OpenGL GL_DST_COLOUR Blending Factor
	= 3 – Interpret same as OpenGL GL_SRC_COLOUR Blending Factor
	= 4 – Interpret same as OpenGL GL_ONE_MINUS_DST_COLOUR Blending Factor
	= 5 – Interpret same as OpenGL GL_ONE_MINUS_SRC_COLOUR Blending Factor
	= 6 – Interpret same as OpenGL GL_SRC_ALPHA Blending Factor
	= 7 – Interpret same as OpenGL GL_ONE_MINUS_SRC_ALPHA Blending Factor
	= 8 – Interpret same as OpenGL GL_DST_ALPHA Blending Factor
	= 9 – Interpret same as OpenGL GL_ONE_MINUS_DST_ALPHA Blending Factor
	= 10 – Interpret same as OpenGL GL_SRC_ALPHA_SATURATE Blending Factor

RGBA : Ambient Colour

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

RGBA : Diffuse Colour and Alpha

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

RGBA : Specular Colour

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

RGBA : Emission Colour

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

F32 : Shininess

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

F32 : Reflectivity

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

6.1.2.3 Texture Image Attribute Element

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

Texture Image Attribute Element defines a texture image and its mapping environment. ISO 14306 format LSG traversal semantics state that texture image attributes accumulate down the LSG by replacement on a *per channel* basis. See below for more information on texture image channels.

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

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

Table 21 — Texture Image Attribute data field inhibit values

Field Inhibit Flag Bit	Data Field(s) Bit Applies To
0	<u>I32 : Texture Type</u> , <u>Mipmap Image Texel Data</u> , <u>MbString : External Storage Name</u> , <u>Shared Image Flag</u>
1	<u>Border Mode</u> , <u>Border Colour</u>
2	<u>Mipmap Minification Filter</u> , <u>Mipmap Magnification Filter</u>
3	<u>S-Dimen Wrap Mode</u> , <u>T-Dimen Wrap Mode</u> , <u>R-Dimen Wrap Mode</u>

4	<u>Blend Type, Blend Colour</u>
5	<u>Texture Transform</u>
6	<u>Tex Coord Gen Mode, Tex Coord Reference Plane</u>
8	<u>Internal Compression Level</u>

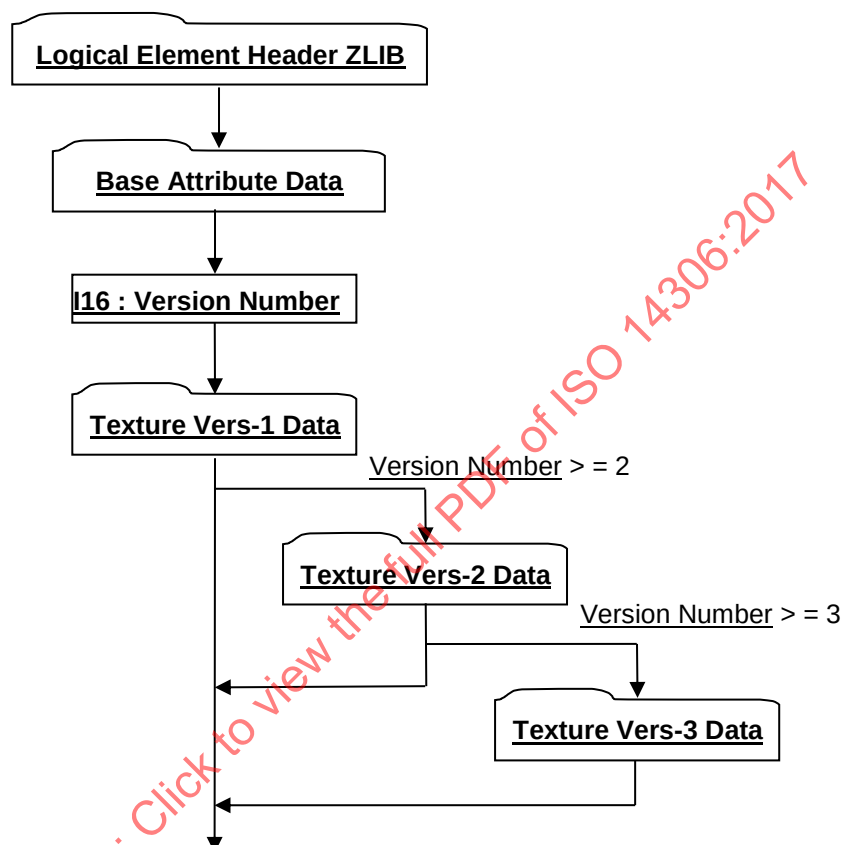


Figure 53 — Texture Image Attribute Element data collection

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

Complete description for Base Attribute Data can be found in [6.1.2.1.1 Base Attribute Data](#).

Complete description for Texture Vers-1 Data can be found in [6.1.2.3.1 Texture Vers-1 Data](#).

Complete description for Texture Vers-2 Data can be found in [6.1.2.3.2 Texture Vers-2 Data](#).

Complete description for Texture Vers-3 Data can be found in [6.1.2.3.3 Texture Vers-3 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.

Table 22 — Texture Image Version Number values

= 1	Version-1 Format
= 2	Version-2 Format
= 3	Version-3 Format

When a data element in the JT file is versioned, it is for the purpose of adding a few pieces of new data onto the end of the existing data format. In this way, older viewers and readers of the JT file that do not yet know about higher local versions will naturally read the lower-numbered version blocks and ignore the higher-numbered ones they do not know how to read. This is sometimes the case with Texture Image Attribute Element, but sometimes not. Entirely new texture types with no analogous lower-level functionality have been added. In these cases, the most sensible thing for an older reader to do it to ignore the texture image entirely as if it were not even present in the JT file.

In order to support this sensible fallback mechanism, the following two general rules are followed: 1) a given texture image is written at the lowest version level that completely captures its contents, and 2) lower-order Texture Vers Data blocks are written with a "stub" texture.

6.1.2.3.1 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 [6.1.2.3.2 Texture Vers-2 Data](#) and [6.1.2.3.3 Texture Vers-3 Data](#)). However, advanced textures *also* write a Texture Vers-1 Data block because of the need to be backward-compatible with older readers that may not understand Vers-2 and Vers-3 data.

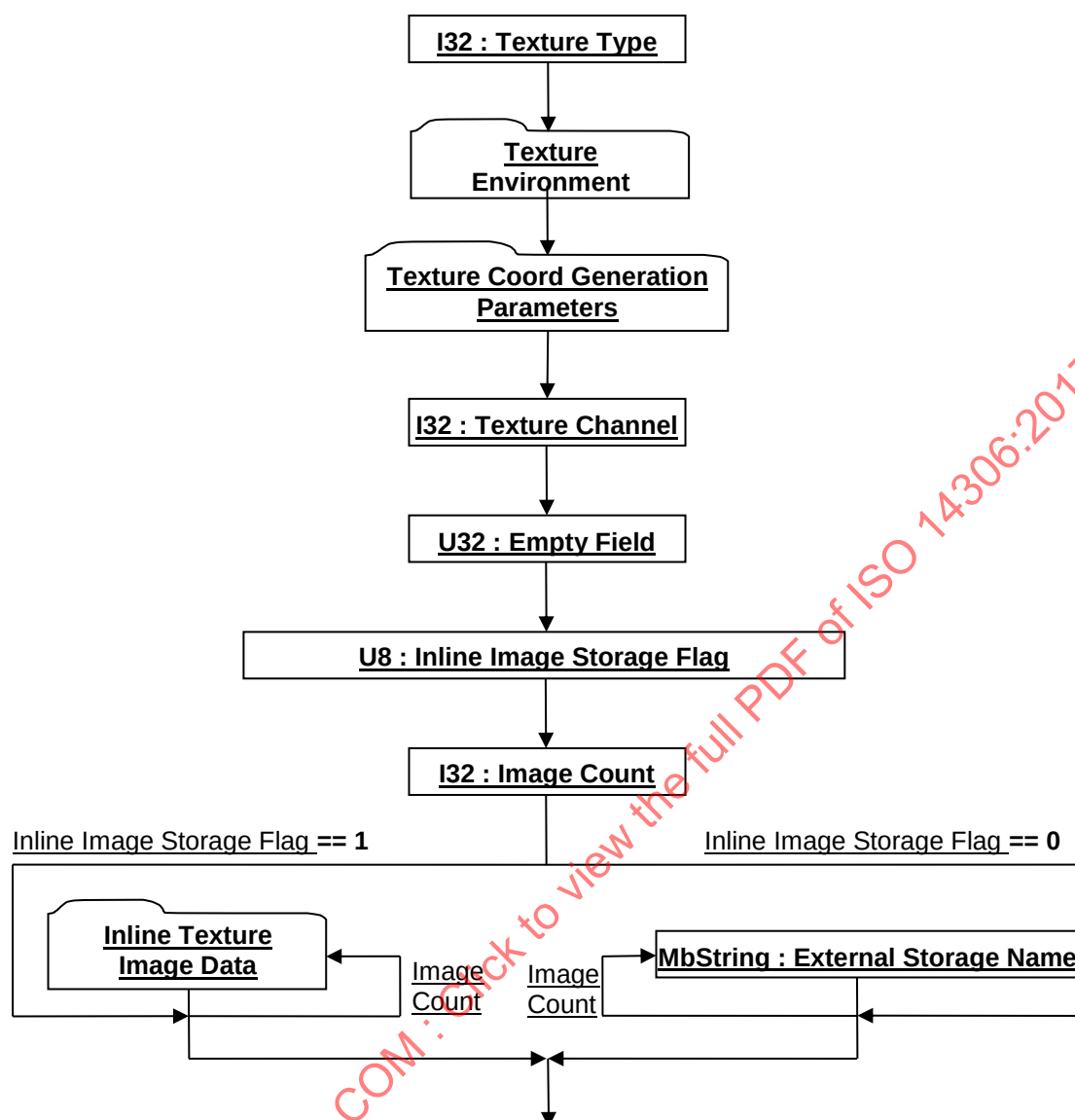


Figure 54 — Texture Vers-1 Data collection

Complete details for Texture Environment can be found in [Texture Environment](#).

Complete details for Texture Coord Generation Parameters can be found in [Texture Coord Generation Parameters](#).

Complete details for Inline Texture Image Data can be found in [Inline Texture Image Data](#).

I32 : Texture Type

Texture Type specifies the type of texture.

Table 23 — Texture Vers-1 type values

= 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 colour values) represents surface normal XYZ components.
= 5	Cube Map. A cube map texture is a texture cube centred at the origin and formed by a set of six two-dimensional texture images.
= 6	Depth Map. A depth map texture is a texture where the image texel data represents depth values.

I32 : Texture Channel

Texture Channel specifies the texture channel number for the Texture Image Element. For purposes of multi-texturing, the JT concept of a texture channel corresponds to the OpenGL concept of a "texture unit." The Texture Channel value shall be between 0 and 31 inclusive. Common Data Conventions and Constructs suggest that renderers of JT data ignore all but channel-0 if the renderer does not support multi-textured geometry. Also for purposes of blending, renderers of JT data should assume that higher numbered texture channels "blend over" lower numbered ones.

U32 : Empty Field

Refer to best practice [11.4 Empty Field](#).

U8 : Inline Image Storage Flag

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

Table 24 — Texture Vers-1 Inline Image Storage Flag values

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

I32 : Image Count

Image Count specifies the number of texture images. A "Cube Map" [I32 : Texture Type](#) shall have six images while all other Texture Types should only have one image.

MbString : External Storage Name

External Storage Name is a string identifying the name of an external texture image storage. External Storage Name is only present if data field [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.

Texture Environment

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

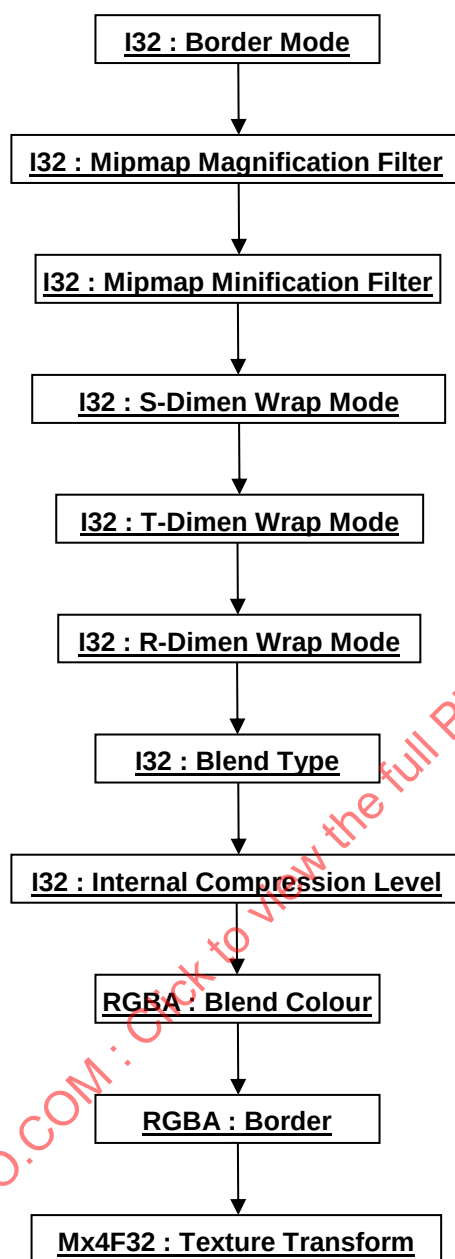


Figure 55 — Texture Environment data collection

I32 : Border Mode

Border Mode specifies the texture border mode.

Table 25 — Texture Vers-1 Texture Environment Border Mode values

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

I32 : Mipmap Magnification Filter

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

Table 26 — Texture Vers-1 Texture Environment Mipmap Magnification Filter values

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

I32 : Mipmap Minification Filter

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

Table 27 — Texture Vers-1 Texture Environment Mipmap Minification Filter values

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

I32 : S-Dimen Wrap Mode

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

Table 28 — Texture Vers-1 Texture Environment S-Dimen Wrap Mode values

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

I32 : T-Dimen Wrap Mode

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

I32 : R-Dimen Wrap Mode

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

I32 : Blend Type

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

Table 29 — Texture Vers-1 Texture Environment Blend Type values

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

I32 : Internal Compression Level

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

Table 30 — Texture Vers-1 Texture Environment Internal Compression Level values

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

RGBA : Blend Colour

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

RGBA : Border Colour

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

Mx4F32 : Texture Transform

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

Texture Coord Generation Parameters

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

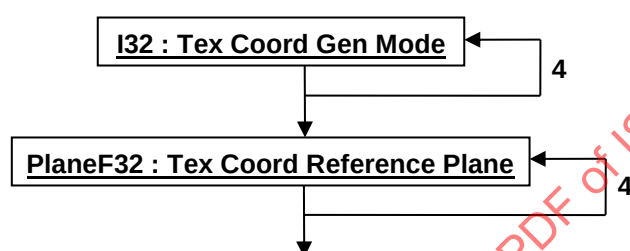


Figure 56 — Texture Coord Generation Parameters data collection

I32 : Tex Coord Gen Mode

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

Table 31 — Texture Vers-1 Texture Coord Generation Gen Mode values

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

PlaneF32 : Tex Coord Reference Plane

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

Inline Texture Image Data

Inline Texture Image Data is a collection of data defining the texture format properties and image texel data for one texture image. Inline Texture Image Data is only present if data field Inline Image Storage

Flag equals “1.” If present there will be data field Image Count number of Inline Texture Image Data instances.

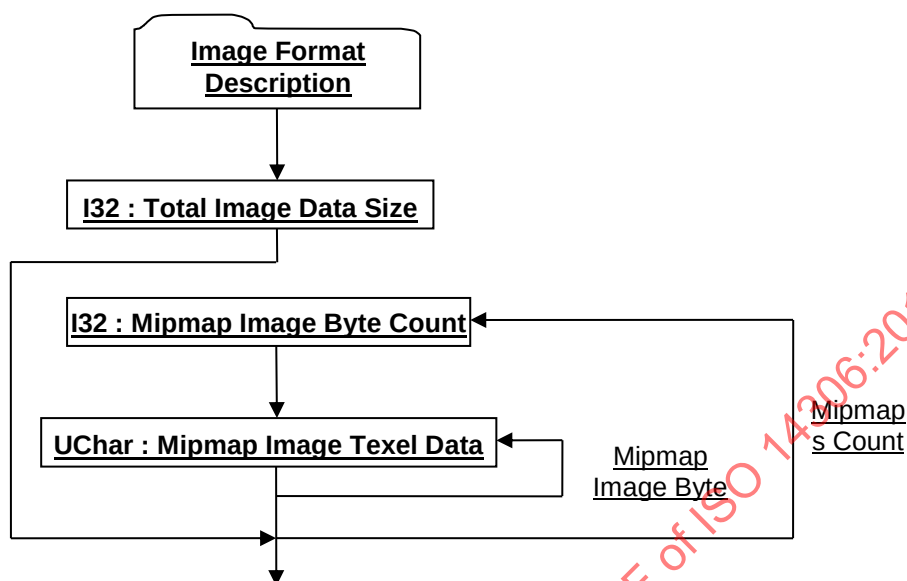


Figure 57 — Inline Texture Image Data collection

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

I32 : Total Image Data Size

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

I32 : Mipmap Image Byte Count

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

UChar : Mipmap Image Texel Data

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

Image Format Description

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

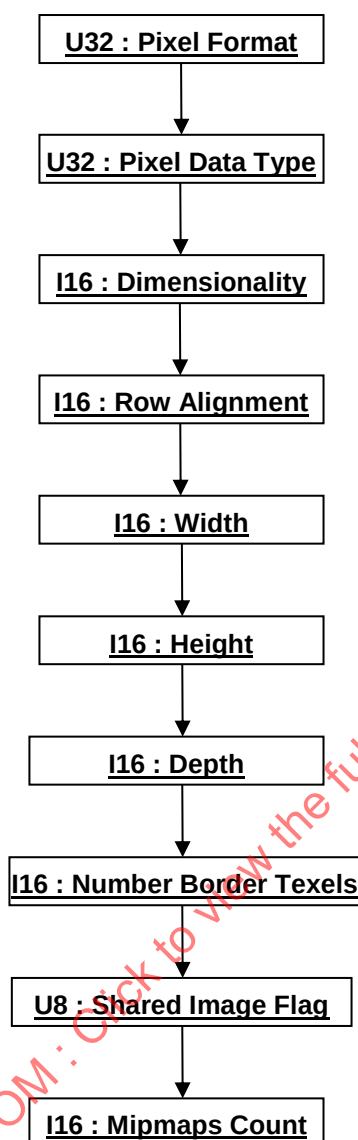


Figure 58 — Image Format Description data collection

U32 : Pixel Format

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

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

= 0	No format specified. Texture mapping is not applied.
= 1	RGB: A red colour component followed by green and blue colour components
= 2	RGBA: A red colour component followed by green, blue, and alpha colour components
= 3	LUM: A single luminance component
= 4	LUMA: A luminance component followed by an alpha colour component.
= 5	A single stencil index.
= 6	A single depth component
= 7	A single red colour component
= 8	A single green colour component
= 9	A single blue colour component

= 10	A single alpha colour component
= 11	A blue colour component, followed by green and red colour components
= 12	A blue colour component, followed by green , red, and alpha colour components
= 13	A depth component, followed by a stencil component

U32 : Pixel Data Type

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

Table 33 — Texture Vers-1 Image Format Description Pixel Data values

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

I16 : Dimensionality

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

Table 34 — Texture Vers-1 Image Format Description Dimensionality values

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

I16 : Row Alignment

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

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

I16 : Width

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

I16 : Height

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

I16 : Depth

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

I16 : Number Border Texels

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

U8 : Shared Image Flag

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

Table 35 — Texture Vers-1 Image Format Description Shared Image Flag values

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

I16 : Mipmaps Count

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

6.1.2.3.2 Texture Vers-2 Data

Texture Vers-2 Data collection supports texturing effects not representable in the Texture Vers-1 Data format (e.g. more precise texture types, automatic texture channel, etc.) Any Texture Image Attribute Element using the Texture Vers-2 Data format will contain a “degenerate” Texture Vers-1 Data block, where Image Count data field has a value of “0”.

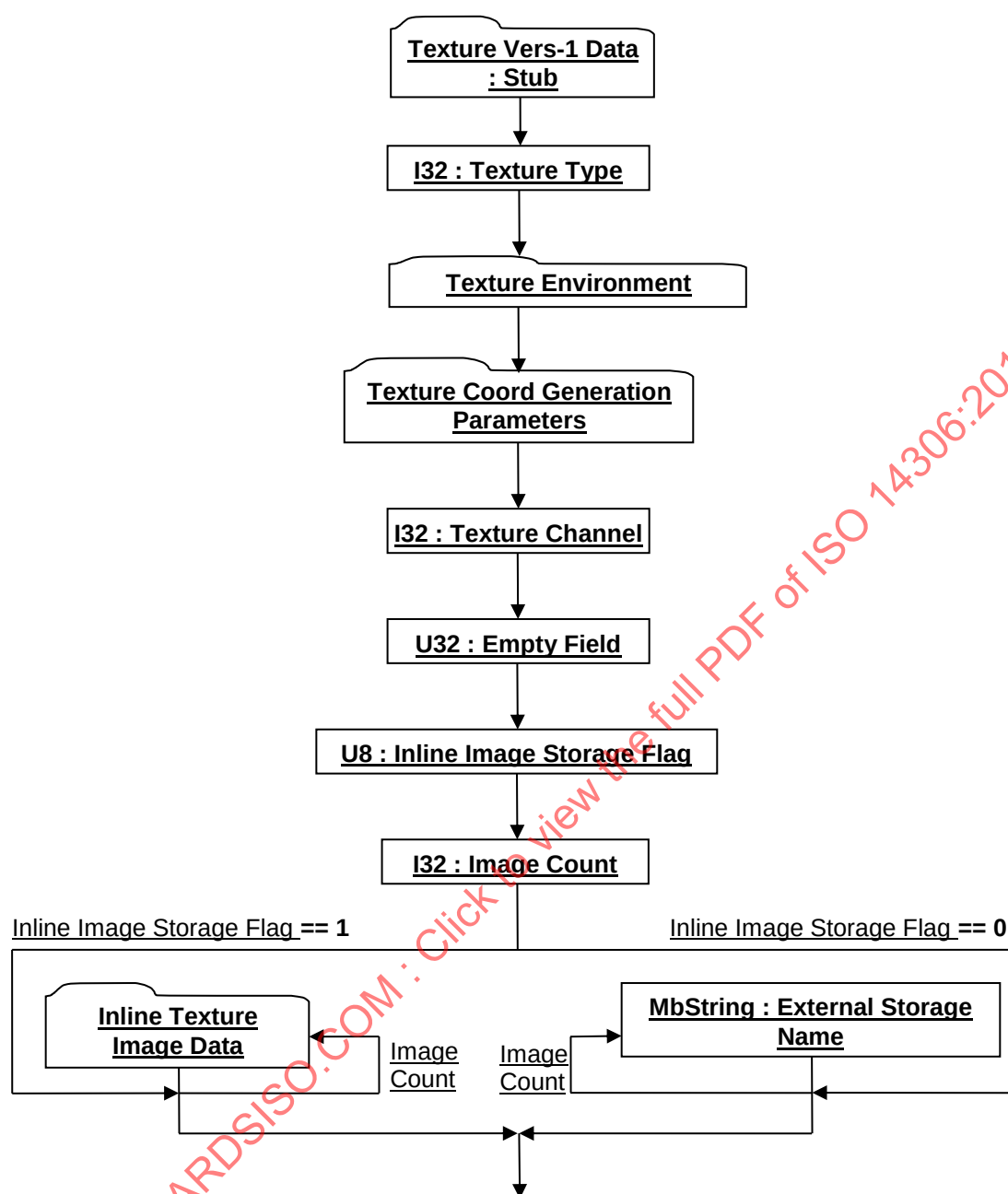


Figure 59 — Texture Vers-2 Data collection

Complete details for Texture Environment can be found in [Texture Environment](#).

Complete details for Texture Coord Generation Parameters can be found in [Texture Coord Generation Parameters](#).

Complete details for Inline Texture Image Data can be found in [Inline Texture Image Data](#).

Texture Vers-1 Data : Stub

This is a dummy block written with its [I32 : Texture Type](#) field set to "None". This block is included so that older readers that do not understand [Texture Vers-2 Data](#) will read an "empty" texture.

I32 : Texture Type

Texture Type specifies the type of texture. There is a complete restructuring and redefinition of what a “texture type” implies in Texture Vers-2 Data. It is a much stronger concept now, that not only describes generally what the texture image contains, but also defines precisely what the texture is being used for. In the following list, “image” refers to an image texture, “pre-lit” indicates that the image texture is to be applied before lighting when rendering the object to which it is applied, and “post-lit” indicates that the image texture is to be applied after lighting. A gloss map is a pre-lit texture that applies itself to the specular material component of lighting instead of the diffuse component. A light map is an environment texture (texture at infinity surrounding the whole model) that serves as a source of illumination during shading calculations.

Table 36 — Texture Vers-2 Texture Type values

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

I32 : Texture Channel

Texture Channel specifies the texture channel number for the Texture Image Element. For purposes of multi-texturing, the JT concept of a texture channel corresponds to the OpenGL concept of a “texture unit.” The Texture Channel value shall be between -1 and 31 inclusive. The value -1 is accepted to denote a texture whose channel number is to be automatically assigned. This assignment will never displace another texture with an explicit texture channel assignment from its slot. Common Data Conventions and Constructs 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, any renderer of JT data should ensure that higher numbered texture channels “blend over” lower numbered ones.

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

U32 : Empty Field

Refer to best practice [11.4 Empty Field](#).

U8 : Inline Image Storage Flag

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

Table 37 — Texture Vers-2 Inline Image Storage Flag values

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

I32 : Image Count

Image Count specifies the number of texture images. A “Cube Map” [I32 : Texture Type](#) shall have six images while all other Texture Types may 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.

6.1.2.3.3 Texture Vers-3 Data

Texture Vers-3 Data collection supports texturing effects not representable in the [Texture Vers-1 Data](#) format or the [Texture Vers-2 Data](#) format (e.g. texture coordinate channel, separator texture type, and texture channel greater than 31). Any [Texture Image Attribute Element](#) using the [Texture Vers-3 Data](#) format will contain a “degenerate” [Texture Vers-1 Data](#) block, and a “degenerate” [Texture Vers-2 Data](#) block, where [Image Count](#) data field has a value of 0 and the Texture Type will be set to None.

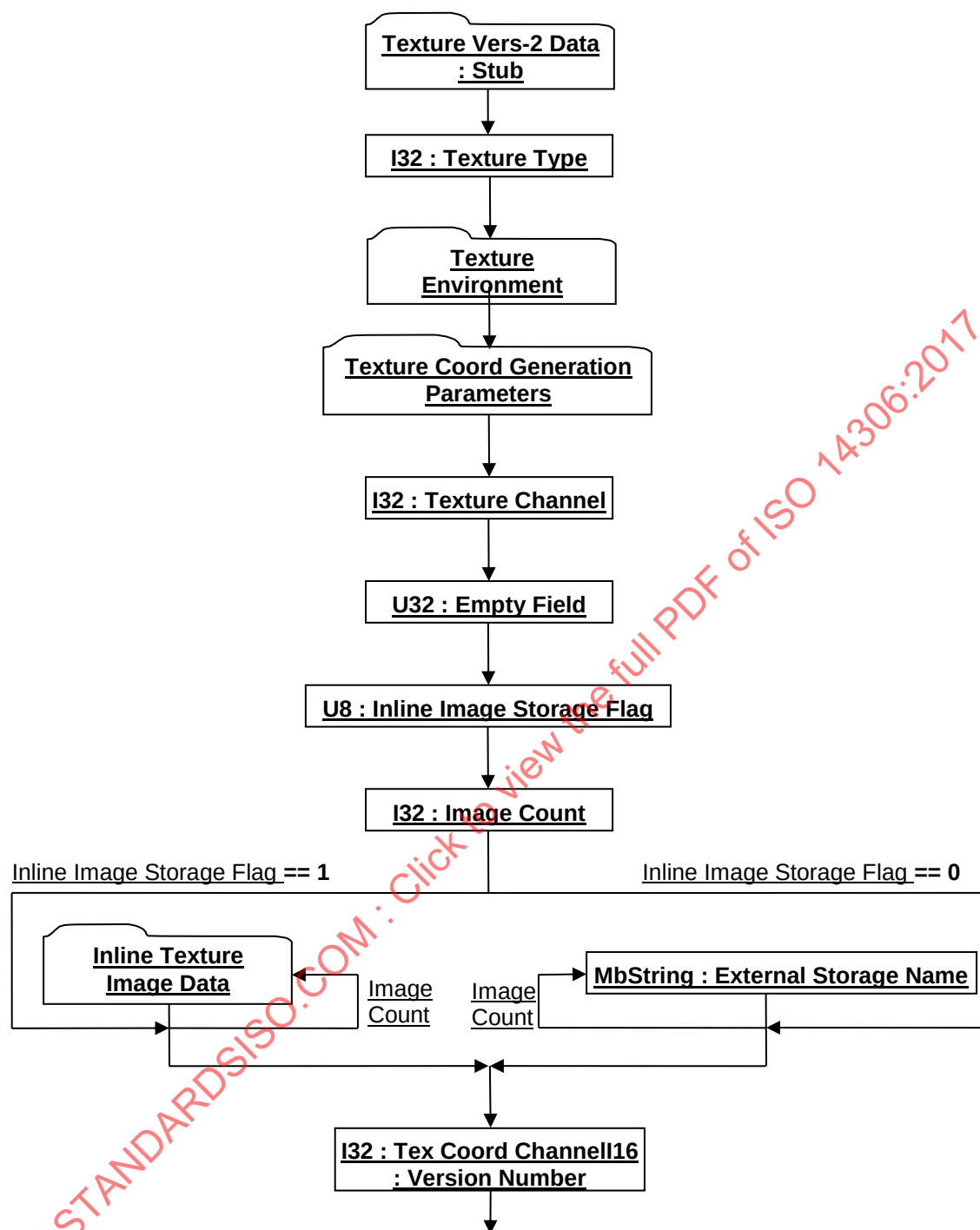


Figure 60 — Texture Vers-3 Data collection

Complete details for Texture Environment can be found in [Texture Environment](#).

Complete details for Texture Coord Generation Parameters can be found in [Texture Coord Generation Parameters](#).

Complete details for Inline Texture Image Data can be found in [Inline Texture Image Data](#).

Texture Vers-2 Data : Stub

This is a dummy block written with its I32 : Texture Type field set to "None". This block is included so that older readers that do not understand Texture Vers-3 Data will read an "empty" texture.

I32 : Texture Type

Texture Type specifies the type of texture. A new texture type, separator texture, is defined in Texture Vers-3 Data to support resetting the texture accumulation state mid-graph. Shadow maps and prefiltered light maps, however, are a general exception to this rule. In the following list, "image" refers to an image texture, "pre-lit" indicates that the image texture is to be applied before lighting when rendering the object to which it is applied, and "post-lit" indicates that the image texture is to be applied after lighting. A gloss map is a pre-lit texture that applies itself to the specular material component of lighting instead of the diffuse component. A light map is an environment texture (texture at infinity surrounding the whole model) that serves as a source of illumination during shading calculations.

Table 38 — Texture Vers-3 Texture Type values

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

I32 : Texture Channel

Texture Channel specifies the texture channel number for the Texture Image Element. For purposes of multi-texturing, the JT concept of a texture channel corresponds to the OpenGL concept of a "texture unit." The Texture Channel value shall be between -1 and 2,147,483,647 inclusive. The value -1 is accepted to denote a texture whose channel number is to be automatically assigned. This assignment will never displace another texture with an explicit texture channel assignment from its slot. Common Data Conventions and Constructs 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, any renderer of JT data should ensure that higher numbered texture channels "blend over" lower numbered ones.

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

U32 : Empty Field

Refer to best practice [11.4 Empty Field](#).

U8 : Inline Image Storage Flag

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

Table 39 — Texture Vers-3 Inline Image Storage Flag values

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

I32 : Image Count

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

MbString : External Storage Name

External Storage Name is a string identifying the name of an external texture image storage. External Storage Name is only present if data field [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.

I32 : Tex Coord Channel

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

6.1.2.4 Draw Style Attribute Element

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

Draw Style Attribute Element contains information defining various aspects of the graphics state/style that should be used for rendering associated geometry. ISO 14306 format LSG traversal semantics state that draw style attributes accumulate down the LSG by replacement.

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

Table 40 — Draw Style Attribute Field Inhibit flag values

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

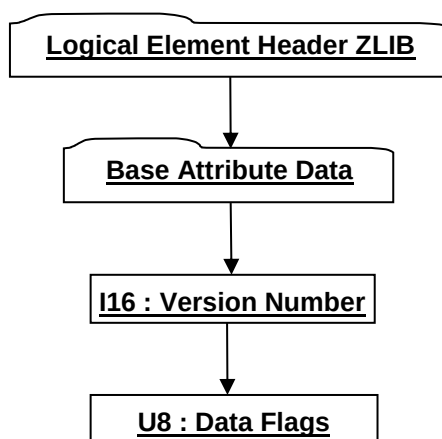


Figure 61 — Draw Style Attribute Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice 11.5 Local version numbers.

U8 : Data Flags

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

Table 41 — Draw Style Attribute Data Flag values

0x01	Back-face Culling Flag. Indicates if back-facing polygons should be discarded (culled). = 0 – Back-facing polygons not culled. = 1 – Back-facing polygons culled.
0x02	Two Sided Lighting Flag. Indicates if two sided lighting should be enabled to insure that polygons are illuminated on both sides. = 0 – Disable two sided lighting. = 1 – Enable two sided lighting.
0x04	Outlined Polygons Flag. Indicates if polygons should be draw as “wireframes” i.e. not filled. = 0 – Polygons drawn as filled. = 1 – Only polygon’s outline drawn.
0x08	Lighting Enabled Flag. Indicates if lighting should be enabled. If lighting disabled, then renderer should perform no calculations concerning normals, light sources, material properties, etc. = 0 – Disable lighting. = 1 – Enable lighting.
0x10	Flat Shading Flag.

	Indicates if the geometry should be rendered with single colour (flat shading) or with many different colour (smooth/Gouraud) shading. = 0 – Disable flat shading (i.e. use smooth/Gouraud shading). = 1 – Enable flat shading.
0x20	Separate Specular Flag. Indicates if the application of the specular colour should be delayed until after texturing. If no texture mapping then this flag setting is irrelevant. = 0 – Apply specular colour contribution before texture mapping. = 1 – Apply specular colour contribution after texture mapping.

6.1.2.5 Light Set Attribute Element

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

Light Set Attribute Element holds an unordered list of Lights. ISO 14306 format LSG traversal semantics state that light set attributes accumulate down the LSG through addition of lights to an attribute list.

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

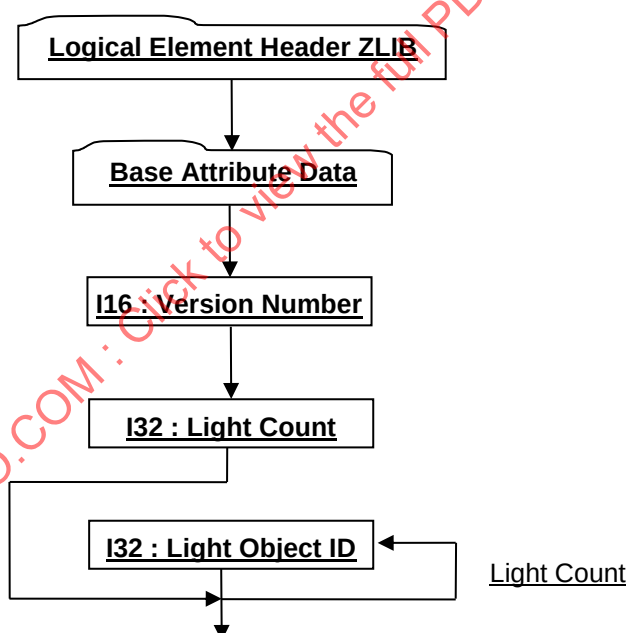


Figure 62 — Light Set Attribute Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice 11.5 Local version numbers.

I32 : Light Count

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

I32 : Light Object ID

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

6.1.2.6 Infinite Light Attribute Element

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

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

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

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

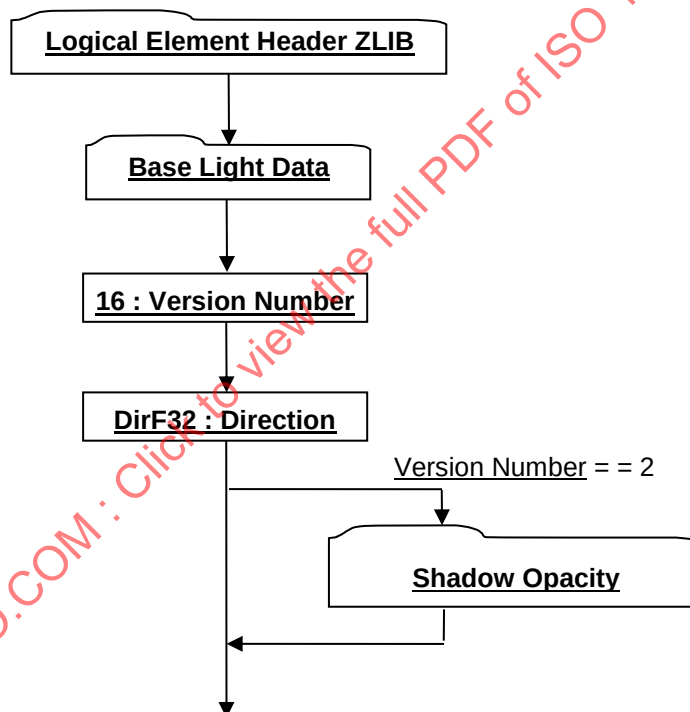


Figure 63 — Infinite Light Attribute Element data collection

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

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

Complete description for Shadow Parameters can be found in [6.1.2.6.2 Shadow Parameters](#).

16 : Version Number

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

Table 42 — Light Set Attribute Version Number values

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

DirF32 : Direction

Direction specifies the direction the light is pointing in.

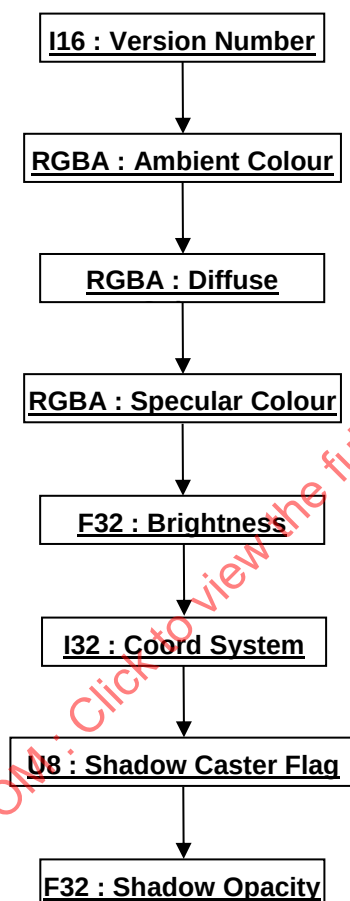
6.1.2.6.1 Base Light Data

Figure 64 — Base Light Data collection

I16 : Version Number

Version number is the version identifier for this element. For information on local version numbers see best practice **11.5 Local version numbers**.

RGBA : Ambient Colour

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

RGBA : Diffuse Colour

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

RGBA : Specular Colour

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

F32 : Brightness

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

I32 : Coord System

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

Table 43 — Base Light Data Coord System values

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

U8 : Shadow Caster Flag

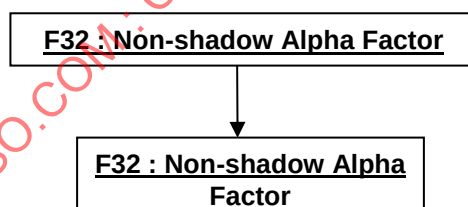
Shadow Caster Flag is a flag that indicates whether the light is a shadow caster or not.

Table 44 — Base Light Data Shadow Caster Flag values

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

F32 : Shadow Opacity

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

6.1.2.6.2 Shadow Parameters**Figure 65 — Shadow Parameters data collection****F32 : Non-shadow Alpha Factor**

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

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

F32 : Shadow Alpha Factor

Shadow Alpha Factor is one of a matched pair of fields intended to govern how a shadowing light source (one whose Shadow Caster Flag is set) casts "alpha light" into areas that it does not illuminate (i.e. are

in shadow). Those fragments in shadow from this light source will have their alpha values scaled by Shadow Alpha Factor. Shadow Alpha Factor value shall lie on the range [0.0, 1.0] inclusive.

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

6.1.2.7 Point Light Attribute Element

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

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

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

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

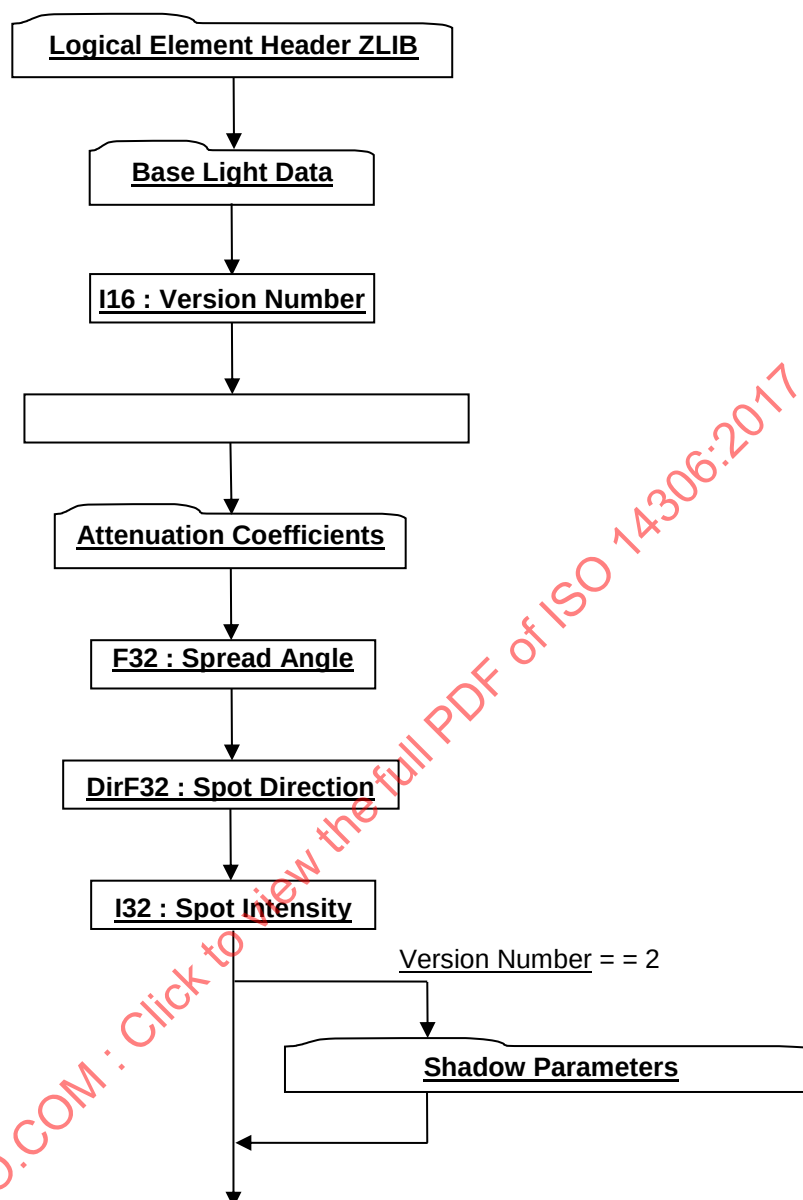


Figure 66 — Point Light Attribute Element data collection

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

Complete description for Base Light Data can be found in [6.1.2.6.1 Base Light Data](#).

Complete description for Attenuation Coefficients can be found in [6.1.2.7.1 Attenuation Coefficients](#).

Complete description for Shadow Parameters can be found in [6.1.2.6.2 Shadow Parameters](#).

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.

Table 45 — Point Light Attribute Version Number values

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

HCoordF32 : Position

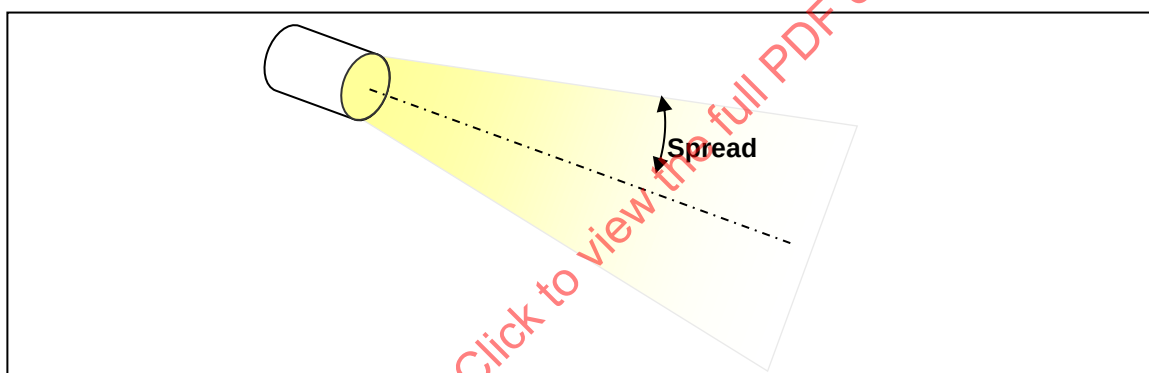
Position specifies the light position in homogeneous coordinates.

F32 : Spread Angle

Spread Angle, as shown in [Figure 67](#) — Spread Angle value with respect to the light cone below, specifies in degrees the half angle of the light cone. Valid Spread Angle values are clamped and interpreted as follows:

Table 46 — Point Light Attribute Spread Angle values

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

**Figure 67 — Spread Angle value with respect to the light cone**

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

I32 : Spot Intensity

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

6.1.2.7.1 Attenuation Coefficients

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

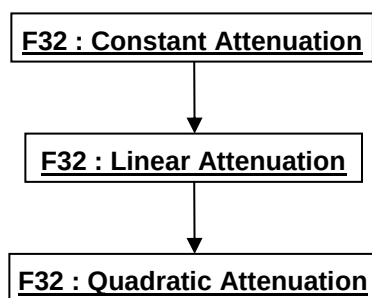


Figure 68 — Attenuation Coefficients data collection

F32 : Constant Attenuation

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

F32 : Linear Attenuation

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

F32 : Quadratic Attenuation

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

6.1.2.8 Linestyle Attribute Element

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

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

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

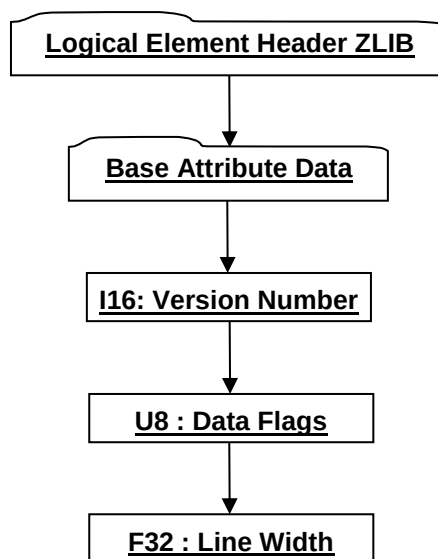


Figure 69 — Linestyle Attribute Element data collection

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

Complete description for Base Attribute Data can be found in [6.1.2.1.1 Base Attribute Data](#).

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice [11.5 Local version numbers](#).

U8 : Data Flags

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

Table 47 — Linestyle Attribute Data Flag values

0x0 F	Line Type (stored in bits 0 – 3 or in binary notation 00001111) Line type specifies the polyline rendering stipple-pattern. = 0 - Solid = 1 - Dash = 2 - Dot = 3 - Dash_Dot = 4 - Dash_Dot_Dot = 5 - Long_Dash = 6 - Centre_Dash = 7 - Centre_Dash_Dash
0x1 0	Antialiasing Flag (stored in bit 4 or in binary notation 00010000) Indicates if antialiasing should be applied as part of rendering polylines. = 0 – Antialiasing disabled. = 1 – Antialiasing enabled.

F32 : Line Width

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

6.1.2.9 Pointstyle Attribute Element

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

Pointstyle Attribute Element contains information defining the graphical properties that should be used for rendering points. ISO 14306 format LSG traversal semantics state that Pointstyle attributes accumulate down the LSG by replacement.

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

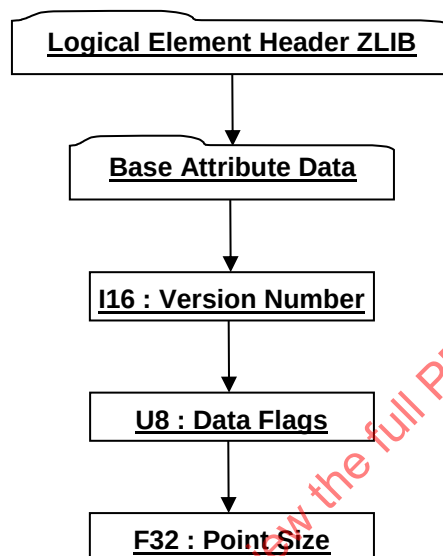


Figure 70 — Pointstyle Attribute Element data collection

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

Complete description for Base Attribute Data can be found in [6.1.2.1.1 Base Attribute Data](#).

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice [11.5 Local version numbers](#).

U8 : Data Flags

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

Table 48 — Pointstyle Attribute Data Flag values

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

F32 : Point Size

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

6.1.2.10 Geometric Transform Attribute Element

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

Geometric Transform Attribute Element contains a 4x4 homogeneous transformation matrix that positions the associated LSG node's coordinate system relative to its parent LSG node. ISO 14306 format LSG traversal semantics state that geometric transform attributes accumulate down the LSG through matrix multiplication as follows:

$$p' = pAM$$

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

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

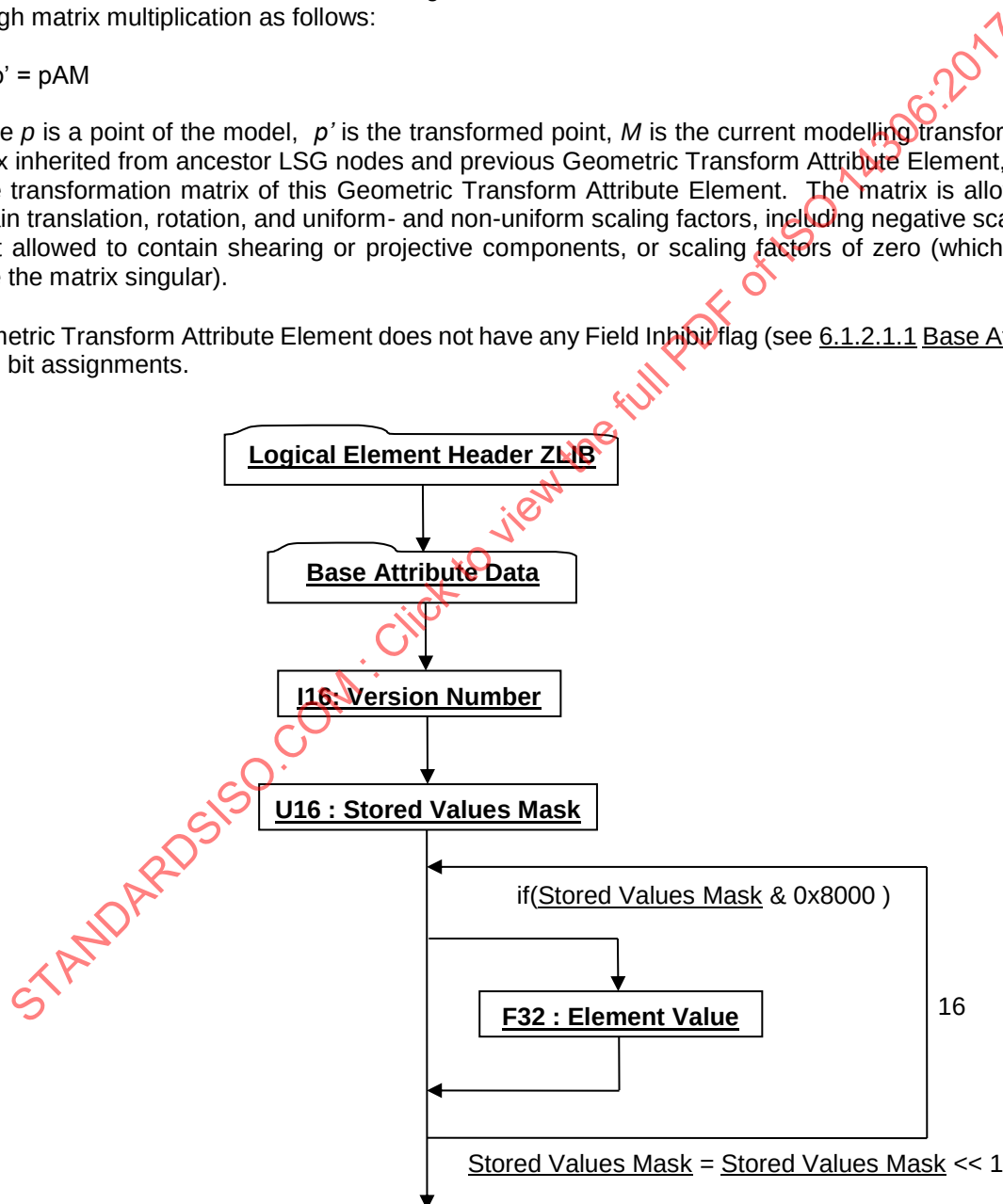


Figure 71 — Geometric Transform Attribute Element data collection

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

Complete description for Base Attribute Data can be found in [6.1.2.1.1 Base Attribute Data](#).

I16: Version Number

Version Number is the version identifier for this node. For information on local version numbers see best practice [11.5 Local version numbers](#).

U16 : Stored Values Mask

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

Bit15	Bit14	Bit13	Bit12
Bit11	Bit10	Bit9	Bit8
Bit7	Bit6	Bit5	Bit4
Bit3	Bit2	Bit1	Bit0

The individual bit-flag values are interpreted as follows:

Table 49 — Geometric Transform Attribute Stored Value Mask individual bit-flag values

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

F32 : Element Value

Element Value specifies a particular matrix element value.

6.1.2.11 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 state that shader effects attributes accumulate down the LSG by replacement.

Shader Effects Attribute Element does not have any Field Inhibit flag (see [6.1.2.1.1 Base Attribute Data](#)) bit assignments.

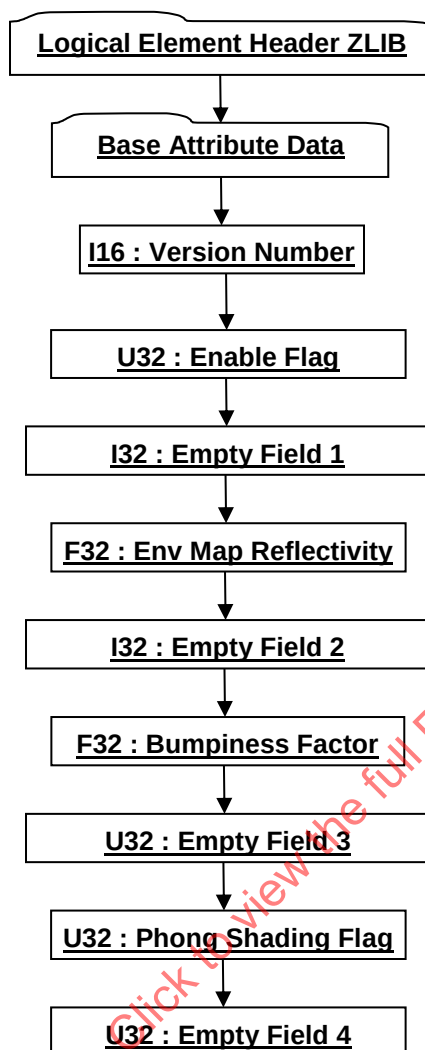


Figure 72 — Shader Effects Attribute Element data collection

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

Complete description for Base Attribute Data can be found in [6.1.2.1.1 Base Attribute Data](#).

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice [11.5 Local version numbers](#).

U32 : Enable Flag

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

Table 50 — Shader Effects Attribute Enable Flag values

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

I32 : Empty Field 1

Refer to best practice [11.4](#) Empty Field.

F32 : Env Map Reflectivity

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

I32 : Empty Field 2

Refer to best practice [11.4](#) Empty Field.

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 than 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 : Empty Field 3

Refer to best practice [11.4](#) Empty Field.

U32 : Phong Shading Flag

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

Table 51 — Shader Effects Attribute Phong Shading Flag values

= 0	Phong Shading disabled
= 1	Phong Shading enabled

U32 : Empty Field 4

Refer to best practice [11.4](#) Empty Field.

6.1.2.12 Texture Coordinate Generator Attribute Element

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

Texture Coordinate Generator Attribute Element defines texture coordinate generation for texture mapping. Multiple texture coordinate generation at a given node is supported by way of the “texture coordinate channel” concept. ISO 14306 format LSG traversal semantics state that Texture Coordinate Generator attributes accumulate down the LSG by replacement on a per-channel basis.

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

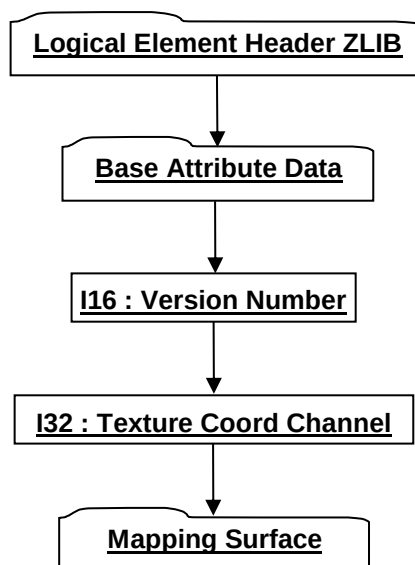


Figure 73 — Texture Coordinate Generator Attribute Element data collection

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

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

Complete description for Mapping Surface can be found in 6.1.2.12.1 Mapping Surface.

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice **11.5 Local version numbers**.

I32 : Texture Coord Channel

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

6.1.2.12.1 Mapping Surface

Mapping Surface defines the mapping surface for texture coordinate generation. Four kinds of mapping surfaces, Mapping Plane Element, Mapping Cylinder Element, Mapping Sphere Element, and Mapping TriPlanar Element are defined to support texture coordinate generation.

Mapping Plane Element

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

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

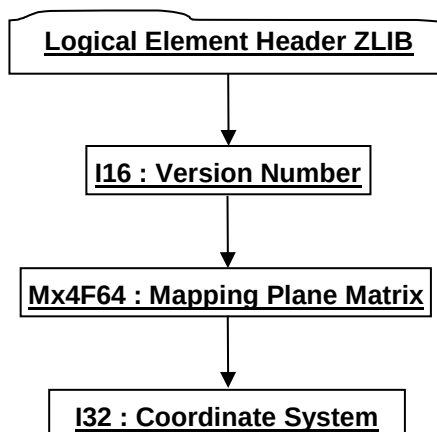


Figure 74 — Mapping Plane Element data collection

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

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice **11.5 Local version numbers**.

Mx4F64 : Mapping Plane Matrix

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

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

I32 : Coordinate System

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

Table 52 — Mapping Plane Matrix Coordinate System values

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

Mapping Cylinder Element

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

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

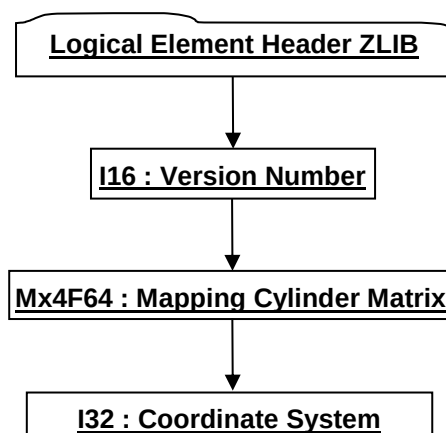


Figure 75 — Mapping Cylinder Element data collection

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

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice **11.5 Local version numbers**.

Mx4F64 : Mapping Cylinder Matrix

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

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

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

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

I32 : Coordinate System

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

Table 53 — Mapping Cylinder Matrix Coordinate System values

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

Mapping Sphere Element

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

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

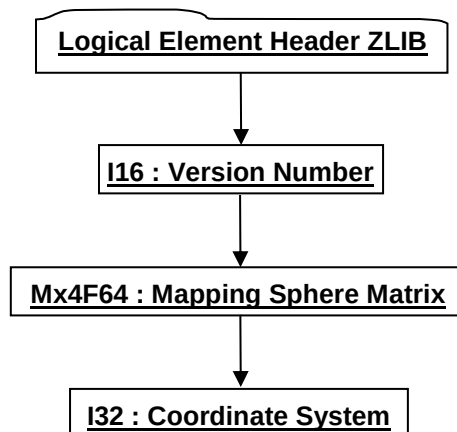


Figure 76 — Mapping Sphere Element data collection

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

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice **11.5 Local version numbers**.

Mx4F64 : Mapping Sphere Matrix

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

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

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

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

I32 : Coordinate System

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

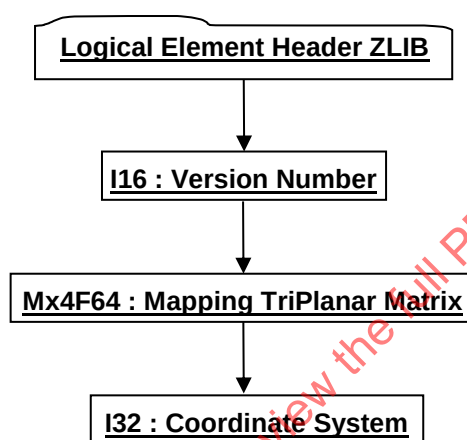
Table 54 — Mapping Sphere Matrix Coordinate System values

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

Mapping TriPlanar Element

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

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

**Figure 77 — Mapping TriPlanar Element data collection**

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

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice **11.5 Local version numbers**.

Mx4F64 : Mapping TriPlanar Matrix

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

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

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

I32 : Coordinate System

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

Table 55 — Mapping TriPlanar Matrix Coordinate System values

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

6.2 Property Atom Elements

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

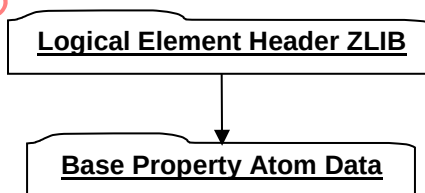
An individual property is specified as a *key/value* Property Atom Element pair, where the *key* identifies the type and meaning of the *value*. The ISO 14306 format supports many different Property Atom Element key/value object types. The different Property Atom Element key/value object types are documented in the following subsections.

Some “Common Data Conventions and Constructs” for placing application or enterprise properties/meta-data on Nodes in JT files can be found in **11.8 Metadata Conventions** section of this reference.

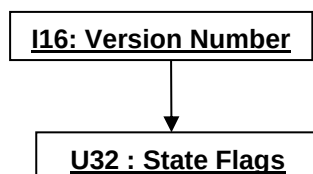
6.2.1 Base Property Atom Element

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

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

**Figure 78 — Base Property Atom Element data collection**

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

Base Property Atom Data**Figure 79 — Base Property Atom Data collection****I16: Version Number**

Version Number is the version identifier for this data collection. For information on local version numbers see best practice [11.5 Local version numbers](#).

U32 : State Flags

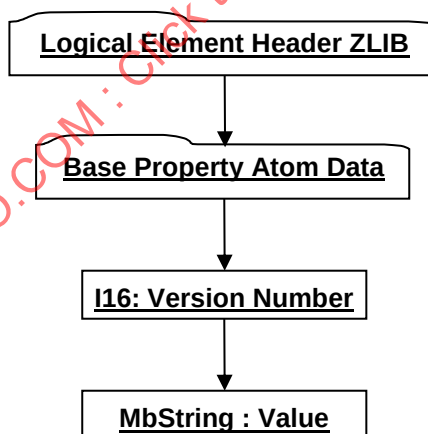
State Flags is a collection of flags. The flags are combined using the binary OR operator and store various state information for property atoms. Bits 0 – 7 are freely available for an application to store whatever property atom information desired. All other bits are reserved for future expansion of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

The State Flags shall be defaulted to 0x40000000 and persisted as such.

6.2.2 String Property Atom Element

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

String Property Atom Element represents a character string property atom.

**Figure 80 — String Property Atom Element data collection**

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

Complete description for Base Property Atom Data can be found in [Base Property Atom Data](#).

I16: Version Number

Version Number is the version identifier for this data collection. For information on local version numbers see best practice [11.5 Local version numbers](#).

MbString : Value

Value contains the character string value for this property atom.

6.2.3 Integer Property Atom Element

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

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

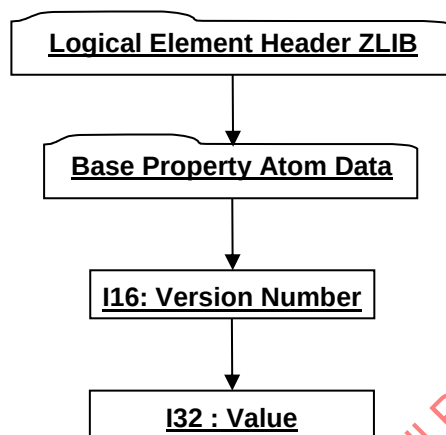


Figure 81 — Integer Property Atom Element data collection

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

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

I16: Version Number

Version Number is the version identifier for this data collection. For information on local version numbers see best practice 11.5 Local version numbers.

I32 : Value

Value contains the integer value for this property atom.

6.2.4 Floating Point Property Atom Element

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

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

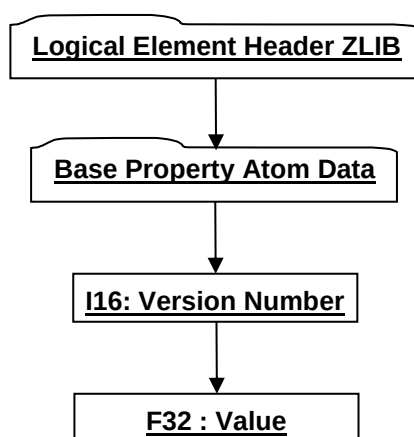


Figure 82 — Floating Point Property Atom Element data collection

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

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

I16: Version Number

Version Number is the version identifier for this data collection. For information on local version numbers see best practice 11.5 Local version numbers.

F32 : Value

Value contains the floating point value for this property atom.

6.2.5 JT Object Reference Property Atom Element

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

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

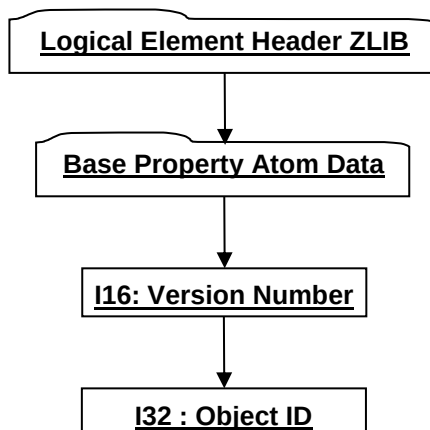


Figure 83 — JT Object Reference Property Atom Element data collection

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

Complete description for Base Property Atom Data can be found in [Base Property Atom Data](#).

I16: Version Number

Version Number is the version identifier for this data collection. For information on local version numbers see best practice [11.5 Local version numbers](#).

I32 : Object ID

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

6.2.6 Date Property Atom Element

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

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

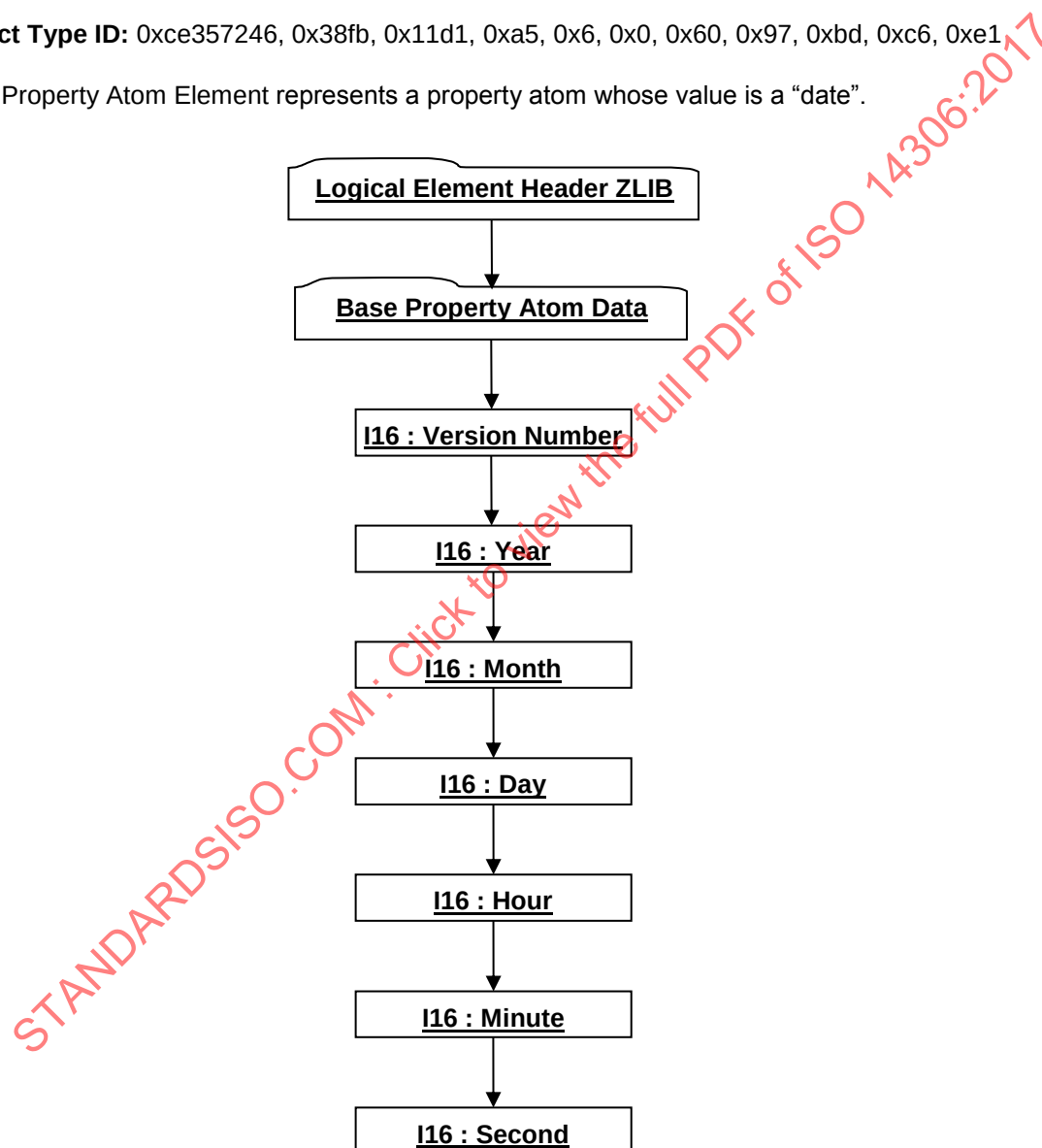


Figure 84 — Date Property Atom Element data collection

Complete description for **Logical Element Header ZLIB** can be found in [Logical Element Header ZLIB](#).

Complete description for Base Property Atom Data can be found in [Base Property Atom Data](#).

I16 : Version Number

Version Number is the version identifier for this data collection. For information on local version numbers see best practice **11.5 Local version numbers**.

I16 : Year

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

I16 : Month

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

I16 : Day

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

I16 : Hour

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

I16 : Minute

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

I16 : Second

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

6.2.7 Late Loaded Property Atom Element

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

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

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

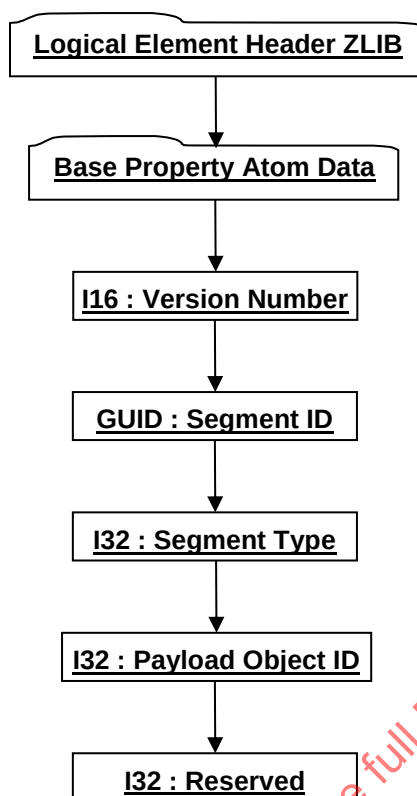


Figure 85 — Late Loaded Property Atom Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this data collection. For information on local version numbers see best practice 11.5 Local version numbers.

GUID : Segment ID

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

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

I32 : Segment Type

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

I32 : Payload Object ID

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

I32 : Reserved

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

6.2.8 Vector4f Property Atom Element

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

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

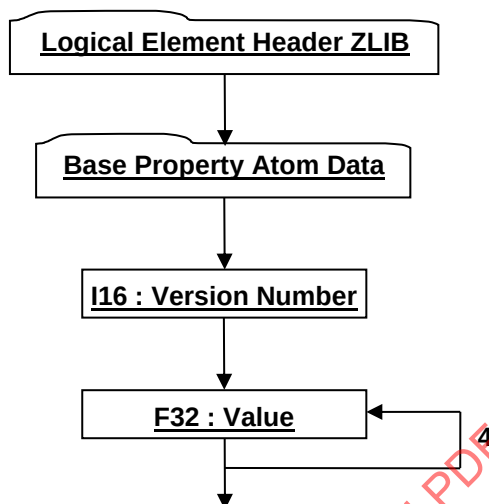


Figure 86 — Vector4f Property Atom Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this data collection. For information on local version numbers see best practice 11.5 Local version numbers.

F32 : Value

Value contains the floating point value for this property atom.

6.3 Property Table

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

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

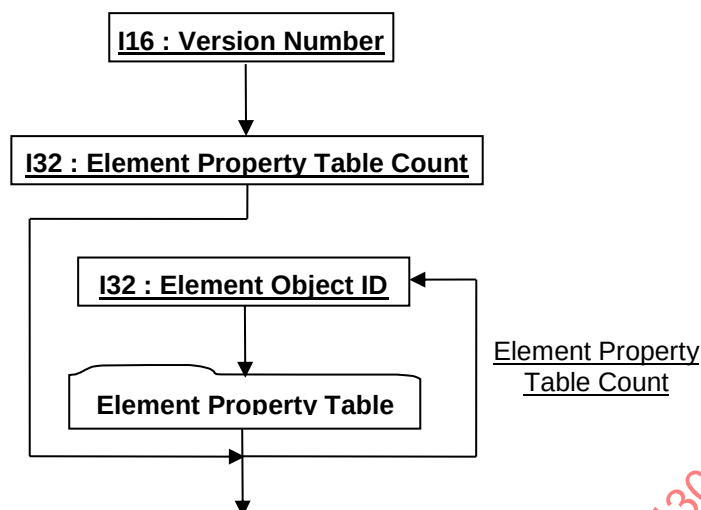


Figure 87 — Property Table data collection

I16 : Version Number

Version Number is the version identifier for this Property Table. For information on local version numbers see best practice **11.5 Local version numbers**.

I32 : Element Property Table Count

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

I32 : Element Object ID

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

6.3.1 Element Property Table

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

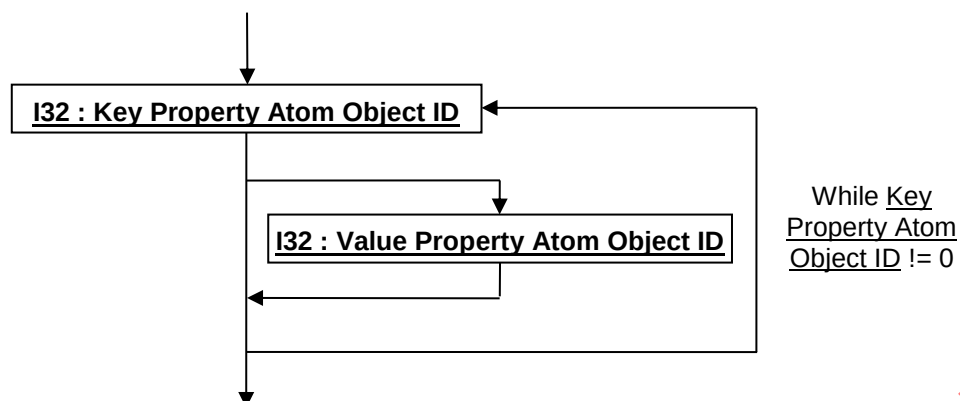


Figure 88 — Element Property Table data collection

I32 : Key Property Atom Object ID

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

I32 : Value Property Atom Object ID

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

7 Shape LOD Segment

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

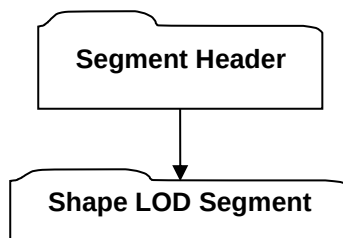


Figure 89 — Shape LOD Segment data collection

7.1 Shape LOD Element

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

There are several types of Shape LOD Elements which the ISO 14306 format supports. The following sub-sections document the various Shape LOD Element types.

7.1.1 Base Shape LOD Element

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

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

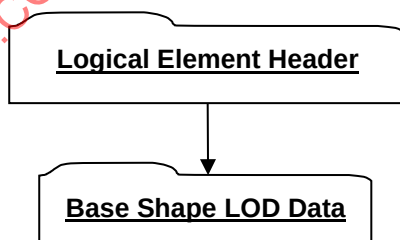


Figure 90 — Base Shape LOD Element data collection

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

7.1.1.1 Base Shape LOD Data

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

I16 : Version Number

Figure 91 — Base Shape LOD Data collection

I16 : Version Number

Version Number is the version identifier for this Base Shape LOD Data. For information on local version numbers see best practice **11.5 Local version numbers**.

7.1.2 Vertex Shape LOD Element

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

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

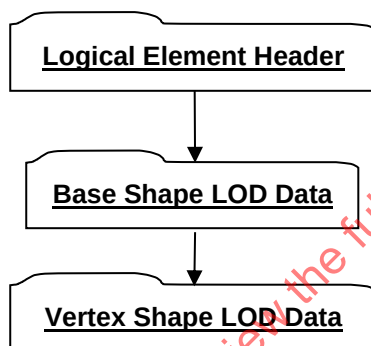


Figure 92 — Vertex Shape LOD Element data collection

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

7.1.2.1 Vertex Shape LOD Data

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

One use for face groups is to establish a correspondence between Brep faces and their triangle representation. A convention for mapping JT Brep and XT Brep faces to face groups is described in section **11.12 Brep Face Group Associations**.

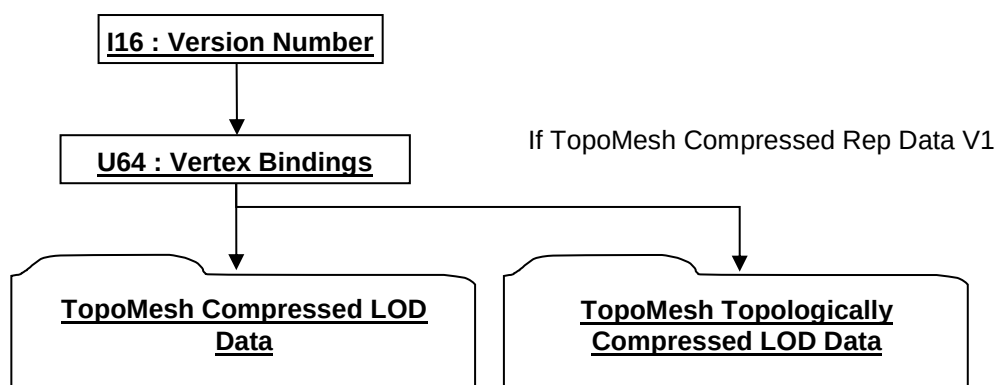


Figure 93 — Vertex Shape LOD Data collection

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

I16 : Version Number

Version Number is the version identifier for this Vertex Shape LOD Data. For information on local version numbers see best practice 11.5 Local version numbers.

U64 : Vertex Bindings

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

Table 56 — Vertex Shape LOD Bindings values

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

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

7.1.2.2 TopoMesh LOD Data

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

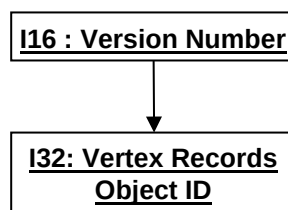


Figure 94 — TopoMesh LOD Data collection

I16 : Version Number

Version Number is the version identifier for this TopoMesh LOD Data. For information on local version numbers see best practice [11.5 Local version numbers](#).

I32: Vertex Records Object ID

Vertex Records Object ID is the identifier for the vertex records associated with this Object. Other objects referencing these vertex records will do so using this Object ID.

7.1.2.2.1 TopoMesh Compressed LOD Data

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

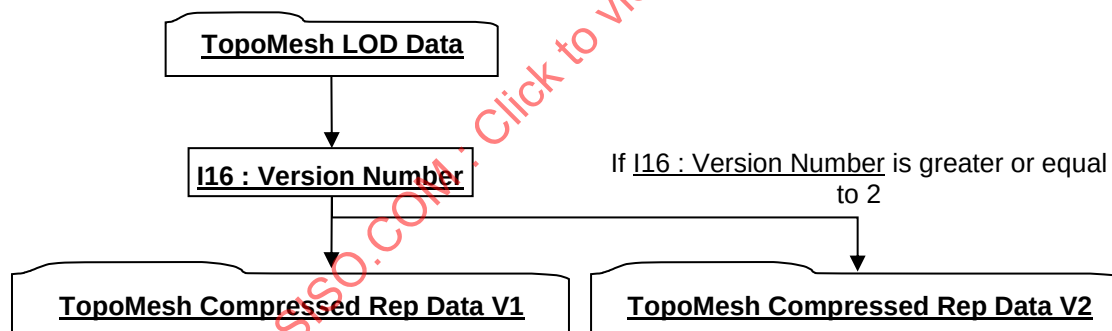


Figure 95 — TopoMesh LOD Data collection

Complete description for TopoMesh LOD Data, TopoMesh Compressed Rep Data V1, and TopoMesh Compressed Rep Data V2 can be found in [TopoMesh LOD Data](#), [TopoMesh Compressed Rep Data V1](#), and [TopoMesh Compressed Rep Data V2](#).

I16 : Version Number

Version Number is the version identifier for this TopoMesh LOD Data. For information on local version numbers see best practice [11.5 Local version numbers](#).

7.1.2.2.2 TopoMesh Topologically Compressed LOD Data

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

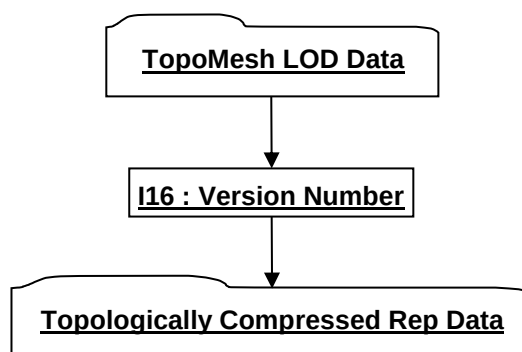


Figure 96 — TopoMesh Topologically Compressed LOD Data collection

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

I16 : Version Number

Version Number is the version identifier for this TopoMesh Topologically Compressed LOD Data. For information on local version numbers see best practice 11.5 Local version numbers.

7.1.2.2.3 Topologically Compressed Rep Data

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

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

To begin the decoding process, first read the compressed data fields shown in Figure 97. These fields provide all the information necessary to reconstruct the per face-group organized sets of triangles. The first 22 fields represent the topological information, and the remaining fields constitute the set of unique vertex records to be used. The next step is to run the topological decoder algorithm detailed in Annex D on this data to reconstruct the topologically connected representation of the triangle mesh in a so-called "dual VFMesh." The triangles in this heavy-weight data structure can then be exported to a lighter-weight form, and the dual VFMesh discarded if desired.

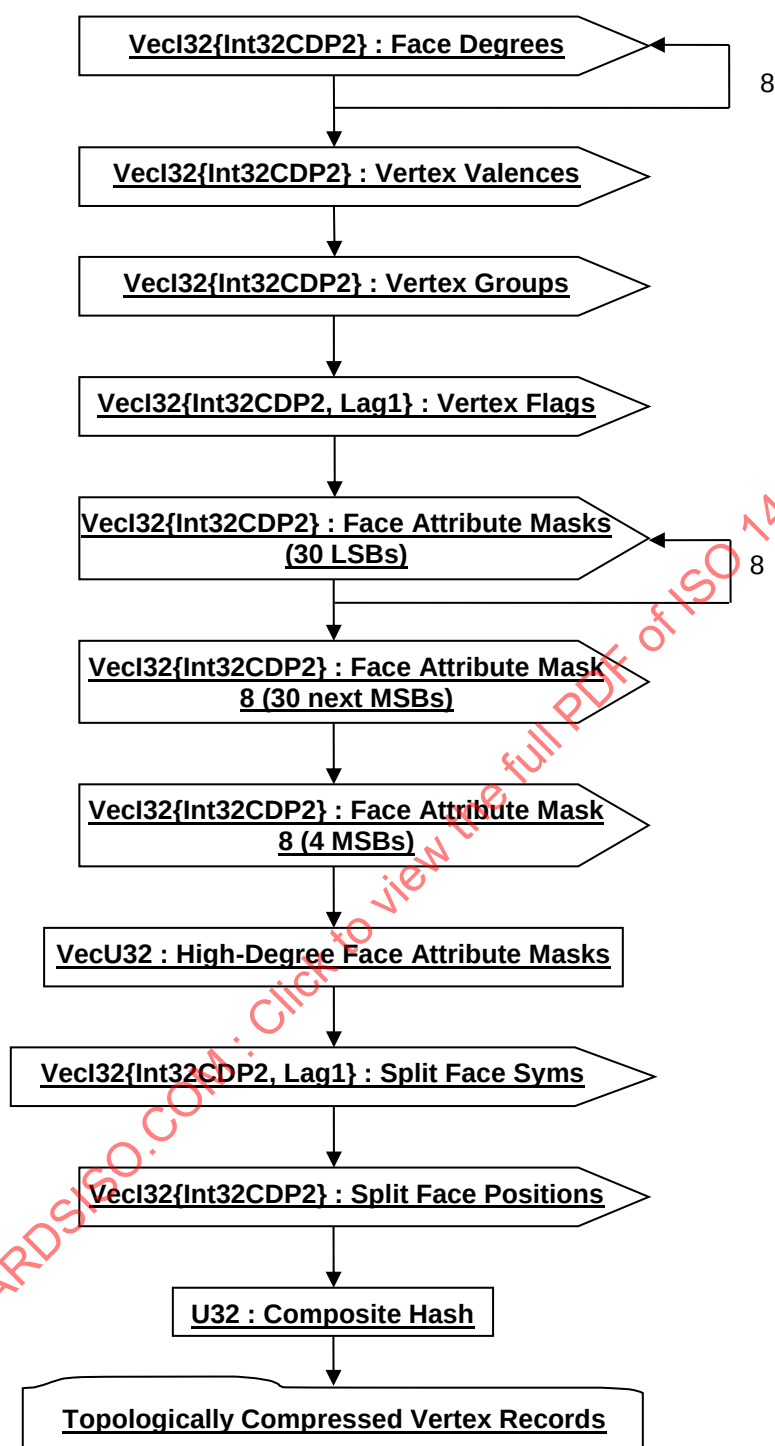


Figure 97 — Topologically Compressed Rep Data Collection

VecI32{Int32CDP2} : Face Degrees

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

VecI32{Int32CDP2} : Vertex Valences

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

VecI32{Int32CDP2} : Vertex Groups

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

VecI32{Int32CDP2, Lag1} : Vertex Flags

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

VecI32{Int32CDP2} : Face Attribute Masks (30 LSBs)

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

VecI32{Int32CDP2} : Face Attribute Mask 8 (30 next MSBs)

This field encodes the next 30 most significant bits of the 8th context group of face attribute bit vectors.

VecI32{Int32CDP2} : Face Attribute Mask 8 (4 MSBs)

This field encodes the 4 most significant bits of the 8th context group of face attribute bit vectors, rounding out its full 64-bit width.

VecU32 : High-Degree Face Attribute Masks

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

VecI32{Int32CDP2, Lag1} : Split Face Syms

Encodes the list of “split face” ID numbers in the order the coder encountered them.

VecI32{Int32CDP2} : Split Face Positions

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

U32 : Composite Hash

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

```
UInt32 uHash          = 0;
UInt32 anDegSyms[8]   = {0},
      nValSyms = 0,
      nVGrpSyms = 0,
      nVtxFlags = 0,
      anAttrMasks[8] = {0},
      nLrgAttrMasks = 0,
      nSplitVtxSyms = 0,
      nSplitVtxPos = 0;
VecI32 vFaceDegreeSymbols[8], vviValenceSymbols, vFaceGroupSyms,
```

```

        vvuAttrMasks[8], viSplitVtxSyms, viSplitVtxPos;
VecI16 vFaceFlags;
VecU32 vuTmp, vuAttrMasksLrg;
...
for (i=0 ; i<8 ;i++)
    uHash = hash32((UInt32*) vFaceDegreeSymbols[i].ptr(), anDegSyms[i], uHash );
uHash = hash32((UInt32*) vviValenceSymbols.ptr(), nValSyms, uHash );
uHash = hash32((UInt32*) vVtxGroupSyms.ptr(), nGrpSyms, uHash );
uHash = hash16((UInt16*) vVtxFlags.ptr(), nFlags, uHash );
for (i=0 ; i<7 ;i++)
    uHash = hash32((UInt32*) vvuAttrMasks[i].ptr(), anAttrMasks[i], uHash );
vuTmp = vvuAttrMasks[7] & 0x3fffffff; // Lower 30 bits of each element
uHash = hash32(vuTmp.ptr(), anAttrMasks[7], uHash );
vuTmp = (vvuAttrMasks[7] >> 30) & 0x3fffffff; // Next 30 bits of each element
uHash = hash32(vuTmp.ptr(), anAttrMasks[7], uHash );
vuTmp = (vvuAttrMasks[7] >> 60) & 0x0f; // Upper 4 bits of each element
uHash = hash32(vuTmp.ptr(), anAttrMasks[7], uHash );
uHash = hash32(vuAttrMasksLrg.ptr(), nLrgAttrMasks, uHash );
uHash = hash32((UInt32*) viSplitVtxSyms.ptr(), nSplitVtxSyms, uHash );
uHash = hash32((UInt32*) viSplitVtxPos.ptr(), nSplitVtxPos, uHash );

```

7.1.2.3 Topologically Compressed Vertex Records

Documented here is the format of the vertex data written by the topological encoder from Annex F. Some additional explanation is necessary, however, because only the *unique* vertex coordinates are written to the JT file, while the remaining vertex attributes (normals, colours, texture coordinates, vertex flags) may not be unique.

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

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

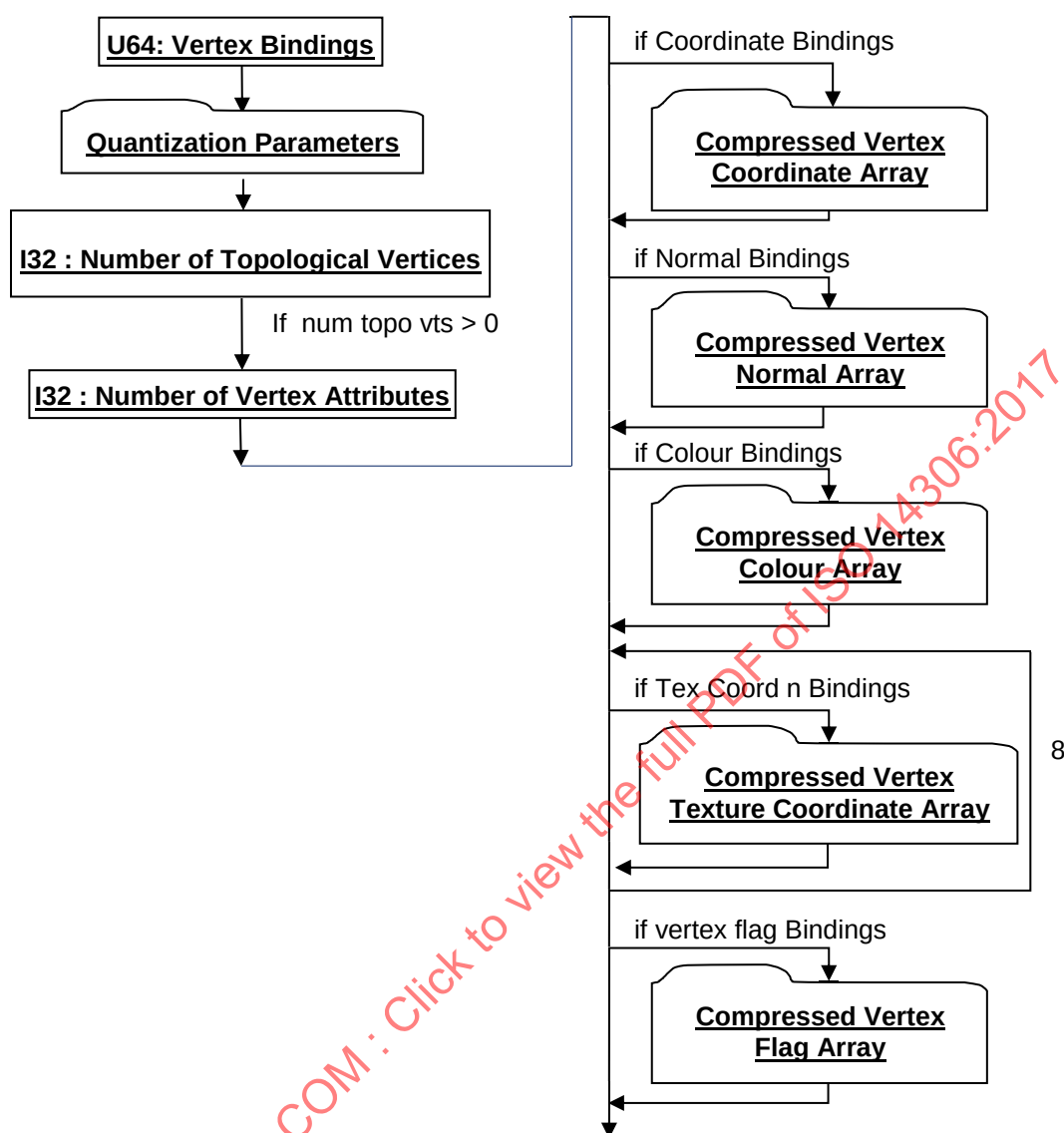


Figure 98 — Topologically Compressed Vertex Records data collection

U64: Vertex Bindings

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

I32: Number of Topological Vertices

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

I32: Number of Vertex Attributes

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

7.1.2.4 TopoMesh Compressed Rep Data V1

TopoMesh Compressed Rep Data V1 contains the geometric shape definition data (e.g. vertices, colours, normals, etc.) in a lossy or lossless compressed formed.

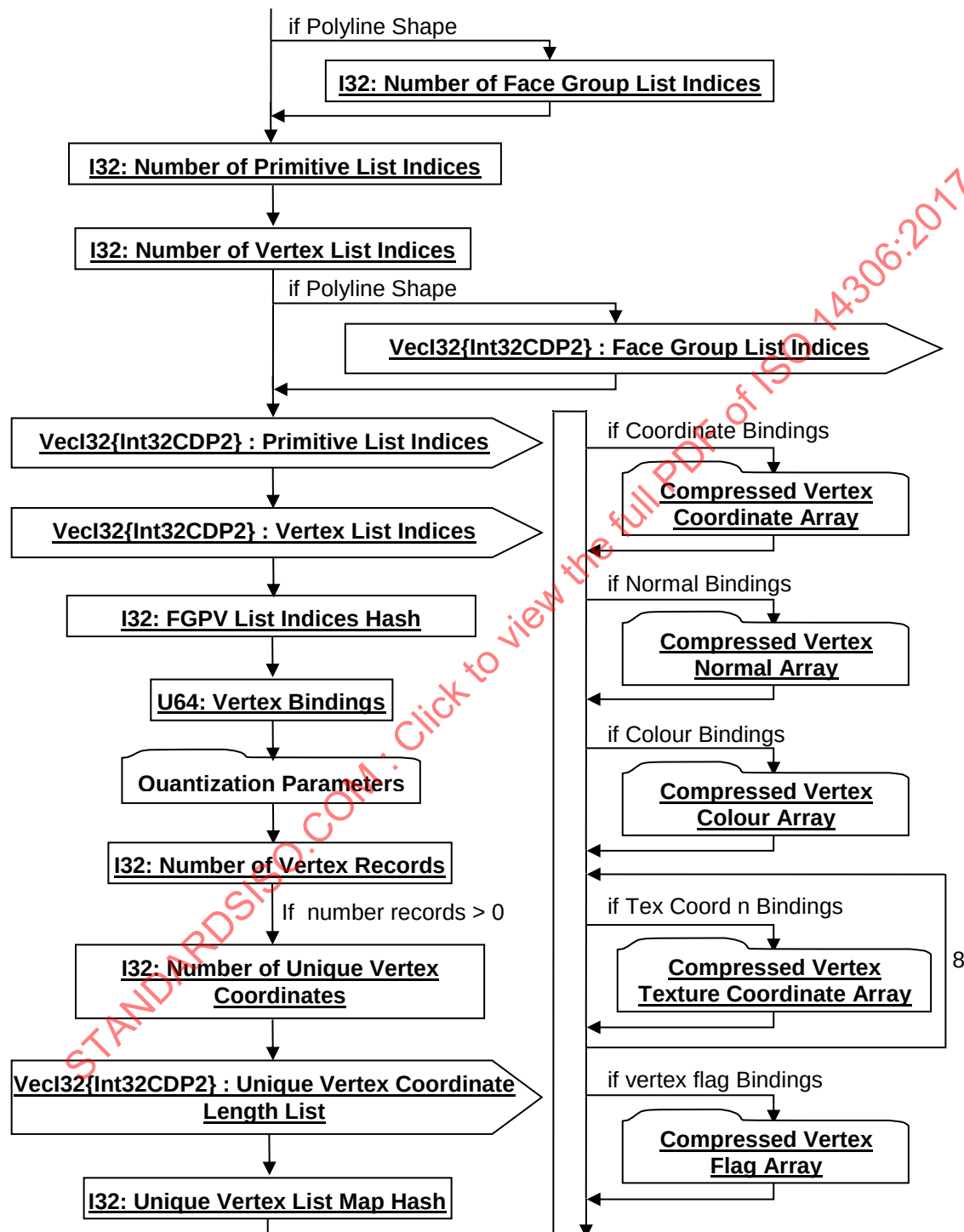


Figure 99 — TopoMesh Compressed Rep Data V1 data collection

Complete description for Quantization Parameters can be found in [Quantization Parameters](#).

I32: Number of Face Group List Indices

Number of Face Group List Indices.

I32: Number of Primitive List Indices

Number of Primitive List Indices.

I32: Number of Vertex List Indices

Number of Vertex List Indices.

VecI32{Int32CDP2} : Face Group List Indices

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

VecI32{Int32CDP2} : Primitive List Indices

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

VecI32{Int32CDP2} : Vertex List Indices

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

I32: FGPV List Indices Hash

The FGPV Hash is the combined hash value of the Face Group List Indices (if Polyline), Primitive List Indices, and Vertex List Indices. Refer to section **11.6** for a more detailed description on hashing.

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

U64: Vertex Bindings

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

I32: Number of Vertex Records

Number of vertex records.

I32: Number of Unique Vertex Coordinates

Number of unique vertex coordinates values in the Compressed Vertex Coordinate Array.

VecI32{Int32CDP2} : Unique Vertex Coordinate Length List

The Unique Vertex Length List contains the number of vertex records containing each of the unique vertex coordinates and should sum to the number of vertex records. When read in the Compressed

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

I32: Unique Vertex List Map Hash

The Unique Vertex List Map Hash is the hash value of Unique Vertex Coordinate Length List. Refer to section 11.6 for a more detailed description on hashing.

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

7.1.2.5 TopoMesh Compressed Rep Data V2

TopoMesh Compressed Rep Data V2 data contains additional geometric shape data (auxiliary vertex fields) that were not included in V1. Each Auxiliary field contains data that is parallel to the existing vertex record fields in order capture additional information about each vertex (e.g. Vertex Identifiers, Weights, ...).

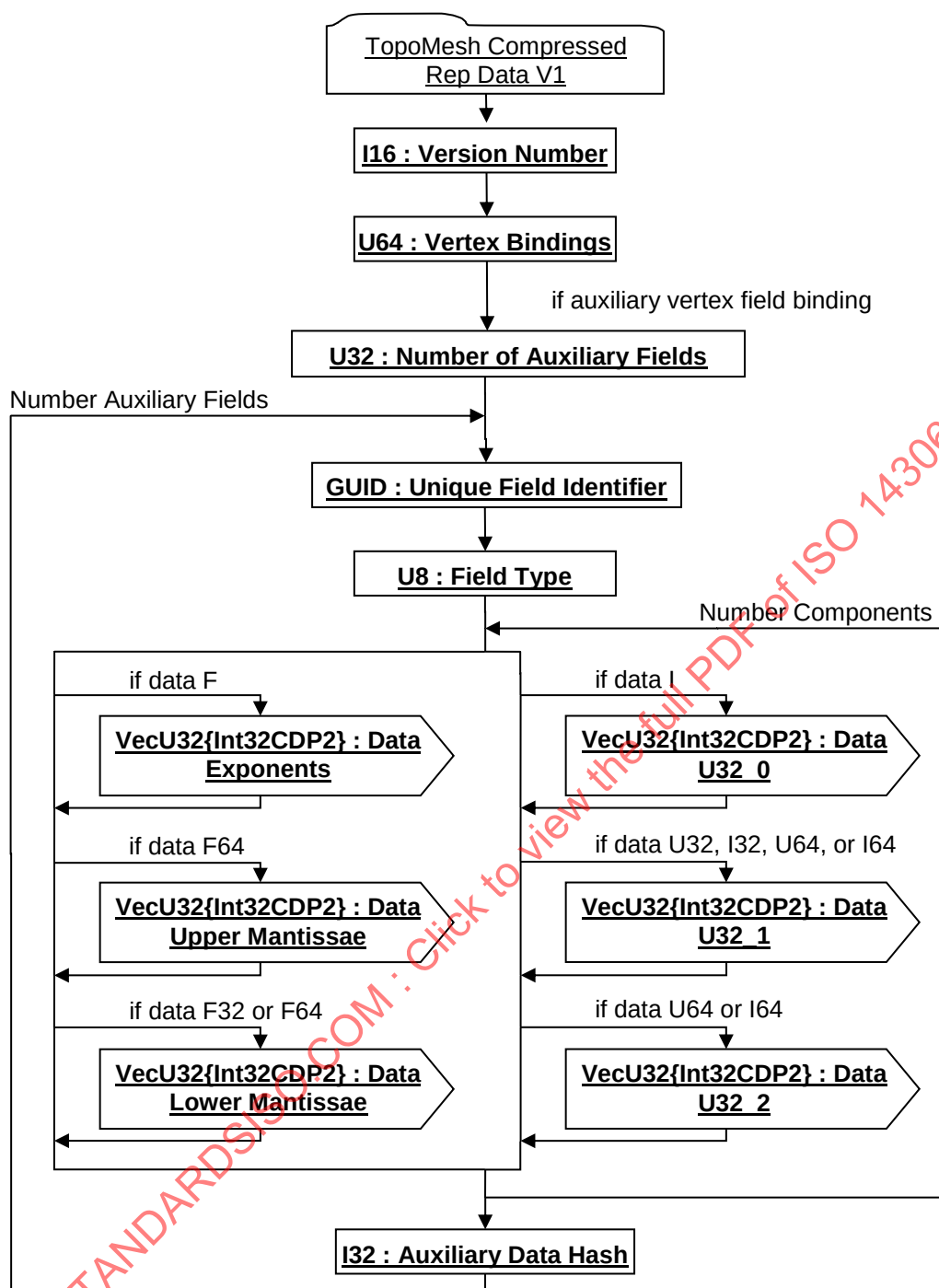


Figure 100 — TopoMesh Compressed Rep Data V2 data collection

Complete description for TopoMesh Compressed Rep Data V1 can be found in TopoMesh Compressed Rep Data V1.

I16 : Version Number

Version Number is the version identifier for this TopoMesh Compressed Rep Data V2. For information on local version numbers see best practice 11.5 Local version numbers.

U64 : Vertex Bindings

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

U32 : Number of Auxiliary Fields

Number of Auxiliary Vertex Fields

GUID : Unique Field Identifier

Each Auxiliary Vertex Field is associated with Unique Field Identifier to denote the usage of the contained data. This field is intended to be user defined. Any valid **GUID**, as defined in Table 4 - Composite Data Types, can be used as a Unique Field Identifier.

U8 : Field Type

Defines the number of components and type of data contained within the auxiliary field based upon the below table.

Table 57 — TopoMesh Compressed Rep Data V2 Field Type values

Type	Data	Components	Type	Data	Components
1	U8	1	24	I32	4
2	U8	2	25	U64	1
3	U8	3	26	U64	2
4	U8	4	27	U64	3
5	I8	1	28	U64	4
6	I8	2	29	I64	1
7	I8	3	30	I64	2
8	I8	4	31	I64	3
9	U16	1	32	I64	4
10	U16	2	33	F32	1
11	U16	3	34	F32	2
12	U16	4	35	F32	3
13	I16	1	36	F32	4
14	I16	2	37	F32	2x2
15	I16	3	38	F32	3x3
16	I16	4	39	F32	4x4
17	U32	1	40	F64	1
18	U32	2	41	F64	2
19	U32	3	42	F64	3
20	U32	4	43	F64	4
21	I32	1	44	F64	2x2
22	I32	2	45	F64	3x3
23	I32	3	46	F64	4x4

VecU32{Int32CDP2} : Data U32_0

Data U32_0 contains the low order bits from the data i'th data component for each vertex record in an U32 vector. For U8, I8, U16, and I16 data types this contains all bits. For U32, I32, U64, and I64 data types it contains bits 0 through 30. Data U32_0 uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Data U32_1

Data U32_1 contains the middle order bits from the data i'th data component for each vertex record in an U32 vector. For U32 and I32 data types it only contains bit 31. For U64 and I64 data types it contains bits 31 through 61. Data U32_1 uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Data U32_2

Data U32_2 contains the upper order bits from the data i'th data component for each vertex record in an U32 vector. For U64 and I64 data types it contains bits 62 and 63. Data U32_2 uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Data Lower Mantissae

Vertex Coord Components is a vector of lower bits of Floating Point Mantissae for all the i'th component values of a set of vertex coordinates. For F32 data type this contains all bits of the mantissa, however for F64 data type it only contains bits 0 through 30. Data Lower Mantissae uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Data Upper Mantissae

Vertex Coord Components is a vector of upper bits of the Floating Point Mantissae for all the i'th component values of a set of vertex coordinates. For the F64 data type it contains bits 31 through 51. Data Upper Mantissae uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Data Exponents

Vertex Coord Components is a vector of Floating Point Exponents and Sign for all the i'th component values of a set of vertex coordinates. Data Exponents uses the Int32 version of the CODEC to compress and encode data.

I32 : Auxiliary Data Hash

The Auxiliary Data Hash is the combined hash of auxiliary field data arrays. Refer to section 11.6 for a more detailed description on hashing.

```

UInt32 uHash      = 0;
UInt32 nVtxRec    = 0,
      nComp       = 0;
vecU32 vU32_0, vU32_1, vU32_2, vLMANT, vUMANT, vEXP;
...
if ( bU8 || bI8 || bU16 || bI16 ) {
    for ( int i=0 ; i<nComp ; i++ ) {
        uHash = hash32( &vU32_0[i], nVtxRec, uHash );
    }
} else if ( bU32 || bI32 ) {
    for ( int i=0 ; i<nComp ; i++ ) {
        uHash = hash32( &vU32_0[i], nVtxRec, uHash );
        uHash = hash32( &vU32_1[i], nVtxRec, uHash );
    }
} else if ( bU64 || bI64 ) {
    for ( int i=0 ; i<nComp ; i++ ) {
        uHash = hash32( &vU32_0[i], nVtxRec, uHash );
        uHash = hash32( &vU32_1[i], nVtxRec, uHash );
        uHash = hash32( &vU32_2[i], nVtxRec, uHash );
    }
} else if ( bF32 ) {
    for ( int i=0 ; i<nComp ; i++ ) {
        uHash = hash32( &vLMANT[i], nVtxRec, uHash );
        uHash = hash32( &vEXP[i], nVtxRec, uHash );
    }
} else {
    for ( int i=0 ; i<nComp ; i++ ) {
        uHash = hash32( &vLMANT[i], nVtxRec, uHash );
    }
}

```

```

    uHash = hash32( &vUMANT[i], nVtxRec, uHash );
    uHash = hash32( &vEXP[i], nVtxRec, uHash );
}
}

```

7.1.3 Tri-Strip Set Shape LOD Element

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

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

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

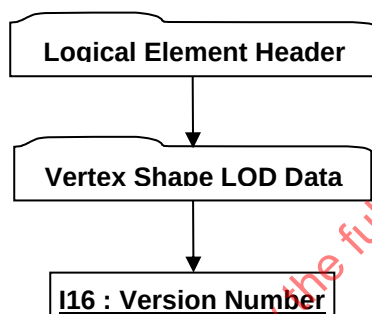


Figure 101 — Tri-Strip Set Shape LOD Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this Tri-Strip Set Shape LOD. For information on local version numbers see best practice 11.5 Local version numbers.

7.1.4 Polyline Set Shape LOD Element

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

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

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

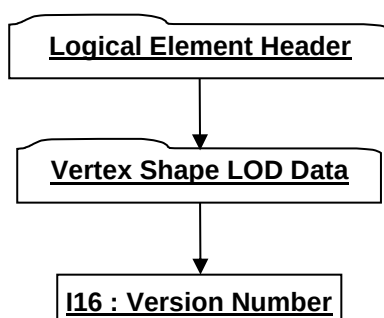


Figure 102 — Polyline Set Shape LOD Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this Polyline Set Shape LOD. For information on local version numbers see best practice 11.5 Local version numbers.

7.1.5 Point Set Shape LOD Element

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

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

A Point Set Shape LOD Element is typically referenced by a Point Set Shape Node Element using Late Loaded Property Atom Elements (see 6.1.1 10.5 Point Set Shape Node Element and 6.2.7 Late Loaded Property Atom Element respectively).

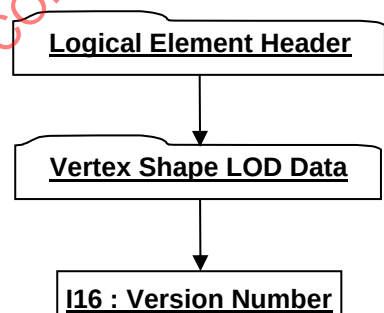


Figure 103 — Point Set Shape LOD Element data collection

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

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

I16 : Version Number

Version Number is the version identifier for this Point Set Shape LOD. For information on local version numbers see best practice 11.5 Local version numbers.

7.1.6 Null Shape LOD Element

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

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

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

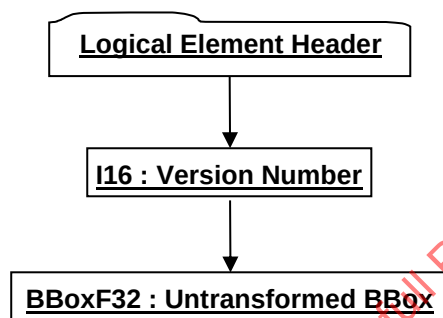


Figure 104 — Null Shape LOD Element data collection

Complete description for **Logical Element Header** can be found in [Logical Element Header](#).

I16 : Version Number

Version Number is the version identifier for this Null Shape LOD Element. For information on local version numbers see best practice [11.5 Local version numbers](#).

BBoxF32 : Untransformed BBox

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

7.2 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 centre and radius) from which LODs can be generated.

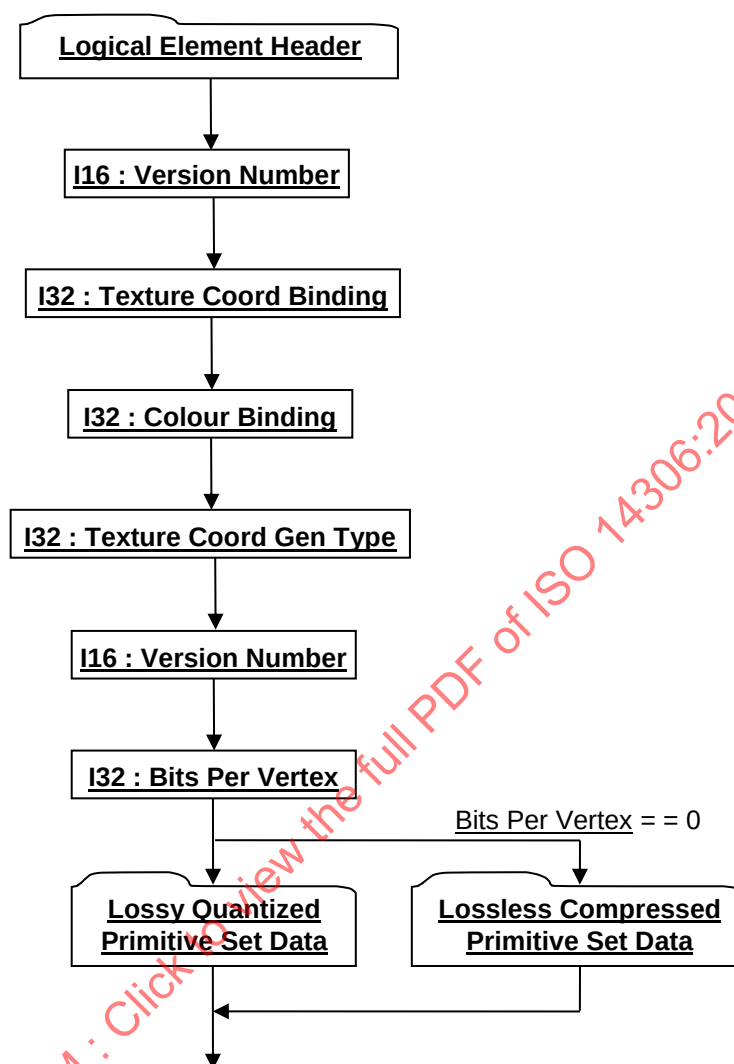


Figure 105 — Primitive Set Shape Element data collection

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

I16 : Version Number

Version Number is the version identifier for this element. For information on local version numbers see best practice **11.5 Local version numbers**.

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:

Table 58 — Primitive Set Shape Texture Coord Binding values

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

I32 : Colour Binding

Colour Binding specifies how (at what granularity) colour 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.

Table 59 — Primitive Set Shape Version Number values

= 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 shall be within range [0:32] inclusive.

I32 : Texture Coord Gen Type

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

Table 60 — Primitive Set Shape Texture Coord Gen Type values

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

7.2.1 Lossless Compressed Primitive Set Data

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

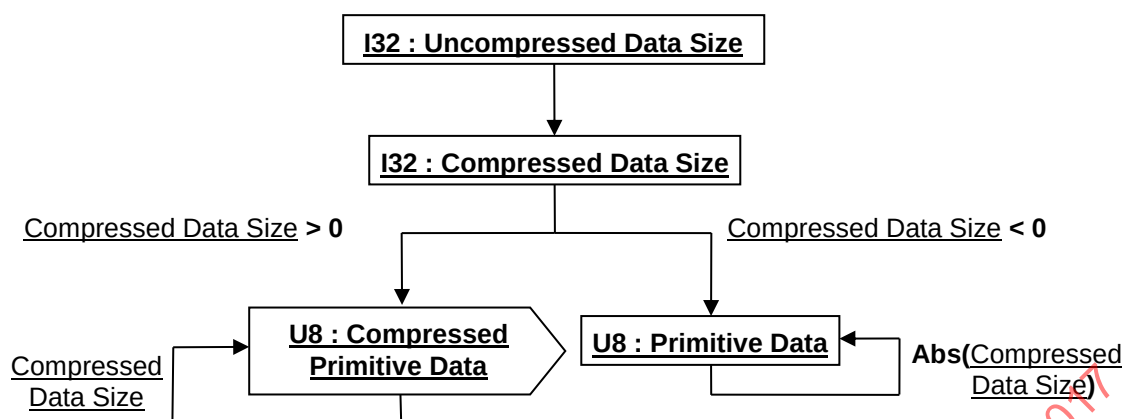


Figure 106 — Lossless Compressed Primitive Set Data collection

I32 : Uncompressed Data Size

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

I32 : Compressed Data Size

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

U8 : Primitive Data

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

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

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

Where the data elements have the following size and meaning:

Table 61 — Lossless Compressed Primitive Set Data Field values

Element	Data Type	Description
reserved	I32	This is a field reserved for future expansion of the ISO 14306 format.
params1	CoordF32	Interpretation is Primitive Type specific (see below table)
params2	DirF32	Interpretation is Primitive Type specific (see below table)
params3	Quaternion	Interpretation is Primitive Type specific (see below table)
colour	RGB	Red, Green, Blue colour component values
type	I32	Primitive Type = 0 – Box = 1 – Cylinder = 2 – Pyramid = 3 – Sphere = 4 – Tri-Prism

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

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

Table 62 — Primitive Set “params#” Data Fields Interpretation

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

U8 : Compressed Primitive Data

The Compressed Primitive Data field represents the same data as documented in Primitive Data field above except that the data is compressed using the general “ZLIB deflation compression” method. The Compressed Primitive Data field is only present if Compressed Data Size value is greater than zero. See 10 Data Compression and Encoding for more details on ZLIB compression and ZLIB library version used.

7.2.2 Lossy Quantized Primitive Set Data

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

The interpretation of the three per-primitive “params#” data fields is primitive type dependent. See Table 62 — Primitive Set “params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the “params#” data fields.

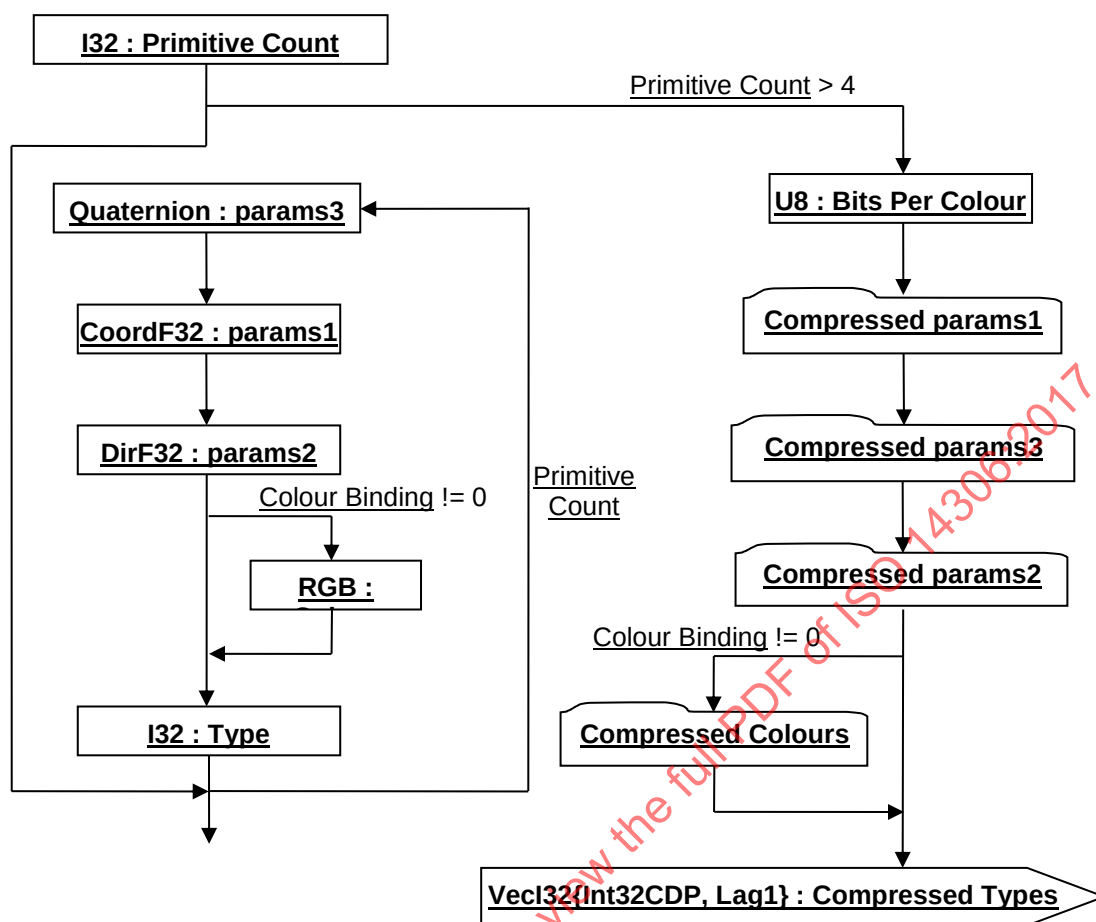


Figure 107 — Lossy Quantized Primitive Set Data collection

I32 : Primitive Count

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

Quaternion : params3

Interpretation of params3 data field is primitive Type dependent. See Table 62 — Primitive Set “params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the params3 data fields.

CoordF32 : params1

Interpretation of params1 data field is primitive Type dependent. See Table 62 — Primitive Set “params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the params1 data fields.

DirF32 : params2

Interpretation of params1 data field is primitive Type dependent. See Table 62 — Primitive Set “params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the params1 data fields.

RGB : Colour

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

I32 : Type

Type specifies the primitive type. See Table 61 — Lossless Compressed Primitive Set Data Field values in Lossless Compressed Primitive Set Data for valid primitive Type values.

U8 : Bits Per Colour

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

VecI32{Int32CDP, Lag1} : Compressed Types

The Compressed Types data field is a vector of Type data for all the primitives in the Primitive Set. Compressed Types uses the Int32 version of the CODEC to compress and encode data. In an uncompressed form the valid primitive Type values are as documented in Table 61 — Lossless Compressed Primitive Set Data Field values in Lossless Compressed Primitive Set Data.

7.2.2.1 Compressed params1

Compressed params1 is the compressed representation of the *params1* data for all the primitives in the Primitive Set. Note that the interpretation of the uncompressed *params1* data is primitive Type dependent. See Table 62 — Primitive Set “params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data for per-primitive type description of the *params1* data fields.

The *params1* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Vertex number of quantization bits) for each collection of ordinate values. Since *params1* is of type “CoordF32”, it has three ordinate values (three F32 values), and thus three Uniform Quantizers (where a Uniform Quantizer is a scalar quantizer/encoder whose range is divided into levels of equal spacing). See 10 Data Compression and Encoding for more complete description of Uniform Quantizer.

The ISO 14306 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.

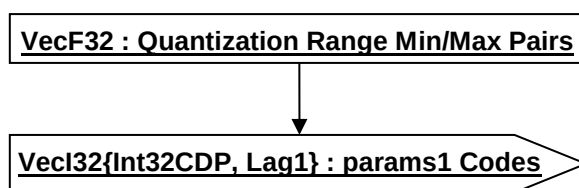


Figure 108 — Compressed params1 data collection

VecF32 : Quantization Range Min/Max Pairs

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

VecI32{Int32CDP, Lag1} : params1 Codes

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

7.2.2.2 Compressed params3

Compressed *params3* is the compressed representation of the *params3* data for all the primitives in the Primitive Set. Note that the interpretation of the uncompressed *param31* data is primitive Type dependent. See [Table 62 — Primitive Set “params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data](#) for per-primitive type description of the *params3* data fields.

The *params3* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Vertex number of quantization bits) for each collection of ordinate values. Since *params1* is of type “Quaternion”, it has four ordinate values (four F32 values), and thus four Uniform Quantizers (where a Uniform Quantizer is a scalar quantizer/encoder whose range is divided into levels of equal spacing). See [10 Data Compression and Encoding](#) for more complete description of Uniform Quantizer.

The ISO 14306 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 108 — Compressed params1 data collection](#).

7.2.2.3 Compressed params2

Compressed *params2* is the compressed representation of the *params2* data for all the primitives in the Primitive Set. Note that the interpretation of the uncompressed *params2* data is primitive Type dependent. See [Table 62 — Primitive Set “params#” Data Fields Interpretation in Lossless Compressed Primitive Set Data](#) for per-primitive type description of the *params2* data fields.

The *params2* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Vertex number of quantization bits) for each collection of ordinate values. Since *params2* is of type “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 10 Data Compression and Encoding for more complete description of Uniform Quantizer.

The ISO 14306 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 108 — Compressed *params1* data collection.

7.2.2.4 Compressed Colours

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

The *colour* data for all primitives in the Primitive Set is compressed/encoded on a per ordinate basis using a separate Uniform Quantizer (with Bits Per Colour number of quantization bits) for each collection of ordinate values. Since *colour* is of type “RGB”, it has three ordinate values (three F32 values), and thus three Uniform Quantizers (where a Uniform Quantizer is a scalar quantizer/encoder whose range is divided into levels of equal spacing). See 10 Data Compression and Encoding for more complete description of Uniform Quantizer.

The ISO 14306 format packs all the *colour* data for all primitives into a single array using an ordinate dependent order (as shown below) and then encodes each of the lists of ordinate values using a separate Uniform Quantizer per ordinate list.

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

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

The storage format of Compressed Colours is exactly the same as that documented in Figure 108 — Compressed *params1* data collection.

8 Geometry Segments

The geometry segment contains an Element that defines the precise geometric Boundary Representation data for a particular part. There are currently 6 geometry section options for an ISO 14306 file; STEP, XT B-Rep, ULP, LWPA, Wireframe and JT B-Rep (deprecated).

Part Node Elements may reference a geometry segment as Late Loaded Property Atom Elements.

The geometry segments supports ZLIB compression on all element data, so all elements in a geometry segment use the Logical Element Header ZLIB form of element header data

8.1 STEP B-Rep Segment

The STEP B-Rep Segment defines the precise geometry of a part or an assembly of parts in STEP boundary representation normative format and shall be as defined in ISO 10303-42.

The STEP B-Rep segment contains a compressed STEP file which can be produced and processed by a STEP processor. This includes all necessary product structure and property information to constitute a valid STEP file.

The schema version of the embedded STEP file in the STEP B-Rep segment shall be identified by the version field defined in Annex B.

The file format shall be encoded according to the syntax and EXPRESS language mapping defined in ISO 10303-21 and to the iso_14306_schema EXPRESS specified in Annex C.

8.2 XT B-Rep Segment

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

See Annex G for a full description of the XT B-Rep Segment.

8.3 JT ULP Segment

JT ULP Segment contains an Element that defines the semi-precise geometric Boundary Representation data for a particular Part in JT ULP format.

See Annex M for a full description of the JT ULP Segment.

8.4 JT LWPA Segment

JT LWPA Segment contains an Element that defines light weight precise analytic data for a particular part. More specifically LWPA contains the collection of analytic surfaces in the B-Rep definition of the part.

See Annex M section 2 JT LWPA Segment for a full description of the LWPA Segment.

8.5 Wireframe Segment

Wireframe Segment contains an Element that defines the precise 3D wireframe data for a particular Part.

See Annex L for a full description of the Wireframe Segment.

8.6 JT B-Rep Element (deprecated)

JT B-Rep Element represents a particular Part's precise data in JT boundary representation format.

See Annex K for a full description of the JT B-Rep Segment.

NOTE: JT B-Rep is deprecated and should be considered read only for application creation.

STANDARDSISO.COM : Click to view the full PDF of ISO 14306:2017

9 Meta Data Segment

Meta Data Segments are used to store large collections of meta-data in separate addressable segments of the JT File. Storing meta-data in a separate addressable segment allows references (from within the JT file) to these segments to be constructed such that the meta-data can be late-loaded (i.e. JT file reader can be structured to support the “best practice” of delaying the loading/reading of the referenced meta-data segment until it is actually needed).

Meta Data Segments are typically referenced by Part Node Elements (see [6.1.1.5 Part Node Element](#)) using Late Loaded Property Atom Elements (see [6.2.7 Late Loaded Property Atom Element](#)).

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

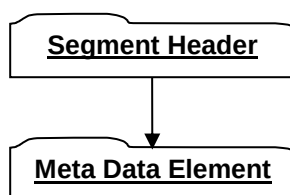


Figure 109 — Meta Data Segment data collection

Complete description for **Segment Header** can be found in [5.1.3.1 Segment Header](#).

The following sub-sections document the various [I32 : Texture](#) Coord Channel types.

9.1 Property Proxy Meta Data Element

Object Type ID: 0xce357247, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1

A Property Proxy Meta Data Element serves as a “proxy” for all meta-data properties associated with a particular Meta Data Node Element (see [6.1.1.6 Meta Data Node Element](#)). The proxy is in the form of a list of key/value property pairs where the *key* identifies the type and meaning of the *value*. Although the property *key* is always in the form of a String data type, the *value* can be one of several data types.

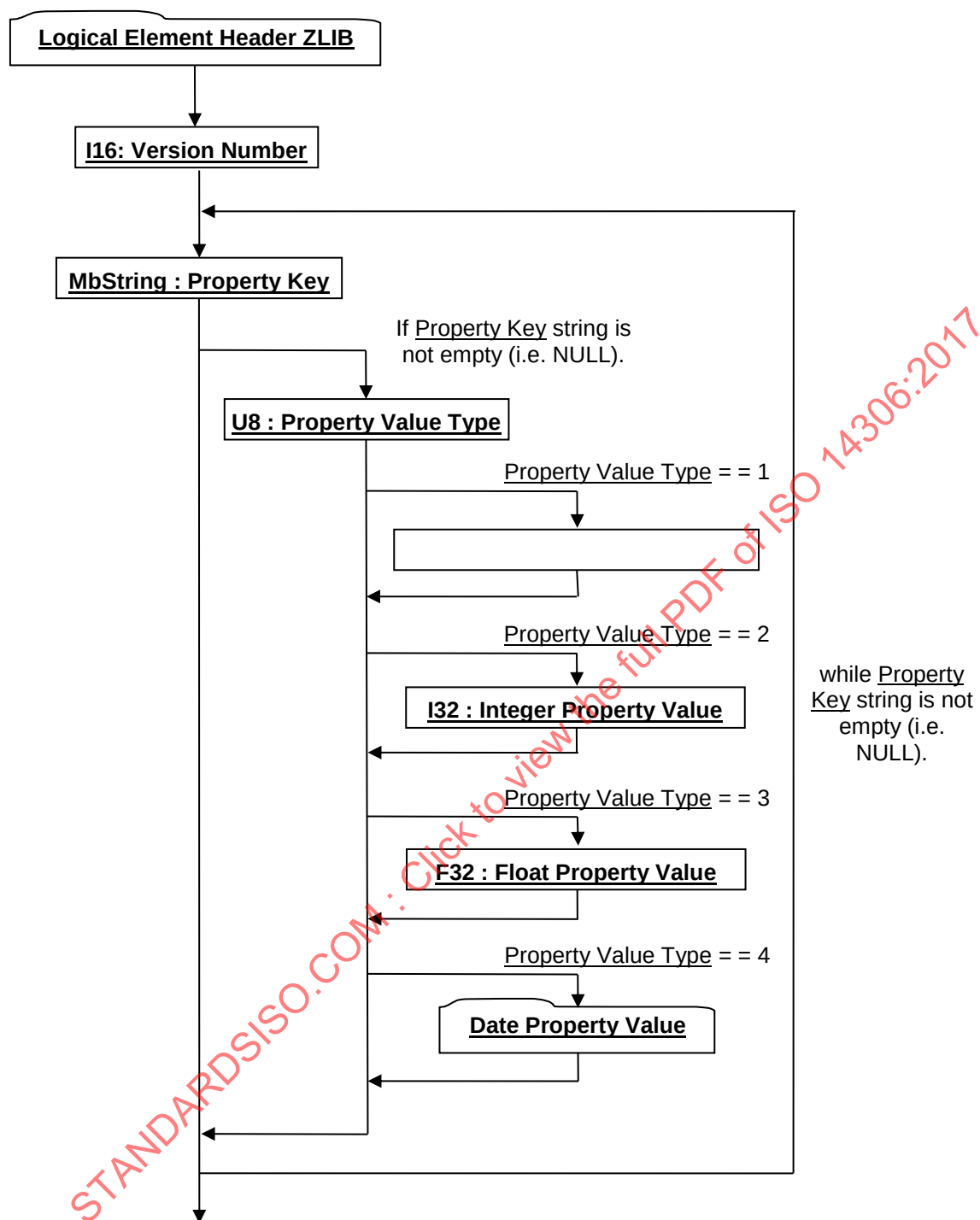


Figure 110 — Meta Data Segment data collection

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

I16: Version Number

Version Number is the version identifier for this data collection. For information on local version numbers see best practice **11.5 Local version numbers**.

MbString : Property Key

Property Key specifies the *key* string for the property.

U8 : Property Value Type

Property Value Type specifies the data type for the Property Value. If the type equals “0” then no Property Value is written. Valid types include the following:

Table 63 — Property Proxy Meta Data Property Value Type values

= 0	Unknown
= 1	MbString data type value
= 2	I32 data type value
= 3	F32 data type value
= 4	Date value

MbString : String Property Value

String Property Value represents the property value when Property Value Type = = 1.

I32 : Integer Property Value

Integer Property Value represents the property value when Property Value Type = = 2.

F32 : Float Property Value

Float Property Value represents the property value when Property Value Type = = 3.

9.1.1 Date Property Value

Date Property Value represents the property value when Property Value Type = = 4. Date Property Value data collection represents a date as a combination of year, month, day, hour, minute, and second data fields.

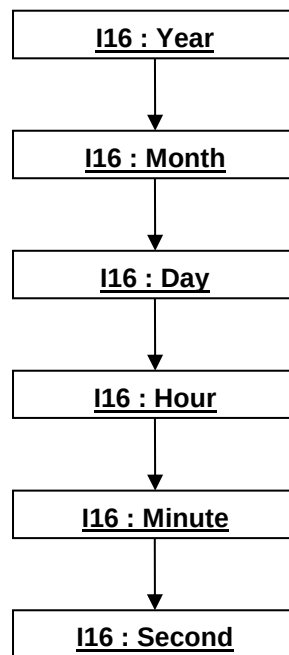


Figure 111 — Date Property Value data collection

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.

9.2 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 I32 : Texture Coord Channel which contains the Product and Manufacturing Information for a part/assembly.

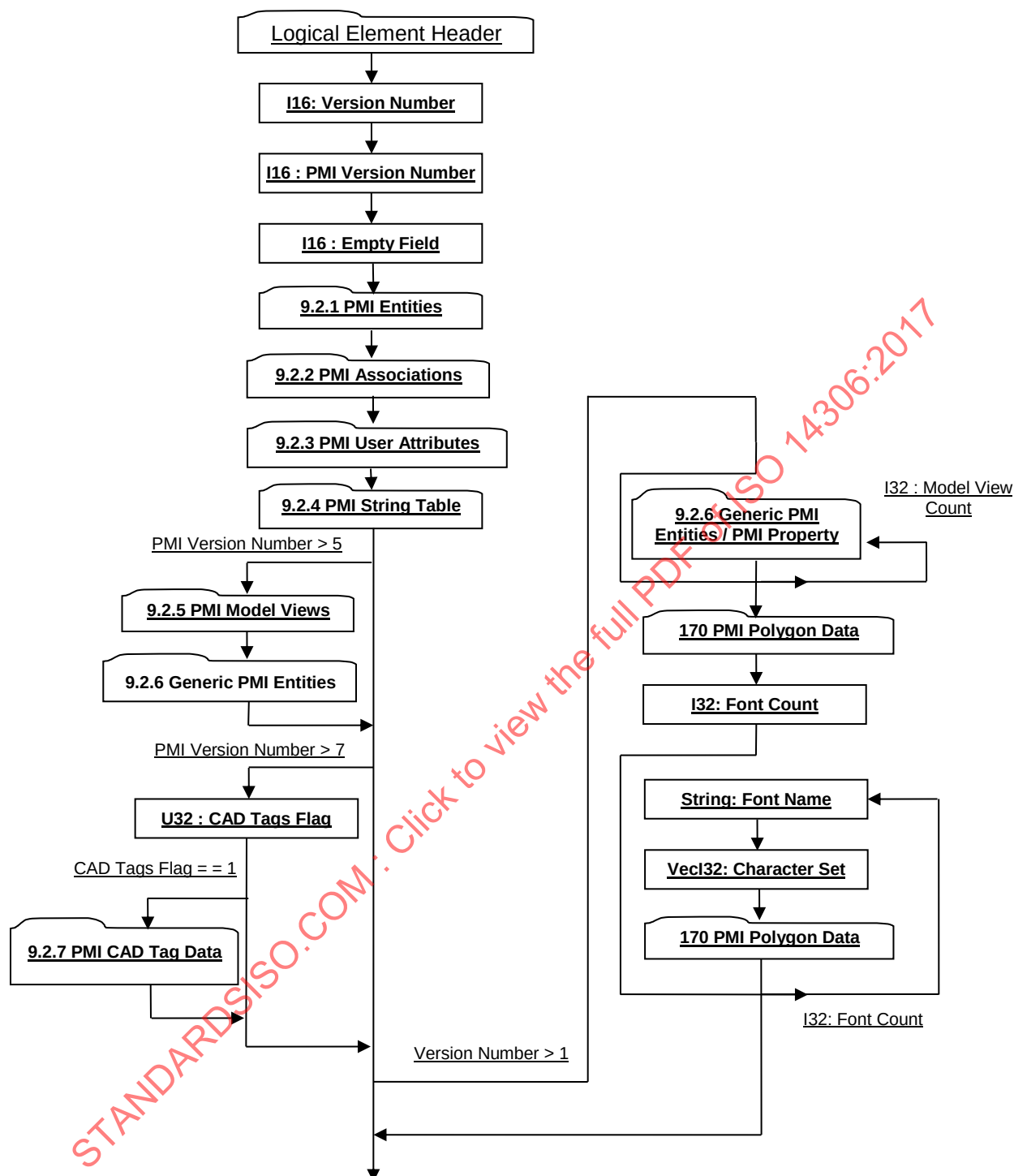


Figure 112 — PMI Manager Meta Data Element data collection

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

I16: Version Number

Version Number is the version identifier for this PMI Manager Element. For information on local version numbers see best practice **11.5 Local version numbers**.

I16 : PMI Version Number

PMI Version Number is the identifier that defines which classes of PMI data will be available in the PMI Manager Meta Data Element. PMI Version Number data is additive.

The valid PMI version numbers for V9.5 JT are as follows:

Table 64 — PMI Manager Meta Data Version Number values

= 3	Version 3 includes; I32 : Attribute Count, Design Group Attribute (see Figure 135 — PMI Design Group Entities data collection)
= 4	Version 4 includes version 3 plus; - CoordF32 : Weld Point, DirF32 : Approach Direction, DirF32 : Clamping Direction, DirF32 : Normal Direction (see Figure 129 — PMI Spot Weld Entities data collection) - CoordF32 : Location, DirF32 : Measurement Direction, DirF32 : Coordinate Direction, DirF32 : Normal Direction (see Figure 132 — PMI Measurement Point Entities data collection)
= 5	Version 5 Includes version 4 plus; - U8 : Symbol Valid Flag (see Figure 116 — PMI Base Data collection) - I32 : Polyline Type Count, I16 : Polyline Type (see Figure 123 — Non-Text Polyline Data collection)
= 6	Version 6 includes version 5 plus; 10.2.5 PMI Model Views, 10.2.6 Generic PMI Entities (see Figure 112 — PMI Manager Meta Data Element data collection) - U32 : URL Flag (see Figure 124 — PMI Note Entities data collection) - I32 : Source Owning Entity String ID, I32 : Destination Owning Entity String ID (see Figure 138 — PMI Associations data collection)
= 7	Version 7 includes version 6 plus ; - U16 : U16 : User Flags (see Figure 142 — Generic PMI Entities data collection) - U32 : Hidden Flag (Figure 144 — PMI Property Atom data collectionsee)
= 8	Version 8 Includes version 7 plus; U32 : CAD Tags Flag (see Figure 112 — PMI Manager Meta Data Element data collection)

I16 : Empty Field

Refer to best practice [11.4](#) Empty Field.

U32 : CAD Tags Flag

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

I32: Font Count

Number of sets of glyph definitions. Each set of glyphs represents a single font definition that consists of a name, a character set and polygonal glyph definition for each character in the set.

String: Font Name

Font name specifies a representative name for the font set.

VecI32: Character Set

Integer identifiers for each character whose symbol is defined in the ensuing PolygonData segment.

9.2.1 PMI Entities

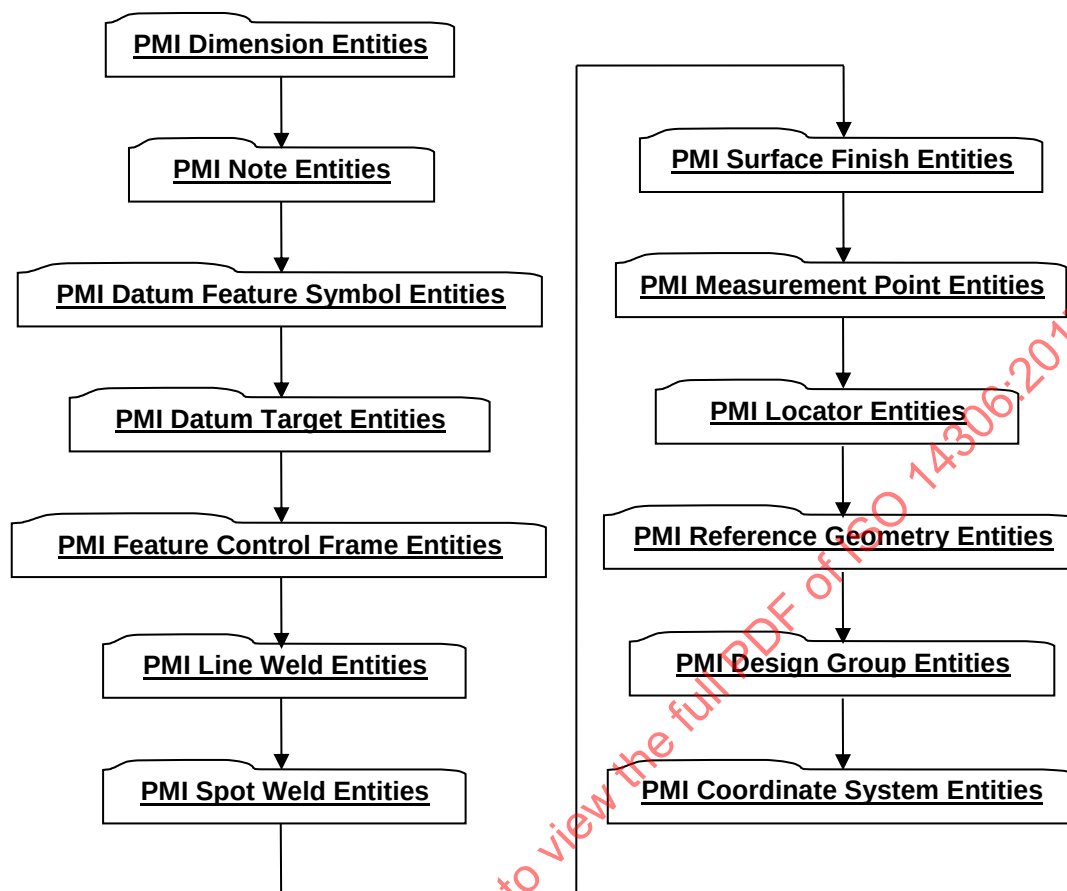


Figure 113 — PMI Entities data collection

9.2.1.1 PMI Dimension Entities

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

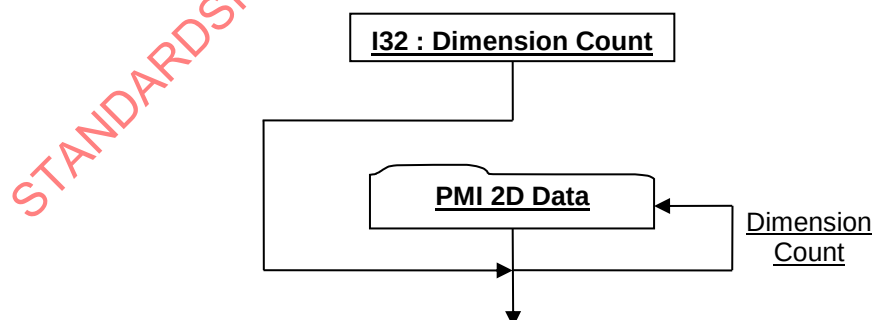


Figure 114 — PMI Dimension Entities data collection

I32 : Dimension Count

Dimension Count specifies the number of Dimension entities.

9.2.1.1.1 PMI 2D Data

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

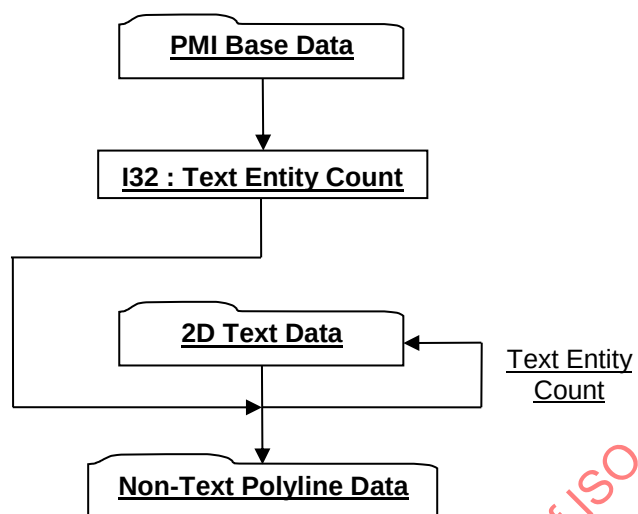


Figure 115 — PMI 2D Data collection

I32 : Text Entity Count

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

PMI Base Data

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

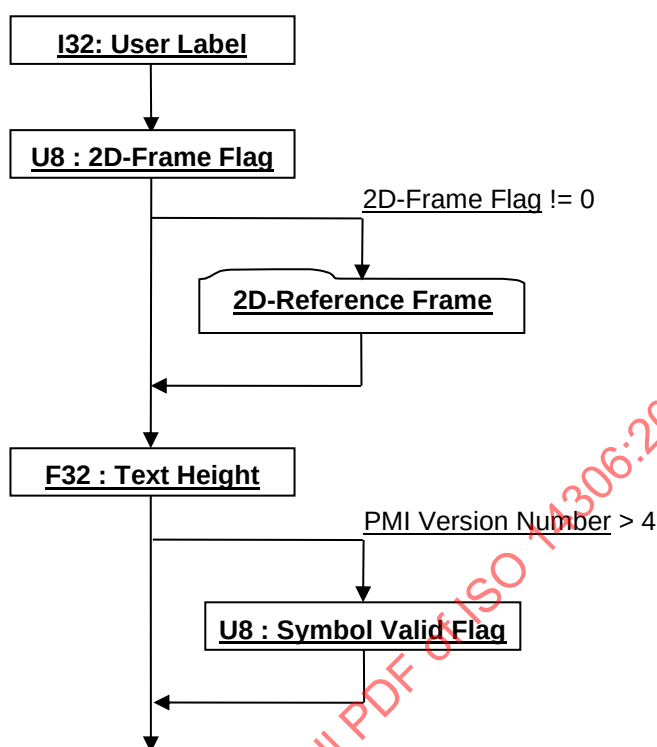


Figure 116 — PMI Base Data collection

I32 : User Label

User Label specifies the particular PMI entity identifier.

U8 : 2D-Frame Flag

2D-Frame Flag is a flag specifying whether 2D-Reference Frame data is stored. If 2D-Frame Flag has a non-zero value then 2D-Reference Frame data is included. If 2D-Frame Flag has a value of “2”, then dummy (i.e. all zeros) 2D-Reference Frame data is written. The “2D-Frame Flag = 2” case is used by 9.2.6 Generic PMI Entities because for Generic PMI Entities all the Non-Text Polyline Data is already in 3D form (i.e. XYZ coordinate data).

F32 : Text Height

Text Height specifies the PMI text height in WCS.

U8 : Symbol Valid Flag

Symbol Valid Flag is a flag specifying whether the particular PMI entity is valid. If Symbol Valid Flag has a non-zero value then PMI entity is valid. This flag is only stored if the Version Number as defined in 9.2 PMI Manager Meta Data Element is greater than “4.”

2D-Reference Frame

The 2D-Reference Frame data collection defines a reference frame (2D coordinate system) where the PMI entity is displayed in 3D space. All the PMI entity's 2D and 3D polyline data is assumed to lie on the defined plane.

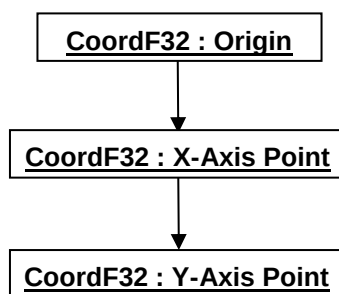


Figure 117 — 2D-Reference Frame data collection

CoordF32 : Origin

Origin defines the origin (min-corner) of the 2D coordinate system

CoordF32 : X-Axis Point

X-Axis Point defines a point along the X-Axis of the 2D coordinate system.

CoordF32 : Y-Axis Point

Y-Axis Point defines a point along the Y-Axis of the 2D coordinate system.

2D Text Data

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

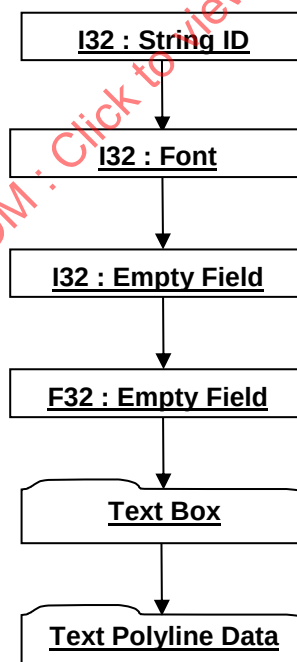


Figure 118 — 2D Text Data collection

I32 : String ID

String ID specifies the identifier for the character string. This identifier is an index to a particular character string in the PMI String Table as defined in 9.2.4 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:

Table 65 — PMI 2D Base Data Font values

= 1	Simplex
= 2	Din
= 3	Military
= 4	ISO
= 5	Lightline
= 6	IGES 1001
= 7	Century
= 8	IGES 1002
= 9	IGES 1003
= 101	Japanese JISX 0208 coded character set
= 102	Japanese Extended Unix Codes JISX 0208 coded character set
= 103	Chinese GB 2312.1980 Simplified coded character set
= 104	Korean KSC 5601 coded character set
= 105	Chinese Big5 Traditional coded character set

I32 : Empty Field

Refer to best practice [11.4](#) Empty Field.

F32 : Empty Field

Refer to best practice [11.4](#) Empty Field.

Text Box

The Text Box data collection specifies a 2D box that particular text fits within. All values are with respect to 2D-Reference Frame documented in [0 2D-Reference Frame](#).

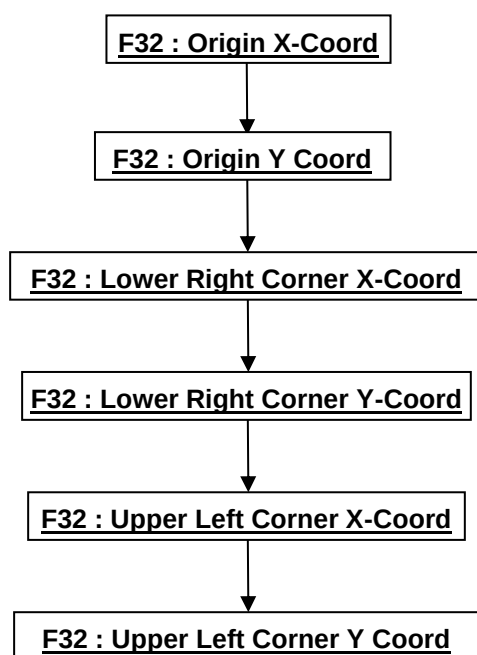


Figure 119 — Text Box data collection

F32 : Origin X-Coord

Origin X-Coord defines the 2D X-coordinate of the text origin with respect to 2D-Reference Frame.

F32 : Origin Y Coord

Origin Y-Coord defines the 2D Y-coordinate of the text origin with respect to 2D-Reference Frame.

F32 : Lower Right Corner X-Coord

Lower Right Corner X-Coord defines the 2D X-coordinate of the lower right corner of the text with respect to 2D-Reference Frame.

F32 : Lower Right Corner Y-Coord

Lower Right Corner Y-Coord defines the 2D Y-coordinate of the lower right corner of the text with respect to 2D-Reference Frame.

F32 : Upper Left Corner X-Coord

Upper Left Corner X-Coord defines the 2D X-coordinate of the upper left corner of the text with respect to 2D-Reference Frame.

F32 : Upper Left Corner Y Coord

Upper Left Corner Y-Coord defines the 2D Y-coordinate of the upper left corner of the text with respect to 2D-Reference Frame.

Text Polyline Data

The Text Polyline Data collection defines any polyline segments which are part of the text representation. This existence of this polyline data is conditional (i.e. not all text has it) and is made up of an array of indices into an array of polyline segments packed as 2D vertex coordinates, specifying where each polyline segment begins and ends. Polylines are constructed from these arrays of data as follows:

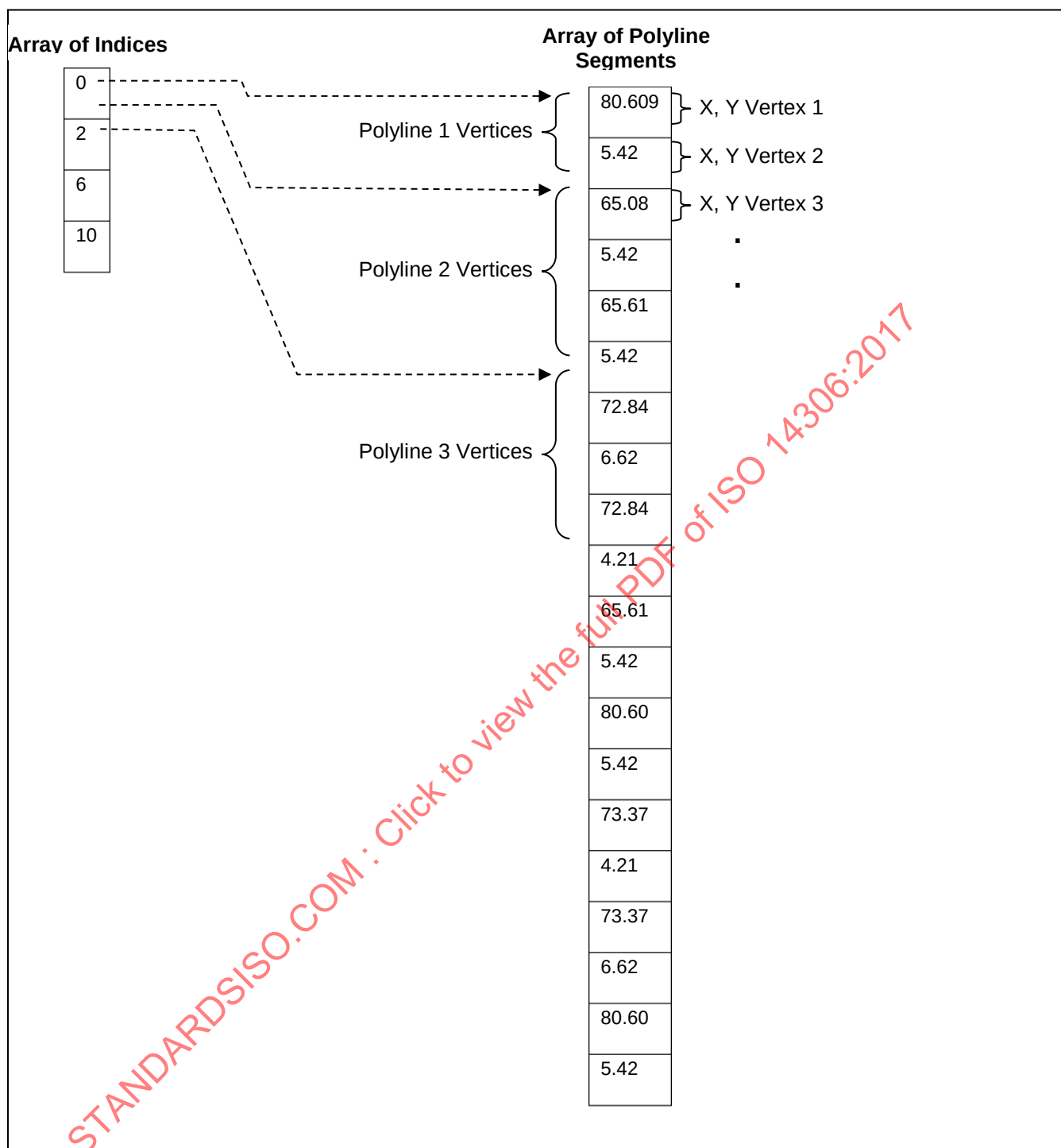


Figure 120 — Constructing Text Polylines from data arrays

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

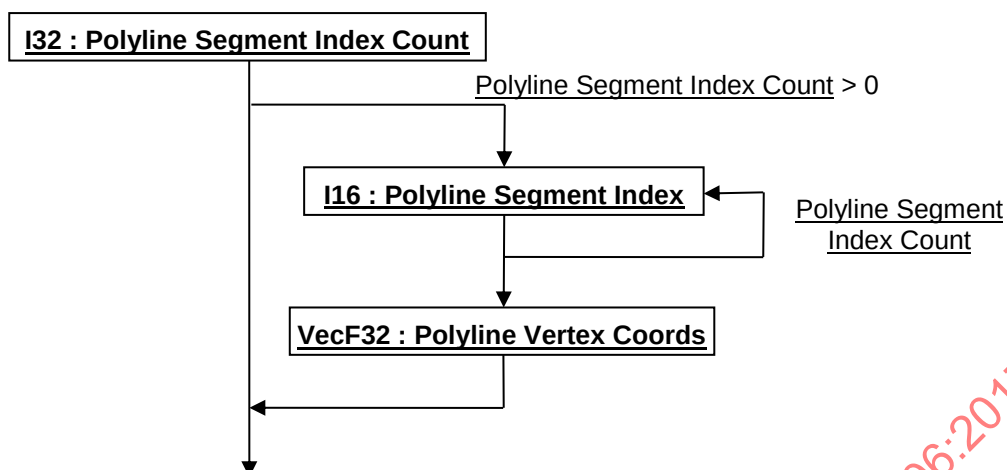


Figure 121 — Text Polyline Data collection

I32 : Polyline Segment Index Count

Polyline Segment Index Count specifies the number of polyline segment indices.

I16 : Polyline Segment Index

Polyline Segment Index is an index into the Polyline Vertex Coords array specifying where polyline segment begins or ends. This index is a vertex coordinate index so the absolute index into the Polyline Vertex Coords array is computed by multiplying the index value by “2” (i.e. for 2D coordinates).

VecF32 : Polyline Vertex Coords

Polyline Vertex Coords is an array of polyline segments packed as 2D point coordinates. These 2D point coordinates are with respect to the 2D-Reference Frame documented in 0 2D-Reference Frame.

Non-Text Polyline Data

The Non-Text Polyline Data collection contains all the non-text polylines making up the particular PMI entity. Examples of non-text polylines include line attachments, text boxes, symbol box dividers, etc. The Non-Text Polyline Data collection is made up of an array of indices into an array of polyline segments packed as either 2D or 3D vertex coordinates, specifying where each polyline segment begins and ends. Whether vertex coordinates are 2D or 3D is dependent upon the PMI entity type using this data collection. If it is a 9.2.6 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 9.2 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 122 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]...”

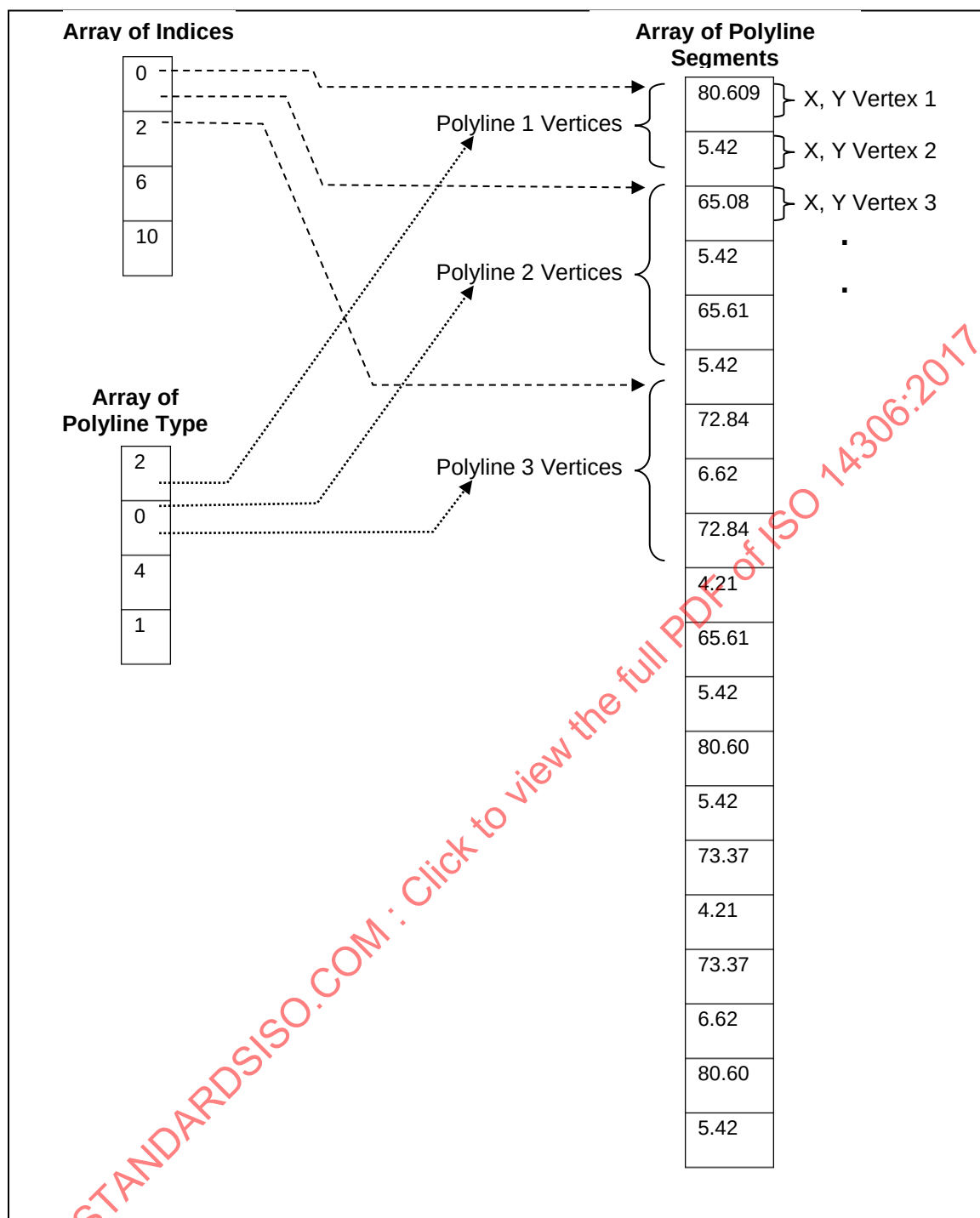


Figure 122 — Constructing Non-Text Polylines from packed 2D data arrays

This data is represented in the JT Open format as follows:

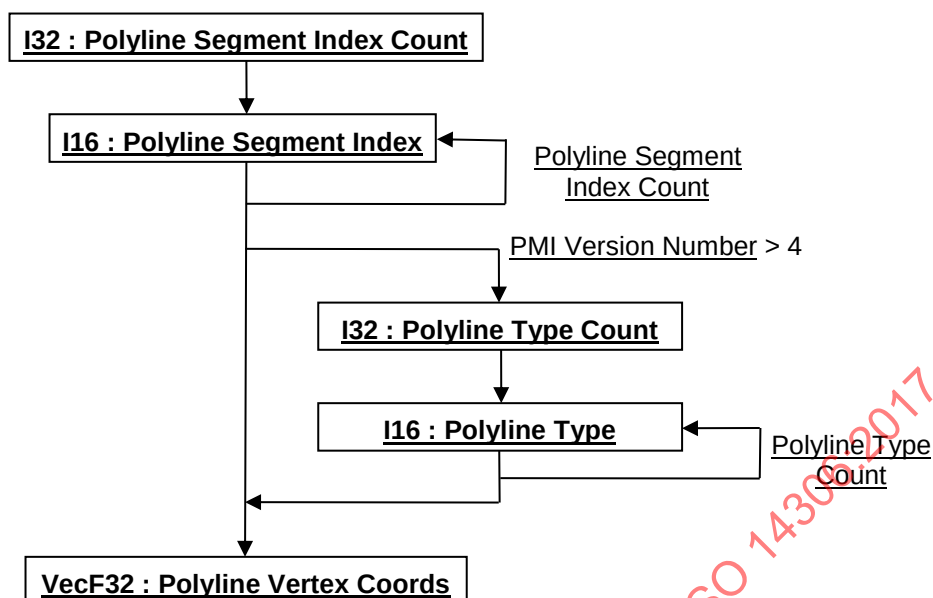


Figure 123 — Non-Text Polyline Data collection

I32 : Polyline Segment Index Count

Polyline Segment Index Count specifies the number of polyline segment indices.

I16 : Polyline Segment Index

Polyline Segment Index is an index into the Polyline Vertex Coords array specifying where polyline segment begins or ends. This index is a vertex/coordinate index so the absolute index into the Polyline Vertex Coords array is computed by multiplying the index value by “2” (i.e. for 2D coordinates).

I32 : Polyline Type Count

Polyline Type Count specifies the number of polyline type values.

I16 : Polyline Type

Polyline Type specifies the type of polyline segment in Polyline Vertex Coords array. See Figure 122 — Constructing Non-Text Polylines from packed 2D data arrays for interpretation of this array of type values relative to the defined polylines. Valid values include the following:

Table 66 — PMI 2D Non-Text Polyline Type values

= 0	General line
= 1	General arrow
= 2	General circle
= 3	General arc
= 4	Extended line 1
= 5	Extended line 2
= 6	Extended arc
= 7	Extended circle
= 8	Text line (used in text boxes and symbol box dividers)
= 9	Text string

VecF32 : Polyline Vertex Coords

Polyline Vertex Coords is an array of polyline segments packed as 2D point coordinates. These 2D point coordinates are with respect to the 2D-Reference Frame documented in 0 2D-Reference Frame.

9.2.1.2 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.

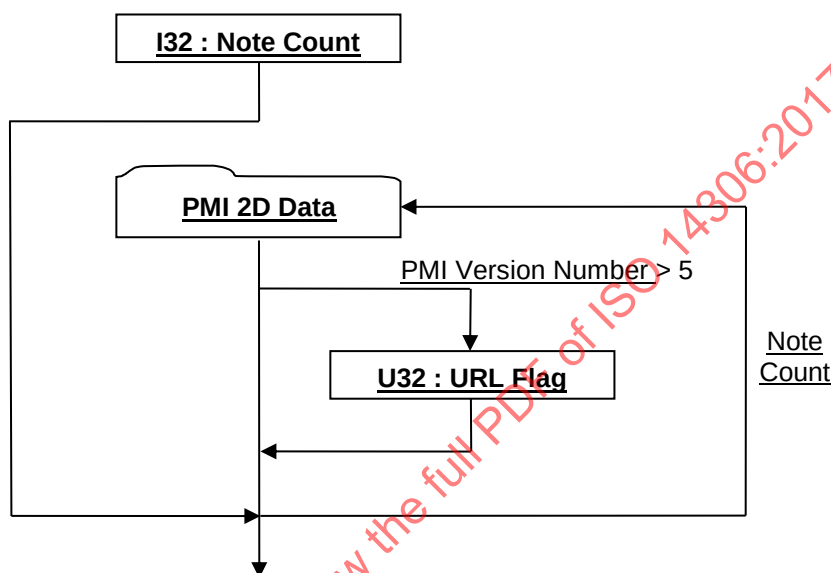


Figure 124 — PMI Note Entities data collection

Complete description for PMI 2D Data can be found in 9.2.1.1.1 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 9.2 PMI Manager Meta Data Element, is greater than “5”. The URL is the actual text of the note as specified in PMI 2D Data.

9.2.1.3 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.

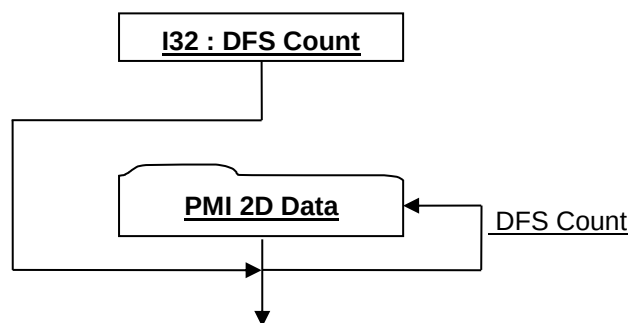


Figure 125 — PMI Datum Feature Symbol Entities data collection

Complete description for PMI 2D Data can be found in [9.2.1.1.1 PMI 2D Data](#).

I32 : DFS Count

DFS Count specifies the number of Datum Feature Symbol entities.

9.2.1.4 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.

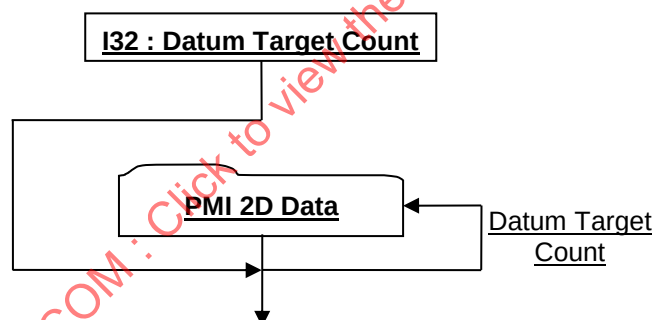


Figure 126 — PMI Datum Target Entities data collection

Complete description for PMI 2D Data can be found in [9.2.1.1.1 PMI 2D Data](#).

I32 : Datum Target Count

Datum Target Count specifies the number of Datum Target entities.

9.2.1.5 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.

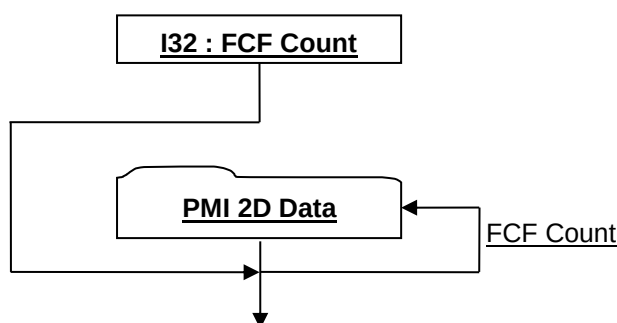


Figure 127 — PMI Feature Control Frame Entities data collection

Complete description for PMI 2D Data can be found in [9.2.1.1.1 PMI 2D Data](#).

I32 : FCF Count

FCF Count specifies the number of Feature Control Frame entities.

9.2.1.6 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.

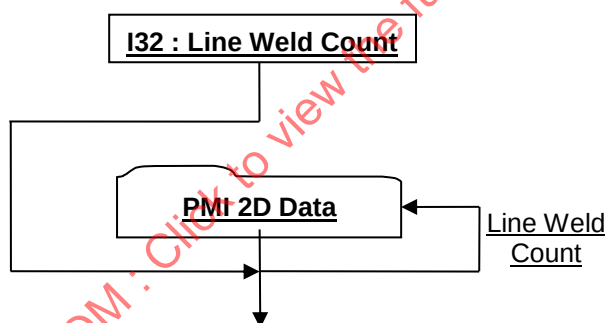


Figure 128 — PMI Line Weld Entities data collection

Complete description for PMI 2D Data can be found in [9.2.1.1.1 PMI 2D Data](#).

I32 : Line Weld Count

Line Weld Count specifies the number of Line Weld entities.

9.2.1.7 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 [9.2 PMI Manager Meta Data Element](#), is greater than or equal to “4”.

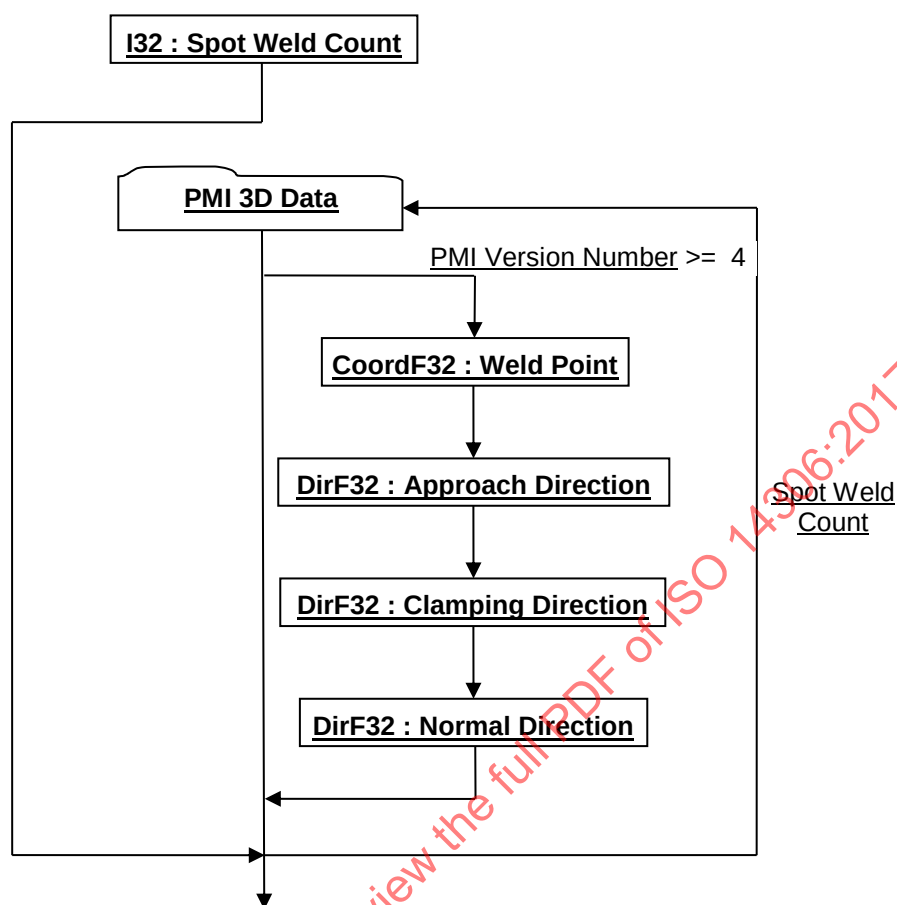


Figure 129 — PMI Spot Weld Entities data collection

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.

9.2.1.7.1 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 120](#) of [0 Text Polyline Data](#).

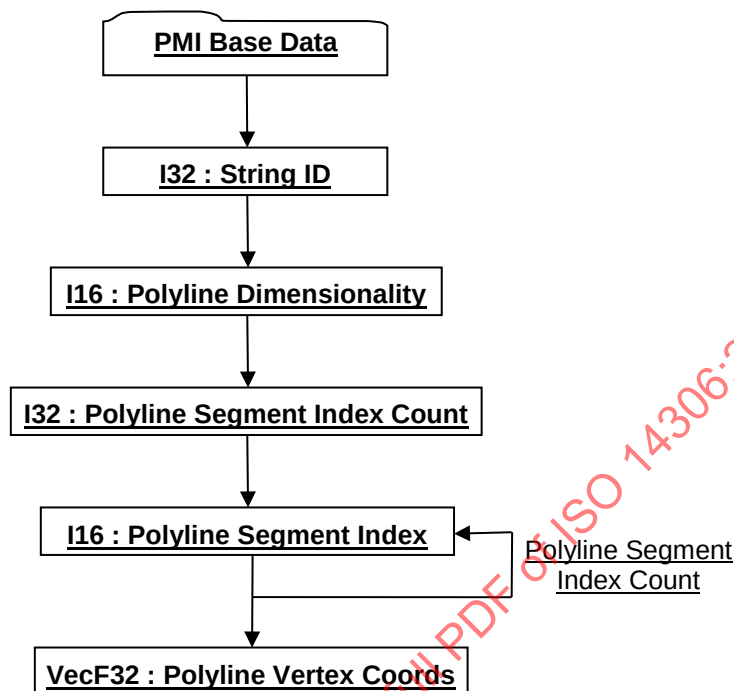


Figure 130 — PMI 3D Data collection

Complete description for PMI Base Data can be found in [PMI Base Data](#).

I32 : String ID

String ID specifies the identifier for the character string. This identifier is an index to a particular character string in the PMI String Table as defined in [9.2.4 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:

Table 67 — PMI 3D Data Polyline Dimensionality values

= 2	Indicates 2-dimensional (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.

9.2.1.8 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).

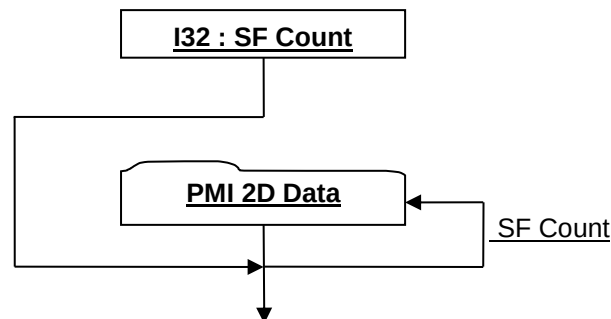


Figure 131 — PMI Surface Finish Entities data collection

Complete description for PMI 2D Data can be found in [9.2.1.1.1 PMI 2D Data](#).

I32 : SF Count

SF Count specifies the number of Surface Finish symbol entities.

9.2.1.9 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 [9.2 PMI Manager Meta Data Element](#), is greater than or equal to "4".

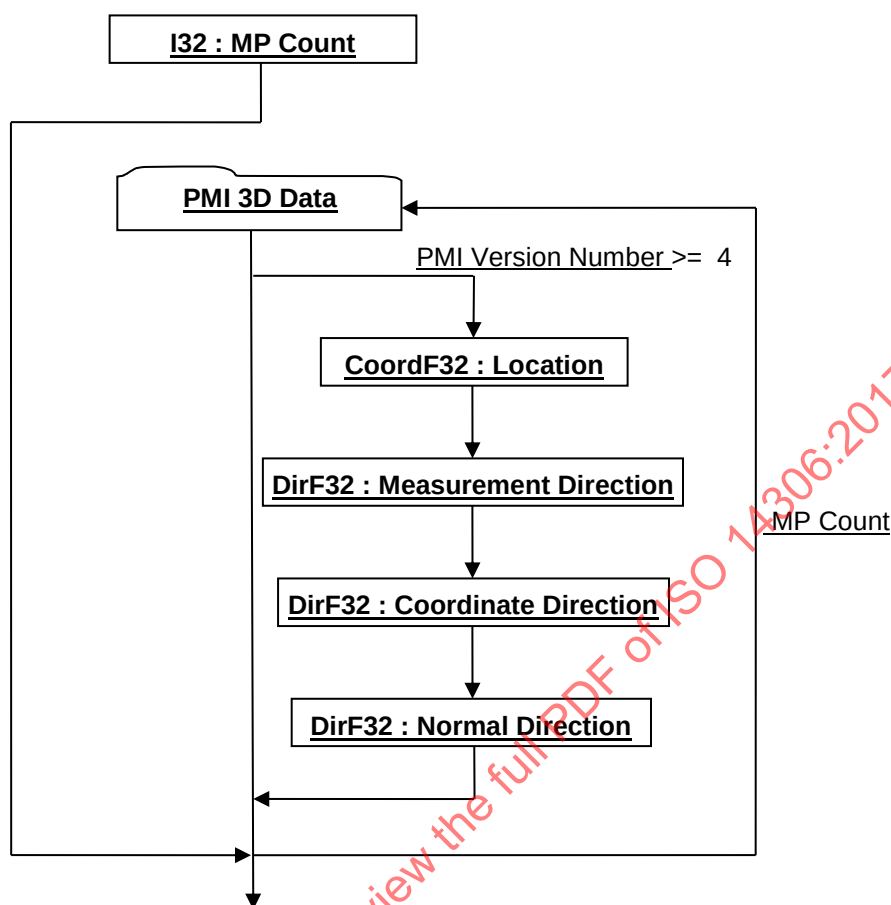


Figure 132 — PMI Measurement Point Entities data collection

Complete description for PMI 3D Data can be found in [9.2.1.7.1 PMI 3D Data](#).

I32 : MP Count

MP Count specifies the number of Measurement Point entities.

CoordF32 : Location

Location specifies the coordinates of the Measurement Point.

DirF32 : Measurement Direction

Measurement Direction specifies the components of the direction vector from which a CCM (Coordinate Measuring Machine) approaches when taking a measurement.

DirF32 : Coordinate Direction

Coordinate Direction specifies the components of the direction vector another Measurement Point on a mating part would like to align with a Measurement Point on the first part.

DirF32 : Normal Direction

Normal Direction specifies the components of the direction vector normal to the actual Measurement Point.

9.2.1.10 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.

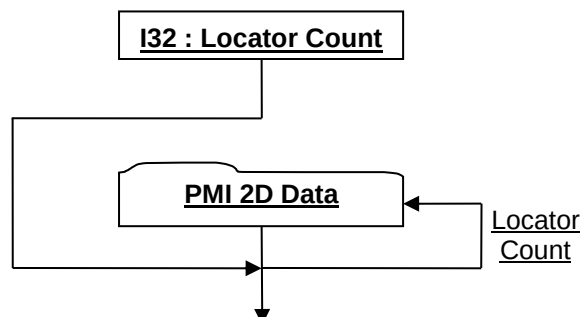


Figure 133 — PMI Locator Entities data collection

Complete description for PMI 2D Data can be found in [9.2.1.1.1 PMI 2D Data](#).

I32 : Locator Count

Locator Count specifies the number of Locator symbol entities.

9.2.1.11 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 [9.2.1.7.1 PMI 3D Data](#)) as follows:

Table 68 — PMI Reference Geometry Entity values

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

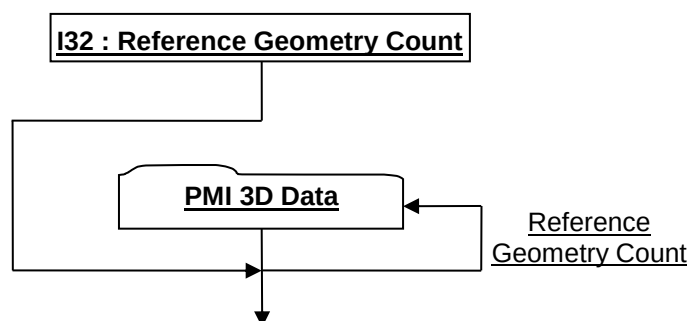


Figure 134 — PMI Reference Geometry Entities data collection

Complete description for PMI 3D Data can be found in [9.2.1.7.1 PMI 3D Data](#).

I32 : Reference Geometry Count

Reference Geometry Count specifies the number of Reference Geometry entities.

9.2.1.12 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 9.2.2 PMI Associations), where specific PMI entities are associated as “destinations” to a “source” PMI Design Group.

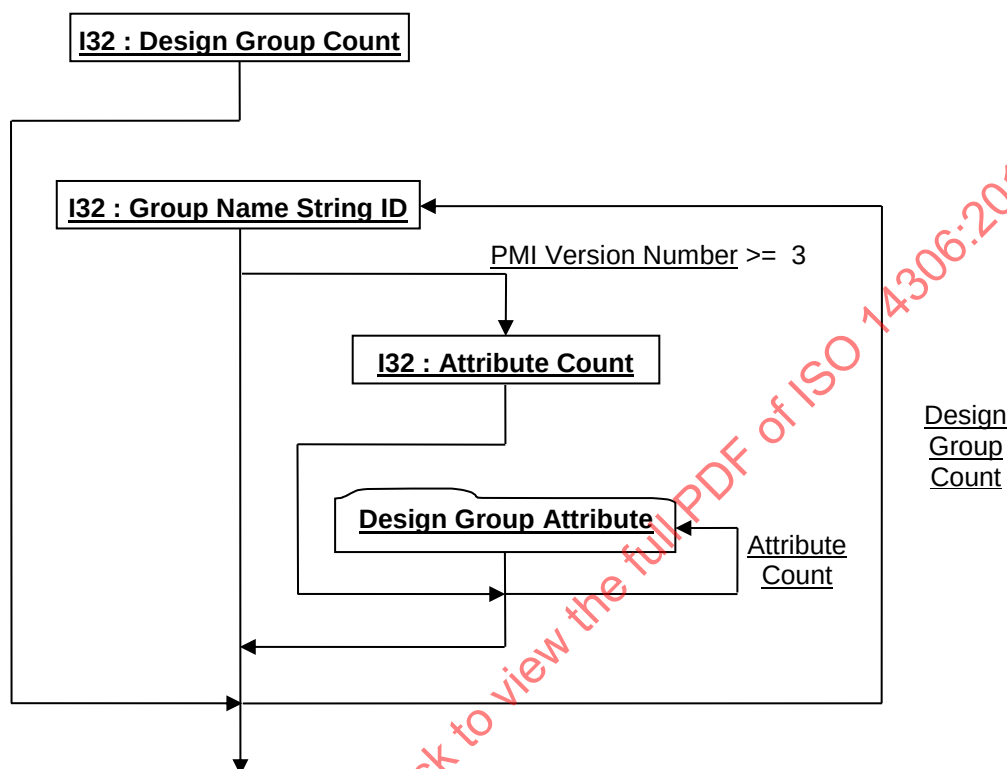


Figure 135 — PMI Design Group Entities data collection

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 9.2.4 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

9.2.1.12.1 Design Group Attribute

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

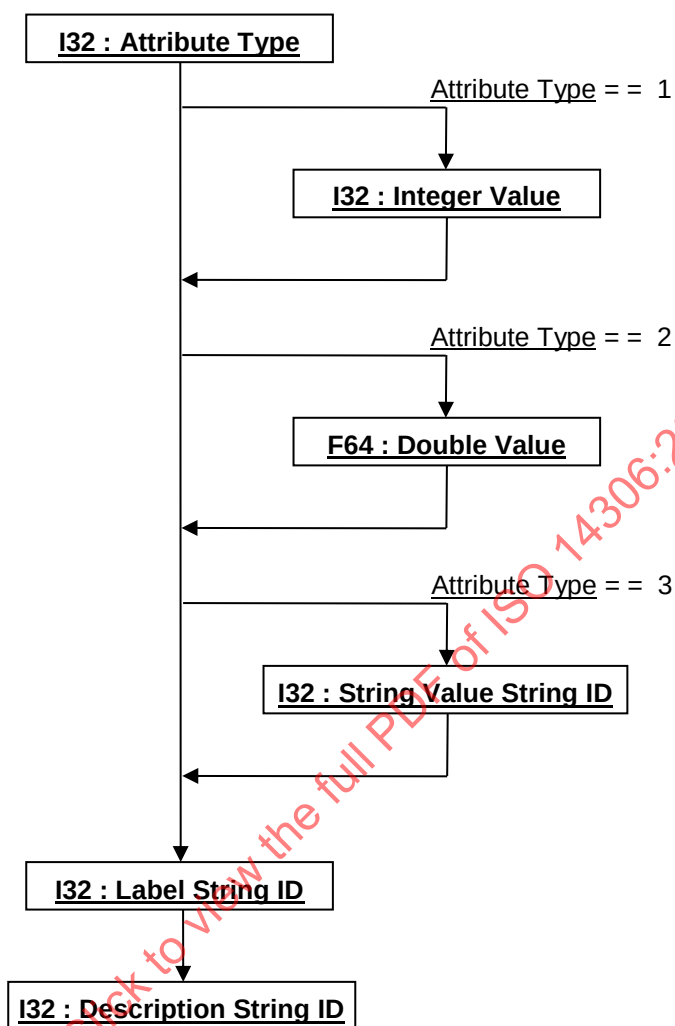


Figure 136 — Design Group Attribute data collection

I32 : Attribute Type

Attribute Type specifies the attribute type. Valid types include the following:

Table 69 — PMI Design Group Attribute Type values

= 1	Integer
= 2	Double
= 3	String

I32 : Integer Value

Integer Value specifies the value for “integer” Attribute Types.

F64 : Double Value

Double Value specifies the value for “double” Attribute Types.

I32 : String Value String ID

String Value String ID specifies the string identifier value for “string” Attribute Types. This identifier is an index to a particular character string in the PMI String Table as defined in 9.2.4 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 9.2.4 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 9.2.4 PMI String Table. An identifier value of “-1” indicates no string.

9.2.1.13 PMI Coordinate System Entities

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

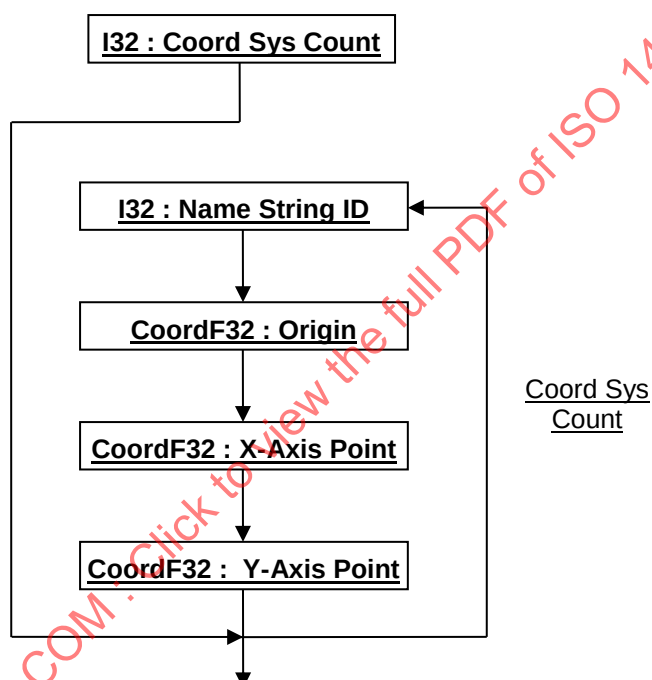


Figure 137 — PMI Coordinate System Entities data collection

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 9.2.4 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.

9.2.2 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”.

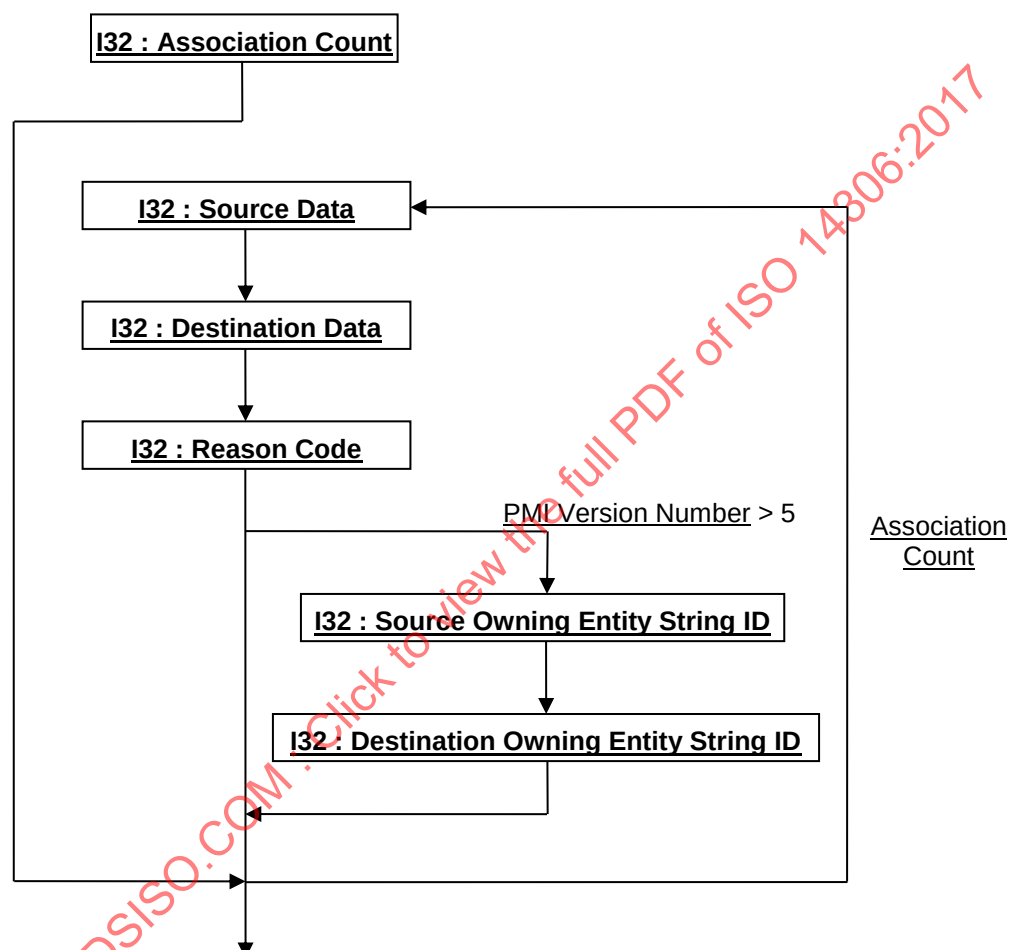


Figure 138 — PMI Associations data collection

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 bits fields that are not defined as in use should be set to “0”.

Table 70 — PMI Associations Source Data values

Bits 0 - 23	Source Entity Identifier. The interpretation of this identifier data is dependent upon the value of Bit 31 documented below.
Bits 24 -30	Source Entity PMI or B-Rep type. Valid types include the following:

	= 0 PMI - Dimension = 1 PMI - Note = 2 PMI - Datum Feature Symbol = 3 PMI - Datum Target = 4 PMI - Feature Control Frame = 5 PMI - Line Weld = 6 PMI - Spot Weld = 7 PMI - Measurement Point = 8 PMI - Surface Finish = 9 PMI - Locator Designator = 10 PMI - Reference Geometry = 11 PMI - Coordinate System = 12 PMI - Design Group = 13 PMI - User Attribute = 14 B-Rep - Vertex = 15 B-Rep - Edge = 16 B-Rep - Face = 17 PMI - Model View = 18 PMI - Generic = 19 Wireframe Rep - Edge = 20 PMI - Unspecified type = 21 Part Instance
Bit 31	Indirect Identifier Flag = 0 – Value in Bits 0-23 is not the actual CAD identifier, instead Bits 0-23 is an index into the source type's PMI array or index of the edge/face in B-Rep or Wireframe Rep for the source entity. = 1 – Value in Bits 0-23 is not the actual CAD identifier; instead Bits 0-23 is an index into the list of CAD Tags (as documented in 9.2.7 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:

Table 71 — PMI Associations Reason Code values

= 0	Association is to the primary entity being dimensioned
= 1	Association is to the secondary entity being dimensioned
= 2	Association is to the dimension plane
= 5	Association is to the entity used to specify the Z-Axis of a coordinate system
= 10	Association is to an entity "associated" to or "included in" a PMI symbol
= 11	Association is to an entity used to "attach" a PMI symbol.
= 12	Association is to first entity used to “attach” a PMI symbol
= 13	Association is to second entity used to “attach” a PMI symbol
= 14	Specifying PMI grouping, source is PMI/B-Rep entity and destination is design group.
= 15	Association is to a weld line entity
= 16	Association is to a “hot spot”
= 17	Association is to a child in a PMI stack

= 72	Association is for PMI miscellaneous relation.
= 73	Association is for PMI related entity.
= 98	Association is to show the PMI when associated Model View is selected. Source is the PMI, and destination is Model View.
= 99	Association is to show/select PMI B, if showing/selecting PMI A. Source is PMI A, and destination is PMI B. This is different from an “attached” PMI, where the convention is to show the PMI visibly linked to one another.
= 100	Association is to show all parts except the associated part instance. Source is the part instance, and destination is Model View

I32 : 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 9.2.4 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 9.2 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 9.2.4 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 9.2 PMI Manager Meta Data Element, is greater than “5”.

9.2.3 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.

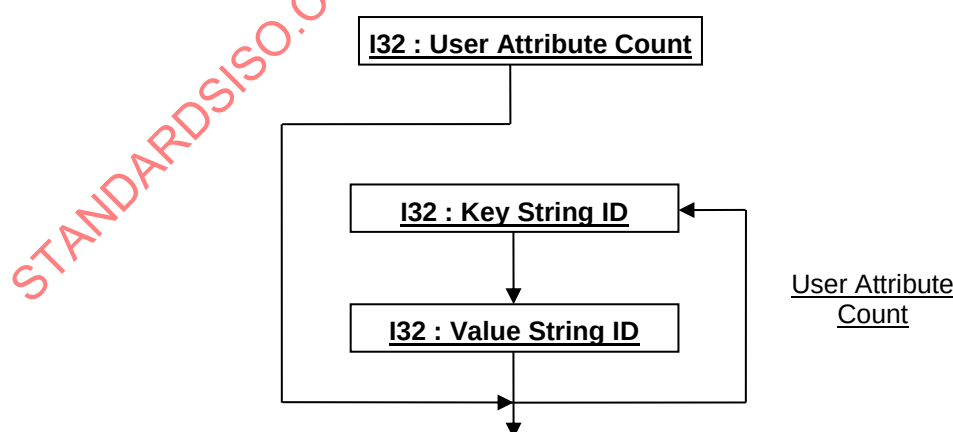


Figure 139 — PMI User Attributes data collection

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 9.2.4 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 9.2.4 PMI String Table. An identifier value of “-1” indicates no string.

9.2.4 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).

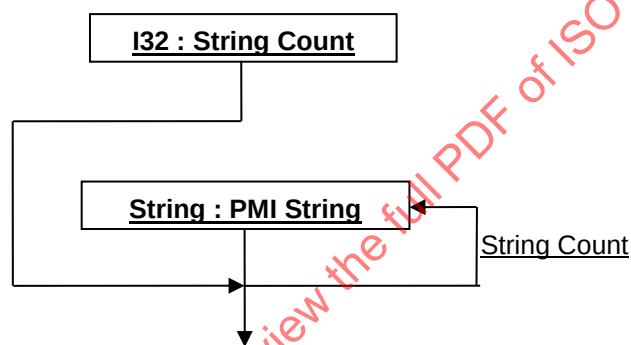


Figure 140 — PMI String Table data collection

I32 : String Count

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

String : PMI String

PMI String specifies the character string.

9.2.5 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 9.2.2 PMI Associations), where specific PMI entities are associated as “destinations” to a “source” PMI Model View.

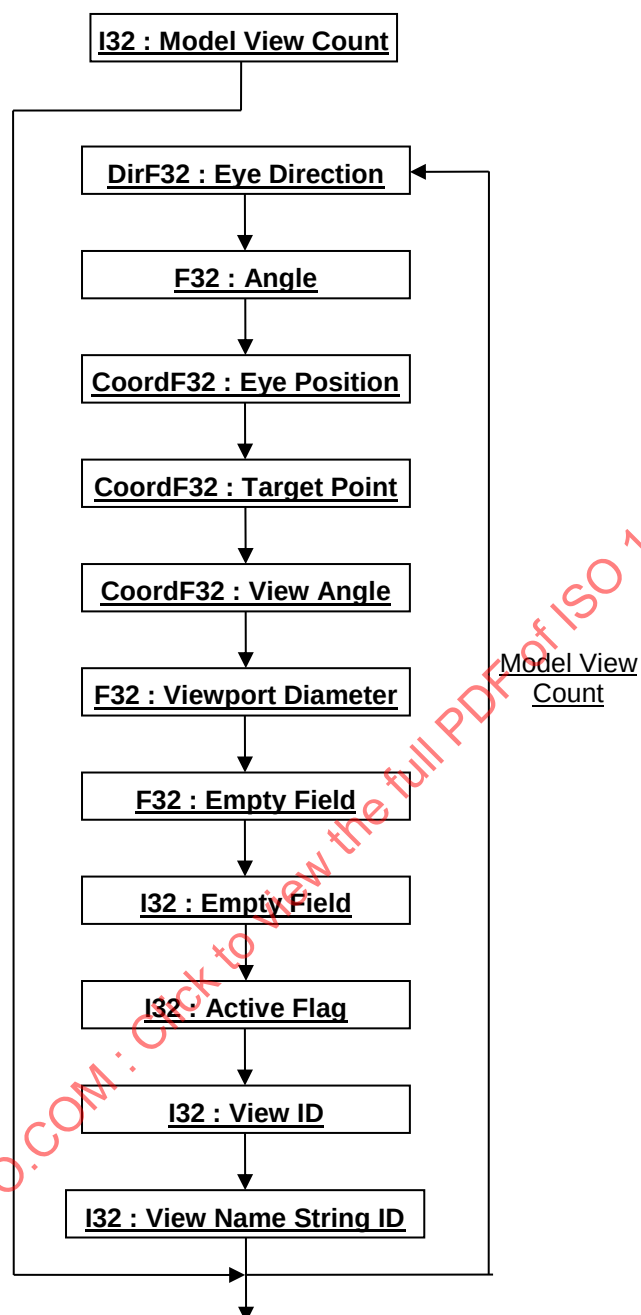


Figure 141 — PMI Model Views data collection

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 : Empty Field

Refer to best practice [11.4 Empty Field](#)

I32 : Empty Field

Refer to best practice [11.4 Empty Field](#)

I32 : Active Flag

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

Table 72 — PMI Model Views Active Flag values

= 0	Is not the active Model View.
= 1	Is the active Model View

I32 : View ID

View ID specifies the Model View unique identifier.

I32 : View Name String ID

View Name String ID specifies the string identifier for the Model View's name. This identifier is an index to a particular character string in the PMI String Table as defined in [9.2.4 PMI String Table](#). An identifier value of “-1” indicates no string.

9.2.6 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.

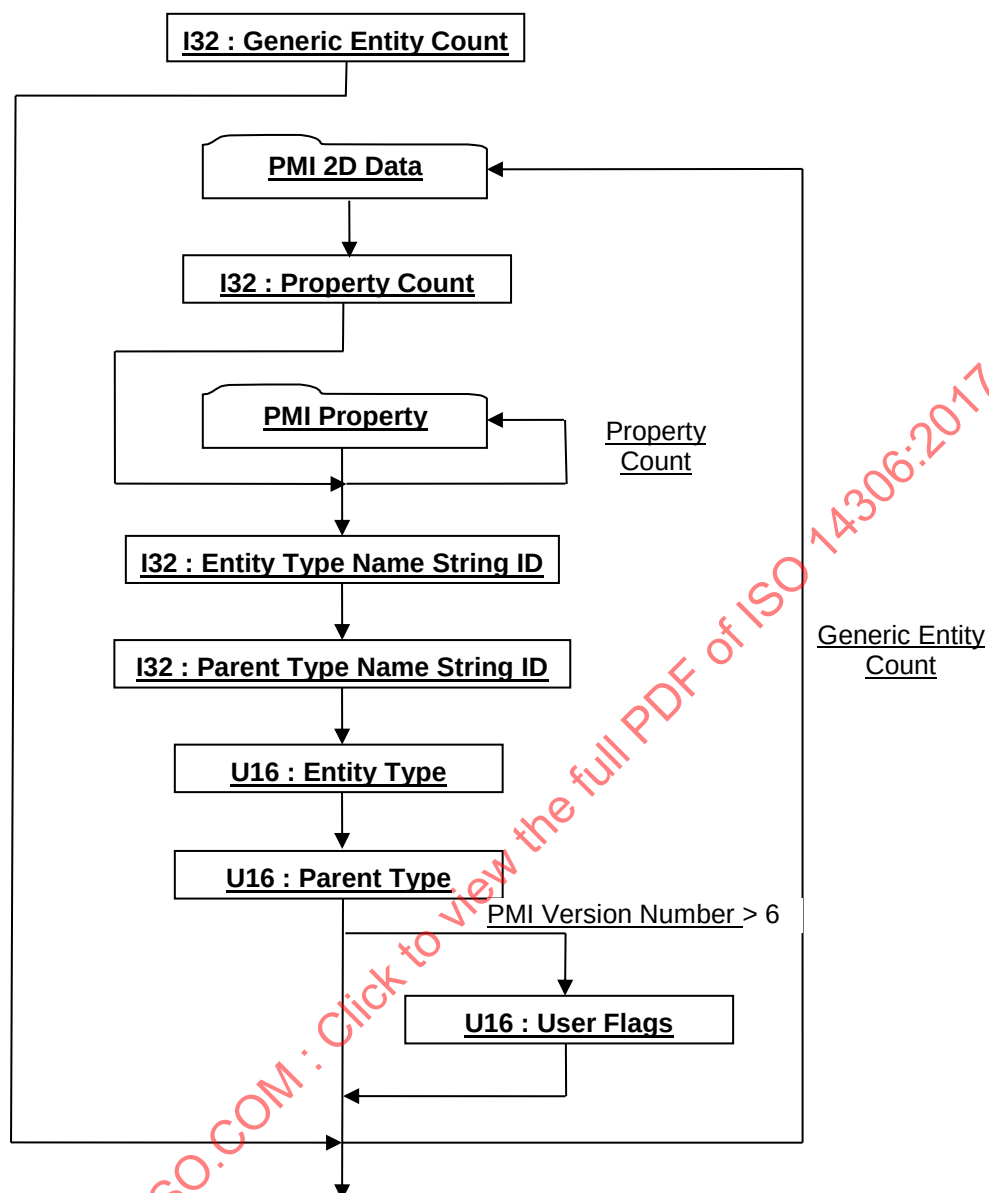


Figure 142 — Generic PMI Entities data collection

Complete description for PMI 2D Data can be found in [9.2.1.1.1 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 [9.2.4 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 9.2.4 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.

Table 73 — Generic PMI Entity Type values

0x0001	PMI (generally only used as a <u>Parent Type</u>)
0x0002	Weld
0x0004	Spot Weld
0x0008	Line Weld
0x0010	Groove Weld
0x0011	Fillet Weld
0x0012	Slot Weld
0x0014	Edge Weld
0x0018	Arc Spot Weld
0x0020	Resistance Spot Weld
0x0021	Resistance Seam Weld
0x0022	Structural Adhesive Bead Shaped
0x0024	Structural Adhesive Tape Shaped
0x0028	Structural Adhesive Dollop Shaped
0x0040	Mechanical Clinch Connector
0x0041	Surface Finish
0x0042	Measurement Point
0x0044	Datum Locator
0x0048	Certification Point
0x0080	Geometric Dimensioning and Tolerancing
0x0081	Feature Control Frame
0x0082	Dimension
0x0084	Datum Feature Symbol
0x0088	Datum Target
0x0100	Note
0x0101	Face Attribute Note
0x0102	Model View Label Note
0x0104	Coordinate System
0x0108	Reference Geometry
0x0110	Reference Point
0x0111	Reference Axis
0x0112	Reference Plane
0x0114	User Defined
0x0118	Measurement Locator
0x0120	Datum Point
0x0121	Surface Vector Measurement Point
0x0122	Hole Vector Measurement Point
0x0124	Trimmed Sheet Vector Measurement Point
0x0128	Hem Vector Measurement Point

0x0230	Fastener PMI
0x0231	Material specification
0x0232	Process specification
0x0233	Part specification
0x0235	Balloon Note
0x0238	Circle Centre
0x0239	Coordinate Note
0x0240	AttributeNote
0x0241	Bundle or Dressing Note
0x0242	Cutting Plane Symbol
0x0243	Crosshatch
0x0244	E Marking (Note)
0x0245	Organization
0x0246	Region
0x0305	Section
0x0306	Centreline
0x0307	Fit Designation
0x0308	Composite Feature Control Frame

U16 : Parent Type

Parent Type specifies the parent Generic PMI Entity Type. The valid Parent Type values are the same as that documented above for Entity Type. The Parent Type is used to create a class hierarchy of PMI when presenting the PMI contents from a JT file.

U16 : User Flags

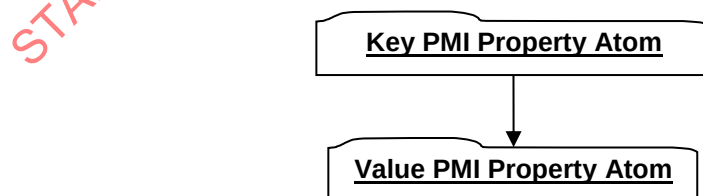
User Flags is a collection of flags. The flags are combined using the binary OR operator and store various state information for the Generic PMI Entity. All bits fields that are not defined as in use should be set to "0".

Table 74 — Generic PMI User Flag values

0x0001	Show PMI Entity "flat to screen only" flag = 0 – Allow PMI display plane to rotate with model. = 1 – Display PMI entity in the plane of the screen, so that it does not rotate with model.

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 143 — PMI Property data collection**

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

Although there is no reference compliant requirements for what the PMI Property key/value pairs shall be for each Generic PMI Entity type, there are some common PMI Property keys and corresponding

value formats that appear in JT File. The below table documents these common PMI Property keys (i.e. the keys encoded string value) and what the format of the value data is in the values encoded string (see PMI Property Atom for an explanation of what is meant by “encoded string value” for the “key” and “value” data, see [8] for an explanation of the “%f” format).

Table 75 — Common Property Keys and Their Value Encoding formats

“Key” Property Atom Value String	“Value” Property Atom Value String Encoding Format	Decoding Notes
“PMI_PROP_ANCHOR_POINT”	“Px Py Pz”	Each Px, Py, Pz is a F32 value using “%f” format
“PMI_PROP_NOTE_HAS_URL”	“0” or “1”	0 = = False; 1 = = True
“PMI_PROP_NORMAL_DIR”	“Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using “%f” format
“PMI_PROP_APPROACH_DIR”	“Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using “%f” format
“PMI_PROP_CLAMPING_DIR”	“Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using “%f” format
“PMI_PROP_MEAS_DIR”	“Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using “%f” format
“PMI_PROP_COORD_DIR”	“Dx Dy Dz”	Each Dx, Dy, Dz is a F32 value using “%f” format
“PMI_PROP_MEAS_LEVEL”	“#”	Integer representing level number
“PMITextForegroundColour”	“#”	Hexadecimal integer representing RGB colour where value has “0x00bbgrr” form. The low-order byte contains a value for the relative intensity of red; the second byte contains a value for the relative intensity of green; and the third byte contains a value for the relative intensity of blue. The high-order byte shall be zero. The maximum value for a single byte is 0xFF (i.e. intensity value is in the range [0:255]).
“PMITextBackgroundColour”	“#”	Same as “PMITextForegroundColour”
“PMITextBackgroundOpacity”	“#”	Unsigned decimal integer representing opacity percentage. Actual opacity is: decoded# / 100.0
“PMITextShowBorder”	“#”	Unsigned decimal integer: 0 = = False; 1 = = True
“PMITextSize”	“#”	Unsigned decimal integer representing text size in units of pixels.
“PMITextInPlane”	“#”	Unsigned decimal integer: 0 = = False; 1 = = True where “1” indicates that text should be displayed in the plane of the entity so that it rotates with view.
“PMIGeometryColour”	“#”	Same as “PMITextForegroundColour”
“PMIGeometryWidth”	“#”	Unsigned decimal integer representing line width in units of pixels.
CLIP_NORMAL	“#,#,#”	Used for <u>Entity Type</u> = “0x0114” and <u>Entity Type Name String</u> = “Section” to specify the normal to the clipping plane. The clipping

“Key” Property Atom Value String	“Value” Property Atom Value String Encoding Format	Decoding Notes
		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 <u>Entity Type</u> = “0x0114” and <u>Entity Type Name String</u> = “Section” to specify one point on the clipping plane. Each # is a F64 value using “%lf” format.
TRANSFORMATION_MATRIX	“#,#,#,#,#,#,#,#,#,#,#,#,#”	Used for <u>Entity Type</u> = “0x0114” and <u>Entity Type Name String</u> = “Part Transform” to specify a transformation matrix. Each # is a F32 value using “%f” format.

9.2.6.1.1 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.

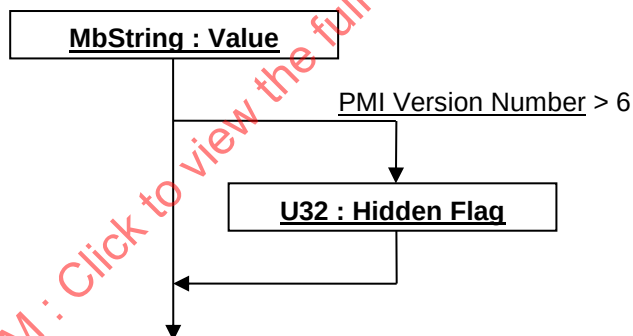


Figure 144 — PMI Property Atom data collection

MbString : Value

Value specifies the property atom value encoded into a String. See Table 75 — Common Property Keys and Their Value Encoding formats above for encoding formats of the Value string.

U32 : Hidden Flag

Hidden Flag specifies if the property is “hidden” or not. A JT file reader could use this flag to control whether read properties should be exposed to the end user of the application reading the JT file. Valid values include the following:

Table 76 — PMI Property Atom Hidden Flag values

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

9.2.7 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 9.2 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.

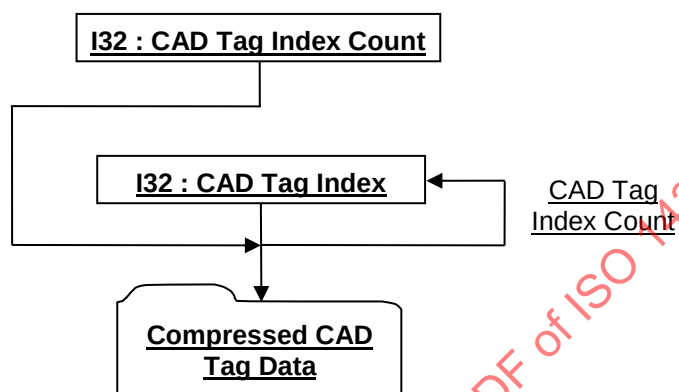


Figure 145 — PMI CAD Tag Data collection

Complete description for Compressed CAD Tag Data can be found in 10.1.16 Compressed CAD Tag Data.

I32 : CAD Tag Index Count

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

- Line Weld Count ;
- Spot Weld Count ;
- SF Count ;
- MP Count ;
- Reference Geometry Count ;
- Datum Target Count ;
- FCF Count ;
- Locator Count ;
- Dimension Count ;
- DFS Count ;
- Note Count ;

- Model View Count ;
- Design Group Count ;
- Coord Sys Count ;
- Generic Entity Count.

I32 : CAD Tag Index

CAD Tag Index specifies an index into a list of CAD Tags, identifying the CAD Tag belonging to a particular PMI entity. There will be a total of CAD Tag Index Count number of CAD Tag Indices and the order of the indices will be as defined by the above documented CAD Tag Index Count formula (i.e. Line Weld CAD Tag Indices are first, followed by the Spot Weld CAD Tag Indices, followed by the Surface Finish CAD Tag Indices, etc.)

9.2.8 PMI Polygon Data

The PMI Polygon Data collection contains a list of vertices classified as polygonal primitives. Its composition is shown in the figure 177. Each block of PMI PolygonData contains a list of 0 or more PolygonData elements. Empty PolygonData elements are written with 0 vertices and no additional fields.

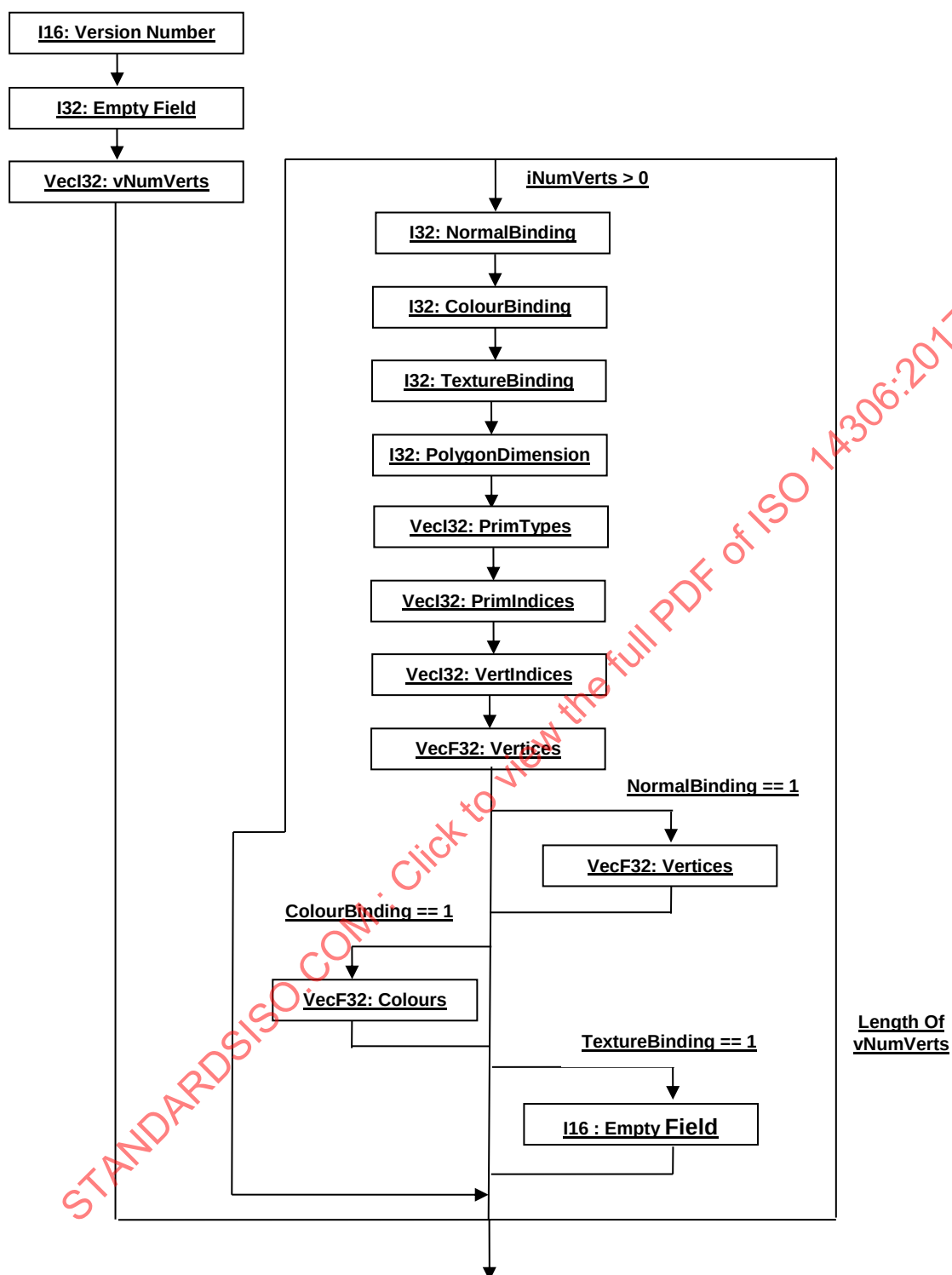


Figure 146 — PMI Polygon Data

I16: Version Number

Version number is the version identifier for this PMI Polygon Data Element. For information on local version numbers see best practice **11.5 Local version numbers.**

I32: Empty Field

Refer to best practice [11.4 Empty Field](#).

VecI32: vNumVerts

An integer vector used to record the number of vertices in each polygon data element. The length of this vector is equal to the number of PolygonData elements written in this block of PMI PolygonData. The presence of additional data fields in each PolygonData element is hinged upon that element having more than 0 vertices recorded in this vector.

Retrieve next vertCount from vNumVerts.

If the next element in the vNumVerts vector is non-zero, proceed to read other fields that make up a single PMI PolygonData element. Otherwise, skip reading more data for this element and loop back to seek the next element in the vector.

iNumVerts.

Number of vertices for the i^{th} PolygonData element.

Length Of vNumVerts.

Number of Polygon Data elements.

I32: NormalBinding

A Boolean value that indicates if there are normals present along with the list of coordinates at each vertex.

I32: ColourBinding

A Boolean value that indicates if there are colours present along with the list of coordinates at each vertex.

I32: TextureBinding

A Boolean value that indicates if there are Texture Coordinates present along with the list of coordinates at each vertex.

I32: PolygonDimension

Indicates the dimension of vertex coordinates.

VecI32: PrimTypes

An array indicating the type of each of the primitive stored in the PrimIndices array. Adjacent numbers in the array form tuples of the form [PrimIndex, PrimType]. All primitives to the left of the PrimIndex are of type PrimType unless they are already to the left of an earlier PrimIndex in this array.

VecI32: PrimIndices

Indices of vertices that form a single primitive. The difference between two adjacent values in this array determines the length of the primitive. An extra element is stored at the end of this array to identify the length of the last primitive. Values in this array are indices into the VertIndices array.

VecI32: VertIndices

An array of indices into the Vertices array. This index array eliminates the need to duplicate floating point vertices that are shared by multiple primitives.

VecF32: Vertices

The list of vertex coordinates. Each vertex is made of PolygonDimension coordinates. The length of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Normals

An optional list of Normals for each vertex. Presence of this list is indicated by the NormalBinding flag. Each normal consists of PolygonDimension components. The size of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Colours

An optional list of Colours for each vertex. Presence of this list is indicated by the ColourBinding flag. Each colour consists of PolygonDimension components. The size of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Texture Coords

An optional list of Texture coordinates for each vertex. Presence of this list is indicated by the TexCoordBinding flag. Each TexCoord consists of 2 components. The size of this list is equal to number of vertices multiplied by 2.

10 Data Compression and Encoding

The JT File format utilizes compression and encoding algorithms to produce compact and efficient representations of data. The types of compression algorithms supported by the ISO 14306 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 ISO 14306 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 ISO 14306 format of varying qualitative levels of compression allows producers of JT data to fine tune the tradeoff between compression ratio and fidelity of the data.

In some instances, data may be encoded/compressed using multiple techniques applied on top of one another in a serial fashion (i.e. encoding applied to the output of another encoder). One common example of this multiple encoding is when an array/vector of floating point data is first quantized into some integer codes and then these resulting integer codes are further compressed/encoded using an Arithmetic or BitLength CODEC (see [10.2 Encoding Algorithms](#)).

Beyond the data collection specific compression/encoding, some ISO 14306 format Data Segment types (see **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 additionally 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 F.

10.1 Common Compression Data Collection Formats

For convenience and brevity in documenting the ISO 14306 format, this section of the reference documents the format for several common “data compression/encoding” related data collections that can exist in the ISO 14306 format. You will find references to these common compression data collections in the [5.2 Data Segments](#) section of the document.

10.1.1 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 ISO 14306 format data compression algorithms and Int32 Compressed Data Packet, “out-of-band data” has the following meaning.

CODECs (e.g. Arithmetic, see [10.2 Encoding Algorithms](#) for technical description) exploit the statistics present in the relative frequencies of the values being encoded. Values that occur frequently enough allow 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 [Int32 Probability Context Table Entry](#) for further details on the representation of “escape” symbol).

Essentially the “out-of-band data” is the high-entropy residue left over after the CODECs have squeezed all the advantage out of the original data stream that they can. However, this “out-of-band data” is sent back around for another pass because sometimes there are *different* statistics to be taken advantage of. When all other coding options have been exhausted, the Bitlength CODEC is invoked. The Bitlength CODEC directly encodes all values given to it, does not require a probability context, and hence never produces additional “out-of-band data”. The byte stops there, in other words.

In some cases, all values may be written as "out of band" when the Codec cannot perform *any* useful compression. In this case, the encoded I32 : CodeText Length field will be 0, and the I32 : Out-Of-Band Value Count will be equal to I32 : Value Element Count. The implied action in this case is to merely copy the Out-Of-Band value data into the output Value Element array instead of invoking the Codec.

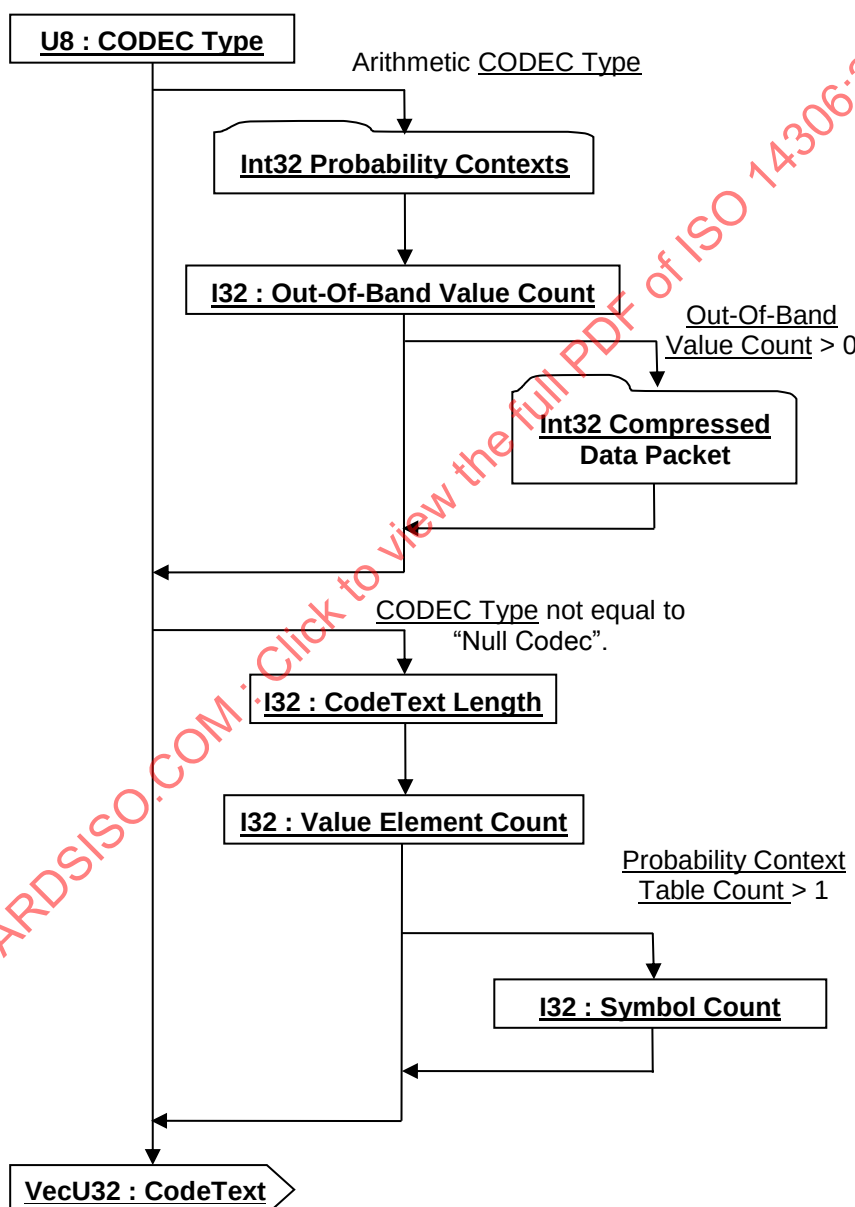


Figure 147 — Int32 Compressed Data Packet data collection

U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See [10.2 Encoding Algorithms](#) for complete explanation of each of the encoding algorithms.

Table 77 — Common Compression CODEC Type values

= 0	Null CODEC
= 1	Bitlength CODEC
= 3	Arithmetic CODEC
= 4	Chopper 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 the Arithmetic CODEC Type.

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 is like the “length” field written if you are 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 Values should be equal to Value Element Count. When only a single Probability Context Table is used, Value Element Count will also be equal to the number of Symbols decoded upon completion of decoding.

I32 : Symbol Count

When two Probability Context Tables are being used, Symbol Count specifies the number of Symbols to be decoded by the Arithmetic CODEC. There is a subtlety present in the method `CodecDriver::addOutputSymbol()` when it is passed an Escape symbol. Only if the Codec is using Probability Context Table 0 when it receives an Escape symbol does it emit a Value from the “Out-Of-Band” data array. Because of this subtlety, the number of Symbols decoded can be larger than the number of Values produced, thus the reason for writing this field distinct from 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.

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 the Arithmetic CODEC Type. A Probability Context Table is a trimmed and scaled histogram of the input values. It tallies the frequencies of the several most frequently occurring values. It is central to the operation of the arithmetic CODEC.

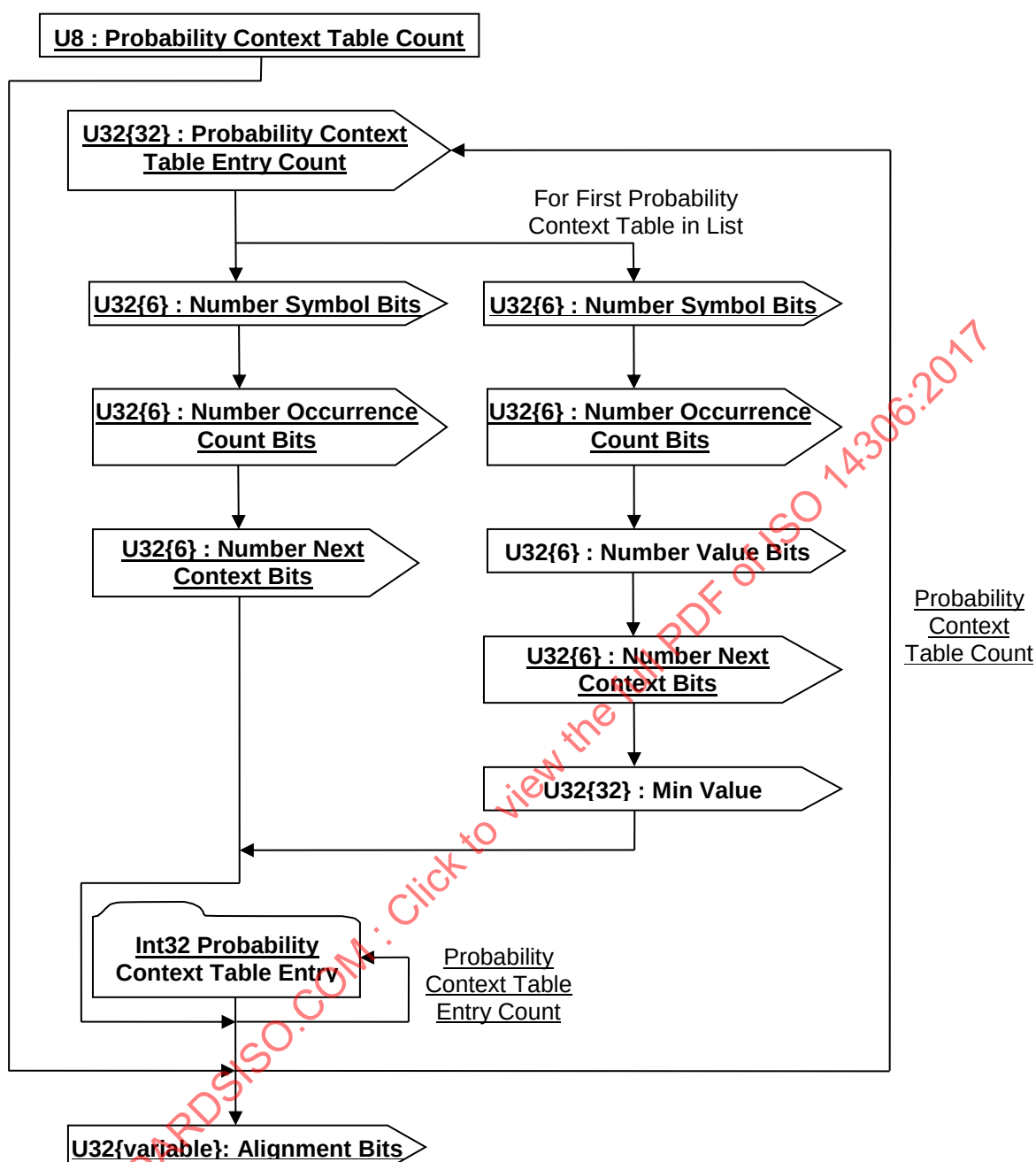


Figure 148 — Int32 Probability Contexts data collection

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. Note that Number Value Bits is only specified in the JT file for the *first* Probability Context Table. If a second Probability Context Table is present, the Number Value Bits from the first should be used for the second as well.

U32{6} : Number Next Context Bits

Number Next Context Field Bits specifies the number of bits used for the Next Context Field in 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 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 JT file is always aligned on bytes. Therefore after reading in a block of bit data such as the probability context tables it is necessary to discard any remaining bits on the last byte that is read in. This is represented by the "Alignment Bits" entry.

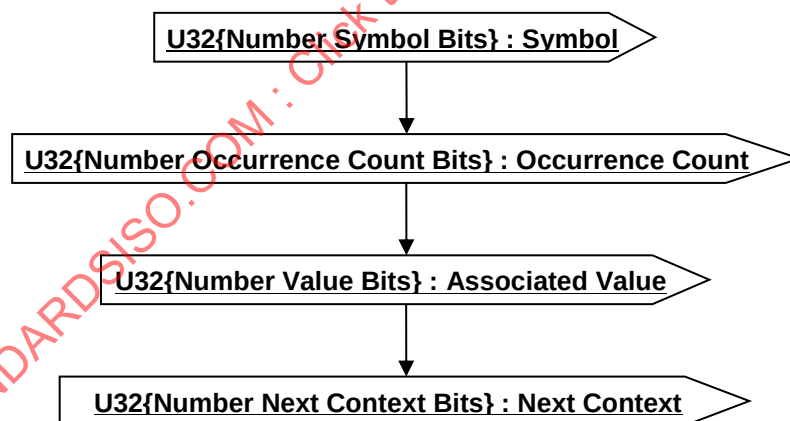
Int32 Probability Context Table Entry

Figure 149 — Int32 Probability Context Table Entry data collection

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 shall subtract "2" from the read value to get the true symbol value. Complete description for Number Symbol Bits can be found in 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 [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 10.1.1 for number of symbols to be decoded). During Arithmetic decoding as described in **Annex D**.

pDriver->numSymbolsToRead() – Refers to the total number of symbols to be decoded (i.e. [Value Element Count](#) in section 10.1.1 when the number of Probability Context Tables is equal to 1, or [Symbol Count](#) when the number of Probability Context Tables is 2).

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 do not directly encode values, they encode symbols. Symbols, then, are associated with specific values, so when the CODEC decodes an array of symbols, you can reconstruct the array of values that was intended by looking up the symbols in the Probability Context Table. This value is stored with “Min Value” subtracted from the value. Complete descriptions for “Min Value” and Number Value Bits can be found in [Int32 Probability Contexts](#).

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

U32{Number Next Context Bits} : Next Context

Next Context field specifies which Probability Context Table to use when decoding the next symbol. The value of this field will be greater than or equal to 0, and less than Probability Context Table Count.

10.1.2 Int32 Compressed Data Packet Mk. 2

The Int32 Compressed Data Packet Mk. 2 collection represents an enhanced form of the original Int32 Compressed Data Packet. Note that the Int32 Compressed Data Packet Mk. 2 collection can in itself contain another Int32 Compressed Data Packet Mk. 2 collection if there are any “Out-Of-Band data.” In the context of the ISO 14306 format data compression algorithms and Int32 Compressed Data Packet Mk. 2, “out-of-band data” has the meaning described below.

Entropy CODECs (e.g. Arithmetic) exploit the statistics present in the relative frequencies of the values being encoded. Values that occur frequently enough allow 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 [Int32 Probability Context Table Entry Mk. 2](#) for further details on the representation of “escape” symbol).

Essentially the “out-of-band data” is the high-entropy residue left over after the CODEC has squeezed all the advantage out of the original data stream that it can. However, this “out-of-band data” is sent back around for another pass because sometimes there are *new* or *different* statistics to be exploited.

The Int32 Compressed Data Packet Mk. 2 brings the new *Chopper* pseudo-CODEC to the table. Its job is to identify fields of bits in a sequence of otherwise incompressible data that may be hiding low-entropy statistics that can be profitably exploited. In other words, it “chops” the input data up into bit fields, and

then encodes them separately using the Arithmetic or BitLength CODECs, or in some cases, another round of chopping. The Chopper also removes *value bias* from the original input data array. Some input data arrays may contain values that are clustered around a certain central value. In these cases, it is profitable to first subtract out a *bias value* from the original input data. In some cases, this simple expedient may dramatically reduce the apparent field width necessary to code the variation in the original sequence.

In some cases, all values may be written as "out of band" when the Codec cannot perform *any* useful compression. In this case, the encoded I32 : CodeText Length field will be 0, and the I32 : Out-Of-Band Value Count will be equal to I32 : Value Element Count. The implied action in this case is to merely copy the Out-Of-Band value data into the output Value Element array instead of invoking the Codec.

When all other coding options have been exhausted, the Bitlength CODEC is invoked. The Bitlength CODEC directly encodes all values given to it, does not require a probability context, and hence never produces additional "out-of-band data". The byte stops there, in other words.

Note that in the diagram below, encoding can loop back recursively for Out-Of-Band data and chopper fields. *For JT v9 files, the maximum recursion depth may not exceed three.*

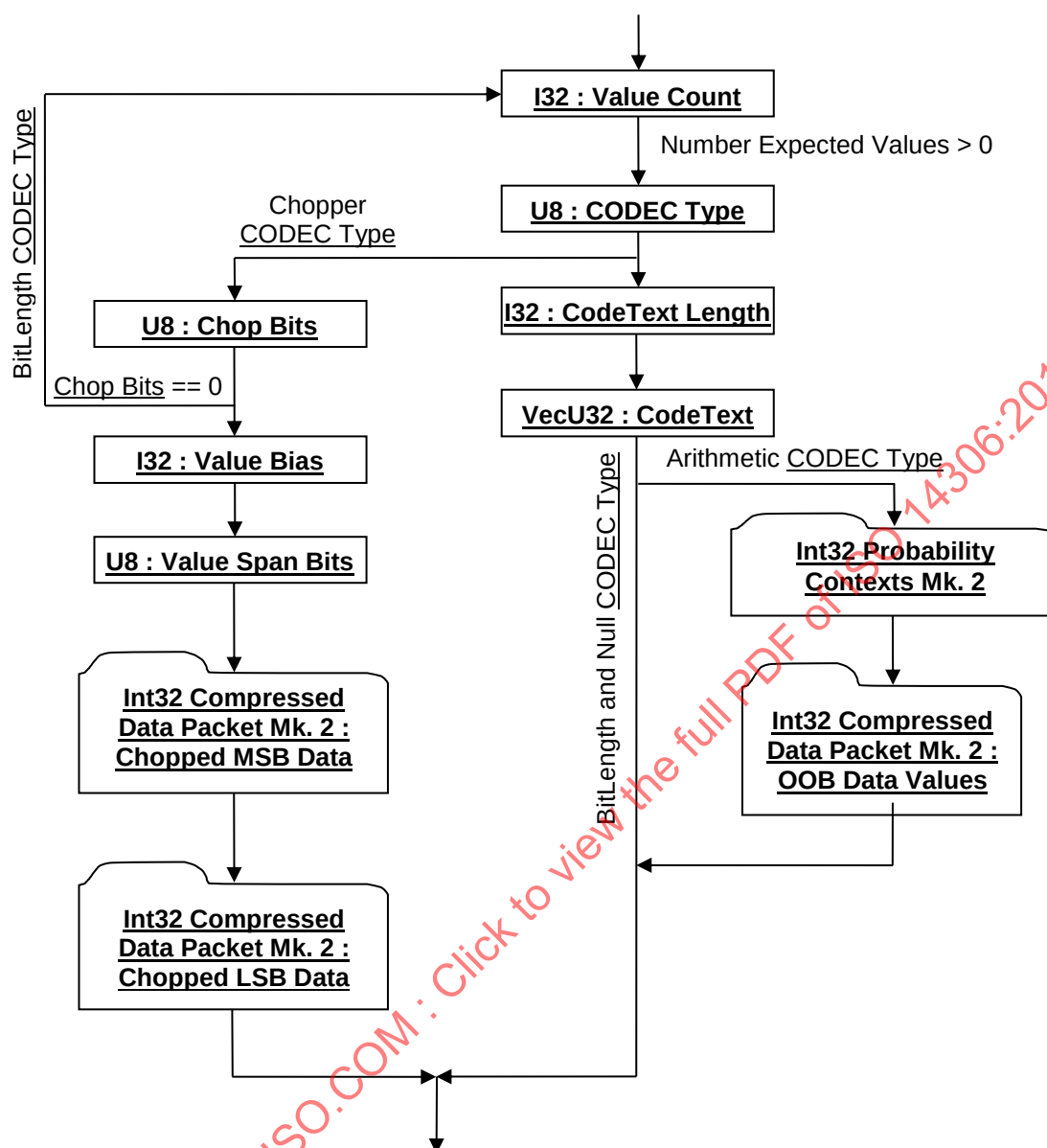


Figure 150 — Int32 Compressed Data Packet Mk. 2 data collection

I32 : Value Count

Value Count specifies the number of values that the CODEC is expected to decode (i.e. it is like the “length” field written if you are just writing out a vector of integers). Upon completion of decoding the CodeText data field below, the number of decoded Values should be equal to Value Count. When only a single Probability Context Table is used, Value Element Count will also be equal to the number of Symbols decoded upon completion of decoding.

U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See 10.2 Encoding Algorithms for complete explanation of each of the encoding algorithms.

Table 78 — Int32 Probability Contexts CODEC Type values

= 0	Null CODEC
= 1	Bitlength CODEC
= 3	Arithmetic CODEC
= 4	Chopper CODEC

I32 : CodeText Length

CodeText Length specifies the total number of bits of CodeText data (CodeText data field is described below).

VecU32 : CodeText

CodeText is the array/vector of encoded symbols. For CODEC Type not equal to "Null CODEC", the total number of bits of encoded data in this array is indicated by the previously described CodeText Length data field.

U8 : Chop Bits

Chop Bits specifies the number of high-order bits "chopped off" from the *biased* input data array and coded separately from the low-order bits. Repeated applications of the Chopper pseudo-CODEC can expose low-entropy bit fields that would be inaccessible by directly coding the data array. Chop Bits is the number of bits coded into the Chopped MSB Data field.

I32 : Value Bias

Value Bias is the (signed) number that is subtracted from the original input data array elements *before* computing Value Span Bits and Chop Bits. See Chopped LSB Data below for a full explanation of how to reconstitute the original data values using Value Bias and the two chopped fields.

U8 : Value Span Bits

Value Span Bits specifies the total bit width of the *biased* input data array. Note that Value Span Bits minus Chop Bits is the number of low-order bits present in the Chopped LSB Data field.

Int32 Compressed Data Packet Mk. 2 : Chopped MSB Data

This field contains the separately compressed most significant bits of the *biased* input data array, whose elements contain Value Span Bits bits of significance. In other words, this field contains the bit field from the *biased* data array beginning at bit number ValueSpan-ChopBits and ending at bit number ValueSpan-1 inclusive. This field may contain negative numbers.

Int32 Compressed Data Packet Mk. 2 : Chopped LSB Data

This field contains the separately compressed most significant bits of the original input data array, whose elements contain Value Span Bits bits of significance. In other words, this field contains the bit field from the original data array beginning at bit number 0 and ending at bit number ValueSpan-ChopBits-1 inclusive. This field may only contain positive numbers; all bits above this range shall encode to 0. A pseudo-code representation of the re-constituting the original data values is as follows:

$$\text{OrigValue}[i] = (\text{LSBValue}[i] \mid (\text{MSBValue}[i] \ll (\text{ValSpanBits} - \text{ChopBits}))) + \text{ValueBias};$$

Int32 Compressed Data Packet Mk. 2 : OOB Data Values

This field encodes the out-of-band values associated with the Arithmetic CODEC.

Int32 Probability Contexts Mk. 2

Int32 Probability Contexts Mk. 2 data collection encodes a Probability Context Table, and is present only for the Arithmetic CODEC Type. A Probability Context Table is a trimmed and scaled histogram of the

input values. It tallies the frequencies of the several most frequently occurring values. It is central to the operation of the Arithmetic CODEC.

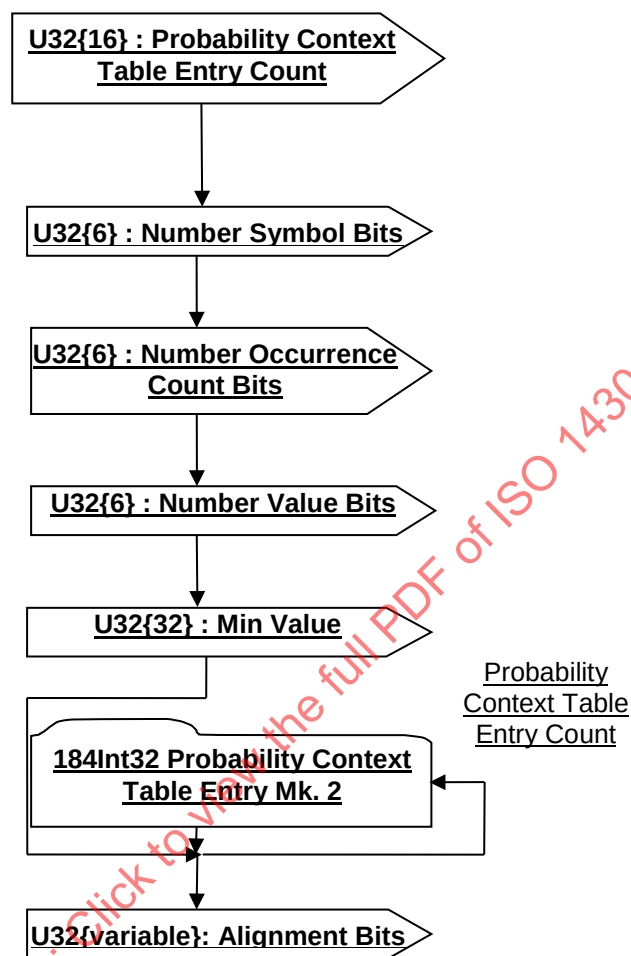


Figure 151 — Int32 Probability Contexts Mk. 2 data collection

U32{16} : Probability Context Table Entry Count

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

U32{6} : Number 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. Note that Number Value Bits is only specified in the JT file for the *first* Probability Context Table. If a second Probability Context Table is present, the Number Value Bits from the first should be used for the second as well.

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 [Int32 Probability Context Table Entry](#).

U32{variable}: Alignment Bits

Alignment Bits represents the number of additional padding bits stored to arrive at the next even multiple of 8 bits. Values of “0” are stored in the alignment bits.

Note: Data written into a JT file is always aligned on bytes. Therefore after reading in a block of bit data such as the probability context tables it is necessary to discard any remaining bits on the last byte that is read in. This is represented by the “Alignment Bits” entry.

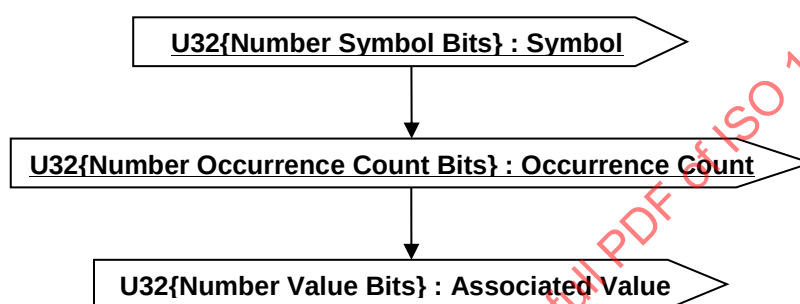
Int32 Probability Context Table Entry Mk. 2

Figure 152 — Int32 Probability Context Table Entry Mk. 2 data collection

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 shall subtract “2” from the read value to get the true symbol value. Complete description for Number Symbol Bits can be found in [Int32 Probability Contexts Mk. 2](#).

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 [Int32 Probability Contexts Mk. 2](#).

Note: Occurrence Counts for all symbols are normalized (converted to a relative frequency) during the write process in order to ensure the minimum amount of bits possible is used to write them while closely approximating their actual frequency. This has several implications the reader should be aware of:

The sum of all Occurrence Counts is not guaranteed to equal the number of symbols to be decoded (see [I32 : Value Count](#) in section [10.1.2](#) for number of symbols to be decoded).

During Arithmetic decoding as described in **Annex D**.

pDriver->numSymbolsToRead () – Refers to the total number of symbols to be decoded (i.e. [I32 : Value Count](#) in section [10.1.2](#)).

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 do not directly encode values, they encode symbols. Symbols, then, are associated with specific values, so when the CODEC decodes an array of symbols, you can reconstruct the array of values that was intended by looking up the symbols in the Probability Context Table. This value is stored with “Min Value” subtracted from the value. Complete descriptions for “Min Value” and Number Value Bits can be found in Int32 Probability Contexts Mk. 2.

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

10.1.3 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 ISO 14306 format data compression algorithms and Float64 Compressed Data Packet, “out-of-band data” has the following meaning.

The Arithmetic CODEC (see 10.2 Encoding Algorithms for technical description) can exploit the statistics present in the relative frequencies of the values being encoded. Values that occur frequently enough allow the CODEC 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 marker 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 it can.

Whereas the Int32 Compressed Data Packet (see 10.1.1 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.

In some cases, all values may be written as “out of band” when the Codec cannot perform *any* useful compression. In this case, the encoded I32 : CodeText Length field will be 0, and the I32 : Out-Of-Band Value Count will be equal to I32 : Value Element Count. The implied action in this case is to merely copy the Out-Of-Band value data into the output Value Element array instead of invoking the Codec.

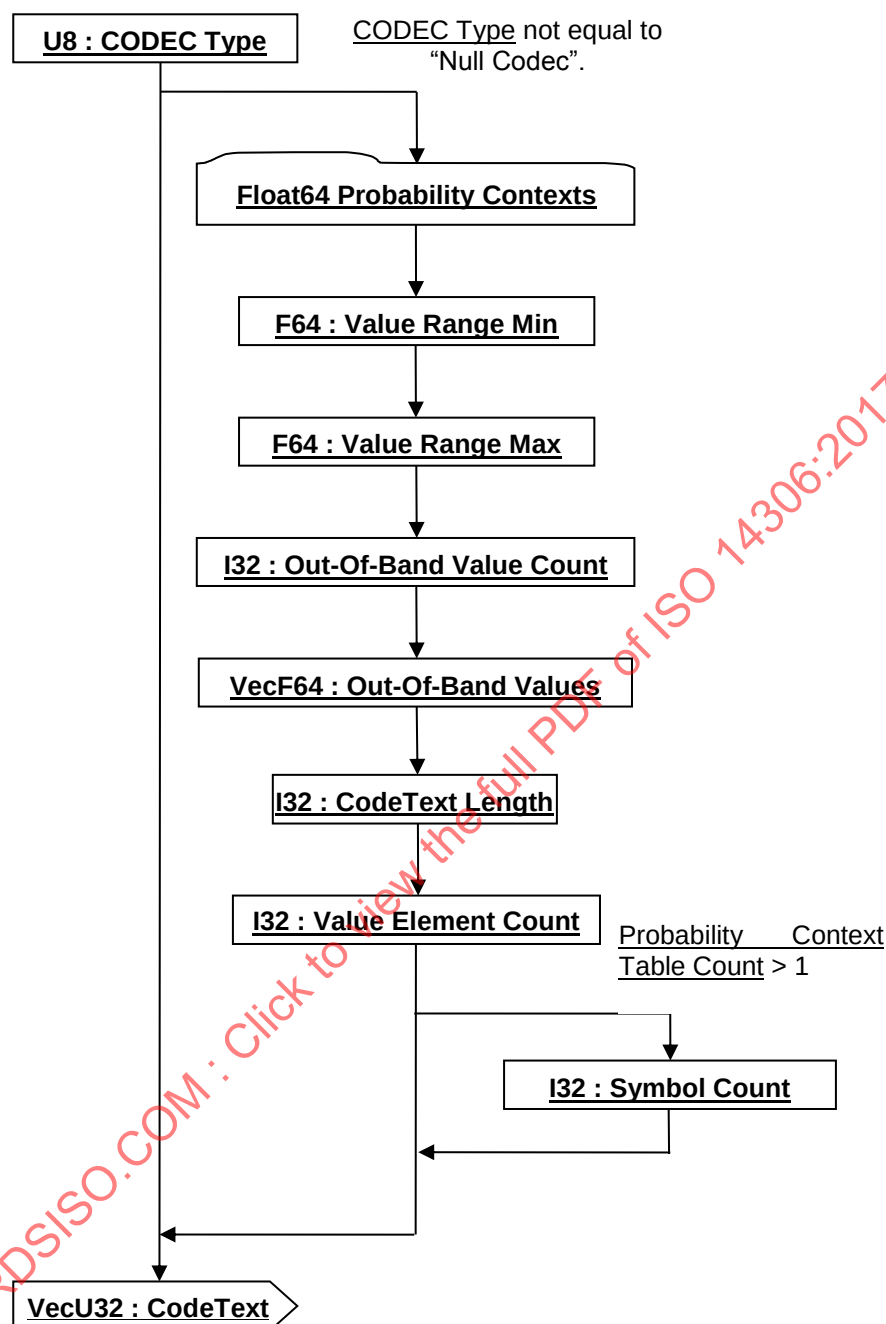


Figure 153 — Float64 Compressed Data Packet data collection

U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See [10.2 Encoding Algorithms](#) for complete explanation of each of the encoding algorithms.

Table 79 — Float64 Compressed Data Packet CODEC Type values

= 0	Null CODEC
= 1	Bitlength CODEC
= 3	Arithmetic CODEC
= 4	Chopper CODEC

F64 : Value Range Min

Value Range Min specifies the minimum of the value range used to encode the values. This data field is only present if CODEC Type is not equal to "Null CODEC."

F64 : Value Range Max

Value Range Max specifies the maximum of the value range used to encode the values. This data field is only present if CODEC Type is not equal to "Null 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 if CODEC Type is not equal to "Null CODEC."

VecF64 : Out-Of-Band Values

Out-Of-Band Values specifies the vector/list of "Out-Of-Band" values. This data field is only present if CODEC Type is not equal to "Null CODEC."

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 is like the "length" field written if you are 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.

I32 : Symbol Count

When two Probability Context Tables are being used, Symbol Count specifies the number of Symbols to be decoded by the Arithmetic CODEC. There is a subtlety present in the method `CodecDriver::addOutputSymbol()` when it is passed an Escape symbol. Only if the Codec is using Probability Context Table 0 when it receives an Escape symbol does it emit a Value from the "Out-Of-Band" data array. Because of this subtlety, the number of Symbols decoded can be larger than the number of Values produced, thus the reason for writing this field distinct from 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.

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.

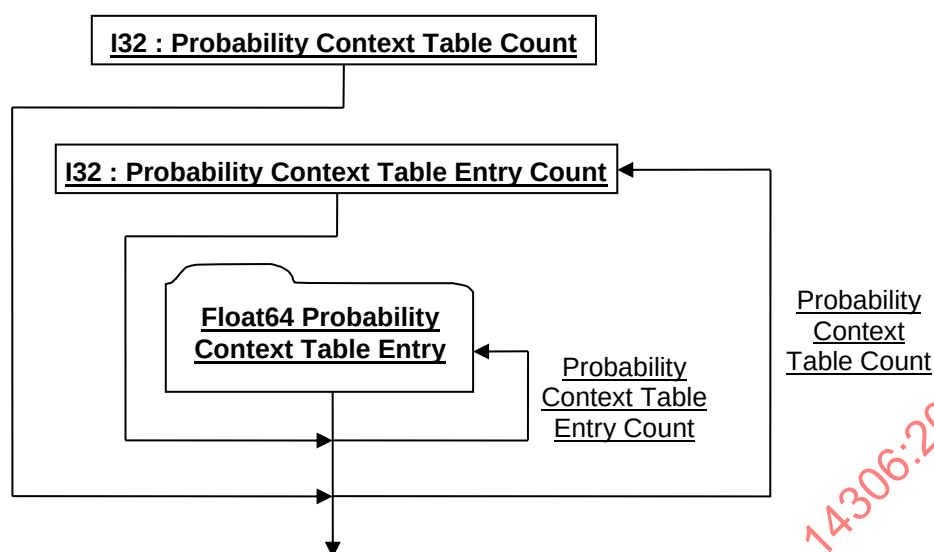


Figure 154 — Float64 Probability Contexts data collection

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.

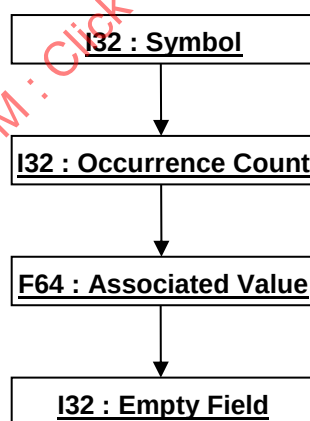
Float64 Probability Context Table Entry

Figure 155 — Float64 Probability Context Table Entry data collection

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 [10.1.3 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 do not 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 : Empty Field

Refer to best practice [11.4 Empty Field](#).

10.1.4 Compressed Vertex Coordinate Array

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

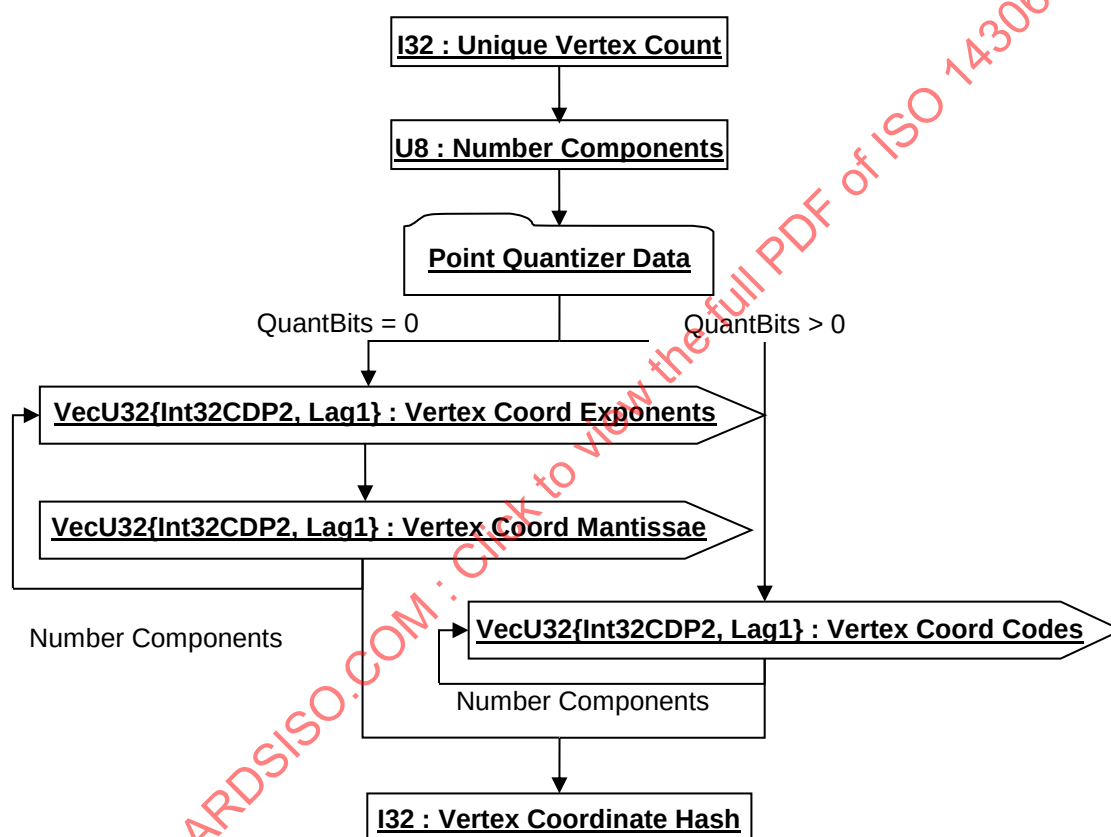


Figure 156 — Compressed Vertex Coordinate Array data collection

Complete description for Point Quantizer Data can be found in Point Quantizer Data.

The above predicates “QuantBits = 0” and “QuantBits > 0” refer to the value of the field U8 : Number Of Bits stored in the three components of [10.1.9 Point Quantizer Data](#). All three of these fields are required to be equal.

I32 : Unique Vertex Count

Vertex Count specifies the count (number of unique) vertices in the Vertex Codes arrays. Identical values are only stored once therefore it may be necessary to smear out the vertices as described in [TopoMesh Compressed Rep Data V1](#) and [TopoMesh Topologically Compressed LOD Data](#).

U8 : Number Components

Number Components specifies the number of vertex components present for each vertex record in the set of vertex records.

VecU32{Int32CDP2, Lag1} : Vertex Coord Exponents

Vertex Coord Exponents is a vector of Floating Point Exponents and Sign for all the *i*th component values of a set of vertex coordinates. Vertex Coord Exponents uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Vertex Coord Mantissae

Vertex Coord Mantissae is a vector of Floating Point Mantissae for all the *i*th component values of a set of vertex coordinates. Vertex Coord Mantissae uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Vertex Coord Codes

Vertex Coord Codes is a vector of quantizer “codes” for all the *i*th component values of a set of vertex coordinates. Vertex Coord Codes uses the Int32 version of the CODEC to compress and encode data.

I32 : Vertex Coordinate Hash

The Vertex Coordinate Hash is the combined hash of the unique vertex coordinate records. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex coordinate values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the vertex coordinates codes for each of the component arrays. Refer to section 11.6 for a more detailed description on hashing.

```
UInt32 uHash      = 0;
uInt32 nUniqVtx   = 0;
vecF32 vCoord[nUniqVtx][3];
vecU32 vCodes[3];
...
if ( uQuantBits == 0 ) {
    for ( int i=0 ; i<nComp ; i++ ) {
        for ( int j=0 ; j<nUniqVtx ; j++ ) {
            uHash = hash32( (UInt32*)&vCoord[j][i], 1, uHash );
        }
    }
} else {
    for ( int i=0 ; i<nComp ; i++ ) {
        uHash = hash32( &vCodes[i], nUniqVtx, uHash );
    }
}
```

10.1.5 Compressed Vertex Normal Array

The Compressed Vertex Normal Array data collection contains the compressed data/representation for a set of vertex normals. Compressed Vertex Normal Array data collection is only present if previously read vertex bindings denote normals are presents (See [Vertex Shape LOD Data U64 : Vertex Bindings](#) for complete explanation of the vertex bindings).

A variation of the CODEC developed by Michael Deering at Sun Microsystems is used to encode the normals when quantization is enabled. The variation being that the “Sextants” are arranged differently than in Deering’s scheme [9], for better delta encoding. See [10.2.4 Deering Normal CODEC](#) for a complete explanation on the Deering CODEC used.

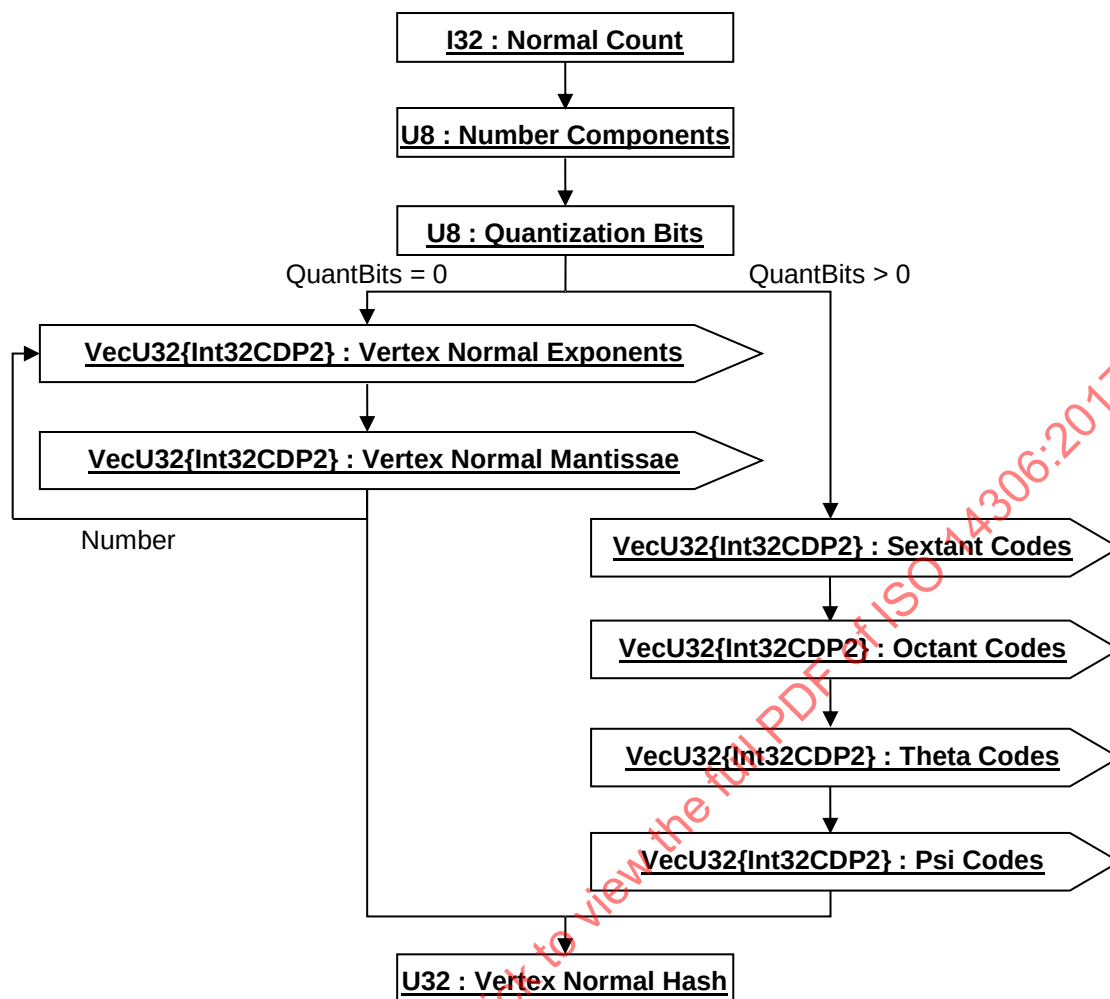


Figure 157 — Compressed Vertex Normal Array data collection

I32 : Normal Count

Normal count specifies the number of normals. This number should equal the total number of vertex records.

U8 : Number Components

Number Components specifies the number of normal components present for each vertex record in the set of vertex records.

U8 : Quantization Bits

The number of bits used when the Deering Normal CODEC if quantization is enabled. A value of 0 denotes that quantization is disabled.

VecU32{Int32CDP2} : Vertex Normal Exponents

Vertex Normal Components is a vector of Floating Point Exponents for all the *i*th component values of a set of vertex coordinates. Vertex Normal Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Vertex Normal Mantissae

Vertex Normal Components is a vector of Floating Point Mantissae for all the *i*th component values of a set of vertex coordinates. Vertex Normal Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : 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{Int32CDP2} : 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{Int32CDP2} : 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{Int32CDP2} : 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.

U32 : Vertex Normal Hash

The Vertex Normal Hash is the combined hash of the vertex normals. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex normal values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the Sextant, Octant, Theta, and Psi Codes for all vertex records. Refer to section 11.6 for a more detailed description on hashing.

```
UInt32 uHash    = 0;
uInt32 nVtxRec = 0;
vecF32 vNorm[nVtxRec][3];
vecU32 vSextant, vOctant, vTheta, vPsi;
...
if ( uQuantBits == 0 ) {
    for ( int i=0 ; i<nComp ; i++ ) {
        for ( int j=0 ; j<nVtxRec ; j++ ) {
            uHash = hash32( (UInt32*)&vNorm[j][i], 1, uHash );
        }
    }
} else {
    uHash = hash32( &vSextant, nVtxRec, uHash );
    uHash = hash32( &vOctant, nVtxRec, uHash );
    uHash = hash32( &vTheta, nVtxRec, uHash );
    uHash = hash32( &vPsi, nVtxRec, uHash );
}
```

10.1.6 Compressed Vertex Texture Coordinate Array

The Compressed Vertex Texture Coordinate Array data collection contains the quantization data/representation for a set of vertex texture coordinates. Compressed Vertex Texture Coordinate Array data collection is only present if previously read vertex bindings denote texture coordinates are

presents (See [Vertex Shape LOD Data U64 : Vertex Bindings](#) for complete explanation of the vertex bindings).

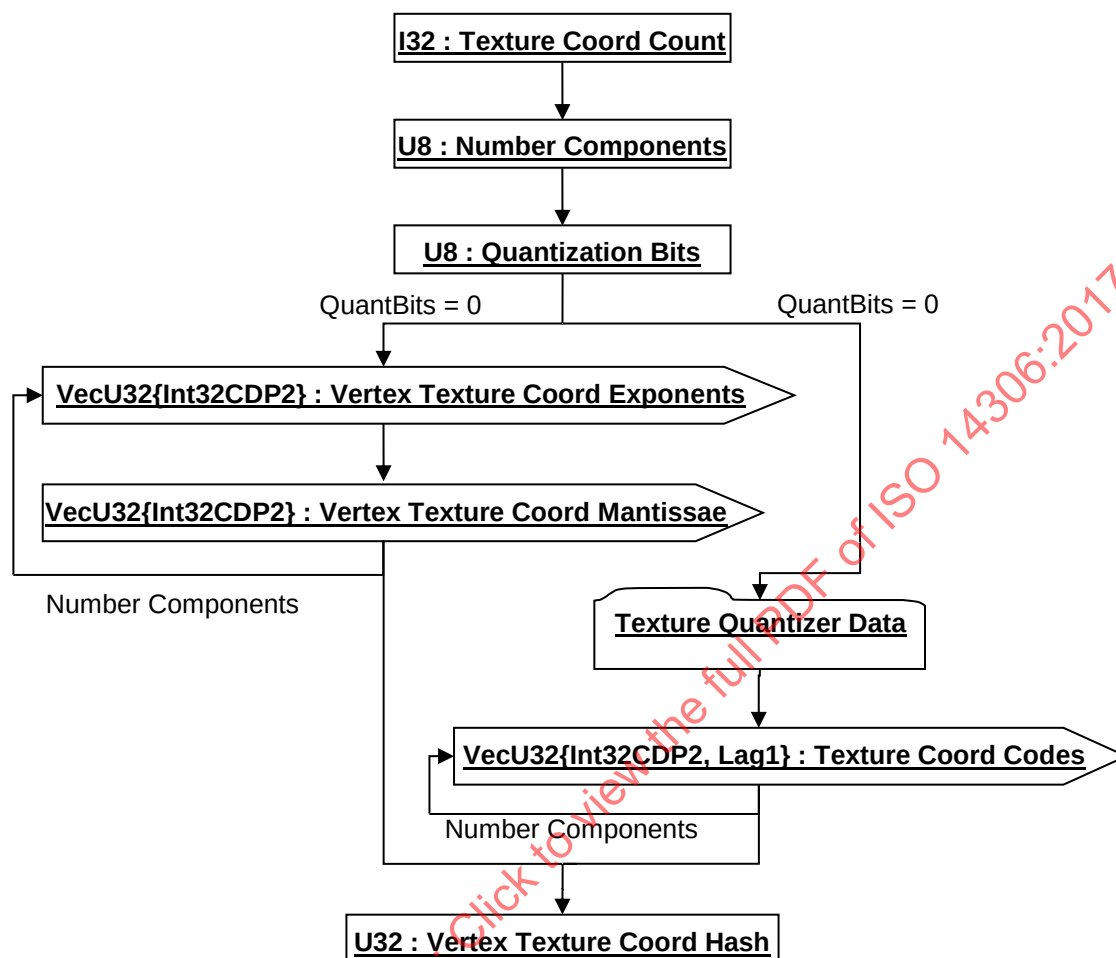


Figure 158 — Compressed Vertex Texture Coordinate Array data collection

Complete description for Texture Quantizer Data can be found in [10.1.10 Texture Quantizer Data](#).

I32 : Texture Coord Count

Colour count specifies the number of Texture Coordinates. This number should equal the total number of vertex records.

U8 : Number Components

Number Components specifies the number of Texture Coordinate components present for each vertex record in the set of vertex records.

U8 : Quantization Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision) for each of the components. The actual number of quantization bits used is specified within [Texture Quantizer Data](#). Value shall be within range [0:24] inclusive.

VecU32{Int32CDP2} : Vertex Texture Coord Exponents

Vertex Texture Coordinate Components is a vector of Floating Point Exponents for all the *i*th component values of a set of vertex coordinates. Vertex Texture Coordinate Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Vertex Texture Coord Mantissae

Vertex Texture Coordinate Components is a vector of Floating Point Mantissae for all the *i*th component values of a set of vertex coordinates. Vertex Texture Coordinate Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Texture Coord Codes

V-Texture Coord Codes is a vector of quantizer “codes” for all the *n*th-component of a set of vertex texture coordinates. V-Texture Coord Codes uses the Int32 version of the CODEC to compress and encode data.

U32 : Vertex Texture Coord Hash

The Vertex Texture Coord Hash is the combined hash of the Vertex Texture Coordinates. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex texture coordinate values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the vertex texture coordinates codes for each of the component arrays. Refer to section 11.6 for a more detailed description on hashing.

```

UInt32 uHash      = 0;
uInt32 nVtxRec    = 0;
vecF32 vTexCoord[nVtxRec][4];
vecU32 vCodes[4];
...
if ( uQuantBits == 0 ) {
    for ( int i=0 ; i<nComp ; i++ ) {
        for ( int j=0 ; j<nVtxRec ; j++ ) {
            uHash = hash32( (UInt32*)(&vTexCoord[j][i]), 1, uHash );
        }
    }
} else {
    for ( int i=0 ; i<nComp ; i++ ) {
        uHash = hash32( &vCodes[i], nVtxRec, uHash );
    }
}

```

10.1.7 Compressed Vertex Colour Array

The Compressed Vertex Colour Array data collection contains the quantization data/representation for a set of vertex colours. Compressed Vertex Colour Array data collection is only present if previously read Colour Binding value is not equal to zero (See [Vertex Shape LOD Data](#) for complete explanation of Colour Binding data field).

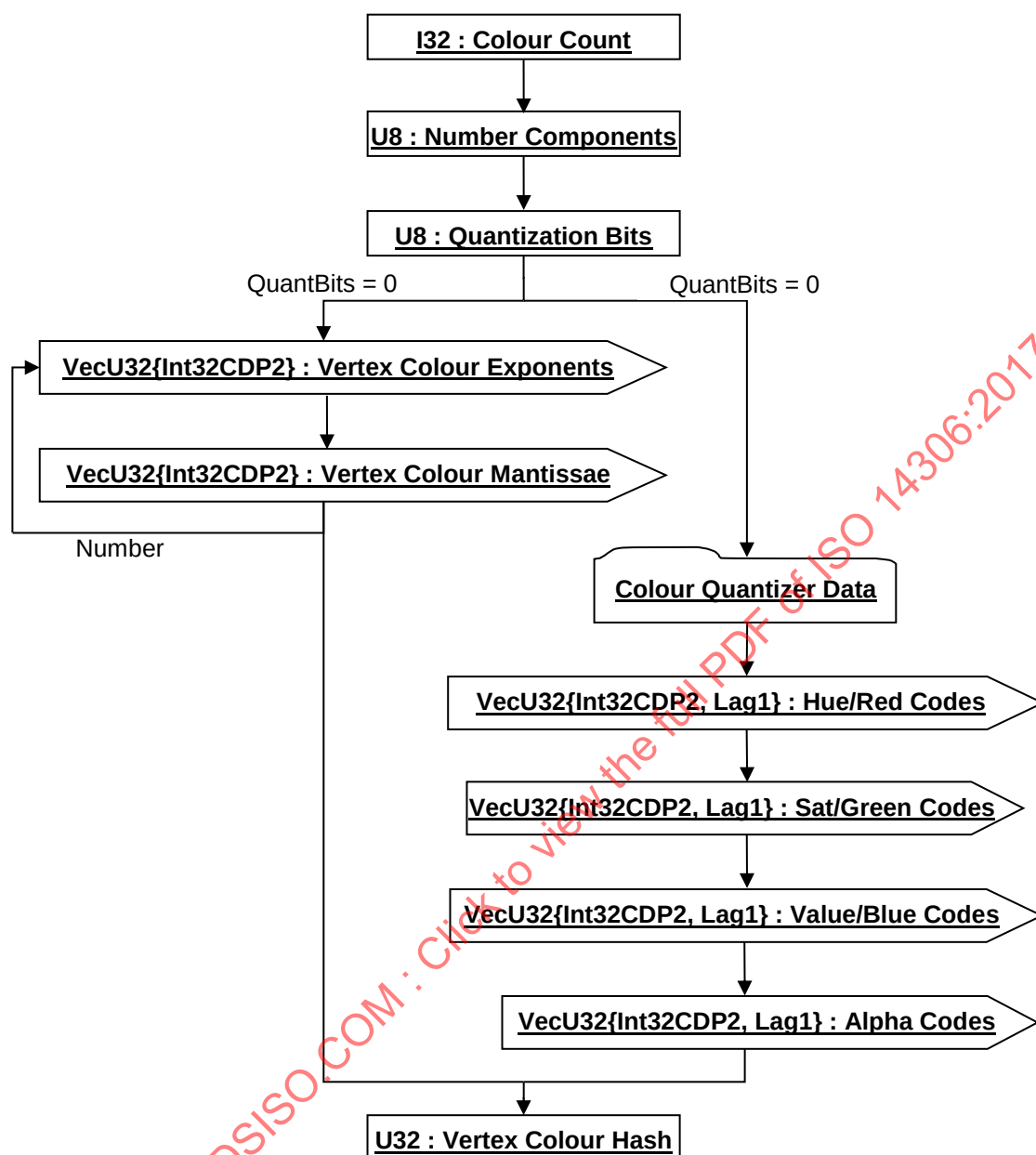


Figure 159 — Compressed Vertex Colour Array data collection

Complete description for Colour Quantizer Data can be found in [10.1.11 Colour Quantizer Data](#).

I32 : Colour Count

Colour count specifies the number of colour records. This number should equal the total number of vertex records.

U8 : Number Components

Number Components specifies the number of Colour components present for each vertex record in the set of vertex records.

U8 : Quantization Bits

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

VecU32{Int32CDP2} : Vertex Colour Exponents

Vertex Normal Components is a vector of Floating Point Exponents for all the ith component values of a set of vertex coordinates. Vertex Normal Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Vertex Colour Mantissae

Vertex Normal Components is a vector of Floating Point Mantissae for all the ith component values of a set of vertex coordinates. Vertex Normal Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Hue/Red Codes

Hue/Red Codes is a vector of quantizer “codes” for all the Hue/Red colour components of a set of vertex colours. Hue/Red Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Sat/Green Codes

Sat/Green Codes is a vector of quantizer “codes” for all the Saturation/Green colour components of a set of vertex colours. Sat/Green Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Value/Blue Codes

Value/Blue Codes is a vector of quantizer “codes” for all the Value/Blue colour components of a set of vertex colours. Value/Blue Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Alpha Codes

Alpha Codes is a vector of quantizer “codes” for all the Alpha colour components of a set of vertex colours. Alpha Codes uses the Int32 version of the CODEC to compress and encode data.

U32 : Vertex Colour Hash

The Vertex Colour Hash is the combined hash of the vertex colours. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex colour values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the Hue/Red, Sat/Green, Value/Blue, and Alpha Codes for all vertex records. Refer to section 11.6 for a more detailed description on hashing.

```
UInt32 uHash      = 0;
UInt32 nVtxRec    = 0;
vecF32 vCol[nVtxRec][3];
vecU32 vHue, vSat, vVal, vAlp;
...
if ( uQuantBits == 0 ) {
    for ( int i=0 ; i<nComp ; i++ ) {
        for ( int j=0 ; j<nVtxRec ; j++ ) {
            uHash = hash32( (UInt32*)&vCol[j][i], nVtxRec, uHash );
        }
    }
} else {
    uHash = hash32( &vHue, nVtxRec, uHash );
    uHash = hash32( &vSat, nVtxRec, uHash );
    uHash = hash32( &vVal, nVtxRec, uHash );
    uHash = hash32( &vAlp, nVtxRec, uHash );
}
```

10.1.8 Compressed Vertex Flag Array

The Compressed Vertex Flag Array data collection contains the quantization data/representation for per vertex flags. Compressed Vertex Flag Array data collection is only present if previously read Vertex Flag Binding value is not equal to zero.

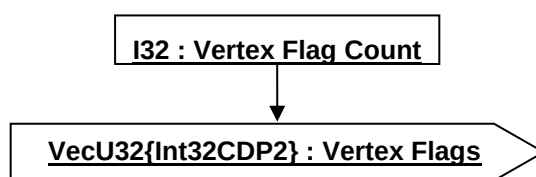


Figure 160 — Compressed Vertex Flag Array data collection

I32 : Vertex Flag Count

Vertex flag count specifies the number of vertex flags. This number should be equal to the total number of vertex records.

VecU32{Int32CDP2} : Vertex Flags

Vertex Flags is a vector of per vertex bit flags encoded as integers with valid values of either 0 (false) or 1 (true). Vertex Flags uses the Int32 version of the CODEC to compress and encode data.

10.1.9 Point Quantizer Data

A Point Quantizer Data collection is made up of three Uniform Quantizer Data collections; there is a separate Uniform Quantizer Data collection for the X, Y, and Z values of point coordinates.

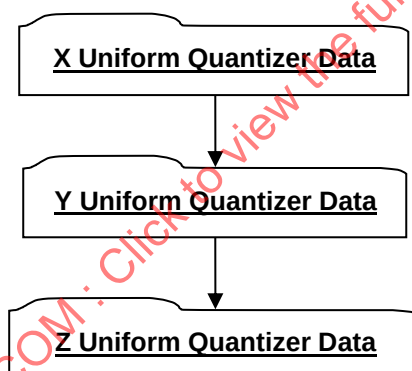


Figure 161 — Point Quantizer Data collection

Complete description for X Uniform Quantizer Data, Y Uniform Quantizer Data and Z Uniform Quantizer Data can be found in [10.1.12 Uniform Quantizer Data](#).

10.1.10 Texture Quantizer Data

A Texture Quantizer Data collection is made up of n Uniform Quantizer Data collections; there is a separate Uniform Quantizer Data collection for each component of the texture coordinates. The number of components is not specified within the quantizer, but rather is determined by the number of texture components present in the underlying data (See [Compressed Vertex Texture Coordinate Arrays U8 : Number Components](#)).

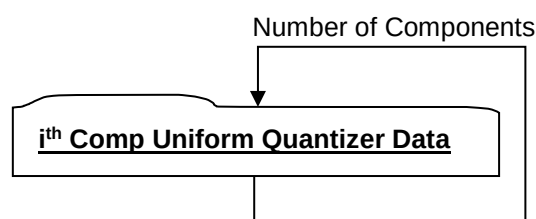


Figure 162 — Texture Quantizer Data collection

Complete description for U Uniform Quantizer Data, and V Uniform Quantizer Data can be found in [10.1.12 Uniform Quantizer Data](#).

10.1.11 Colour Quantizer Data

A Colour Quantizer Data collection contains the quantizer information for each of the colour components. The Colour Quantizer utilizes a separate Uniform Quantizer Data collection for each of the 4 colour components, but if the HSV colour model is being used, then it is not necessary to store a complete Uniform Quantizer Data Collection.

For the HSV model, since the range values for each colour component are constant, only the Number of Bits of precision for each colour component's Uniform Quantizer is stored. The Uniform Quantizer range values for the HSV colour components should always be assumed to be the following:

Table 80 — Colour Quantizer values

Component	Quantizer Range	
	Min	Max
Hue	0.0	6.0
Saturation	0.0	1.0
Value	0.0	1.0
Alpha	0.0	1.0

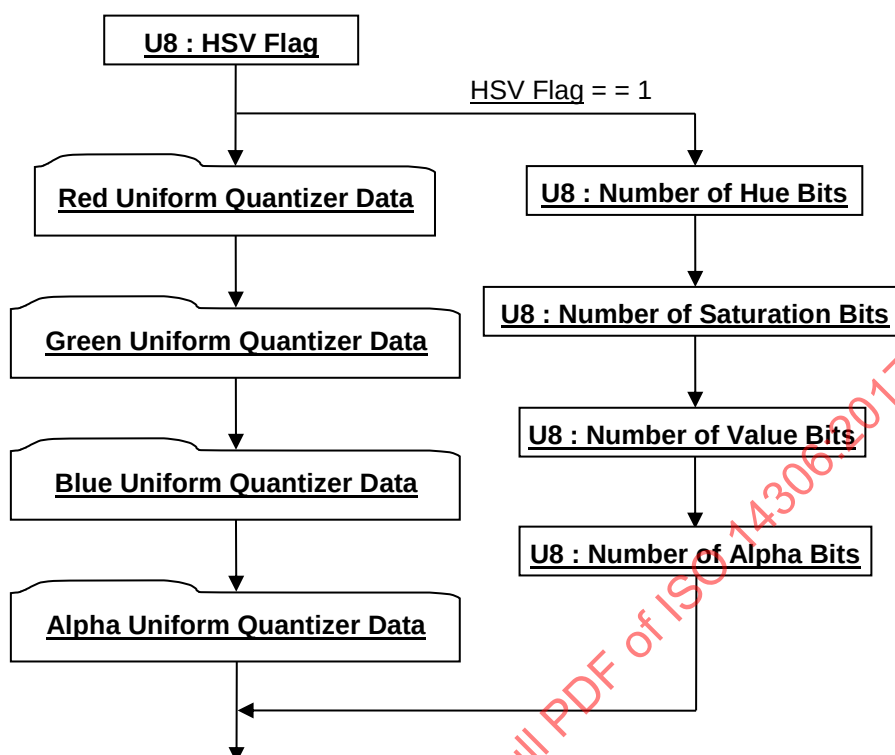


Figure 163 — Colour Quantizer Data collection

Complete descriptions for Red Uniform Quantizer Data, Green Uniform Quantizer Data, Blue Uniform Quantizer Data, and Alpha Uniform Quantizer Data can be found in [10.1.12 Uniform Quantizer Data](#). These four Uniform Quantizer Data collections are only present when data field HSV Flag = 0.

U8 : HSV Flag

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

Table 81 — Colour Quantizer HSV Flag values

= 0	Colour component data stored in RGB colour model form.
= 1	Colour component data stored in HSV colour model form.

U8 : Number of Hue Bits

Number of Hue Bits specifies the quantized size (i.e. the number of bits of precision) for the Hue component of the colour. Number of Hue Bits data is only present when data field HSV Flag = 1.

U8 : Number of Saturation Bits

Number of Saturation Bits specifies the quantized size (i.e. the number of bits of precision) for the Saturation component of the colour. Number of Saturation Bits data is only present when data field HSV Flag = 1.

U8 : Number of Value Bits

Number of Value Bits specifies the quantized size (i.e. the number of bits of precision) for the Value component of the colour. Number of Value Bits data is only present when data field HSV Flag = 1.

U8 : Number of Alpha Bits

Number of Alpha Bits specifies the quantized size (i.e. the number of bits of precision) for the Alpha component of the colour. Number of Alpha Bits data is only present when data field HSV Flag = 1.

10.1.12 Uniform Quantizer Data

The Uniform Quantizer Data collection contains information that defines a scalar quantizer/dequantizer (encoder/decoder) whose range is divided into levels of equal spacing.

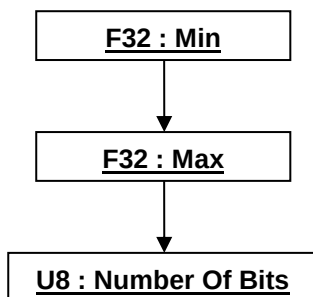


Figure 164 — Uniform Quantizer Data collection

F32 : Min

Min specifies the minimum of the quantized range.

F32 : Max

Max specifies the maximum of the quantized range.

U8 : Number Of Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision). In general, this value shall satisfy the following condition: "0 ≤ Number Of Bits ≤ 32".

10.1.13 Compressed Entity List for Non-Trivial Knot Vector

Compressed Entity List for Non-Trivial Knot Vector data collection specifies index identifiers (i.e. indices to particular entities within a list of entities) for a set of entities that contain Non-Trivial Knot Vectors. The entity types which can contain non-trivial knot vectors include:

JT B-Rep NURBS Surfaces;

JT B-Rep PCS NURBS Curves;

JT B-Rep MCS NURBS Curves;

Wireframe MCS NURBS Curves.

Note that any one occurrence of Compressed Entity List for Non-Trivial Knot Vector data collection will only contain index identifiers for one particular type of the above listed entities. The entity type is inferred based on the data collection which includes/references the Compressed Entity List for Non-Trivial Knot Vector.

A trivial knot vector is one which completely satisfies all conditions of at least one of the following cases:

Case-1 for trivial knot vector:

- number of knots is an even number;
- knot vector has a [0:1] knot range;
- there are no interior knots (i.e. $\text{NumberKnots} = 2 * (\text{NurbsEntityDegree} + 1)$).

Case-2 for trivial knot vector:

- number of knots is an even number;
- Knot vector has a [0:1] knot range;
- $\text{NurbsEntityDegree} < 3$;
- difference between successive non-repeating knots (i.e. KnotDelta) is:
 - $\text{KnotDelta} = 2.0 / (\text{NumberKnots} - (2.0 * \text{NurbsEntityDegree}))$.

Any knot vector which does not satisfy one of the above cases for “trivial knot vector” is classified as a “non-trivial knot vector.”

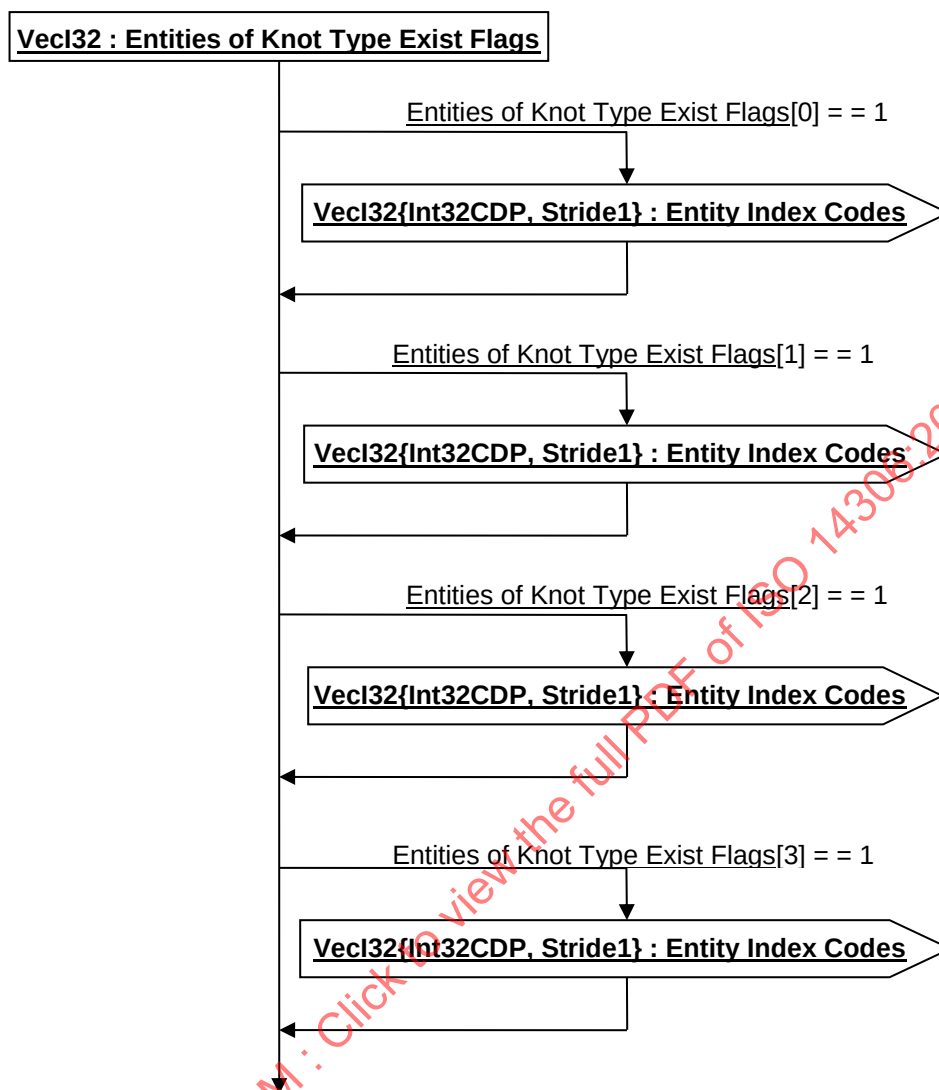


Figure 165 — Compressed Entity List for Non-Trivial Knot Vector data collection

VecI32 : Entities of Knot Type Exist Flags

Entities of Knot Type Exist Flags, is a vector of flags indicating for each knot vector type whether Entity Index ID data collections exist/follow for that knot vector type. Knot Vectors are categorized into types based on the following characteristics: whether internal knots occur in *adjacent pairs* and whether the knot range is [0:1] or some other [x₁: x₂] range.

Currently there are four knot vector types, so this Entities of Knot Type Exist Flags vector should be of length four. The four flags have the following meaning:

Table 82 — Knot Type Exist Flag values

[0]	Flag indicating whether Entity IDs data collection exists for “Even Count [0:1] Range” knot type. Knots in this category have their knot range on [0:1], internal knots occur in <i>adjacent pairs</i>, except when there are no internal knots, in which case Type = 1 instead. = 0 – No Entity IDs data collection exists. = 1 – Entity IDs data collection exists.

[1]	Flag indicating whether Entity IDs data collection exists for “Even Count [x ₁ :x ₂] Range” knot type. Knots in this category have their knot range on [x ₁ :x ₂], and internal knots occur in <i>adjacent pairs</i> . = 0 – No Entity IDs data collection exists. = 1 – Entity IDs data collection exists.
[2]	Flag indicating whether Entity IDs data collection exists for “Odd Count [0:1] Range” knot type. Knots of this type have their knot range on [0:1], and are not Type 0. = 0 – No Entity IDs data collection exists. = 1 – Entity IDs data collection exists.
[3]	Flag indicating whether Entity IDs data collection exists for “Odd Count [x ₁ :x ₂] Range” knot type. Knots of this type have their knot range on [x ₁ :x ₂], and are not Type 1. = 0 – No Entity IDs data collection exists. = 1 – Entity IDs data collection exists.

Examples of knot vectors of Type 0:

```
0 0 X X 1 1
0 0 X X Y Y 1 1
0 0 X X Y Y Z Z 1 1
```

Examples of knot vectors of Type 1:

```
0 0 1 1          (Note: This is the exception to Type 0)
X X Y Y
X X Y Y Z Z
X X Y Y Z Z W W
```

Examples of knot vectors of Type 2:

```
0 0 X 1 1
0 0 X Y 1 1
0 0 X Y Z 1 1
0 0 X X X 1 1
0 0 X X Y Z Z 1 1
```

Examples of knot vectors of Type 3:

```
X X Y Z Z
X X Y Z W W
```

With this information in hand, the reader is able to reconstruct complete knot vectors in the following manner. When reconstructing the knot vector, you only take just enough values from the decoded knot value array. This may be as few as one. All the other values are inferred. Here's a sketch of the reconstruction algorithm:

```
// Number of knots in the knot vector
cNumKnots = numCtlPts + degree + 1;
// Necessary knot multiplicity at both ends of the knot vector
cClamping = degree + 1;
switch (knotType) {
    // Clamping is 0..1, internal knots occur in ADJACENT PAIRS
    // *EXCEPT* when there are no internal knots, in which case
    // Type = 1 instead.
    case 0: numVals = (cNumKnots - 2 * cClamping)/2;
    // Clamping is X1..X2, internal knots occur in ADJACENT PAIRS
    case 1: numVals = (cNumKnots - 2 * cClamping)/2 + 2;
    // Clamping is 0..1, and not Type 0
    case 2: numVals = (cNumKnots - 2 * cClamping);
    // Clamping is X1..X2, and not Type 1
    case 3: numVals = (cNumKnots - 2 * cClamping) + 2;
```

```

}
// numVals is the number of non-inferable knot values needed
// Let vVals be the knot vector value array
// vKnot will be the final output knot vector
if (knotType is either 0 or 2)
    Set vKnot[0 .. cClamping-1] to 0
    Set vKnot[cNumKnots-cClamping .. cNumKnots-1] to 1
else
    Set vKnot[0 .. cClamping-1] to vVals[0]
    Set vKnot[cNumKnots-cClamping .. cNumKnots-1] to vVals[numVals-1]
Set vKnot[cClamping .. cNumKnots-cClamping-1] from vVals[1 .. numVals-2]

```

VecI32{Int32CDP, 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.

10.1.14 Compressed Control Point Weights Data

Compressed Control Point Weights Data collection is the compressed and/or encoded representation of weight data for some set of Control Points. All NURBS based geometry use this data collection to compress/encode Control Point Weight data.

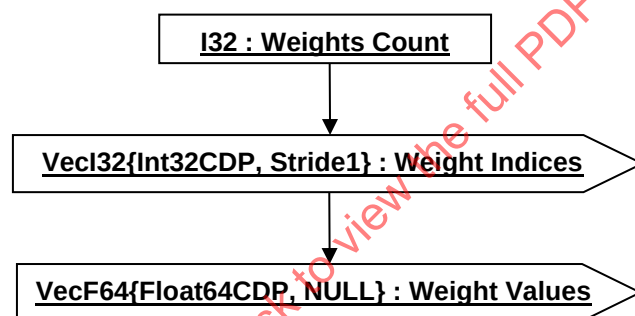


Figure 166 — Compressed Control Point Weights Data collection

I32 : Weights Count

Weights Count specifies the total number of Weights. This count can differ from the Control Point count (see NURBS Surface Control Point Counts) because if the Control Point Dimensionality is non-rational (see data field NURBS Surface Control Point Dimensionality in Surfaces Geometric data), then no Weight values are stored for the particular Control Point. Weights Count value also does not necessarily equate to the actual number of Weights stored, since if a particular Control Point's Weight values is “1”, then no actual Weight value is stored (i.e. JT file loaders/readers can infer that the Weight Value is “1” for Control Points that do not 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.

10.1.15 Compressed Curve Data

Compressed Curve Data collection contains B-Rep description in JT or Wireframe Rep compressed/encoded geometric Curve data. Currently only NURBS Curve types are supported as part of this data collection. Complete documentation for B-Rep description in JT and Wireframe Rep can be found in annex K and L.1 Wireframe Rep Element respectively.

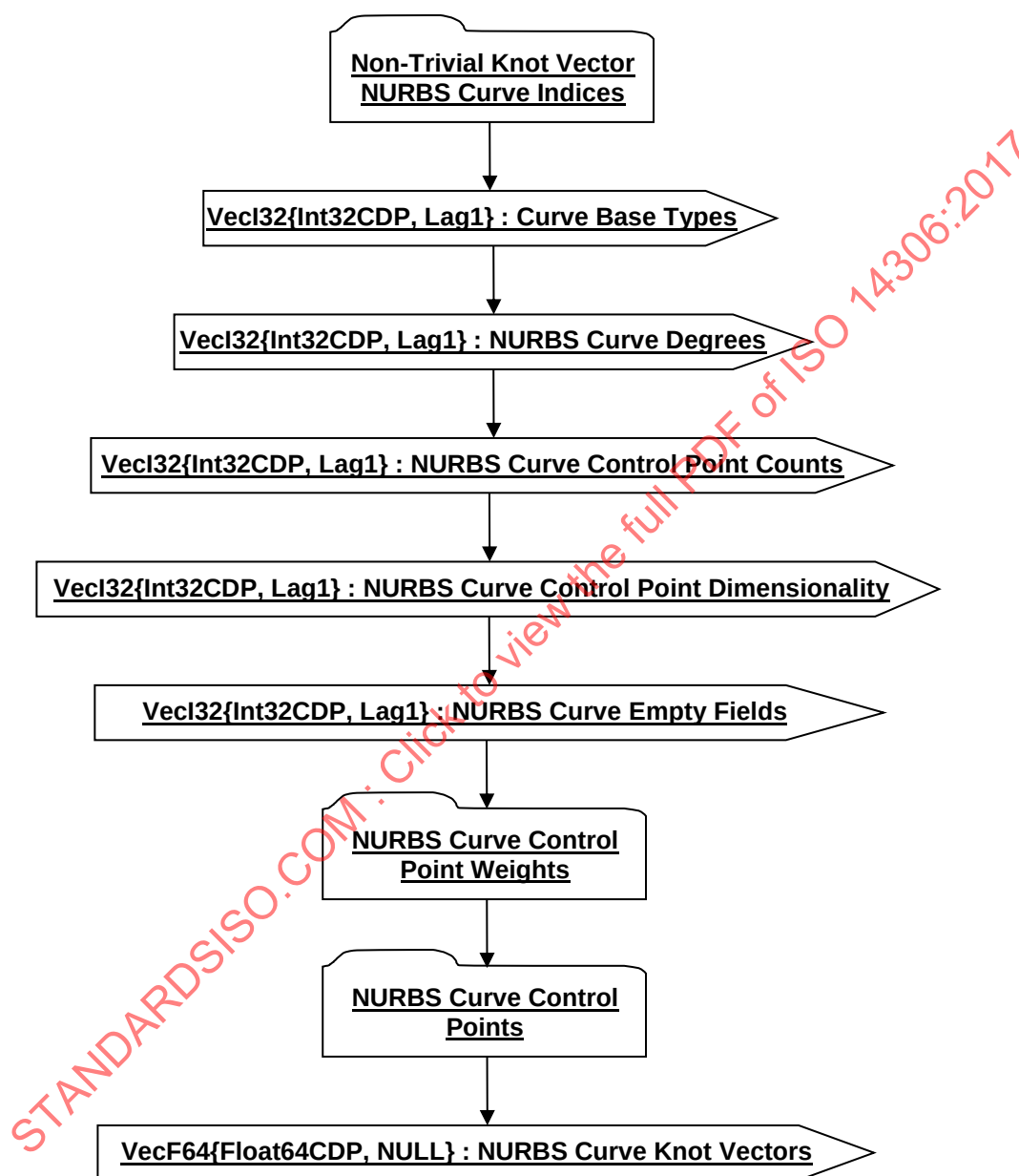


Figure 167 — Compressed Curve Data collection

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 ISO 14306 format to support other curve types.

In an uncompressed/decoded form the Curves base type identifier values have the following meaning:

Table 83 — Compressed Curve Base Type values

= 1	Curve is a NURBS curve

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

VecI32{Int32CDP, 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 Curves (i.e. there is a stored value for each NURBS Curve in the list).

In an uncompressed/decoded form the control point dimensionality values meaning is dependent upon the NURBS Entity type.

For NURBS UV Curve entities the dimensionality value has the following definition:

Table 84 — NURB UV Curve entity dimensionality values

= 2	Non-Rational (each control point has 2 coordinates)
= 3	Rational (each control point has 3 coordinates)

For NURBS XYZ Curve entities the dimensionality value has the following definition:

Table 85 — NURB XYZ Curve entity dimensionality values

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

NURBS Curve Control Point Dimensionality uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : NURBS Curve Empty Fields

NURBS Curve Empty Fields is a vector of data. Each NURBS Curve in a list of Curves has one reserved data field entry in this NURBS Curve Empty Fields vector. NURBS Curve Empty Fields uses the Int32 version of the CODEC to compress and encode data. Refer to best practice 11.4 Empty Field.

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 accumulated into this single list in the same order as the Curve appears in the Curve list (i.e. Curve-N Non-Trivial Knot Vector, Curve-M Non-Trivial Knot Vector, etc.). The NURBS Curves for which knot vectors are stored (i.e. those containing non-trivial knot vectors) are identified in data collection Non-Trivial Knot Vector NURBS Curve Indices

documented in Non-Trivial Knot Vector NURBS Curve Indices. NURBS Curve Knot Vectors uses the Float64 version of the CODEC to compress and encode data.

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 10.1.13 Compressed Entity List for Non-Trivial Knot Vector.

This Curve index data is stored in a compressed format.



**Compressed Entity List for
Non-Trivial Knot Vector**

Figure 168 — Non-Trivial Knot Vector NURBS Curve Indices data collection

Complete description for Compressed Entity List for Non-Trivial Knot Vector can be found in 10.1.13 Compressed Entity List for Non-Trivial Knot Vector.

NURBS Curve Control Point Weights

NURBS Curve Control Point Weights data collection defines the Weight values for a conditional set of Control Points for a list of NURBS Curves. The storing of the Weight value for a particular Control Point is conditional, because if NURBS Curve Control Point Dimension is “non-rational” or the actual Control Point’s Weight value is “1”, then no Weight value is stored for the Control Point (i.e. Weight value can be inferred to be “1”).

The NURBS Curve Control Point Weights data is stored in a compressed format.



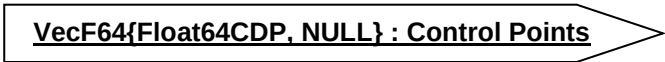
**Compressed Control
Point Weights Data**

Figure 169 — NURBS Curve Control Point Weights data collection

Complete description for Compressed Control Point Weights Data can be found in 10.1.14 Compressed Control Point Weights Data.

NURBS Curve Control Points

NURBS Curve Control Points is the compressed and/or encoded representation of the Control Point coordinates for each NURBS Curve in a list of Curves (i.e. there are stored values for each NURBS Curve in the list). Note that these are non-homogeneous coordinates (i.e. Control Point coordinates have been divided by the corresponding Control Point Weight values).



VecF64{Float64CDP, NULL} : Control Points

Figure 170 — NURBS Curve Control Points data collection

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 accumulated into this single vector in the same order

as the Curve appears in the Curve list (i.e. Curve-1 Control Points, Curve-2 Control Points, etc.). Control Points uses the Float64 version of the CODEC to compress and encode data in a “lossless” manner.

10.1.16 Compressed CAD Tag Data

The Compressed CAD Tag Data collection contains the persistent IDs, as defined in the CAD System, to uniquely identify individual CAD entities (e.g. Faces and Edges of a B-Rep description in JT, 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. K.1.6 B-Rep CAD Tag Data, 9.2.7 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.

STANDARDSISO.COM : Click to view the full PDF of ISO 14306:2017

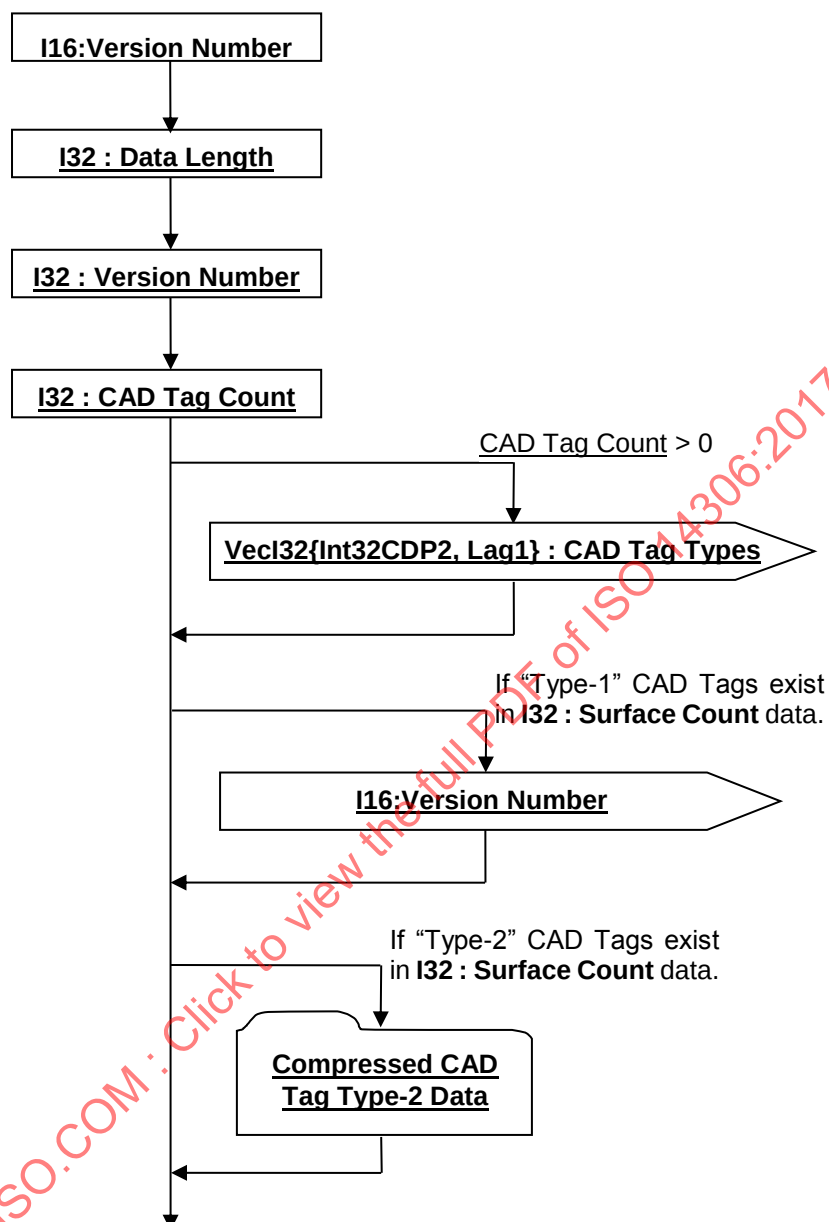


Figure 171 — Compressed CAD Tag Data collection

I16:Version Number

Version Number is the version identifier for the CADTag element. For information on local version numbers see best practice [11.5 Local version numbers.](#)

I32 : Data Length

Data Length specifies the length in bytes of the Compressed CAD Tag Data collection. A JT file loader/reader may use this information to compute the end position of the Compressed CAD Tag Data within the JT file and thus skip reading the remaining Compressed CAD Tag Data.

I32 : Version Number

Version Number is the local version identifier for the Compressed CAD Tag Data. For information on local version numbers see best practice [11.5 Local version numbers.](#)

I32 : CAD Tag Count

CAD Tag Count specifies the number of CAD Tags.

VecI32{Int32CDP2, 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:

Table 86 — Compressed CAD Tag Type values

= 1	32 Bit Integer CAD Tag Type
= 2	64 Bit Integer CAD Tag Type

CAD Tag Types uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP2, 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 shall 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.

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 shall 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.

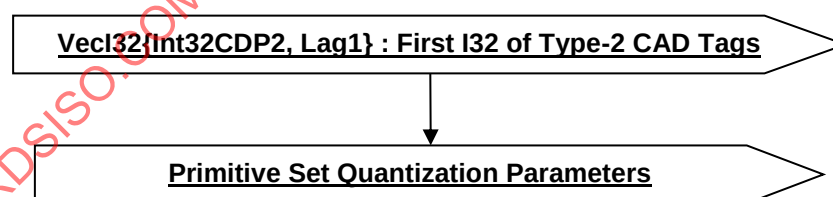


Figure 172 — Compressed CAD Tag Type-2 Data collection

VecI32{Int32CDP2, 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{Int32CDP2, 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.

10.2 Encoding Algorithms

The following sections give a brief technical overview/descriptions of the various encoding algorithms used in the ISO 14306 format. A sample implementation of the encoding and decoding portion of each algorithm can be found in [Annex D](#).

10.2.1 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 multiples (i.e. integers) of a common factor. How close the quantization output approximates the original input data is dependent upon the quantization data range and the number of bits specified to hold the resulting integer value.

The quantization is considered “uniform” because the algorithm divides the data input range into levels of equal spacing (i.e. a uniform scale). The form of Uniform Data Quantization used by the ISO 14306 format is also considered scalar in nature, in that each input value is treated separately in producing the output integer value.

Given the following definitions:

```
inputVal:      Input floating point data to quantize
outputVal:     Resulting quantized output integer value
minInputRange: Specified minimum value of input data range
maxInputRange: Specified maximum value of input data range
nBits:         Specified number of bits of precision (quantized size)
```

The basic algorithm (using C++ style syntax) for Uniform Data Quantization is as follows:

```
UInt32 iMaxCode = (nBits < 32) ? (0x1 << nBits) - 1 : 0xffffffff;
Float64 encodeMultiplier = Float64(iMaxCode) / (maxInputRange - minInputRange);
UInt32 outputVal = UInt32( (inputVal - minInputRange) * encodeMultiplier + 0.5 );
```

Note: For reasons of robustness, “outputVal” shall also be explicitly clamped to the range [0, iMaxCode]. This is because floating-point roundoff error in the calculation of “encodeMultiplier” can otherwise cause “outputVal” to sometimes come out equal to “iMaxCode + 1”.

Note that all compression algorithms in the following sections operate on quantized integer data.

10.2.2 Bitlength CODEC

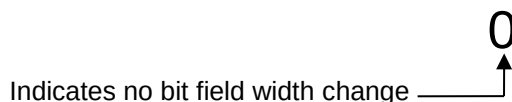
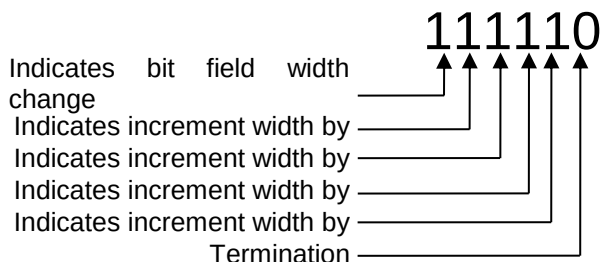
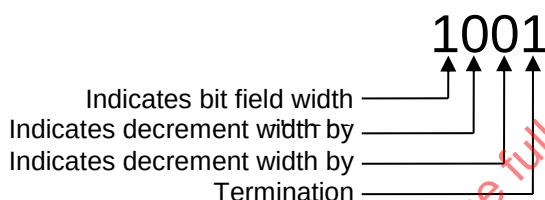
This is a very simple compression algorithm that runs an adaptive-width bit field encoding for each value. As each input value is encountered, the number of bits needed to represent it is calculated and compared to the current “field width”. The current field width is then adjusted upwards or downwards by a constant “step_size” number of bits (i.e. 2 bits for the ISO 14306 format) to accommodate the input value storage. This increment or decrement of the current field width is indicated for each encoded value by a prefix code stored with each value.

The prefix code will be one of the following two forms:

A single '0' bit to denote the same (i.e. current) field width is to be used for the next value.

A '1' bit followed by a series of one or more bits where each bit indicates whether the field width is to be incremented (a '1' bit) or decremented (a '0' bit) by the field step_size, followed by a single terminator bit (which is complement of the previous increment/decrement bit). Note that there can only be increments or decrements in a given prefix code, never both, and that is why the prefix code terminator bit can be recognized as bits are read by simply looking for the complement of the previous increment/decrement bit.

Some examples of prefix codes and their interpretation are as follows:

Example 1: Prefix code to maintain same (current) field width.**Example 2: Prefix code to increment field width four times (8 bits).****Example 3: Prefix code to decrement field width two times.**

A pseudo-code sample implementation of bit length decoding is available in Annex E.

10.2.3 Arithmetic CODEC

In 1948, 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 year, two Doctoral students at the Massachusetts Institute of Technology (MIT) made breakthroughs in the coding of information. The first to press was David Huffman, whose coding scheme we now know as Huffman Coding. In that same class with Huffman was Peter Elias who reportedly developed the first articulation of arithmetic coding, but it lay unpublished until 1976, when Jorma Rissanen and Richard Pasco, of IBM, refined it into a practically useful algorithm.

Arithmetic encoding is a 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 shall be uniquely decodable to create the exact stream of input symbols that were used to create it.

Initially all symbols being encoded have a probability value assigned to them based on the likelihood that the symbol will occur next in the input stream (i.e. the frequency of the symbol in the input stream). Given probability value assignments, each individual symbol is then assigned an interval range along a nominal 0 to 1 “probability line”, where the size of each range corresponds to the symbol’s probability value. Note that a particular symbol owns all values within its assigned range up to, but not including, the range high value, and that it does not matter which symbols are assigned which segment of the range as long it is done in the same manner by both the encoder and the decoder.

Given the above described input stream probability and interval range assignments, a high level description of the arithmetic encoding process is as follows:

Begin with a “current interval” initialized to $[0, 1)$. Note, that in interval range notation (i.e. $[0, 1)$), the “[” symbol indicates inclusive of the interval low limit and “)” symbol indicates exclusive of the interval high limit.

Sequentially for each symbol of the input stream, perform two steps.

Subdivide the current interval into subintervals based on the input stream symbol probability values as described above.

Select the subinterval corresponding to the current input stream symbol being sequentially processed and make it the new “current interval”.

After all input stream symbols have been sequentially processed; output enough bits to distinguish the final “current interval” from all other possible final intervals.

In pseudo code form, the algorithm to accomplish the above described arithmetic encoding for an input stream message of any length could look as follows:

```
Set low to 0.0
Set high to 1.0
While there are still input symbols do
    cur_symbol = get next input symbol
    range = high - low
    high = low + range * high_range(cur_symbol)
    low = low + range * low_range(cur_symbol)
End of While
Output low
```

So the arithmetic encoding process is simply one in which we narrow the range of possible numbers with every new sequentially processed input symbol; where the new narrowed range is proportional to the predefined probability values assigned to each symbol in the input stream.

The arithmetic decoding process is the inverse procedure; where the range is expanded in proportion to the probability of each symbol as it is extracted. For the arithmetic decoding process we find the first symbol in the message by seeing which symbol owns the interval range that our encoded message falls in. Then, since we know the low and high range limit values of the first symbol we can remove their effects by reversing the process that put them in.

In pseudo code form, the algorithm for decoding the incoming number could look as follows:

```
Get encoded_number
Do
    find symbol whose range straddles the encoded_number
    output the symbol
    range = symbol_high_value - symbol_low_value
    encoded_number = encoded_number - symbol_low_value
    encoded_number = encoded_number / range
until no more symbols
```

Example

Following is an example to demonstrate in practice the basic principles of arithmetic coding.

Suppose you want to compress, using arithmetic coding, the following sequence/array of integer data:

{2, 9, 12, 12, 0, 7, 1, 20, 5, 19}

For this input stream of data, the assigned probability values will be as follows:

Table 87 — Example assigned probability values

Number	Probability
0	1/10
1	1/10
2	1/10
5	1/10
7	1/10
9	1/10
12	2/10
19	1/10
20	1/10

Then based on each input numbers probability value, an interval range along a 0 to 1 “probability line” can be assigned to each input number as follows:

Table 88 — Example “probability line” values

Number	Probability	Range
0	1/10	[0.00, 0.10)
1	1/10	[0.10, 0.20)
2	1/10	[0.20, 0.30)
5	1/10	[0.30, 0.40)
7	1/10	[0.40, 0.50)
9	1/10	[0.50, 0.60)
12	2/10	[0.60, 0.80)
19	1/10	[0.80, 0.90)
20	1/10	[0.90, 1.00)

Now proceeding with encoding the example input integer sequence {2, 9, 12, 12, 0, 7, 1, 20, 5, 19}, the first number to be encoded is “2”, so the final encoded value will be a number that is greater than or equal to 0.20 and less than 0.30. Now as each subsequent number in the input stream is sequentially processed for encoding, the possible range of the output number is further restricted. In our example the next number to be encoded is “9” which owns the range [0.50, 0.60) within the new sub-range of [0.20, 0.30); which now further restricts our output number to the range [0.25, 0.26). If we continue this logic for the complete input integer sequence we end up with the following:

Table 89 — Example input integer sequence values

New integer number	Low value	High value
	0.0	1.0
2	0.2	0.3
9	0.25	0.26
12	0.256	0.258
12	0.2572	0.2576
0	0.25720	0.25724
7	0.257216	0.257220
1	0.2572164	0.2572168
20	0.25721676	0.2572168
5	0.257216772	0.257216776
19	0.2572167752	0.2572167756

From the above table, the final low value is “0.2572167752” which is the output number that uniquely encodes the integer number sequence {2, 9, 12, 12, 0, 7, 1, 20, 5, 19}.

Given this encoding scheme, the decoding would simply follow the process previously described. We find the first number in the sequence by looking up in the probability range for the value, whose range, our encoded number “0.2572167752” falls within. In our example this equates to the value “2” and so our first decoded value shall be “2”. Then we apply the previously described decoding subtraction and division steps to arrive at a new encoded value of “0.572167752”. Using this new “0.572167752” encoded value and the same logic of the first step, the second decoded value will be “9”. We continue this process until there are no more numbers to decode.

In practice, due to floating point size (i.e. number of bits) restrictions and possible differences in floating point formats on machines, arithmetic encoding is best implemented using 16 bit or 32 bit integer math. Using 16 bit or 32 bit integer math, an incremental transmission scheme can be implemented, where fixed size integer state variables receive new bits in at the low end and shift them out the high end, forming a single number that can be as many bits long as are available on the computer's storage medium.

Using our example as a guide, define the starting range [0.0, 1.0) to instead be 0 to 0.999 (which is .111 in binary). Then in order to use integer registers to store these numbers, justify the values so that the implied decimal point is at the left hand side of the word. Now load the initial range values based on the word size we are using. In the case of a 16 bit implementation the initial range values will be low equals 0x0000 and high equals 0xFFFF. Since we know these values will go on forever (e.g. 0.999... will continue with FFs) we can shift those extra bits in as needed with no detrimental effects.

Going back to our example and using a 5 digit register, we start with the range:

High: 99999

Low: 00000

Applying the previously described encoding algorithm we first calculate the range between the low and high values; which in this case is 100000 (not 9999 since we assume the high value has an infinite number of 9's). Next, we calculate the new high value which in this example will be 30000. But before we store the new high value we shall decrement it to account for the implied digits appended to it; so new high value will be 29999. Applying similar logic to computing the new low value results in a new range of:

High: 29999 (999...)

Low: 20000 (000...)

In looking at the newly computed high and low range values, it can be seen that the most significant digits of high and low match. A property of arithmetic coding is that as this encoding process continues, the high and low values will continue to get closer, but will never match exactly. Given this property, once the most significant digit of high and low match, it will never change, and thus we can output this most significant digit as the first number in the coded word and continue working with just 16 bit high and low values. This output process is accomplished by shifting both the high and low values left by one digit and shifting in a “9” in the least significant digit of the high value.

Applying the previously described encoding algorithm and continuing the above described process of shifting out most significant digit into the coded word as high and low continually grow closer together looks as follows for encoding our example integer number sequence {2, 9, 12, 12, 0, 7, 1, 20, 5, 19}:

Table 90 — Example integer number sequence values

	High	Low	Range	Cumulative output
Initial State	99999	00000	100000	
Encode “2” [0.2, 0.3)	29999	20000		

	High	Low	Range	Cumulative output
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 condition is known as "underflow" and special logic shall be 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 do not 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 shall perform the following steps to stop the current "underflow":

Delete the 2nd most significant digit from both the High and Low value.

Shift the other digits (those to the right of the deleted 2nd digit) to the left to fill up the space (note that the most significant digit stays in place).

Increment a counter to remember that we threw away a digit and do not know whether it was going to converge to “0” or “9”.

A before and after example of performing the above steps to the High and Low values when ‘underflow’ occurs is as follows:

	Before	After
High	40344	43449
Low	39810	38100
Underflow_counter	0	1

Now as the encoding algorithm continues and the most significant digit of High and Low values once again converge to a common value, then that value shall be output to the coded word along with “Underflow_counter” number of “underflow” digits that were previously deleted. The underflow digits output to the coded word will either be all 9s or 0s, depending on whether the High and Low value converged to the higher or lower value.

A pseudo-code sample implementation of arithmetic decoding is available in Annex E.

10.2.4 Deering Normal CODEC

Michael Deering first published his work on geometry compression in 1995 [7] and later helped present a course on the subject at SIGGRAPH'99 [8]. Although Deering's approach to geometric compression involves compression of vertices, colours and normals, the description detailed here will focus solely on compression of normals since this is the only component of Deering's approach used in the ISO 14306 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 173](#)). 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 173](#)). The particular sextant can be encoded using another three bits. So now unit sphere is divided into 48 identically shaped triangle patches reducing the normal look-up table to about 2000 entries (i.e. $100000/48$);

- Then, a local rectangular orthogonal two dimensional grid is created on the sextant and all normals within the sextant are represented as two n-bit angular addresses (i.e. a quantization of two angular values along the unit sphere) where “n” is in the range from 0 to 6 bits;
- Resulting in a max grand total of 18 bits (3 + 3 + 6 + 6) to represent any normal on the unit sphere.

In the figure below, the sphere is divided into eight octants and each octant is divided into six sextants. Each sextant is assigned an identifying three bit code.

In the figure below, the sphere is divided into eight octants and each octant is divided into six sextants. Each sextant is assigned an identifying three bit code.

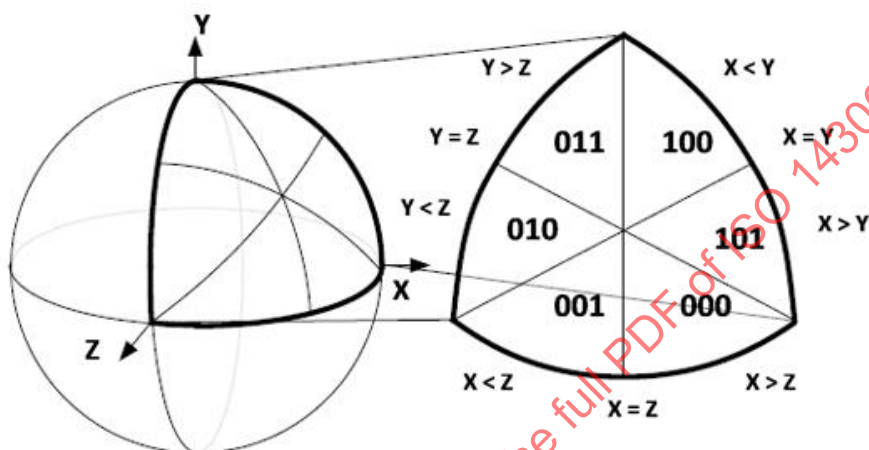


Figure 173 — Sextant Coding on the Sphere

Note that the sextant three bit code assignments used by the ISO 14306 format (as seen in Figure 173) 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:

$$x = \cos\theta * \cos\phi \quad y = \sin\phi \quad z = \sin\theta * \cos\phi;$$

- 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:

$$\theta(\theta'_n) = \arcsin \tan(\phi_{\max} * (n - \theta'_n) / 2n)$$

$$\phi(\phi'_n) = \phi_{\max} * \phi'_n / 2n.$$

Thus to encode (i.e. quantize) a given normal **N** into θ'_n and ϕ'_n :

- **N** shall be first represented (see Figure 173) in the positive octant and appropriate sextant within that octant, resulting in **N'**;

- Then N' shall be dotted with all quantized normals in the sextant;
- For a fixed “ n ”, the corresponding θ'_n and ϕ'_n values of the quantized sextant normal that result in the largest (nearest unity) dot product defines the proper θ'_n and ϕ'_n encoding of N .

With this encoding of normal N into θ'_n and ϕ'_n n -bit integers the complete bit representation of normal N can now be defined as follows:

- Uppermost three bits specify the octant;
- Next three bits specify the sextant code as defined in [Figure 173](#);
- Next two n -bit fields specify θ'_n and ϕ'_n values respectively.

10.3 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 ISO 14306 format uses the following functionality from Version 1.1.2 of the zlib compression library. These provide the basic interface to zlib compression functionality. A link to the web location for Version 1.1.2 of the zlib compression library source code can be found in the bibliography of this document.

- `zlibVersion();`
- `deflate();`
- `deflateReset();`
- `deflateEnd();`
- `inflateInit();`
- `inflate();`
- `inflateReset();`
- `inflateEnd();`
- `zlib_compress();`
- `zlib_compress2();`

11 Common Data Conventions and Constructs

The proceeding sections of this document specify the mandatory clauses for creating a reference compliant Version 9.5 JT file. This section focusing on documenting format conventions that although not required to have a reference compliant JT file, have become commonplace within ISO 14306 format translators.

11.1 Late-Loading Data

The ISO 14306 format was designed and structured to load entities from a JT file on a deferred or as-needed basis. This concept is referred to within this document as “late-loading data”. The ISO 14306 format has many structures in support of this and it is recommended that writers/loaders of JT data leverage these capabilities.

Initial loading only requires the Table of Contents and the LSG. All Meta Data Node Elements, XT B-Rep Element, Wireframe Rep Elements, PMI Manager Meta Data Elements, **JT ULP Elements**, **JT LWPA Elements**, and STEP B-rep may be ignored until they are actually needed. These Late-Loaded data containers are accessed via a Late Loaded Property Atom Element which appears in a LSG Node's Property list. Contained in this Property is the GUID associated with the segment to be loaded. This GUID can be looked up in the TOC Segment, which will give the location in the JT from which to load the actual Element via the Data Segment convention.

11.2 TOC Segment Location

Although there are no ISO 14306 format compliance rules about where the TOC Segment shall be located within the file, it is best placed immediately following the file header.

11.3 Bit Fields

All bits fields that are not defined as in use should be set to “0”.

11.4 Empty Field

In the File Format section of this reference some data fields may be named/documented “Empty Field” (e.g. 6.1.1.7.1LOD Node Data “Empty Field” field). Common conventions suggest that these fields should be treated as follows:

If you are writing a JT file whose data did not originate from reading a previous JT file, then Empty Fields should be set to a value a “0” when writing the field to a JT file.

If you are writing a JT file whose data originated from reading a previous JT file (i.e. rewriting a JT File), then “Empty Fields” should be written with the same value that was read from the originating JT file.

11.5 Local version numbers

The version numbers seen throughout the data collections are version numbers local to those data types. They provide a simple means by which those data collections can be extended. All version information for 9.5 JT data is included within this document.

For each data collection, data for each local version should be written in sequence. When reading the data, the local version number allows readers to read up to the maximum local version they support and then use the segment length that was read in the Segment Header to skip over additional data. For example when version one and two data is present the user can choose to read only the version one data or additionally read the version two data as required.

Local version numbers are used for conditional branching as depicted in the element figures.

11.5.1 Version numbers

Local version number values defined at the publication of this document are “0x0001” and “0x0002”. Listed below are the data collections that use local version numbers by value.

1. “0x0001” is the only valid value for Version 9.5 JT files for the listed data elements;

6.1.1.1.1 Base Node Data;

6.1.1.3.1 Group Node Data;

6.1.1.4 Instance Node Element;

6.1.1.5 Part Node Element;

6.1.1.6.1 Meta Data Node Data;

6.1.1.7.1 LOD Node Data;

6.1.1.8 Range LOD Node Element;

6.1.1.9 Switch Node Element;

Base Shape Data;

6.1.1.10.7 NULL Shape Node Element;

6.1.1.10.8 Primitive Set Shape Node Element;

6.1.2.1.1 Base Attribute Data;

6.1.2.4 Draw Style Attribute Element;

6.1.2.5 Light Set Attribute Element;

6.1.2.6.1 Base Light Data;

6.1.2.8 Linestyle Attribute Element;

6.1.2.9 Pointstyle Attribute Element;

6.1.2.10 Geometric Transform Attribute Element;

6.1.2.11 Shader Effects Attribute Element;

6.1.2.12 Texture Coordinate Generator Attribute Element;

Mapping Plane Element;

Mapping Cylinder Element;

Mapping Sphere Element;

Mapping TriPlanar Element;

Base Property Atom Data;

6.2.2 String Property Atom Element;

- 6.2.3 Integer Property Atom Element;
 - 6.2.4 Floating Point Property Atom Element;
 - 6.2.5 JT Object Reference Property Atom Element;
 - 6.2.6 Date Property Atom Element;
 - 6.2.7 Late Loaded Property Atom Element;
 - 6.2.8 Vector4f Property Atom Element;
 - 6.3 Property Table;
 - 7.1.1.1 Base Shape LOD Data;
 - 7.1.2.1 Vertex Shape LOD Data;
 - 7.1.2.5 TopoMesh Compressed Rep Data V2;
 - 7.1.3 Tri-Strip Set Shape LOD Element;
 - 7.1.4 Polyline Set Shape LOD Element;
 - 7.1.5 Point Set Shape LOD Element;
 - 7.1.6 Null Shape LOD Element;
 - 7.2 Primitive Set Shape Element;
 - 9.1 Property Proxy Meta Data Element;
 - 10.1.16 Compressed CAD Tag Data - I16: Version Number;
 - 9.2.8 PMI Polygon Data;
 - 10.1.16 Compressed CAD Tag Data - I32: Version Number;
 - K.1 JT B-Rep Element;
 - M.3 JT LWPA Element;
 - L.1 Wireframe Rep Element.
2. "0x0002" is the only valid value for Version 9.5 JT files for the listed data elements:
- Vertex Shape Data;
 - 6.1.1.10.4 Polyline Set Shape Node Element;
 - 6.1.1.10.5 Point Set Shape Node Element.
3. "0x0001" and "0x0002" are valid values for:
- 7.1.2.2 TopoMesh LOD Data;
 - 7.1.2.2.1 TopoMesh Compressed LOD Data;
 - 7.1.2.2.2 TopoMesh Topologically Compressed LOD Data;

9.2 PMI Manager Meta Data Element;

M.1 JT ULP Element.

11.6 Hash Value

Hashing is a means by which a large chunk of values can be represented by single value through the use of a mathematical function that provides a distinctive value for each unique set of ordered values. The hash function used within the v9.x format was published by Bob Jenkins in Dr Dobbs back in 1997 and its implementation is provided in Annex E.

The hash function takes a pointer to a set of values, the number of values, and a seed hash value. It returns the resulting hash value. Initially the seed value is set to 0, however when hashing multiple data fields together the hash of previous data field is used as the seed hash value of the next data field:

```
UInt32 uHash = 0;
uHash = hash32( pVal0, nVal0, uHash );
uHash = hash32( pVal1, nVal1, uHash );
```

The order that individual fields are hashed is extremely important since v9.x readers are strongly encouraged to assert that the stored hash value matches the calculated hash value of the corresponding fields after reading in all the corresponding data. To this end each hash value stored within the v9.x format carefully documents which fields it encompasses and the order in which they should be hashed.

11.7 Scene graph construction

The following guidelines apply for scenegraph construction:

1. use a 6.1.1.6 Meta Data Node Element to denote a CAD “Assembly”,
2. use a 6.1.1.5 Part Node Element to denote a CAD “part”.

Below is an example of a fully-fleshed out small assembly of a three-wheeled motorcycle:

Partition	(6.1.1.2 Partition Node Element)	Root node of JT file
MetaDataNode	(6.1.1.6 Meta Data Node Element)	“Three-wheeler” Top-level assembly
Instance	(6.1.1.4 Instance Node Element)	“Front wheel” assembly
Partition	(6.1.1.2 Partition Node Element)	External reference to JT file for “Wheel” part
PartNode	(6.1.1.5 Part Node Element)	“Wheel” generic part
RangeLOD	(6.1.1.8 Range LOD Node Element)	Level-of-detail node
Group	(6.1.1.3 Group Node Element)	High-LOD group node
TriStripSet	(7.1.3 Tri-Strip Set Shape LOD Element)	Rim geometry
TriStripSet	(7.1.3 Tri-Strip Set Shape LOD Element)	Tire geometry
TriStripSet	(7.1.3 Tri-Strip Set Shape LOD Element)	Low-LOD geometry
MetaDataNode	(6.1.1.6 Meta Data Node Element)	“Rear Axle” assembly
Instance	(6.1.1.4 Instance Node Element)	“Left rear wheel” assembly
[“Wheel” PartNode above]		This instance node’s child is same as the one above.
Instance	(6.1.1.4 Instance Node Element)	“Right rear wheel” assembly
[“Wheel” PartNode above]		This instance node’s child is same as the one above.

6.1.1.4 Instance Node Element are used when referring to an *instanced* part but are not fundamentally different from a 6.1.1.3 Group Node Element having only one child.

11.8 Metadata Conventions

Although there are really no restrictions/limits/requirements on what metadata (i.e. properties) can/shall be attached to nodes in the LSG in order to have a reference compliant JT file, there are some conventions that have been generally followed in the industry when translating CAD data to the JT file format. See Property Atom Elements section of this document for complete description of the file Elements used to attach this property information to nodes.

11.8.1 CAD Properties

The following table lists the conventions that CAD data translators typically (although not always) follow in placing CAD information in a JT file as properties on various LSG nodes. Some of these properties are considered required in order for the data in the file to be interpreted correctly while other properties are optional. See flowing sub-sections for additional information on required versus optional properties.

The convention is to place these Units properties (see 11.8.2 PMI Properties) on every Part and Assembly grouping node in the LSG. By following this convention, JT file format readers/writers are provided maximum flexibility in understanding/indicating the appropriate JT data unit processing for both, monolithic and shattered JT file Assembly structures.

Table 91 — CAD Property Conventions

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values	Required / Optional
CAD_CENTER_OF_GRAVITY	Center of gravity of solids within part	String	3 space separated F64	3 * numeric	Optional
CAD_FORCE_UNITS	The units specified by the originating system for force values. Referenced by CAD_PROP_YOUNGS_MODULUS	String	MbString	MilliNewton Newton PoundForce	Optional
CAD_DENSITY	Density of solids within part	String	F64	numeric	Optional
CAD_MASS	Mass or weight of solids within part	String	F64	numeric	Optional
CAD_MASS_UNITS	Defines the Units of mass	String	MbString	micrograms milligrams grams kilograms ounces pounds	Required
CAD_MOMENT_OF_INERTIA	The components of the Inertia Tensor of the part associated with the node, as calculated by the originating system. By definition the inertia tensor is a 3X3 matrix with values defined in the following order - Ixx, Ixy, Ixz, IYX, Iyy, Iyz, IZX, IZY, Izz. Of these we only need - Ixx, Ixy, Ixz, Iyy, Iyz, Izz, in this order, since the tensor is symmetric so that IYX = Ixy, IZX = Ixz and IZY = Iyz.	String	F64	6 space separated numeric values; xx,yy,zz,xy,xz,yz	Optional

	These components may be assembled as follows. $\begin{Bmatrix} I_{xx} & I_{xy} & I_{xz} \\ [I] = & \begin{Bmatrix} I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{Bmatrix} \end{Bmatrix}$ The units are as specified by JT_PROP_MEASUREMENT_UNITS and CAD_MASS_UNITS				
CAD_PART_NAME	Component name from translator	String	MbString		Optional
CAD_PROP_MATERIAL_THICKNESS	Sheet thickness within part	String	F64	numeric	Optional
CAD_PROP_YOUNG'S_MODULUS	The Young's modulus of the part associated with the node, as defined by the originating system, in units specified by JT_PROP_MEASUREMENT_UNITS and CAD_FORCE_UNITS	String	F64	numeric	Optional
CAD_SOURCE	CAD program the Part originated from	String	MbString		Optional
CAD_SURFACE_AREA	The surface area of the part associated with the node, as calculated by the originating CAD system, measured in the units specified by JT_PROP_MEASUREMENT_UNITS	String	F64	numeric	Optional
CAD_VOLUME	Volume of solids within part	String	F64	numeric	Optional
JT_PROP_MEASUREMENT_UNITS	Important for applications to provide a correct measurement and placement of components. Defines the Model Units and is therefore relevant for the reading and interpretation of the Geometry (B-Rep description in JT) and LOD data. Note: Shall be present as "hidden" parameter. If the property is in the scene graph twice on different nodes, then the definition that the lowest (last) one along a given path through the graph takes precedence.	String	MbString	millimeters centimeters meters inches feet yards micrometers decimeters kilometers mils miles	Required

JTTK_MULTICAD_REFSET	This property shall be set on the PMI To have PMI visible within a reference it shall have a JTTK_MULTICAD_REFSET property defined. The property is a string of Reference set names that the PMI will be visible in. i.e. type="STRING" key="JTTK_MULTICAD_REFSET" value="<refset name>, <refset name>"	String	MbString	Comma delimited string	Optional
REFSET_CURR ENT	Defines which reference set will be displayed in an instance node i.e. type="STRING" key="REFSET_CURRENT" value =<refset name> See also 11.13.1	String	MbString	Comma delimited string	Optional
REFSET_META	Contains a comma delimited list of aliased reference sets via the following convention: REFSET_META="alias_1=name_1, alias_2=name_2" An aliased reference sets is a reference set definition used to record a group of reference sets that crosses multiple CAD Components. See also 11.13.1	String	MbString	Comma delimited string	Optional
REFSETXXX	Specifies the string of part names that are included in the reference set. The property value defines the reference set name that is displayed in viewing systems i.e.; type="STRING" key="REFSET<XXX>" value="REFSET_A_D=[WF]<ref set name>,[LW]<ref set name>,[PT]<ref set name>,<ref set name>,[PMI]"	String	MbString	Comma delimited string of part names and hints	Optional

	There are 3 reserved strings that can be added as prefixes to the JT part names included on the reference set string. They are; [PT] : precedes the referenced part name(s) that contain point cloud geometry [WF] : precedes the referenced part name(s) that contain wireframe geometry [LW]: precedes the referenced part name(s) that contain only facet geometry representations. B-Rep is not present. The reference hint string [PMI] is included without part names. When the [PMI] hint is included corresponding PMI entities with the JT_MULTICAD_REFSET property may exist. See also 11.13.1				
SUBNODE	Specifies the node as being a sub node of the parent assembly.	String	MbString	0 or 1	Optional

11.8.1.1 Required Properties

The required unit properties are really necessary for viewers of JT file data to properly interpret numeric data for analysis operations (e.g. measure) and support the building of assemblies through the reading of multiple JT files in disparate units. There are two units of measure that are relevant, units of distance and units of weight.

The JT_PROP_MEASUREMENT_UNITS property is used to specify units of distance. The CAD_MASS_UNITS property is used to specify units for weight. JT_PROP_MEASUREMENT_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.

The value of JT_PROP_MEASUREMENT_UNITS property defines the unit in which the coordinate values of a LOD are given. It defines further the unit of the BRep data in case of a B-Rep description in JT. If the BRep is of XT type, all BRep values are given in meter [m], regardless of what value is set for the property JT_PROP_MEASUREMENT_UNITS. In the case of B-Rep description in XT type, the "chordal" property (specifying the chordal error of the created LODs) is given only in [m].

11.8.1.2 Optional Properties

Optional properties can be provided, but if the property is a units based value, then the value shall be in units that are consistent with the JT_PROP_MEASUREMENT_UNITS and CAD_MASS_UNITS properties. Thus the units for the optional units based properties shall be as follows:

Table 92 — CAD Optional Property Units

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_CENTRE_OF_GRAVITY	JT_PROP_MEASUREMENT_UNITS
CAD_PROP_MATERIAL_THICKNES S	JT_PROP_MEASUREMENT_UNITS

Note of caution regarding the node placement for the CAD_DENSITY property. Following the recommended convention for the placing of CAD properties (see description in 11.8.1 CAD Properties) implies that all solids within a single JT part are of a uniform density, which may not be true in all cases.

11.8.1.3 Hidden Properties

Properties are used for a range of purposes in a JT file; some are relevant to visual interrogation by users and some not. To enable applications with the ability to differentiate properties a convention for naming key strings should be followed. The key string pattern that is used to denote the visibility of a property is a double colon (":"). The double colon is appended to the end of the name in the property key as shown below. "property" = "hidden"

"property:." = "visible"

The objective of the "hidden" concept is to indicate to a viewing application that a users should not see the property in an application user interface.

For example:

The property with the key "Name:." will list the value "body_4465" in a viewing application when property display is selected.

Property type="STRING" key="Name:." value="body_4465"

The property SUBNODE will not be listed by a viewing application.

Property type="STRING" key="SUBNODE" value="1"

Definition of key names with ":@" included in the key name shall not affect visibility of that property when parsing of the JT data; i.e., "hidden" shall not be interpreted as "encrypted" or "secure" with respect to JT data.

11.8.2 PMI Properties

The table below lists properties that can be found in JT parts that contain Generic PMI Entities (see 9.2.6 Generic PMI Entities). The column titled Supported PMI_Type values defines the expected Generic PMI Entity Types the property will be found with (reference Entity Type). The column Value Format / Expected Values defines the expected representation of the property data. The Property Key and Value Format / Expected Values data forms a key value pair for a standard JT property (see PMI Property Atom).

Table 123 – PMI Properties makes use of C programming string format specifiers in description of values supported by the Property Key. The following values are used

"%d" - Signed 32-bit integer

"%.16g" - 64-bit floating-point number with 16 digit precision

"%s" - Null-terminated array of 8-bit unsigned characters

Note: It is recommended to use the descriptions found in 9.2 PMI Manager Meta Data Element as opposed to working with PMI Data Segment for working with PMI in a JT file.

Table 93 — PMI Properties

Property Key	Supported PMI Type values	Value Format / Expected Values	Prerequisite Key	Additional Note
accountabilityId	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
aftDistance	0x0242	"%.16g"	type	Expects type property to be closed
aftThroughAll	0x0242	"%d"	type	Expects type property to be closed
allAround	0x0041 0x0081 0x0008	"%d"		
allOver	0x0081	"%d"		
AltText[%d]	0x0100 0x0235 0x0244	NULL		
AltText[%d].bold	0x0100 0x0235 0x0244	"%d"	AltText[%d]	
AltText[%d].font	0x0100 0x0235 0x0244	"%s"	AltText[%d]	
AltText[%d].italic	0x0100 0x0235 0x0244	"%d"	AltText[%d]	
AltText[%d].italicAngle	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].Item[%d]	0x0100 0x0235 0x0244	"%s"	AltText[%d]	
AltText[%d].Item[%d].bold	0x0100 0x0235 0x0244	"%d"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].font	0x0100 0x0235 0x0244	"%s"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].format	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	AltText[%d].Item[%d]	
AltText[%d].Item[%d].italic	0x0100 0x0235 0x0244	"%d"	AltText[%d].Item[%d]	

AltText[%d].Item[%d] .italicAngle	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d] .justification	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	AltText[%d].Item[%d]	
AltText[%d].Item[%d] .language	0x0100 0x0235 0x0244	"%s"	AltText[%d].Item[%d]	
AltText[%d].Item[%d] .lineFactor	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d] .outlineType	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	AltText[%d].Item[%d]	
AltText[%d].Item[%d] .spaceFactor	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d] .string	0x0100 0x0235 0x0244	"%s"	AltText[%d].Item[%d]	
AltText[%d].Item[%d] .symbol	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtoleranc e, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout,	AltText[%d].Item[%d]	

		"21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface		
AltText[%d].Item[%d].textAspect	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].textColour	0x0100 0x0235 0x0244	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	AltText[%d].Item[%d]	
AltText[%d].Item[%d].textHeight	0x0100 0x0235 0x0244	"%.16g"	AltText[%d].Item[%d]	
AltText[%d].Item[%d].textThickness	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	AltText[%d].Item[%d]	
AltText[%d].Item[%d].underline	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=over, "1"=under	AltText[%d].Item[%d]	
AltText[%d].justification	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	AltText[%d]	
AltText[%d].language	0x0100 0x0235 0x0244	"%s"	AltText[%d]	
AltText[%d].lineFactor	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].name	0x0100 0x0235 0x0244	"%s"	AltText[%d]	

AltText[%d].outlineType	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	AltText[%d]	
AltText[%d].spaceFactor	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].textAspect	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].textColour	0x0100 0x0235 0x0244	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	AltText[%d]	
AltText[%d].textHeight	0x0100 0x0235 0x0244	"%.16g"	AltText[%d]	
AltText[%d].textOrientation	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=horizontal, "1"=vertical	AltText[%d]	
AltText[%d].textThickness	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	AltText[%d]	
AltText[%d].underline	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=over, "1"=under	AltText[%d]	
angle	0x0306	"%.16g"		
angleFormat	0x0082 0x0081	String representing enumeration such that: "0"=degrees, "1"=seconds, "2"=minutes		
angleNumerator	0x0082	"%.16g"		
approachDirection	0x0004	"%.16g %.16g %.16g"		
Area[%d]	0x0246	NULL		
Area[%d].ConstraintPlane.origin	0x0246	"%.16g %.16g %.16g"	Area[%d].ConstraintPlane	
Area[%d].ConstraintPlane.xaxis	0x0246	"%.16g %.16g %.16g"	Area[%d].ConstraintPlane	

Area[%d].ConstraintPlane.zaxis	0x0246	"%.16g %.16g %.16g"	Area[%d].ConstraintPlane	
Area[%d].curveRef[%d]	0x0246	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formatting is used to semantically tie a singlar semantic aspect with an entity.	Area[%d]	
Area[%d].diameter	0x0246	"%.16g"	Area[%d]	
Area[%d].height	0x0246	"%.16g"	Area[%d]	
Area[%d].innerDiameter	0x0246	"%.16g"	Area[%d]	
Area[%d].insidePoint	0x0246	"%.16g %.16g %.16g"	Area[%d]	
Area[%d].length	0x0246	"%.16g"	Area[%d]	
Area[%d].originAnchor	0x0246	String representing enumeration such that: "0"=topleft, "1"=topcentre, "2"=topright, "3"=middleleft, "4"=middlecentre, "5"=middleright, "6"=bottomleft, "7"=bottomcentre, "8"=bottomright	Area[%d]	
Area[%d].type	0x0246	String representing enumeration such that: "0"=rectangular, "1"=circular, "2"=annular, "3"=cylindrical, "4"=general	Area[%d]	
Area[%d].width	0x0246	"%.16g"	Area[%d]	
areaShape	0x0088	String representing enumeration such that: "0"=diameter, "1"=sphericaldiameter, "2"=square		
arrowSideFinishSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=unspecified, "2"=machining, "3"=grinding, "4"=chipping, "5"=none, "6"=hammering, "7"=peening, "8"=rolling		
arrowSideGrooveAngle	0x0008	"%s"		

arrowSideGrooveGap	0x0008	"%s"		
arrowSideLongitudinalSize	0x0008	"%s"		
arrowSideMainSize	0x0008	"%s"		
arrowSideStaggeredSize	0x0008	"%s"		
arrowSideSupplementalSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=convex, "2"=toesblended, "3"=concave, "4"=flush		
arrowSideSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=jisseam, "2"=singlebevelgroove, "3"=singleugroove, "4"=edgeflange, "5"=surface, "6"=steepflankedsingle bevel, "7"=flaresinglevgroove, "8"=surfacejoint, "9"=singlevgroove, "10"=inclinedjoint, "11"=plug, "12"=fillet, "13"=squaregroove, "14"=spot, "15"=singlej, "16"=square, "17"=overlay, "18"=singleu, "19"=singlejgroove, "20"=doubleflange, "21"=singleflange, "22"=spotprojected, "23"=plugandslot, "24"=singlebevelbroadr ootface, "25"=edge, "26"=jisstaggeredfillet1, "27"=singlev, "28"=flaresinglebevelgr oove, "29"=singlevbroadrootf ace, "30"=steepflankedsingl ev, "31"=removablebackin gstrip, "32"=permanentbackin gstrip, "33"=jisfillet, "34"=singlebevel, "35"=backing, "36"=foldjoint, "37"=jisstaggeredfillet2, "38"=seam, "39"=bead		

attachmentType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=angulardimension, "1"=ongeometry, "2"=lineardimension, "3"=leader, "4"=stacked, "5"=ordinatedimension, "6"=radialdimension, "7"=diametraldimension, "8"=noleader		
AttributeName[%d]	0x0240 0x0241	"%s"		
axisDirection	0x0230 0x0238	"%.16g %.16g %.16g"		
balloonId	0x0235	"%s"		
basic	0x0082	"%d"		
blanked	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
bold	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
bottomExtensionLine	0x0041	"%d"		
category	0x0100 0x0235 0x0244	"%s"		
causality	0x0082	String representing enumeration such that: "0"=key, "1"=functional, "2"=reference, "3"=associated, "4"=pmi		
centrecrossSize	0x0306	"%.16g"		

centrelineType	0x0306	String representing enumeration such that: "0"=centreline2d, "1"=centreline3d, "2"=centremark, "3"=circularcentreline, "4"=boltcirclecentreline, "5"=symmetriccentreline, "6"=offsetcentreline		
changeLevel	0x0118	String representing enumeration such that: "0"=soft, "1"=medium, "2"=hard		
characteristic	0x0081	String representing enumeration such that: "0"=profileofaline, "1"=circularrunout, "2"=perpendicularity, "3"=position, "4"=totalrunout, "5"=profileofasurface, "6"=cylindricity, "7"=symmetry, "8"=angularity, "9"=parallelism, "10"=concentricity, "11"=flatness, "12"=circularity, "13"=straightness		
clampingDirection	0x0004	"%.16g %.16g %.16g"		
CLIP_CAPPING	0x0305	"%d"		
CLIP_CAPPING_COLOUR	0x0305	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string		
CLIP_CURVE	0x0305	"%d"		
CLIP_CURVE_COLOUR	0x0305	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string		
CLIP_NORMAL	0x0305	"%.16g,%.16g,%.16g"		
CLIP_POSITION	0x0305	"%.16g,%.16g,%.16g"		
CLIP_RENDER	0x0305	String representing enumeration such that: "0"=shaded, "1"=wireframe		
CLIP_STYLE	0x0305	Enumerated string supporting values "Both", "Near", "Far" and "None"		

commaAsDecimal	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
conformsToFaces	0x0246	"%d"		
ConstraintPlane.origin	0x0242	"%.16g %.16g %.16g"	ConstraintPlane	
ConstraintPlane.xaxis	0x0242	"%.16g %.16g %.16g"	ConstraintPlane	
ConstraintPlane.zaxis	0x0242	"%.16g %.16g %.16g"	ConstraintPlane	
CoordinatedEntity[%d]	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	NULL		
CoordinatedEntity[%d].bold	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%d"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].font	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].italic	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%d"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].italicAngle	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].Item[%d]	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].Item[%d].bold	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%d"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].font	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].format	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].italic	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%d"	CoordinatedEntity[%d].Item[%d]	

CoordinatedEntity[%d].Item[%d].italicAngle	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].justification	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].language	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].lineFactor	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].outlineType	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].spaceFactor	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].string	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d].Item[%d]	

CoordinatedEntity[%d].Item[%d].symbol	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].textAspect	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	

CoordinatedEntity[%d].Item[%d].textColour	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].textHeight	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].textThickness	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].Item[%d].underline	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=over, "1"=under	CoordinatedEntity[%d].Item[%d]	
CoordinatedEntity[%d].justification	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	CoordinatedEntity[%d]	
CoordinatedEntity[%d].language	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].lineFactor	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].name	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%s"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].outlineType	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	CoordinatedEntity[%d]	
CoordinatedEntity[%d].spaceFactor	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].textAspect	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	

CoordinatedEntity[%d].textColour	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	CoordinatedEntity[%d]	
CoordinatedEntity[%d].textHeight	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	"%.16g"	CoordinatedEntity[%d]	
CoordinatedEntity[%d].textOrientation	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=horizontal, "1"=vertical	CoordinatedEntity[%d]	
CoordinatedEntity[%d].textThickness	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	CoordinatedEntity[%d]	
CoordinatedEntity[%d].underline	0x0230 0x0004 0x0008 0x0042 0x0118 0x0238	String representing enumeration such that: "0"=over, "1"=under	CoordinatedEntity[%d]	
coordinatePlane	0x0118	String representing enumeration such that: "0"=y, "1"=xz, "2"=xy, "3"=yz, "4"=z, "5"=xyz, "6"=x		
coordinationDirection	0x0042	"%.16g %.16g %.16g"		
counterbored	0x0230	"%d"		
CpsLetters	0x0242	NULL		
CpsLetters.bold	0x0242	"%d"	CpsLetters	
CpsLetters.font	0x0242	"%s"	CpsLetters	
CpsLetters.italic	0x0242	"%d"	CpsLetters	
CpsLetters.italicAngle	0x0242	"%.16g"	CpsLetters	
CpsLetters.Item[%d]	0x0242	"%s"	CpsLetters	
CpsLetters.Item[%d].bold	0x0242	"%d"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].font	0x0242	"%s"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].format	0x0242	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	CpsLetters.Item[%d]	
CpsLetters.Item[%d].italic	0x0242	"%d"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].italicAngle	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].justification	0x0242	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	CpsLetters.Item[%d]	

CpsLetters.Item[%d].language	0x0242	"%s"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].lineFactor	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].outlineType	0x0242	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	CpsLetters.Item[%d]	
CpsLetters.Item[%d].spaceFactor	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].string	0x0242	"%s"	CpsLetters.Item[%d]	

CpsLetters.Item[%d]. symbol	0x0242	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	CpsLetters.Item[%d]	
CpsLetters.Item[%d]. textAspect	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d]. textColour	0x0242	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	CpsLetters.Item[%d]	

CpsLetters.Item[%d].textHeight	0x0242	"%.16g"	CpsLetters.Item[%d]	
CpsLetters.Item[%d].textThickness	0x0242	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	CpsLetters.Item[%d]	
CpsLetters.Item[%d].underline	0x0242	String representing enumeration such that: "0"=over, "1"=under	CpsLetters.Item[%d]	
CpsLetters.justification	0x0242	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	CpsLetters	
CpsLetters.language	0x0242	"%s"	CpsLetters	
CpsLetters.lineFactor	0x0242	"%.16g"	CpsLetters	
CpsLetters.name	0x0242	"%s"	CpsLetters	
CpsLetters.outlineType	0x0242	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	CpsLetters	
CpsLetters.spaceFactor	0x0242	"%.16g"	CpsLetters	
CpsLetters.textAspect	0x0242	"%.16g"	CpsLetters	
CpsLetters.textColour	0x0242	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	CpsLetters	
CpsLetters.textHeight	0x0242	"%.16g"	CpsLetters	
CpsLetters.textOrientation	0x0242	String representing enumeration such that: "0"=horizontal, "1"=vertical	CpsLetters	
CpsLetters.textThickness	0x0242	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	CpsLetters	

CpsLetters.underline	0x0242	String representing enumeration such that: "0"=over, "1"=under	CpsLetters	
CrosshatchPattern.angle	0x0243 0x0246	"%.16g"	CrosshatchPattern	
CrosshatchPattern.pattern	0x0243 0x0246	String representing enumeration such that: "0"=unset, "1"=rubber, "2"=lead, "3"=user, "4"=steel, "5"=brass, "6"=solid, "7"=iron, "8"=aluminum, "9"=glass, "10"=electricalwinding, "11"=refractory, "12"=thermalinsulation	CrosshatchPattern	
CrosshatchPattern.rate	0x0243 0x0246	"%d"	CrosshatchPattern	
CrosshatchPattern.spacing	0x0243 0x0246	"%.16g"	CrosshatchPattern	
Curve[%d]	0x0305	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formatting is used to semantically tie a singlar semantic aspect with an entity.		
curveRef	0x0242	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formatting is used to semantically tie a singlar semantic aspect with an entity.	type	Expects type property to not be plane
datumOnLeader	0x0081 0x0084 0x0100 0x0235 0x0244	String representing enumeration such that: "0"=none, "1"=solid, "2"=filled		
delta	0x0004 0x0008	"%d"		
depth[%d]	0x0230	"%.16g"		
Description	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243	"%s"		

	0x0244 0x0246 0x0306 0x0307			
descriptiveModifier	0x0233	"%s"		
deviation	0x0082	"%s"		
diameter	0x0088 0x0004 0x0238	"%.16g"		
diameter[%d]	0x0230	"%.16g"		
diameterOverride	0x0238	"%d"		
diameterPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
dimensionLeadingZero	0x0082	"%d"		
DimensionText[%d]	0x0082	NULL		DimensionText[%d] index expected to range from 0-3 inclusive. Each instance of this is expected to have a different position such that DimensionText[x].position != DimensionText[y].position for any combination of x and y
DimensionText[%d].position	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after	DimensionText[%d]	
DimensionText[%d].Text	0x0082	NULL	DimensionText[%d]	
DimensionText[%d].Text.bold	0x0082	"%d"	DimensionText[%d].Text	
DimensionText[%d].Text.font	0x0082	"%s"	DimensionText[%d].Text	
DimensionText[%d].Text.italic	0x0082	"%d"	DimensionText[%d].Text	
DimensionText[%d].Text.italicAngle	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.Item[%d]	0x0082	"%s"	DimensionText[%d].Text	

DimensionText[%d].Text.Item[%d].bold	0x0082	"%d"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].font	0x0082	"%s"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].format	0x0082	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].italic	0x0082	"%d"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].italicAngle	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].justification	0x0082	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].language	0x0082	"%s"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].lineFactor	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].outlineType	0x0082	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].spaceFactor	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].string	0x0082	"%s"	DimensionText[%d].Text.Item[%d]	

DimensionText[%d].Text.Item[%d].symbol	0x0082	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].textAspect	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	

DimensionText[%d].Text.Item[%d].textColour	0x0082	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].textHeight	0x0082	"%.16g"	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].textThickness	0x0082	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.Item[%d].underline	0x0082	String representing enumeration such that: "0"=over, "1"=under	DimensionText[%d].Text.Item[%d]	
DimensionText[%d].Text.justification	0x0082	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	DimensionText[%d].Text	
DimensionText[%d].Text.language	0x0082	"%s"	DimensionText[%d].Text	
DimensionText[%d].Text.lineFactor	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.name	0x0082	"%s"	DimensionText[%d].Text	
DimensionText[%d].Text.outlineType	0x0082	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	DimensionText[%d].Text	
DimensionText[%d].Text.spaceFactor	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.textAspect	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.textColour	0x0082	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	DimensionText[%d].Text	

DimensionText[%d].Text.textHeight	0x0082	"%.16g"	DimensionText[%d].Text	
DimensionText[%d].Text.textOrientation	0x0082	String representing enumeration such that: "0"=horizontal, "1"=vertical	DimensionText[%d].Text	
DimensionText[%d].Text.textThickness	0x0082	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	DimensionText[%d].Text	
DimensionText[%d].Text.underline	0x0082	String representing enumeration such that: "0"=over, "1"=under	DimensionText[%d].Text	
dimensionTrailingZero	0x0082	"%d"		
direction	0x0082 0x0041 0x0081	"%.16g %.16g %.16g"		
displayClearance	0x0307	"%d"		
displayOption	0x0306	String representing enumeration such that: "0"=none, "1"=centreline, "2"=centrelinewithextension, "3"=centrepoin		
displayTolerance	0x0307	"%d"		
displayType	0x0230 0x0004 0x0100 0x0235 0x0244 0x0042 0x0238	String representing enumeration such that: "0"=urface, "1"=creen		
documentation	0x0082	"%d"		
DualDimensionText[%d]	0x0082	NULL		DualDimensionText[%d] index expected to range from 0-3 inclusive. Each instance of this is expected to have a different position such that DualDimensionText[x].position != DualDimensionText[y].position for any combination of x and y
DualDimensionText[%d].position	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after	DualDimensionText[%d]	
DualDimensionText[%d].Text	0x0082	NULL	DualDimensionText[%d]	
DualDimensionText[%d].Text.bold	0x0082	"%d"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.font	0x0082	"%s"	DualDimensionText[%d].Text	

DualDimensionText[%d].Text.italic	0x0082	"%d"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.italicAngle	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.Item[%d]	0x0082	"%s"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.Item[%d].bold	0x0082	"%d"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].font	0x0082	"%s"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].format	0x0082	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].italic	0x0082	"%d"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].italicAngle	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].justification	0x0082	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].language	0x0082	"%s"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].lineFactor	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	

DualDimensionText[%d].Text.Item[%d].outlineType	0x0082	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].spaceFactor	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].string	0x0082	"%s"	DualDimensionText[%d].Text.Item[%d]	

DualDimensionText[%d].Text.Item[%d].symbol	0x0082	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].textAspect	0x0082	%.16g"	DualDimensionText[%d].Text.Item[%d]	

DualDimensionText[%d].Text.Item[%d].textColour	0x0082	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].textHeight	0x0082	"%.16g"	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].textThickness	0x0082	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.Item[%d].underline	0x0082	String representing enumeration such that: "0"=over, "1"=under	DualDimensionText[%d].Text.Item[%d]	
DualDimensionText[%d].Text.justification	0x0082	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.language	0x0082	"%s"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.lineFactor	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.name	0x0082	"%s"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.outlineType	0x0082	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.spaceFactor	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.textAspect	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.textColour	0x0082	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	DualDimensionText[%d].Text	

DualDimensionText[%d].Text.textHeight	0x0082	"%.16g"	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.textOrientation	0x0082	String representing enumeration such that: "0"=horizontal, "1"=vertical	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.textThickness	0x0082	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	DualDimensionText[%d].Text	
DualDimensionText[%d].Text.underline	0x0082	String representing enumeration such that: "0"=over, "1"=under	DualDimensionText[%d].Text	
dualLowerDeltaDenominator	0x0082	"%d"		
dualPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
dualPrecision	0x0082	"%d"		
dualTolerancePrecision	0x0082	"%d"		
dualType	0x0082	String representing enumeration such that: "0"=positional, "1"=bracket		
dualUnits	0x0082	"%s"		
dualUnitText	0x0082	"%d"		
dualUpperDeltaDenominator	0x0082	"%d"		
dualValueDenominator	0x0082	"%d"		
fastenerSubType	0x0230	"%s"		
FCF[%d].	0x0308	"%.16g %.16g %.16g"	FCF[%d]	
FCF[%d].Text.bold	0x0308	"%d"	FCF[%d].Text	
FCF[%d].Text.font	0x0308	"%s"	FCF[%d].Text	
FCF[%d].Text.italic	0x0308	"%d"	FCF[%d].Text	
FCF[%d].Text.italicAngle	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.Item[%d]	0x0308	"%s"	FCF[%d].Text	
FCF[%d].Text.Item[%d].bold	0x0308	"%d"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].font	0x0308	"%s"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].format	0x0308	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference,	FCF[%d].Text.Item[%d]	

		"8"=enum, "9"=list, "10"=datetime		
FCF[%d].Text.Item[%d].italic	0x0308	"%d"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].italicAngle	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].justification	0x0308	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].language	0x0308	"%s"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].lineFactor	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].outlineType	0x0308	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoreddirectangle, "21"=parallelogram	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].spaceFactor	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].string	0x0308	"%s"	FCF[%d].Text.Item[%d]	

FCF[%d].Text.Item[%d].symbol	0x0308	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].textAspect	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].textColour	0x0308	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	FCF[%d].Text.Item[%d]	

FCF[%d].Text.Item[%d].textHeight	0x0308	"%.16g"	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].textThickness	0x0308	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	FCF[%d].Text.Item[%d]	
FCF[%d].Text.Item[%d].underline	0x0308	String representing enumeration such that: "0"=over, "1"=under	FCF[%d].Text.Item[%d]	
FCF[%d].Text.justification	0x0308	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	FCF[%d].Text	
FCF[%d].Text.language	0x0308	"%s"	FCF[%d].Text	
FCF[%d].Text.lineFactor	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.name	0x0308	"%s"	FCF[%d].Text	
FCF[%d].Text.outlineType	0x0308	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	FCF[%d].Text	
FCF[%d].Text.spaceFactor	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.textAspect	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.textColour	0x0308	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	FCF[%d].Text	
FCF[%d].Text.textHeight	0x0308	"%.16g"	FCF[%d].Text	
FCF[%d].Text.textOrientation	0x0308	String representing enumeration such that: "0"=horizontal, "1"=vertical	FCF[%d].Text	

FCF[%d].Text.textThickness	0x0308	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	FCF[%d].Text	
FCF[%d].Text.underline	0x0308	String representing enumeration such that: "0"=over, "1"=under	FCF[%d].Text	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.bold	0x0308	"%d"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.font	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.italic	0x0308	"%d"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.italic Angle	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].bold	0x0308	"%d"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].font	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].format	0x0308	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].italic	0x0308	"%d"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].italicAngle	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].justification	0x0308	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].language	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].lineFactor	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].outlineType	0x0308	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].spaceFactor	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].string	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].symbol	0x0308	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textAspect	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textColour	0x0308	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textHeight	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textThickness	0x0308	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].underline	0x0308	String representing enumeration such that: "0"=over, "1"=under	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.justification	0x0308	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.language	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.lineFactor	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.name	0x0308	"%s"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	

FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.outlineType	0x0308	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.spaceFactor	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textAspect	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textColour	0x0308	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textHeight	0x0308	"%.16g"	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textOrientation	0x0308	String representing enumeration such that: "0"=horizontal, "1"=vertical	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textThickness	0x0308	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.underline	0x0308	String representing enumeration such that: "0"=over, "1"=under	FCF[%d].ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	
fieldWeld	0x0008	"%d"		
filename	0x0114	"%s"		

fitGrade	0x0082	"%d"		
flag	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
font	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
foreDistance	0x0242	"%.16g"	type	Expects type property to be closed
foreThroughAll	0x0242	"%d"	type	Expects type property to be closed
fraction	0x0082	"%d"		
fractionSize	0x0082	String representing enumeration such that: "0"=full, "1"=twothirds, "2"=threequarters		
FreeState	0x0082	NULL		
fullCircle	0x0306	"%d"		
functionalSubscript	0x0118	"%s"		
GeneralText	0x0230	NULL		
GeneralText.bold	0x0230	"%d"	GeneralText	
GeneralText.font	0x0230	"%s"	GeneralText	
GeneralText.italic	0x0230	"%d"	GeneralText	
GeneralText.italicAngle	0x0230	"%.16g"	GeneralText	
GeneralText.Item[%d]	0x0230	"%s"	GeneralText	
GeneralText.Item[%d].bold	0x0230	"%d"	GeneralText.Item[%d]	
GeneralText.Item[%d].font	0x0230	"%s"	GeneralText.Item[%d]	

GeneralText.Item[%d].format	0x0230	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	GeneralText.Item[%d]	
GeneralText.Item[%d].italic	0x0230	"%d"	GeneralText.Item[%d]	
GeneralText.Item[%d].italicAngle	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].justification	0x0230	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	GeneralText.Item[%d]	
GeneralText.Item[%d].language	0x0230	"%s"	GeneralText.Item[%d]	
GeneralText.Item[%d].lineFactor	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].outlineType	0x0230	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	GeneralText.Item[%d]	
GeneralText.Item[%d].spaceFactor	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].string	0x0230	"%s"	GeneralText.Item[%d]	

GeneralText.Item[%d].symbol	0x0230	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	GeneralText.Item[%d]	
GeneralText.Item[%d].textAspect	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].textColour	0x0230	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	GeneralText.Item[%d]	

GeneralText.Item[%d].textHeight	0x0230	"%.16g"	GeneralText.Item[%d]	
GeneralText.Item[%d].textThickness	0x0230	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	GeneralText.Item[%d]	
GeneralText.Item[%d].underline	0x0230	String representing enumeration such that: "0"=over, "1"=under	GeneralText.Item[%d]	
GeneralText.justification	0x0230	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	GeneralText	
GeneralText.language	0x0230	"%s"	GeneralText	
GeneralText.lineFactor	0x0230	"%.16g"	GeneralText	
GeneralText.name	0x0230	"%s"	GeneralText	
GeneralText.outlineType	0x0230	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	GeneralText	
GeneralText.spaceFactor	0x0230	"%.16g"	GeneralText	
GeneralText.textAspect	0x0230	"%.16g"	GeneralText	
GeneralText.textColour	0x0230	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	GeneralText	
GeneralText.textHeight	0x0230	"%.16g"	GeneralText	
GeneralText.textOrientation	0x0230	String representing enumeration such that: "0"=horizontal, "1"=vertical	GeneralText	

GeneralText.textThickness	0x0230	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	GeneralText	
GeneralText.underline	0x0230	String representing enumeration such that: "0"=over, "1"=under	GeneralText	
grade	0x0082	"%d"		
group	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
groupid	0x0004	"%s"		
holeDeviation	0x0307	"%s"		
holeGrade	0x0307	"%d"		
holeLowerDelta	0x0307	"%.16g"		
holeUpperDelta	0x0307	"%.16g"		
hotSpotPosition	0x0118	"%.16g %.16g %.16g"		
identifier	0x0231 0x0232 0x0233 0x0100 0x0235 0x0244	"%s"		
includeI	0x0239	"%d"		
includeJ	0x0239	"%d"		
includeK	0x0239	"%d"		
includeLabel	0x0239	"%d"		
includeLabels	0x0240 0x0241	"%d"		
includeLevel	0x0239	"%d"		
includeX	0x0239	"%d"		
includeY	0x0239	"%d"		
includeZ	0x0239	"%d"		
index	0x0088	"%d"		
indexDirection	0x0230	"%.16g %.16g %.16g"		
individualExtensions	0x0306	"%d"		
individuallyApplied	0x0084	"%d"		
innerDiameter	0x0088	"%.16g"		
invertText	0x0041	"%d"		
iPrefix	0x0239	"%s"		
isReference	0x0082	"%d"		
iSuffix	0x0239	"%s"		

italic	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		
italicAngle	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
itemName	0x0233	"%s"		
jisArrowSidePartialPenetrationGrooveWeld	0x0008	"%d"		
jisOtherSidePartialPenetrationGrooveWeld	0x0008	"%d"		
jisStaggeredFilletWeld	0x0008	"%d"		
jointId	0x0004	"%s"		
jPrefix	0x0239	"%s"		
jSuffix	0x0239	"%s"		
justification	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=right, "1"=centre, "2"=left		

Keyword[%d]	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
kPrefix	0x0239	"%s"		
kSuffix	0x0239	"%s"		
label	0x0084 0x0088	"%s"		
labelPrefix	0x0239	"%s"		
labelSuffix	0x0239	"%s"		
language	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
lay	0x0041	String representing enumeration such that: "0"=parallel, "1"=multidirectional, "2"=perpendicular, "3"=circular, "4"=particulate, "5"=angularboth, "6"=radial		
layer	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"		

Leader[%d]	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	NULL		
Leader[%d].arrowAngle	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].arrowColour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Leader[%d]	
Leader[%d].arrowLength	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	

Leader[%d].arrowLineThickness	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Leader[%d]	
Leader[%d].arrowLineType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=unset, "1"=longdashed, "2"=dotted, "3"=dotteddashed, "4"=phantom, "5"=dashed, "6"=solid, "7"=centreline	Leader[%d]	
Leader[%d].arrowOutsideLength	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].arrowPlacement	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=out, "1"=in	Leader[%d]	

Leader[%d].arrowType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=closed, "1"=open, "2"=filled, "3"=none, "4"=origin, "5"=cross, "6"=dot, "7"=filledot, "8"=wedge, "9"=plus, "10"=x, "11"=closedsolid, "12"=closeddouble, "13"=closeddoublesolid, "14"=opendouble, "15"=filledouble, "16"=integral, "17"=box, "18"=filledbox, "19"=datum, "20"=filleddatum, "21"=solidorigin, "22"=filledorigin, "23"=xorigin, "24"=solidbox	Leader[%d]	
Leader[%d].colour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Leader[%d]	
Leader[%d].dotDiameter	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	

Leader[%d].extensionLineColour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Leader[%d]	
Leader[%d].extensionLineExtension	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].extensionLineGap	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].extensionLineThickness	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Leader[%d]	

Leader[%d].extensionLineType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=unset, "1"=longdashed, "2"=dotted, "3"=dotteddashed, "4"=phantom, "5"=dashed, "6"=solid, "7"=centreline	Leader[%d]	
Leader[%d].lineTextGap	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].lineType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=unset, "1"=longdashed, "2"=dotted, "3"=dotteddashed, "4"=phantom, "5"=dashed, "6"=solid, "7"=centreline	Leader[%d]	
Leader[%d].radiusToCentre	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d"	Leader[%d]	

Leader[%d].Reference	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formatting is used to semantically tie a singlar semantic aspect with an entity.	Leader[%d]	
Leader[%d].stubDirection	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=left, "1"=right	Leader[%d]	
Leader[%d].stubLength	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].terminator	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g %.16g %.16g"	Leader[%d]	

Leader[%d].thickness	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Leader[%d]	
Leader[%d].tParm	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"	Leader[%d]	
Leader[%d].uvParm. U	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		Expect to also have Leader[%d].uvParam .V.
Leader[%d].uvParm. V	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		Expect to also have Leader[%d].uvParam .U.
leadingZero	0x0081	"%d"		
length	0x0088	"%.16g"		
level	0x0042	"%s"		
levelPrefix	0x0239	"%s"		
levelSuffix	0x0239	"%s"		

limitFitOrder	0x0082 0x0307	String representing enumeration such that: "0"=valuelimitfittolerance, "1"=tolerancevaluelimitfit, "2"=valuetolerancelimitfit		
limitFitParenthesis	0x0082 0x0307	String representing enumeration such that: "0"=none, "1"=valuelimitfit, "2"=limitfit, "3"=tolerance		
linearPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
lineFactor	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
lineStyle	0x0114	String representing enumeration such that: "0"=unset, "1"=longdashed, "2"=dotted, "3"=dotteddashed, "4"=phantom, "5"=dashed, "6"=solid, "7"=centreline		
locationOnCurve	0x0241	"%.16g"		
lowerDelta	0x0082 0x0041	"%.16g"		
lowerDeltaDenominator	0x0082	"%d"		
machiningAllowance	0x0041	"%s"		
mainEdgeType	0x0118	String representing enumeration such that: "0"=ce, "1"=e, "2"=e, "3"=ek, "4"=co		
mainHoleType	0x0118	String representing enumeration such that: "0"=dl, "1"=h, "2"=hk, "3"=ch, "4"=hk, "5"=ac, "6"=h, "7"=bl, "8"=ch		

mainRectSlotFreeformType	0x0118	String representing enumeration such that: "0"=dl, "1"=h, "2"=hk, "3"=bl, "4"=hk, "5"=ac, "6"=h, "7"=ch		
mainSurfaceType	0x0118	String representing enumeration such that: "0"=sk, "1"=co, "2"=j, "3"=s, "4"=sl, "5"=sk, "6"=cs, "7"=s, "8"=ck		
materialSide	0x0242	"%d"		
maxBonus	0x0081	"%d"		
maxBonusPrecision	0x0081	"%d"		
maxBonusValue	0x0081	"%.16g"		
maxRoughness	0x0041	"%s"		
maxRoughness2	0x0041	"%s"		
measurementDirection	0x0042	"%.16g %.16g %.16g"		
minRoughness	0x0041	"%s"		
minRoughness2	0x0041	"%s"		
modifier	0x0041	"%d"		
movable	0x0088	"%d"		
movableTargetStubDirection	0x0088	String representing enumeration such that: "0"=left, "1"=right		
movableTargetStubLength	0x0088	"%.16g"		
nameModifier	0x0233	"%s"		
Nomenclature	0x0231 0x0232	NULL		
Nomenclature.bold	0x0231 0x0232	"%d"	Nomenclature	
Nomenclature.font	0x0231 0x0232	"%s"	Nomenclature	
Nomenclature.italic	0x0231 0x0232	"%d"	Nomenclature	
Nomenclature.italicAngle	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.Item[%d]	0x0231 0x0232	"%s"	Nomenclature	
Nomenclature.Item[%d].bold	0x0231 0x0232	"%d"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].font	0x0231 0x0232	"%s"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].format	0x0231 0x0232	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	Nomenclature.Item[%d]	
Nomenclature.Item[%d].italic	0x0231 0x0232	"%d"	Nomenclature.Item[%d]	

Nomenclature.Item[%d].italicAngle	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].justification	0x0231 0x0232	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Nomenclature.Item[%d]	
Nomenclature.Item[%d].language	0x0231 0x0232	"%s"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].lineFactor	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].outlineType	0x0231 0x0232	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Nomenclature.Item[%d]	
Nomenclature.Item[%d].spaceFactor	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].string	0x0231 0x0232	"%s"	Nomenclature.Item[%d]	

Nomenclature.Item[%d].symbol	0x0231 0x0232	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	Nomenclature.Item[%d]	
Nomenclature.Item[%d].textAspect	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].textColour	0x0231 0x0232	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Nomenclature.Item[%d]	

Nomenclature.Item[%d].textHeight	0x0231 0x0232	"%.16g"	Nomenclature.Item[%d]	
Nomenclature.Item[%d].textThickness	0x0231 0x0232	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Nomenclature.Item[%d]	
Nomenclature.Item[%d].underline	0x0231 0x0232	String representing enumeration such that: "0"=over, "1"=under	Nomenclature.Item[%d]	
Nomenclature.justification	0x0231 0x0232	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Nomenclature	
Nomenclature.language	0x0231 0x0232	"%s"	Nomenclature	
Nomenclature.lineFactor	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.name	0x0231 0x0232	"%s"	Nomenclature	
Nomenclature.outlineType	0x0231 0x0232	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Nomenclature	
Nomenclature.spaceFactor	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.textAspect	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.textColour	0x0231 0x0232	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Nomenclature	
Nomenclature.textHeight	0x0231 0x0232	"%.16g"	Nomenclature	
Nomenclature.textOrientation	0x0231 0x0232	String representing enumeration such that: "0"=horizontal, "1"=vertical	Nomenclature	

Nomenclature.textThickness	0x0231 0x0232	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Nomenclature	
Nomenclature.underline	0x0231 0x0232	String representing enumeration such that: "0"=over, "1"=under	Nomenclature	
normalDirection	0x0004 0x0042 0x0118 0x0238	"%.16g %.16g %.16g"		
Note	0x0118	NULL		
Note.bold	0x0118	"%d"	Note	
Note.font	0x0118	"%s"	Note	
Note.italic	0x0118	"%d"	Note	
Note.italicAngle	0x0118	"%.16g"	Note	
Note.Item[%d]	0x0118	"%s"	Note	
Note.Item[%d].bold	0x0118	"%d"	Note.Item[%d]	
Note.Item[%d].font	0x0118	"%s"	Note.Item[%d]	
Note.Item[%d].format	0x0118	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	Note.Item[%d]	
Note.Item[%d].italic	0x0118	"%d"	Note.Item[%d]	
Note.Item[%d].italicAngle	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].justification	0x0118	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Note.Item[%d]	
Note.Item[%d].language	0x0118	"%s"	Note.Item[%d]	
Note.Item[%d].lineFactor	0x0118	"%.16g"	Note.Item[%d]	

Note.Item[%d].outline Type	0x0118	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Note.Item[%d]	
Note.Item[%d].space Factor	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].string	0x0118	"%s"	Note.Item[%d]	

Note.Item[%d].symbol	0x0118	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	Note.Item[%d]	
Note.Item[%d].textAspect	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].textColour	0x0118	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Note.Item[%d]	

Note.Item[%d].textHeight	0x0118	"%.16g"	Note.Item[%d]	
Note.Item[%d].textThickness	0x0118	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Note.Item[%d]	
Note.Item[%d].underline	0x0118	String representing enumeration such that: "0"=over, "1"=under	Note.Item[%d]	
Note.justification	0x0118	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Note	
Note.language	0x0118	"%s"	Note	
Note.lineFactor	0x0118	"%.16g"	Note	
Note.name	0x0118	"%s"	Note	
Note.outlineType	0x0118	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Note	
Note.spaceFactor	0x0118	"%.16g"	Note	
Note.textAspect	0x0118	"%.16g"	Note	
Note.textColour	0x0118	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Note	
Note.textHeight	0x0118	"%.16g"	Note	
Note.textOrientation	0x0118	String representing enumeration such that: "0"=horizontal, "1"=vertical	Note	
Note.textThickness	0x0118	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Note	
Note.underline	0x0118	String representing enumeration such that: "0"=over, "1"=under	Note	

notToScale	0x0082	"%d"		
OpenField	0x0231 0x0232	NULL		
OpenField.bold	0x0231 0x0232	"%d"	OpenField	
OpenField.font	0x0231 0x0232	"%s"	OpenField	
OpenField.italic	0x0231 0x0232	"%d"	OpenField	
OpenField.italicAngle	0x0231 0x0232	"%.16g"	OpenField	
OpenField.Item[%d]	0x0231 0x0232	"%s"	OpenField	
OpenField.Item[%d].bold	0x0231 0x0232	"%d"	OpenField.Item[%d]	
OpenField.Item[%d].font	0x0231 0x0232	"%s"	OpenField.Item[%d]	
OpenField.Item[%d].format	0x0231 0x0232	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	OpenField.Item[%d]	
OpenField.Item[%d].italic	0x0231 0x0232	"%d"	OpenField.Item[%d]	
OpenField.Item[%d].italicAngle	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	
OpenField.Item[%d].justification	0x0231 0x0232	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	OpenField.Item[%d]	
OpenField.Item[%d].language	0x0231 0x0232	"%s"	OpenField.Item[%d]	
OpenField.Item[%d].lineFactor	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	
OpenField.Item[%d].outlineType	0x0231 0x0232	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	OpenField.Item[%d]	

OpenField.Item[%d].spaceFactor	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	
OpenField.Item[%d].string	0x0231 0x0232	"%s"	OpenField.Item[%d]	
OpenField.Item[%d].symbol	0x0231 0x0232	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	OpenField.Item[%d]	
OpenField.Item[%d].textAspect	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	

OpenField.Item[%d].textColour	0x0231 0x0232	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	OpenField.Item[%d]	
OpenField.Item[%d].textHeight	0x0231 0x0232	"%.16g"	OpenField.Item[%d]	
OpenField.Item[%d].textThickness	0x0231 0x0232	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	OpenField.Item[%d]	
OpenField.Item[%d].underline	0x0231 0x0232	String representing enumeration such that: "0"=over, "1"=under	OpenField.Item[%d]	
OpenField.justification	0x0231 0x0232	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	OpenField	
OpenField.language	0x0231 0x0232	"%s"	OpenField	
OpenField.lineFactor	0x0231 0x0232	"%.16g"	OpenField	
OpenField.name	0x0231 0x0232	"%s"	OpenField	
OpenField.outlineType	0x0231 0x0232	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	OpenField	
OpenField.spaceFactor	0x0231 0x0232	"%.16g"	OpenField	
OpenField.textAspect	0x0231 0x0232	"%.16g"	OpenField	
OpenField.textColour	0x0231 0x0232	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	OpenField	
OpenField.textHeight	0x0231 0x0232	"%.16g"	OpenField	
OpenField.textOrientation	0x0231 0x0232	String representing enumeration such that: "0"=horizontal, "1"=vertical	OpenField	

OpenField.textThickn ess	0x0231 0x0232	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	OpenField	
OpenField.underline	0x0231 0x0232	String representing enumeration such that: "0"=over, "1"=under	OpenField	
optDirection	0x0230	"%.16g %.16g %.16g"		
origin	0x0082	String representing enumeration such that: "0"=first, "1"=second		
otherSideFinishSymb ol	0x0008	String representing enumeration such that: "0"=unset, "1"=unspecified, "2"=machining, "3"=grinding, "4"=chipping, "5"=none, "6"=hammering, "7"=peening, "8"=rolling		
otherSideGrooveAngl e	0x0008	"%s"		
otherSideGrooveGap	0x0008	"%s"		
otherSideLongitudina lSize	0x0008	"%s"		
otherSideMainSize	0x0008	"%s"		
otherSideStaggeredS ize	0x0008	"%s"		
otherSideSupplemen talSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=convex, "2"=toesblended, "3"=concave, "4"=flush		

otherSideSymbol	0x0008	String representing enumeration such that: "0"=unset, "1"=jisseam, "2"=singlebevelgroove, "3"=singleugroove, "4"=edgeflange, "5"=surface, "6"=steepflankedsingle bevel, "7"=flaresinglevgroove, "8"=surfacejoint, "9"=singlevgroove, "10"=inclinedjoint, "11"=plug, "12"=fillet, "13"=squaregroove, "14"=spot, "15"=singlej, "16"=square, "17"=overlay, "18"=singleu, "19"=singlejgroove, "20"=doubleflange, "21"=singleflange, "22"=spotprojected, "23"=plugandslot, "24"=singlebevelbroadr ootface, "25"=edge, "26"=jisstaggeredfillet1, "27"=singlev, "28"=flaresinglebevelgr oove, "29"=singlevbroadrootf ace, "30"=steepflankedsingl ev, "31"=removablebackin gstrip, "32"=permanentbackin gstrip, "33"=jisfillet, "34"=singlebevel, "35"=backing, "36"=foldjoint, "37"=jisstaggeredfillet2, "38"=seam, "39"=bead		

outlineType	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram		
overrun	0x0306	"%.16g"		
parenthesis	0x0041	String representing enumeration such that: "0"=none, "1"=left, "2"=right, "3"=both		
PartNumber	0x0118	NULL		
PartNumber.bold	0x0118	"%d"	PartNumber	
PartNumber.font	0x0118	"%s"	PartNumber	
PartNumber.italic	0x0118	"%d"	PartNumber	
PartNumber.italicAngle	0x0118	"%.16g"	PartNumber	
PartNumber.Item[%d]	0x0118	"%s"	PartNumber	
PartNumber.Item[%d].bold	0x0118	"%d"	PartNumber.Item[%d]	
PartNumber.Item[%d].font	0x0118	"%s"	PartNumber.Item[%d]	
PartNumber.Item[%d].format	0x0118	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	PartNumber.Item[%d]	
PartNumber.Item[%d].italic	0x0118	"%d"	PartNumber.Item[%d]	
PartNumber.Item[%d].italicAngle	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].justification	0x0118	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	PartNumber.Item[%d]	

PartNumber.Item[%d].language	0x0118	"%s"	PartNumber.Item[%d]	
PartNumber.Item[%d].lineFactor	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].outlineType	0x0118	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	PartNumber.Item[%d]	
PartNumber.Item[%d].spaceFactor	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].string	0x0118	"%s"	PartNumber.Item[%d]	

PartNumber.Item[%d].symbol	0x0118	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	PartNumber.Item[%d]	
PartNumber.Item[%d].textAspect	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].textColour	0x0118	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	PartNumber.Item[%d]	

PartNumber.Item[%d].textHeight	0x0118	"%.16g"	PartNumber.Item[%d]	
PartNumber.Item[%d].textThickness	0x0118	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	PartNumber.Item[%d]	
PartNumber.Item[%d].underline	0x0118	String representing enumeration such that: "0"=over, "1"=under	PartNumber.Item[%d]	
PartNumber.justification	0x0118	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	PartNumber	
PartNumber.language	0x0118	"%s"	PartNumber	
PartNumber.lineFactor	0x0118	"%.16g"	PartNumber	
PartNumber.name	0x0118	"%s"	PartNumber	
PartNumber.outlineType	0x0118	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	PartNumber	
PartNumber.spaceFactor	0x0118	"%.16g"	PartNumber	
PartNumber.textAspect	0x0118	"%.16g"	PartNumber	
PartNumber.textColor	0x0118	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	PartNumber	
PartNumber.textHeight	0x0118	"%.16g"	PartNumber	
PartNumber.textOrientation	0x0118	String representing enumeration such that: "0"=horizontal, "1"=vertical	PartNumber	

PartNumber.textThickness	0x0118	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	PartNumber	
PartNumber.underline	0x0118	String representing enumeration such that: "0"=over, "1"=under	PartNumber	
pinDirection	0x0118	"%.16g %.16g %.16g"		
pitchDiaDeviation	0x0082 0x0307	"%s"		
pitchDiaGrade	0x0082 0x0307	"%d"		
planar	0x0088	"%d"		
point1	0x0088	"%.16g %.16g %.16g"		
point2	0x0088	"%.16g %.16g %.16g"		
position	0x0230 0x0004 0x0042 0x0238	"%.16g %.16g %.16g"		
Position[%d]	0x0306	"%.16g %.16g %.16g"		
precision	0x0082 0x0230 0x0041 0x0088 0x0004 0x0008 0x0118 0x0238 0x0239 0x0241 0x0307	"%d"		
ProcessText	0x0041 0x0004 0x0008	NULL		
ProcessText.bold	0x0041 0x0004 0x0008	"%d"	ProcessText	
ProcessText.font	0x0041 0x0004 0x0008	"%s"	ProcessText	
ProcessText.italic	0x0041 0x0004 0x0008	"%d"	ProcessText	
ProcessText.italicAngle	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.Item[%d]	0x0041 0x0004 0x0008	"%s"	ProcessText	
ProcessText.Item[%d].bold	0x0041 0x0004 0x0008	"%d"	ProcessText.Item[%d]	
ProcessText.Item[%d].font	0x0041 0x0004 0x0008	"%s"	ProcessText.Item[%d]	
ProcessText.Item[%d].format	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	ProcessText.Item[%d]	
ProcessText.Item[%d].italic	0x0041 0x0004 0x0008	"%d"	ProcessText.Item[%d]	
ProcessText.Item[%d].italicAngle	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	

ProcessText.Item[%d].justification	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	ProcessText.Item[%d]	
ProcessText.Item[%d].language	0x0041 0x0004 0x0008	"%s"	ProcessText.Item[%d]	
ProcessText.Item[%d].lineFactor	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].outlineType	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	ProcessText.Item[%d]	
ProcessText.Item[%d].spaceFactor	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].string	0x0041 0x0004 0x0008	"%s"	ProcessText.Item[%d]	

ProcessText.Item[%d].symbol	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	ProcessText.Item[%d]	
ProcessText.Item[%d].textAspect	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].textColour	0x0041 0x0004 0x0008	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	ProcessText.Item[%d]	

ProcessText.Item[%d].textHeight	0x0041 0x0004 0x0008	"%.16g"	ProcessText.Item[%d]	
ProcessText.Item[%d].textThickness	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	ProcessText.Item[%d]	
ProcessText.Item[%d].underline	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=over, "1"=under	ProcessText.Item[%d]	
ProcessText.justification	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	ProcessText	
ProcessText.language	0x0041 0x0004 0x0008	"%s"	ProcessText	
ProcessText.lineFactor	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.name	0x0041 0x0004 0x0008	"%s"	ProcessText	
ProcessText.outlineType	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	ProcessText	
ProcessText.spaceFactor	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.textAspect	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.textColour	0x0041 0x0004 0x0008	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	ProcessText	
ProcessText.textHeight	0x0041 0x0004 0x0008	"%.16g"	ProcessText	
ProcessText.textOrientation	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=horizontal, "1"=vertical	ProcessText	

ProcessText.textThickness	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	ProcessText	
ProcessText.underline	0x0041 0x0004 0x0008	String representing enumeration such that: "0"=over, "1"=under	ProcessText	
profileType	0x0081	String representing enumeration such that: "0"=bilateral, "1"=unilateralinside, "2"=unilateraloutside, "3"=bilateralunequal		
profileValue	0x0081	"%.16g"		
profileValue2	0x0081	"%.16g"		
projected	0x0082	"%d"		
projectedPrecision	0x0081	"%d"		
punchDirection	0x0118	"%.16g %.16g %.16g"		
radialPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		
radius	0x0235	"%.16g"		
referenceDisplay	0x0082	String representing enumeration such that: "0"=parenthesis, "1"=reference, "2"=matched		
referenceSign	0x0008	"%d"		
revision	0x0231 0x0232 0x0233 0x0100 0x0235 0x0244	"%s"		
roughness	0x0041	"%s"		
roughness2	0x0041	"%s"		
roughnessCutoff	0x0041	"%s"		
roughnessSpacing	0x0041	"%s"		
SafetyClassification	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
scale	0x0114	"%.16g"		
shaftDeviation	0x0307	"%s"		
shaftGrade	0x0307	"%d"		

shaftLowerDelta	0x0307	"%.16g"		
shaftUpperDelta	0x0307	"%.16g"		
shapeTextOutsideSymbol	0x0088	"%d"		
shortDash	0x0306	"%.16g"		
singleLine	0x0041 0x0307	"%d"		
size	0x0088	"%s"		
spaceFactor	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
standard	0x0082 0x0081 0x0084 0x0088 0x0100 0x0235 0x0244 0x0239 0x0240 0x0241	String representing enumeration such that: "0"=asme_y145m_1994, "1"=jis, "2"=iso, "3"=bs, "4"=ansi_y145m_1982, "5"=asme_y1441m_2003, "6"=din, "7"=gm_addendum_1994, "8"=asme_y145_2009		
standard	0x0041	String representing enumeration such that: "0"=jis, "1"=ansi_y1436_1993, "2"=iso, "3"=asme_y1436m_1996, "4"=din, "5"=gb, "6"=iso_1302_2002, "7"=eskd, "8"=din_en_iso_1302_2002		
standard	0x0008	String representing enumeration such that: "0"=ansiaws_a24-98 "1"=iso_2556 "2"=jis_z_3021 "3"=din "4"=eskd "5"=gb		
Statistical	0x0082	NULL		
statisticalPlacement	0x0082	String representing enumeration such that: "0"=before, "1"=above, "2"=below, "3"=after		

style	0x0082	String representing enumeration such that: "0"=lineardiametral, "1"=radial, "2"=controlledradial, "3"=diametral, "4"=limits, "5"=ordinate, "6"=sphericalradial, "7"=sphericaldiametral		
subType	0x0118	String representing enumeration such that: "0"=mechanicallyfastened, "1"=manufacturinganda ssembly, "2"=diesmolds, "3"=fixing, "4"=temporarytransferr ed		
symbol	0x0118	String representing enumeration such that: "0"=rectangle, "1"=edge, "2"=hole, "3"=freeform, "4"=slot, "5"=surface		
symbolColour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string		
symbolType	0x0230	"%s"		
Text	0x0081	NULL		
Text.bold	0x0081	"%d"	Text	
Text.font	0x0081	"%s"	Text	
Text.italic	0x0081	"%d"	Text	
Text.italicAngle	0x0081	"%.16g"	Text	
Text.Item[%d]	0x0081	"%s"	Text	
Text.Item[%d].bold	0x0081	"%d"	Text.Item[%d]	
Text.Item[%d].font	0x0081	"%s"	Text.Item[%d]	

Text.Item[%d].format	0x0081	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	Text.Item[%d]	
Text.Item[%d].italic	0x0081	"%d"	Text.Item[%d]	
Text.Item[%d].italicAngle	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].justification	0x0081	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Text.Item[%d]	
Text.Item[%d].language	0x0081	"%s"	Text.Item[%d]	
Text.Item[%d].lineFactor	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].outlineType	0x0081	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Text.Item[%d]	
Text.Item[%d].spaceFactor	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].string	0x0081	"%s"	Text.Item[%d]	

Text.Item[%d].symbol	0x0081	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	Text.Item[%d]	
Text.Item[%d].textAspect	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].textColour	0x0081	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Text.Item[%d]	

Text.Item[%d].textHeight	0x0081	"%.16g"	Text.Item[%d]	
Text.Item[%d].textThickness	0x0081	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Text.Item[%d]	
Text.Item[%d].underline	0x0081	String representing enumeration such that: "0"=over, "1"=under	Text.Item[%d]	
Text.justification	0x0081	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Text	
Text.language	0x0081	"%s"	Text	
Text.lineFactor	0x0081	"%.16g"	Text	
Text.name	0x0081	"%s"	Text	
Text.outlineType	0x0081	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Text	
Text.spaceFactor	0x0081	"%.16g"	Text	
Text.textAspect	0x0081	"%.16g"	Text	
Text.textColour	0x0081	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Text	
Text.textHeight	0x0081	"%.16g"	Text	
Text.textOrientation	0x0081	String representing enumeration such that: "0"=horizontal, "1"=vertical	Text	
Text.textThickness	0x0081	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Text	
Text.underline	0x0081	String representing enumeration such that: "0"=over, "1"=under	Text	

Text[%d]	0x0100 0x0235 0x0244	NULL		
Text[%d].bold	0x0100 0x0235 0x0244	"%d"	Text[%d]	
Text[%d].font	0x0100 0x0235 0x0244	"%s"	Text[%d]	
Text[%d].italic	0x0100 0x0235 0x0244	"%d"	Text[%d]	
Text[%d].italicAngle	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].Item[%d]	0x0100 0x0235 0x0244	"%s"	Text[%d]	
Text[%d].Item[%d].bold	0x0100 0x0235 0x0244	"%d"	Text[%d].Item[%d]	
Text[%d].Item[%d].font	0x0100 0x0235 0x0244	"%s"	Text[%d].Item[%d]	
Text[%d].Item[%d].format	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	Text[%d].Item[%d]	
Text[%d].Item[%d].italic	0x0100 0x0235 0x0244	"%d"	Text[%d].Item[%d]	
Text[%d].Item[%d].italicAngle	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	
Text[%d].Item[%d].justification	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Text[%d].Item[%d]	
Text[%d].Item[%d].language	0x0100 0x0235 0x0244	"%s"	Text[%d].Item[%d]	
Text[%d].Item[%d].lineFactor	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	
Text[%d].Item[%d].outlineType	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoreddirectangle, "21"=parallelogram	Text[%d].Item[%d]	

Text[%d].Item[%d].spaceFactor	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	
Text[%d].Item[%d].string	0x0100 0x0235 0x0244	"%s"	Text[%d].Item[%d]	
Text[%d].Item[%d].symbol	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	Text[%d].Item[%d]	
Text[%d].Item[%d].textAspect	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	

Text[%d].Item[%d].textColour	0x0100 0x0235 0x0244	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Text[%d].Item[%d]	
Text[%d].Item[%d].textHeight	0x0100 0x0235 0x0244	"%.16g"	Text[%d].Item[%d]	
Text[%d].Item[%d].textThickness	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Text[%d].Item[%d]	
Text[%d].Item[%d].underline	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=over, "1"=under	Text[%d].Item[%d]	
Text[%d].justification	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	Text[%d]	
Text[%d].language	0x0100 0x0235 0x0244	"%s"	Text[%d]	
Text[%d].lineFactor	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].name	0x0100 0x0235 0x0244	"%s"	Text[%d]	
Text[%d].outlineType	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	Text[%d]	
Text[%d].spaceFactor	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].textAspect	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	
Text[%d].textColour	0x0100 0x0235 0x0244	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	Text[%d]	
Text[%d].textHeight	0x0100 0x0235 0x0244	"%.16g"	Text[%d]	

Text[%d].textOrientation	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=horizontal, "1"=vertical	Text[%d]	
Text[%d].textThickness	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	Text[%d]	
Text[%d].underline	0x0100 0x0235 0x0244	String representing enumeration such that: "0"=over, "1"=under	Text[%d]	
textAspect	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
textBox	0x0239	"%d"		
textColour	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string		
textDirection	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g %.16g %.16g"		

textHeight	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g"		
textOrigin	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%.16g %.16g %.16g"		
textStubDirection	0x0088	String representing enumeration such that: "0"=left, "1"=right		
textStubLength	0x0088	"%.16g"		
textThickness	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin		
thickness	0x0004	"%.16g"		
thicknesses	0x0004	"%d"		
threadClass	0x0082	"%d"		
threaded	0x0230	"%d"		
ToleranceCompartment[%d]	0x0081	NULL		
ToleranceCompartment[%d].commonZone	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].FreeState	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].maxBonus	0x0081	"%d"	ToleranceCompartment[%d]	

ToleranceCompartment[%d].maxBonusValue	0x0081	"%.16g"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].modifier	0x0081	String representing enumeration such that: "0"=lmc, "1"=mmc, "2"=rfs	ToleranceCompartment[%d]	
ToleranceCompartment[%d].precision	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	0x0081	NULL	ToleranceCompartment[%d].PrimaryDatum	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].datumRef	0x0081	"%d %d %d %s" such that first in is the dst id, second the dst type, third the reason(optional) and the string the path. This property formatting is used to semantically tie a singular semantic aspect with an entity.	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].FreeState	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].label	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].modifier	0x0081	String representing enumeration such that: "0"=lmc, "1"=mmc, "2"=rfs	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	0x0081	NULL	ToleranceCompartment[%d].PrimaryDatum.Reference[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.bold	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.font	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.italic	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.italicAngle	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum.

				TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].bold	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].font	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].format	0x0081	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].italic	0x0081	"%d"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].italicAngle	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].justification	0x0081	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].language	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].lineFactor	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].outlineType	0x0081	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].spaceFactor	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].string	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].symbol	0x0081	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].text Aspect	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textColour	0x0081	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textHeight	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].textThickness	0x0081	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d].underline	0x0081	String representing enumeration such that: "0"=over, "1"=under	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.Item[%d]	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.justification	0x0081	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.language	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.lineFactor	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.name	0x0081	"%s"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.outlineType	0x0081	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.spaceFactor	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textAspect	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textColour	0x0081	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textHeight	0x0081	"%.16g"	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textOrientation	0x0081	String representing enumeration such that: "0"=horizontal, "1"=vertical	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.textThickness	0x0081	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes
ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel.underline	0x0081	String representing enumeration such that: "0"=over, "1"=under	ToleranceCompartment[%d].PrimaryDatum.Reference[%d].TextLabel	PrimaryDatum can be replaced with SecondaryDatum or TertiaryDatum. no notes

ToleranceCompartment[%d].projected	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].projectedValue	0x0081	"%s"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].projectionVector	0x0081	"%.16g %.16g %.16g"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].Statistical	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].tangentPlane	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].unitBasis	0x0081	"%d"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].unitBasisValue	0x0081	"%s"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].value	0x0081	"%s"	ToleranceCompartment[%d]	
ToleranceCompartment[%d].zoneShape	0x0081	String representing enumeration such that: "0"=diameter, "1"=sphericaldiameter, "2"=square	ToleranceCompartment[%d]	
toleranceDisplay	0x0082 0x0041 0x0307	String representing enumeration such that: "0"=minuslimit1, "1"=bilateral, "2"=pluslimit1, "3"=pluslimit2, "4"=equalbilateral, "5"=minuslimit2		
toleranceLeadingZero	0x0082 0x0041 0x0307	"%d"		
tolerancePrecision	0x0082 0x0307	"%d"		
toleranceTrailingZero	0x0082 0x0041 0x0307	"%d"		
trailingZero	0x0081	"%d"		
twoArrows	0x0242	"%d"		
type	0x0082	String representing enumeration such that: "0"=curvelength, "1"=linear, "2"=angular, "3"=radial		
type	0x0230	"%s"		
type	0x0041	String representing enumeration such that: "0"=basic, "1"=mrr, "2"=mrp		

type	0x0088	String representing enumeration such that: "0"=cylindrical, "1"=line, "2"=arbitrary, "3"=circular, "4"=point, "5"=annular, "6"=area, "7"=rectangular, "8"=spherical		
type	0x0004	String representing enumeration such that: "0"=unset, "1"=resistance, "2"=projection		
type	0x0241	String representing enumeration such that: "0"=bundlenote, "1"=dressingnote		
type	0x0242	String representing enumeration such that: "0"=plane, "1"=open, "2"=closed		
type	0x0244	String representing enumeration such that: "0"=exportcontrol, "1"=productproprietary info, "2"=governmentsecurity info		
underline	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=over, "1"=under		
units	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=metres, "1"=millimetres, "2"=inches		
unitText	0x0082	"%d"		
upperDelta	0x0082 0x0041	"%.16g"		

upperDeltaDenominator	0x0082	"%d"		
uriRefs[%d]	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
usage	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
UserDefinedSymbolText[%d]	0x0114	NULL		
UserDefinedSymbolText[%d].Text	0x0114	NULL	UserDefinedSymbolText[%d]	
UserDefinedSymbolText[%d].Text.bold	0x0114	"%d"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.font	0x0114	"%s"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.italic	0x0114	"%d"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.italicAngle	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.Item[%d]	0x0114	"%s"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.Item[%d].bold	0x0114	"%d"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].font	0x0114	"%s"	UserDefinedSymbolText[%d].Text.Item[%d]	

UserDefinedSymbolText[%d].Text.Item[%d].format	0x0114	String representing enumeration such that: "0"=int, "1"=ints, "2"=real, "3"=reals, "4"=boolean, "5"=booleans, "6"=string, "7"=reference, "8"=enum, "9"=list, "10"=datetime	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].italic	0x0114	"%d"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].italicAngle	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].justification	0x0114	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].language	0x0114	"%s"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].lineFactor	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].outlineType	0x0114	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoreddirectangle, "21"=parallelogram	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].spaceFactor	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].string	0x0114	"%s"	UserDefinedSymbolText[%d].Text.Item[%d]	

UserDefinedSymbolText[%d].Text.Item[%d].symbol	0x0114	String representing enumeration such that: "0"=centreline, "1"=partingline, "2"=depth, "3"=countersink, "4"=conicaltaper, "5"=slope, "6"=counterbore, "7"=square, "8"=phi, "9"=plusminus, "10"=degrees, "11"=between, "12"=arclength, "13"=leftparenthesis, "14"=rightparenthesis, "15"=projectedtolerance, "16"=mmc, "17"=lmc, "18"=freestate, "19"=ohm, "20"=circularrunout, "21"=totalrunout, "22"=profileofasurface, "23"=profileofaline, "24"=flatness, "25"=straightness, "26"=symmetry, "27"=perpendicularity, "28"=parallelism, "29"=cylindricity, "30"=concentricity, "31"=circularity, "32"=angularity, "33"=micro, "34"=position, "35"=envelope, "36"=rfs, "37"=tangentplane, "38"=lessthanorequal, "39"=greaterthanorequal, "40"=threadprefix, "41"=triangle, "42"=statistical, "43"=radius, "44"=circledu, "45"=fitfunction, "46"=safetycompliance, "47"=quantity, "48"=independency, "49"=continuousfeature, "50"=spotface	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].textAspect	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	

UserDefinedSymbolText[%d].Text.Item[%d].textColour	0x0114	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].textHeight	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].textThickness	0x0114	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.Item[%d].underline	0x0114	String representing enumeration such that: "0"=over, "1"=under	UserDefinedSymbolText[%d].Text.Item[%d]	
UserDefinedSymbolText[%d].Text.justification	0x0114	String representing enumeration such that: "0"=right, "1"=centre, "2"=left	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.language	0x0114	"%s"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.lineFactor	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.name	0x0114	"%s"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.outlineType	0x0114	String representing enumeration such that: "0"=box, "1"=triangle, "2"=circle, "3"=ellipse, "4"=underline, "5"=square, "6"=scoredcircle, "7"=diamond, "8"=flagright, "9"=flagleft, "10"=flagboth, "11"=oblong, "12"=oblongright, "13"=oblongleft, "14"=sticking, "15"=set, "16"=fixedsupport, "17"=nota, "18"=symmetricalpart, "19"=symmetricalset, "20"=scoredrectangle, "21"=parallelogram	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.spaceFactor	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.textAspect	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	

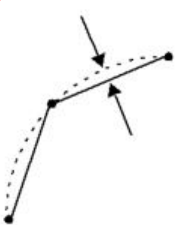
UserDefinedSymbolText[%d].Text.textColour	0x0114	"0x00%02x%02x%02x" such that the values of blue, green and red for the colour are encoded in the string	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.textHeight	0x0114	"%.16g"	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.textOrientation	0x0114	String representing enumeration such that: "0"=horizontal, "1"=vertical	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.textThickness	0x0114	String representing enumeration such that: "0"=normal, "1"=thick, "2"=thin	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].Text.underline	0x0114	String representing enumeration such that: "0"=over, "1"=under	UserDefinedSymbolText[%d].Text	
UserDefinedSymbolText[%d].textDirection	0x0114	"%.16g %.16g %.16g"	UserDefinedSymbolText[%d]	
UserDefinedSymbolText[%d].textOrigin	0x0114	"%.16g %.16g %.16g"	UserDefinedSymbolText[%d]	
valid	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	String representing enumeration such that: "0"=true, "1"=false, "2"=unknown		
value	0x0082 0x0307	"%.16g"		
valueDenominator	0x0082	"%d"		
valueLeadingZero	0x0307	"%d"		
ValueToCustomer	0x0082 0x0230 0x0041 0x0081 0x0084 0x0088 0x0004 0x0008 0x0231 0x0232 0x0233 0x0245 0x0100 0x0042 0x0118 0x0114 0x0238 0x0239 0x0240 0x0241 0x0242 0x0243 0x0244 0x0246 0x0306 0x0307	"%s"		
valueTrailingZero	0x0307	"%d"		
wavinessHeight	0x0041	"%s"		
wavinessSpacing	0x0041	"%s"		

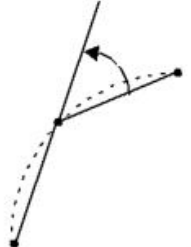
width	0x0088	"%.16g"		
xPrefix	0x0239	"%s"		
xSuffix	0x0239	"%s"		
yPrefix	0x0239	"%s"		
ySuffix	0x0239	"%s"		
zeroToleranceDisplay	0x0082 0x0041 0x0307	String representing enumeration such that: "0"=displayzero, "1"=blank, "2"=suppresstrailingzero, "3"=omit		
zPrefix	0x0239	"%s"		
zSuffix	0x0239	"%s"		

11.8.3 Tessellation Properties

When dealing with faceted graphical representations (i.e. LODs) of precise models (e.g. B-Rep description in JT), depending on the desired use it is often useful/necessary to know what tessellation tolerances were used to generate the faceted representation. To that end, two properties are typically stored on Part Node Elements (if part also has precise model) to indicate the tessellation tolerances used to generate each LOD. These two tessellation properties are as follows:

Table 94 —Tessellation Property values

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
Chordal	Chordal deviation tessellation tolerance in MCS units for each LOD. Measure of maximum allowable distance a linear approximation for a curve/surface may deviate from the true curve/surface. Encoded value string would look as follows for the case of two LODs: "0.045603 0.069245" 	MbString	space separated F32 values	Numeric
Angular::	Angular tessellation tolerance for each LOD in degrees. Two consecutive segments in a linear approximation of a curve/surface form an angle; this value specifies the maximum angle allowed. Encoded value string would look as follows for the case of two LODs:	MbString	space separated F32 values	Numeric

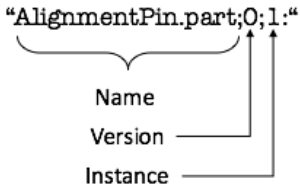
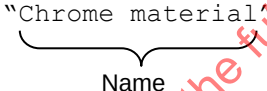
JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
	"30.000000 40.000000" 			
SegLength	The maximum absolute length of (tessellated) line segments in a curve / surface approximation	MbString	space separated F32 values	Numeric

11.8.4 Miscellaneous Properties

The below table documents some miscellaneous properties often placed on various nodes in the LSG to communicate specific information about the node or its contents.

Table 95 — Miscellaneous Property values

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
PMI_TYPE_TABLE	May be attached to <u>Part Node Element</u> to indicate the list of PMI type values and associated names for all PMI types (basically equivalent to the Entity Type field documented in <u>Generic PMI Entities</u>). 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 <u>Shape Node Elements</u> to indicate what geometry type the shape data represents.	MbString	<string>	"Surface" "Wire"
JT_PROP_TRISTRIP_DATA_LAYOUT	This property is deprecated, and is no longer used.			
JT_PROP_ORIGINATING_BREPTYPE	May be attached to <u>Part Node Element</u> to indicate the type of B-Rep associated with the Part.	MbString	<string>	"None" "JtBrep" "XTBrep" "STEP BRep"
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	MbString	<string>	

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
	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.”  For attribute names this string has the following encoded form: “Chrome material” 			

11.9 LSG Attribute Accumulation Semantics

For applications producing or consuming ISO 14306 format data, it is important that the ISO 14306 format semantics of how attributes are meant to be applied and accumulated down the LSG are followed. If not followed, then consistency between the applications in terms of 3D positioning and rendering of LSG model data will not be achieved.

Although each attribute type defines its own application and accumulation LSG semantics (the details of which can be found in each attribute type sub-section under 6.1.2 Attribute Elements), there are some general rules which apply:

Attributes at lower level in the LSG take precedence and replace or accumulate with attributes set at higher levels. When multiple Attributes of the same type are present on a Node, they accumulate in the order they are specified (i.e. from the front of the Attribute list toward the back).

Nodes with no associated attributes inherit those of their parents.

Attributes are inherited only from a node's parents. Thus a given node's attributes do not affect those on the node's siblings.

The root of a partition inherits the attributes in effect at the referring partition node.

Attributes can be marked “final”, which terminates accumulation of that attribute type at that marked attribute and propagates the accumulated value at that point to all descendants of the associated node. Descendants can override a “final” attribute using the “force” flag. Note that “force” does not turn OFF “final” – it is simply a one-shot override of “final” for the specific attribute marked as “forcing.” Multiple attributes of the same type may be marked as “forcing” and in this case, the last one wins. Both of these

flags are OFF by default. An analogy for this “force” and “final” interaction is that “final” is a back-door in the attribute accumulation semantics, and that “force” is the doggy-door in the back-door.

11.10 LSG Part Structure

The ISO 14306 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:

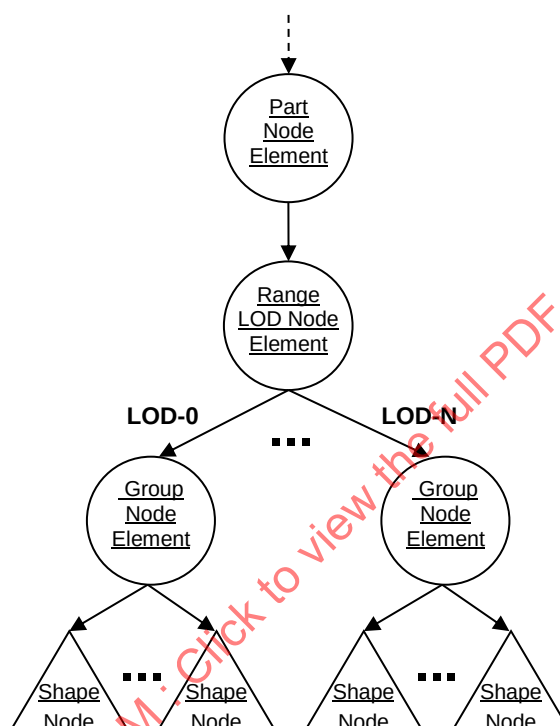


Figure 174 — ISO 14306 format Convention for Modelling each Part in LSG

11.11 Range LOD Node Alternative Rep Selection

Common conventions suggest that LSG traversers apply the following strategy, at Range LOD Nodes (see 6.1.1.8 Range LOD Node Element), when making alternative representation selection decisions based on Range Limits: The first alternate representation is valid when the world coordinate distance between the centre and the eye point is less than or equal to the first range limit (and when no range limits are specified). The second alternate representation is valid when the distance is greater than the first limit and less than or equal to the second limit, and so on. The last alternate representation is valid for all distances greater than the last specified limit.

11.12 Brep Face Group Associations

The original purpose of the face group concept was to provide associativity between Brep faces and geometry. Exactly how a Brep face associates to a face group number is the topic of this section. An implicit scheme has been chosen for face group associativity, rather than storing some kind of explicit data on either the Vertex Shape LOD Data or the Brep. The primary motivation for this implicit scheme is to keep the JT files simple and small; additional association information would not only be redundant, but also wasteful. Tessellators shall exercise this policy when producing Vertex Shape LOD Data from Breps, grouping the triangles into face groups according to its rules. Tristrips may not cross face groups.

Applications shall be able to count on this policy so that, for example, they can map a picking action back to its corresponding Brep face reliably.

STEP BRep: In the case of STEP BRep, the mapping is based on the rank of the faces within the shell. When several shells are defined in a solid (BREP with voids) face index is assigned sequentially by increasing rank of shell within the solid. When several independent solids or shells are present in the STEP segment they are taken par increasing rank within the Shape representation (there is only one shape representation in the STEP segment).

For example: suppose we have two elements in the Shape Representation; the first element is a shell with 2 faces, and the second a Manifold_Solid_Brep with 3 faces, the mapping will be as follows:

```
FG0 ⇔ SH0 F0
FG1 ⇔ SH0 F1
FG2 ⇔ SO0 SH0 F1
FG3 ⇔ SO0 SH0 F2
FG4 ⇔ SO1 SH0 F1
```

JTBrep/ULP: In the case of JtBrep and ULP reps, the mapping is simple. These Reps have a consistent, sequential, index origin-0 numbering scheme for their regions, shells, and faces. So the Brep faces are simply assigned sequentially to face group by increasing region and shell. For example, suppose we have a JTBrep with 2 regions, each with 2 shells, and each with 2 faces. The Face Group ⇔ Region/Shell/Face mapping will be as follows:

```
FG0 ⇔ R0 S0 F0
FG1 ⇔ R0 S0 F1
FG2 ⇔ R0 S1 F0
FG3 ⇔ R0 S1 F1
FG4 ⇔ R1 S0 F0
FG5 ⇔ R1 S0 F1
FG6 ⇔ R1 S1 F0
FG7 ⇔ R1 S1 F1
```

JtXTBrep: In the case of JtXTBrep, the mapping is based on an identifier of each XT face that is persisted on disk. The identifier is unique within each XT body, but it is not an index. XT Brep maintains a zero-based contiguous index of XT face based on increasing identifier value within the same XT body. In the case when multiple bodies are present in JtXTBrep, face index is assigned sequentially by increasing XT body index. For example, suppose we have a JtXTBrep with 2 bodies, each with 2 faces, then the Face Group to Body/Face mapping will be as follows:

```
FG0 ⇔ B0 F0
FG1 ⇔ B0 F1
FG2 ⇔ B1 F0
FG3 ⇔ B1 F1
```

11.13 Handling of different states / variants in a JT file

In order to handle different states of a part, it is necessary to provide methods for activating / deactivating these states and to make them visible for the end user. A typical use case is a semi-finished part, for which it should be possible to manage different states (views). For example, a sheet metal part might be shown in different states before and after deformation ("casting" / "semi-finished" / "final part" or a "sheet metal before" and "after deformation").

A use case from a different domain is to assign different materials to a part for high-end visualization of different designs, i.e. day and night design of the same part.

In general, there are already various possibilities to handle different states. Granularity for layer handling and the reference set is on the part level, which means it is not possible to assign single geometrical bodies but only the full JT Part, including the geometrical bodies and its LOD levels.

11.13.1 Reference Sets

Recommended technology for handling different states is reference sets combined with the subnode definition.

Many CAD systems are able to create a modeled construct whereby a single CAD part can contain a user-defined substructure of geometry and PMI. In CATIA V5, for example, a user-defined substructure can be created using geometrical sets and / or bodies. For the JT world, a CAD part constructed this way is said to have “reference sets” or “reference geometry”. In JT, a CAD part constructed this way is, by convention, referred to as a CAD component.

A JT CAD component is a unique assembly structure, where the top level part in the assembly is the CAD Component and the part(s) that make up the assembly are sub nodes. When this arrangement exists in a JT file, the CAD Component is said to contain Reference Sets. Reference Set definitions are a JT convention made up of assembly nodes, part nodes and properties. The part or assembly nodes defined in the CAD Component assembly are the actual reference set definitions.

CAD Component Assembly Node:

```
Property type="STRING" key="REFSET001" value="REFSET_D=[WF]Curveset001"
Property type="STRING" key="REFSET002" value="REFSET_A=[LW]body_4411,body_4411
Property type="STRING" key="REFSET003" value="REFSET_C=[LW]body_4509,body_4509"
Property type="STRING" key="REFSET004" value="MODEL=[LW]body_4509,[LW]body_4411,"
```

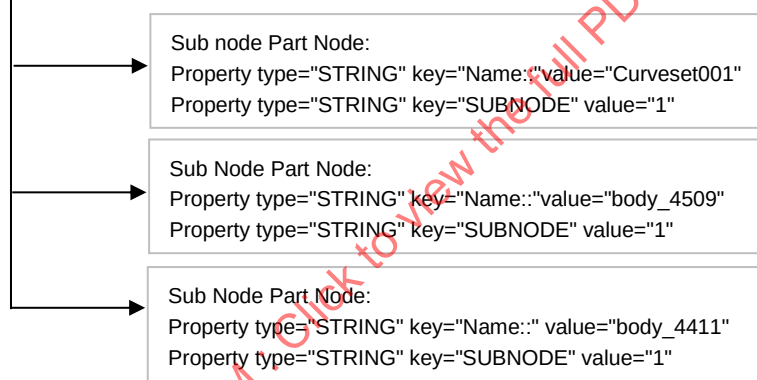


Figure 175 — CAD Component with Reference sets

11.14 Watermarks

11.14.1 Background

JT offers a mechanism to stamp a JT file with a “watermark”, i.e. a text string, TIFF image, or both. Watermarks will be displayed in the foreground of the viewing plane at a fixed position and allow to add confidentiality information and other data that shall always be displayed with the model.

Use cases for watermarks include provision of CAD models to bidders on a contract as JT files including proprietary information, or demonstration models for trade fairs carrying information about the originating company/tool.

Please note that the watermark is just a bitmap / textual information which might be visualized by a JT viewer. It does not protect the data from illicit usage. In JT viewers not supporting watermarks, the JT data will still be visible even without the watermark information.

Another application of watermarks, as discussed in the JT Workflow Forum, is the inclusion of a quality seal / quality check result as watermark into a JT file.



Figure 176 — Watermark Example

11.14.2 Issue – Missing specification in the ISO

Watermarks are supported by the JT file format, but the required data structures are not described in the ISO 14306:2012 JT File Specification.

11.14.3 Recommended Solution

A watermark image can be added to a JT file as a texture. Textures are stored as Attribute Elements in JT. Attribute Elements are placed in the Logical Scene Graph (LSG) as objects associated with nodes. For more information on Attribute Elements, see 5.1.2 Attribute Elements in the ISO JT Edition 1 specification. Use of a JT texture to represent a watermark is implementation dependent. When a texture contains specific properties, an application shall display the texture image on top of any other graphics, effectively displaying it as a watermark.

This implementation requires properties that follow a defined convention. These properties alert the application that the texture information should be displayed in a unique way and at which locations on the display screen the texture should appear. Properties on textures are defined as Property Atom Elements. See section 5.2 Property Atom Elements in the ISO JT Edition 1 specification for full information. These properties are meta-data objects associated with attributes. Each attribute element in an LSG may hold zero or more property atom elements.

Table 96 — Texture watermark properties

Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
WATERMARK / Watermark (*)	Alerts to the application that the texture containing this property is meant to be displayed as a watermark	String	MbString	Watermark
LOCATION / Location (*)	Defines a region on the display for the water mark to appear top left = 1,	Integer	Integer	1-9

	top center = 2, top right = 3, mid left = 4, mid center = 5, mid right = 6, bottom left = 7, bottom center = 8, bottom_right = 9			
Y_DPI	DPI for the output image in Y direction	Float	F32 value	
X_DPI	DPI for the output image in X direction	Float	F32 value	

(*) Property key may appear in upper case or camel style.

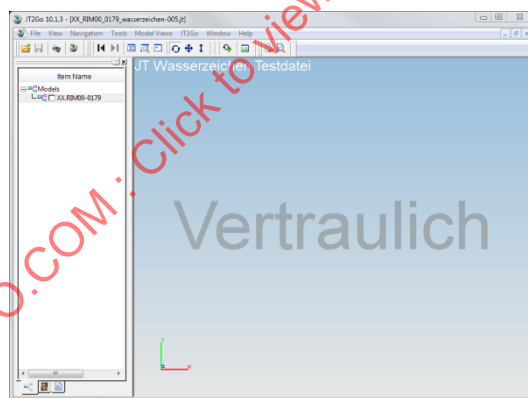


Figure 177 — JT Data with Watermark in JT2GO

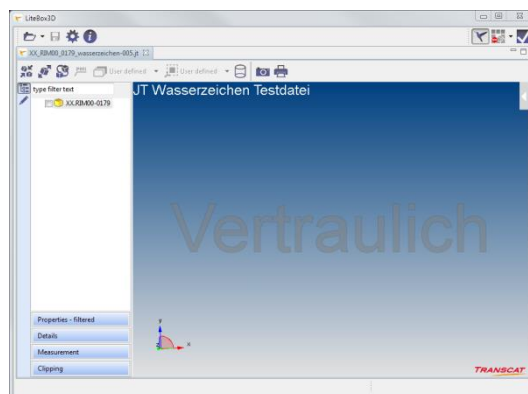


Figure 178 — JT Data with Watermark in LiteBox3D - ISO based JT Viewer

11.14.4 Issue – Protection of data

One point to consider is the fact that the addition of a watermark does not protect the file contents in any way. This also means that in the same way it can be added to a JT file, the watermark can be removed again, even by third-party tools. This could only be prevented by encrypting (part of) the file content, or the application of Digital Rights Management; both approaches have far-reaching consequences on the usability of the data and would require an extension of the file format.

For the time being, this topic is handled as a low-priority issue and no further details are given.

This representation of the CAD Component structure is achieved through the use of the REFSET<XXX> property in the CAD Component part and the SUBNODE property in the parts that contain the reference set definitions.

The reference set property key has the form REFSETXXX with XXX being a three digit incrementing number starting at 001. The numbers shall be assigned concurrently. Up to 999 references set properties can be assigned for a CAD component. A reference set property is a comma-delimited string of part names and hints. These parts can contain geometry such as solid bodies, wireframe, or points that represent content relevant to the owning CAD component. To facilitate identification of the reference set content, hint strings can be combined with the part names. There are four reserved hint strings for reference sets; PMI, PT (point cloud), WF (wireframe) and LW (light weight, facet only). These hint strings provide information about the content in the JT Part, so the application does not need to parse the content first. In case no hint string is provided, the JT Part is of general content (e.g. containing B-Rep data).

Reference set encoding conventions

- Reference set names are not case-sensitive
- The characters = , and \ need to be escaped within the values of the property keys if they are in a part name or reference set name

The REFSET_CURRENT property is defined in instance nodes that are instances of a CAD Component. The property value is the name of a reference set that exists in the instanced CAD Component. Best practice is to use this property to determine the reference set that will become active if the model is set to the “as saved” state.

12 Conformance requirements

Conformance to ISO 14306 includes satisfying the requirements stated in this document and the relevant requirements of the normative references.

A 14306 file shall contain at least all of the following segments:

- the logical scene graph segment (LSG);
- one shape LOD segment.

An implementation shall support at least all the following segments:

- the logical scene graph segment (LSG);
- one shape LOD segment.

On top of these mandatory segments, there is a list of optional segments available to provide complementary capabilities in the 3D Light visualization scope: See Table 6 — Segment Types

The implementers shall not implement deprecated segments in write mode.

It is not recommended to use JT Brep for the creation of new ISO 14306 files.

The use of the STEP segment is recommended for B-Rep.

Annex A (informative)

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 97 — Object Type Identifiers lists the assigned identifier for each Object Type that can exist in a Version 9.5 JT file.

Table 97 — Object Type Identifiers

GUID	Object Type
0xffffffff, 0xffff, 0xffff, 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	<u>Base Node Element</u>
0x10dd101b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Group Node Element</u>
0x10dd102a, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Instance Node Element</u>
0x10dd102c, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>LOD Node Element</u>
0xce357245, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	<u>Meta Data Node Element</u>
0xd239e7b6, 0xdd77, 0x4289, 0xa0, 0x7d, 0xb0, 0xee, 0x79, 0xf7, 0x94, 0x94	<u>NULL Shape Node Element</u>
0xce357244, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	<u>Part Node Element</u>
0x10dd103e, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Partition Node Element</u>
0x10dd104c, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Range LOD Node Element</u>
0x10dd10f3, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Switch Node Element</u>
0x10dd1059, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Base Shape Node Element</u>
0x98134716, 0x0010, 0x0818, 0x19, 0x98, 0x08, 0x00, 0x09, 0x83, 0x5d, 0x5a	<u>Point Set Shape Node Element</u>
0x10dd1048, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Polygon Set Shape Node Element</u>

0x10dd1046, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Polyline Set Shape Node Element</u>
0xe40373c1, 0x1ad9, 0x11d3, 0x9d, 0xaf, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2	<u>Primitive Set Shape Node Element</u>
0x10dd1077, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Tri-Strip Set Shape Node Element</u>
0x10dd107f, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Vertex Shape Node Element</u>
0x10dd1001, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Base Attribute Data</u>
0x10dd1014, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Draw Style Attribute Element</u>
0x10dd1083, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Geometric Transform Attribute Element</u>
0x10dd1028, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Infinite Light Attribute Element</u>
0x10dd1096, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Light Set Attribute Element</u>
0x10dd10c4, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Linestyle Attribute Element</u>
0x10dd1030, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Material Attribute Element</u>
0x10dd1045, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Point Light Attribute Element</u>
0x8d57c010, 0xe5cb, 0x11d4, 0x84, 0xe, 0x00, 0xa0, 0xd2, 0x18, 0x2f, 0x9d	<u>Pointstyle Attribute Element</u>
0xaa1b831d, 0x6e47, 0x4fee, 0xa8, 0x65, 0xcd, 0x7e, 0x1f, 0x2f, 0x39, 0xdb	<u>Shader Effects Attribute Element</u>
0x10dd1073, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Texture Image Attribute Element</u>
0xaa1b831d, 0x6e47, 0x4fee, 0xa8, 0x65, 0xcd, 0x7e, 0x1f, 0x2f, 0x39, 0xdc	<u>Texture Coordinate Generator Attribute Element</u>
0xa3cfb921, 0xbdeb, 0x48d7, 0xb3, 0x96, 0x8b, 0x8d, 0xe, 0xf4, 0x85, 0xa0	<u>Mapping Plane Element</u>
0x3e70739d, 0x8cb0, 0x41ef, 0x84, 0x5c, 0xa1, 0x98, 0xd4, 0x0, 0x3b, 0x3f	<u>Mapping Cylinder Element</u>
0x72475fd1, 0x2823, 0x4219, 0xa0, 0x6c, 0xd9, 0xe6, 0xe3, 0x9a, 0x45, 0xc1	<u>Mapping Sphere Element</u>
0x92f5b094, 0x6499, 0x4d2d, 0x92, 0xaa, 0x60, 0xd0, 0x5a, 0x44, 0x32, 0xcf	<u>Mapping TriPlanar Element</u>
0x10dd104b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Base Property Atom Element</u>
0xce357246, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	<u>Date Property Atom Element</u>
0x10dd102b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Integer Property Atom Element</u>

0x10dd1019, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Floating Point Property Atom Element</u>
0xe0b05be5, 0xfbbd, 0x11d1, 0xa3, 0xa7, 0x00, 0xaa, 0x00, 0xd1, 0x09, 0x54	<u>Late Loaded Property Atom Element</u>
0x10dd1004, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>JT Object Reference Property Atom Element</u>
0x10dd106e, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>String Property Atom Element</u>
Types Stored Within JT B-Rep Segment (Segment Type = 2)	
0x873a70c0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>JT B-Rep Element</u>
Types Stored Within Meta Data Segment (Segment Type = 4)	
0xce357249, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	<u>PMI Manager Meta Data Element</u>
0xce357247, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	<u>Property Proxy Meta Data Element</u>
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	<u>Null Shape LOD Element</u>
0x98134716, 0x0011, 0x0818, 0x19, 0x98, 0x08, 0x00, 0x09, 0x83, 0x5d, 0x5a	<u>Point Set Shape LOD Element</u>
0x10dd10a1, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Polyline Set Shape LOD Element</u>
0xe40373c2, 0x1ad9, 0x11d3, 0x9d, 0xaf, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2	<u>Primitive Set Shape Element</u>
0x10dd10ab, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Tri-Strip Set Shape LOD Element</u>
0x10dd10b0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Vertex Shape LOD Element</u>
Types Stored Within XT B-Rep Segment (Segment Type = 17)	
0x873a70e0, 0x2ac9, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>XT B-Rep Element</u>
Types Stored Within Wireframe Segment (Segment Type = 18)	
0x873a70d0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	<u>Wireframe Rep Element</u>
Types Stored Within JT ULP Segment (Segment Type = 20)	
0xf338a4af, 0xd7d2, 0x41c5, 0xbc, 0xf2, 0xc5, 0x5a, 0x88, 0xb2, 0x1e, 0x73	<u>JT ULP Element</u>
Types Stored Within JT LWPA Segment (Segment Type = 24)	

0xd67f8ea8, 0xf524, 0x4879, 0x92, 0x8c, 0x4c, 0x3a, 0x56, 0x1f, 0xb9, 0x3a	<u>JT LWPA Element</u>
Type Stored Within STEP-B-Rep Segment (Segment Type = 32)	
0x869c7d53, 0xccb0, 0x451b, 0xb2, 0x3, 0xd1, 0x42, 0x81, 0x56, 0x14, 0x56	<u>STEP B-Rep Element</u>

STANDARDSISO.COM : Click to view the full PDF of ISO 14306:2017

Annex B (normative)

STEP B-Rep

STEP B-Rep Segment contains an Element that defines the precise geometric Boundary Representation data for a particular Part in STEP boundary representation format (ISO 10303-42).

STEP B-Rep Segments are typically referenced by Part Node Elements (see Part Node Element) using Late Loaded Property Atom Elements (see Late Loaded Property Atom Element). The STEP B-Rep Segment type supports compression methods described in on all element data. The main component of the STEP B-Rep segment is an ISO 10303-21 file with the compression options as defined by the compression flag in 5.1.3.2 Data for the data segments.

B.1 STEP B-Rep Element

Object Type ID: 0x869c7d53, 0xccb0, 0x451b, 0xb2, 0x3, 0xd1, 0x42, 0x81, 0x56, 0x14, 0x56

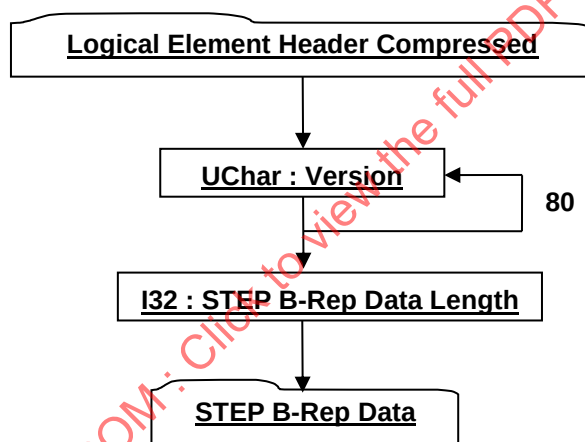


Figure 179 — STEP B-Rep Element data collection

UChar: Version

An 80-character version string defining the version of the file format used to write this file. The Version string is based on the ISO 10303-21 schema_name format:

EXAMPLE {1 0 10303 243 1 1 1}'

I32: STEP B-Rep Data Length

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

B.1.1 STEP B-Rep Data

The STEP B –Rep collection specifies the raw stream of ASCII characters used to represent a Part's STEP B- Rep Body(s) in ISO 10303-21 format based on the iso_14306_schema schema.

The detailed description of all entities is available in ISO 10303-42:2014, Industrial automation systems and integration -- Product data representation and exchange -- Part 42: Integrated generic resource: Geometric and topological representation.

B.1.2 STEP schema

The iso_14306_schema schema describes the data model of the STEP B-Rep for ISO 14306. The complete schema is available in Annex C.

B.1.3 STEP format

The file format shall be encoded according to the syntax and EXPRESS language mapping defined in ISO 10303-21 and to the MIM schema specified in Annex C Computer interpretable listing.

The FILE_SCHEMA element of the header shall specify the name of the EXPRESS schema used and include its object information identifier.

EXAMPLE The instance below identifies the iso_14306_schema extended schema.

```
FILE_SCHEMA (('ISO_14306_SCHEMA {1 0 10303 243 1 1 1}))
```

B.1.4 STEP external element reference in ISO 10303-21

ISO 10303-21 ed3 provide anchor and reference sections to allow entities and values to be defined in external files.

Anchor section structure

The anchor section defines external names for instances in the exchange structure so that they can be referenced.

The anchor section is optional. If an anchor section is included in the exchange structure then it shall be given after the header section and before any reference, data or signature section. The section shall begin with the special token "ANCHOR;" and shall terminate with the special token "ENDSEC;".

Each entry in the anchor section shall define one external name. The anchor name shall not be used for any other anchor in the same exchange structure. The anchor name shall meet the requirements defined in clause 6.5.4.

EXAMPLE

Anchor section structure	Comment
ANCHOR;	
.	
<first> = #10;	/* Anchor defined by an entity instance name */
<second> = @20;	/* Anchor defined by a value instance name */
<third> = 30;	/* Anchor defined by a simple data type */
<fourth> = \$;	/* Anchor defined by a null_value */
<fifth> = (1.1, 2.1, 3.1);	/* Anchor defined by a list */
<sixth> = <picture.jpg>;	/* Anchor defined by a resource */
<seventh> = #INCH;	/* Anchor defined by an EXPRESS constant */
<eighth> = #10 {ratio:196.73};	/* Anchor with a tag */
.	
ENDSEC;	

Detail specifications and descriptions of the anchor section are available in ISO 10303-21 ed3

Reference section structure

The reference section is optional. If a reference section is included in the exchange structure then it shall be given after the header section and any anchor section and before any data section or signature section. The section shall begin with the special token "REFERENCE;" and shall terminate with the special token "ENDSEC;"

Each entry in the reference section shall associate an LHS_OCCURRENCE_NAME with a RESOURCE. If the occurrence name is an ENTITY_INSTANCE_NAME then the resource shall identify an entity. If the occurrence name is a VALUE_INSTANCE_NAME then the resource shall identify a value.

Each occurrence name shall have one association only and it shall not be defined in the data sections of the exchange structure.

EXAMPLE

Reference section structure	Comment
REFERENCE;	
.	
#10 = <http://www.giant.com/product.stp#shape>;	/* Reference to a URI with fragment identifier */
#20 = <building_698.ifc>;	/* Reference to a URI without fragment identifier */
#30 = <#wheel>;	/* Reference to a fragment identifier defined in same file */
#40 = <http://www.giant.com/product.jt#shape>;	/* Reference to a file in another format */
#50 = <http://www.giant.com/product.stp#entity>;	/* Reference to an entity instance */
@60 = <http://www.giant.com/product.stp#value>;	/* Reference to a value */
#50 = <http://www.giant.com/product.stp#100>;	/* Reference using entity instance name */
@50 = <http://www.giant.com/product.stp#100>;	/* Reference using value instance name */
.	
ENDSEC;	

Detail specifications and descriptions of the reference section are available in ISO 10303-21 ed3

Annex C (normative)

STEP schema

Reference link to the EXPRESS schema file on the ISO website:
http://standards.iso.org/iso/14306/ed-2/en/iso_14306_schema.exp

Reference parts:

ISO 10303-41:2014, Industrial automation systems and integration -- Product data representation and exchange -- Part 41: Integrated generic resource: Fundamentals of product description and support.

ISO 10303-42:2014, Industrial automation systems and integration -- Product data representation and exchange -- Part 42: Integrated generic resource: geometric and topological representation.

ISO 10303-43:2011, Industrial automation systems and integration -- Product data representation and exchange -- Part 43: Integrated generic resource: Representation structures.

ISO 10303-44:2014, Industrial automation systems and integration -- Product data representation and exchange -- Part 44: Integrated generic resource: Product structure configuration.

```
(*
$Id: iso_14306_schema,v XXXXXXXXXXXXXXXX Exp $
ISO 114306 TC184/SC4/WG12 N8570
```

EXPRESS Source:

ISO 14306 CD2 JT file format specification for 3D visualisation

The following permission notice and disclaimer shall be included in all copies of this EXPRESS schema ("the Schema"), and derivations of the Schema:

Copyright ISO 2015 All rights reserved
 Permission is hereby granted, free of charge in perpetuity, to any person obtaining a copy of the Schema, to use, copy, modify, merge and distribute free of charge, copies of the Schema for the purposes of developing, implementing, installing and using software based on the Schema, and to permit persons to whom the Schema is furnished to do so, subject to the following conditions:

THE SCHEMA IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SCHEMA OR THE USE OR OTHER DEALINGS IN THE SCHEMA.

In addition, any modified copy of the Schema shall include the following notice:

THIS SCHEMA HAS BEEN MODIFIED FROM THE SCHEMA DEFINED IN
 ISO 14306 CD2 JT file format specification for 3D visualisation
 AND SHOULD NOT BE INTERPRETED AS COMPLYING WITH THAT DOCUMENT
 *)

SCHEMA iso_14306_schema;

CONSTANT

```

dummy_gri : geometric_representation_item :=
representation_item('') || geometric_representation_item();
dummy_tri : topological_representation_item :=
representation_item('') || topological_representation_item();

END_CONSTANT;

TYPE absorbed_dose_measure = REAL;
END_TYPE; -- absorbed_dose_measure

TYPE acceleration_measure = REAL;
END_TYPE; -- acceleration_measure

TYPE amount_of_substance_measure = REAL;
END_TYPE; -- amount_of_substance_measure

TYPE area_measure = REAL;
END_TYPE; -- area_measure

TYPE axis2_placement = SELECT (
    axis2_placement_2d,
    axis2_placement_3d);
END_TYPE; -- axis2_placement

TYPE b_spline_curve_form = ENUMERATION OF
( polyline_form,
  circular_arc,
  elliptic_arc,
  parabolic_arc,
  hyperbolic_arc,
  unspecified );
END_TYPE; -- b_spline_curve_form

TYPE b_spline_surface_form = ENUMERATION OF
( plane_surf,
  cylindrical_surf,
  conical_surf,
  spherical_surf,
  toroidal_surf,
  surf_of_revolution,
  ruled_surf,
  generalised_cone,
  quadric_surf,
  surf_of_linear_extrusion,
  unspecified );
END_TYPE; -- b_spline_surface_form

TYPE capacitance_measure = REAL;
END_TYPE; -- capacitance_measure

TYPE celsius_temperature_measure = REAL;
END_TYPE; -- celsius_temperature_measure

TYPE chained_representation_link = SELECT (
    mapped_item,
    representation_context,
    representation_relationship);
END_TYPE; -- chained_representation_link

TYPE characterized_definition = SELECT (
    characterized_object,
    characterized_product_definition,
    shape_definition);
END_TYPE; -- characterized_definition

TYPE characterized_product_definition = SELECT (
    product_definition,

```



```

    product_definition_relationship);
END_TYPE; -- characterized_product_definition

TYPE conductance_measure = REAL;
END_TYPE; -- conductance_measure

TYPE context_dependent_measure = REAL;
END_TYPE; -- context_dependent_measure

TYPE count_measure = NUMBER;
END_TYPE; -- count_measure

TYPE curve_on_surface = SELECT (
    composite_curve_on_surface,
    pcurve,
    surface_curve);
END_TYPE; -- curve_on_surface

TYPE derived_property_select = SELECT (
    property_definition);
END_TYPE; -- derived_property_select

TYPE description_attribute_select = SELECT (
    application_context,
    external_source,
    property_definition_representation,
    representation);
END_TYPE; -- description_attribute_select

TYPE descriptive_measure = STRING;
END_TYPE; -- descriptive_measure

TYPE dimension_count = INTEGER;
WHERE
    WR1 : SELF > 0;
END_TYPE; -- dimension_count

TYPE dose_equivalent_measure = REAL;
END_TYPE; -- dose_equivalent_measure

TYPE electric_charge_measure = REAL;
END_TYPE; -- electric_charge_measure

TYPE electric_current_measure = REAL;
END_TYPE; -- electric_current_measure

TYPE electric_potential_measure = REAL;
END_TYPE; -- electric_potential_measure

TYPE energy_measure = REAL;
END_TYPE; -- energy_measure

TYPE external_identification_item = SELECT (
    document_file,
    generic_product_definition_reference,
    trimmed_curve);
END_TYPE; -- external_identification_item

TYPE force_measure = REAL;
END_TYPE; -- force_measure

TYPE founded_item_select = SELECT (
    founded_item,
    representation_item);
END_TYPE; -- founded_item_select

TYPE frequency_measure = REAL;

```

```

END_TYPE; -- frequency_measure

TYPE geometric_item_specific_usage_select = SELECT (
    shape_aspect,
    shape_aspect_relationship);
END_TYPE; -- geometric_item_specific_usage_select

TYPE geometric_model_item = SELECT (
    geometric_representation_item);
END_TYPE; -- geometric_model_item

TYPE groupable_item = SELECT (
    geometric_representation_item,
    mapped_item,
    shape_representation_relationship,
    topological_representation_item);
WHERE
    WR1 : NOT ('ISO_14306_SCHEMA.GROUP' IN TYPEOF(SELF));
END_TYPE; -- groupable_item

TYPE identification_item = SELECT (
    document_file,
    representation,
    shape_representation);
END_TYPE; -- identification_item

TYPE identifier = STRING;
END_TYPE; -- identifier

TYPE id_attribute_select = SELECT (
    application_context,
    group,
    property_definition,
    representation,
    shape_aspect,
    shape_aspect_relationship);
END_TYPE; -- id_attribute_select

TYPE illuminance_measure = REAL;
END_TYPE; -- illuminance_measure

TYPE inductance_measure = REAL;
END_TYPE; -- inductance_measure

TYPE item_identified_representation_usage_definition = SELECT (
    represented_definition);
END_TYPE; -- item_identified_representation_usage_definition

TYPE item_identified_representation_usage_select = SELECT (
    representation_item);
END_TYPE; -- item_identified_representation_usage_select

TYPE knot_type = ENUMERATION OF
    ( uniform_knots,
      quasi_uniform_knots,
      piecewise_bezier_knots,
      unspecified );
END_TYPE; -- knot_type

TYPE label = STRING;
END_TYPE; -- label

TYPE length_measure = REAL;
END_TYPE; -- length_measure

TYPE list_of_reversible_topology_item = LIST [0:?] OF reversible_topology_item;
END_TYPE; -- list_of_reversible_topology_item

```

```
TYPE luminous_flux_measure = REAL;
END_TYPE; -- luminous_flux_measure
```

```
TYPE luminous_intensity_measure = REAL;
END_TYPE; -- luminous_intensity_measure
```

```
TYPE magnetic_flux_density_measure = REAL;
END_TYPE; -- magnetic_flux_density_measure
```

```
TYPE magnetic_flux_measure = REAL;
END_TYPE; -- magnetic_flux_measure
```

```
TYPE mass_measure = REAL;
END_TYPE; -- mass_measure
```

```
TYPE measure_value = SELECT (
  absorbed_dose_measure,
  dose_equivalent_measure,
  radioactivity_measure,
  acceleration_measure,
  amount_of_substance_measure,
  area_measure,
  celsius_temperature_measure,
  context_dependent_measure,
  count_measure,
  descriptive_measure,
  capacitance_measure,
  electric_charge_measure,
  conductance_measure,
  electric_current_measure,
  electric_potential_measure,
  energy_measure,
  magnetic_flux_density_measure,
  force_measure,
  frequency_measure,
  illuminance_measure,
  inductance_measure,
  length_measure,
  luminous_flux_measure,
  luminous_intensity_measure,
  magnetic_flux_measure,
  mass_measure,
  numeric_measure,
  non_negative_length_measure,
  parameter_value,
  plane_angle_measure,
  positive_length_measure,
  positive_plane_angle_measure,
  positive_ratio_measure,
  power_measure,
  pressure_measure,
  ratio_measure,
  resistance_measure,
  solid_angle_measure,
  thermodynamic_temperature_measure,
  time_measure,
  velocity_measure,
  volume_measure);
END_TYPE; -- measure_value
```

```
TYPE mechanical_design_geometric_presentation_representation_items = SELECT (
  axis2_placement,
  mapped_item);
END_TYPE; -- mechanical_design_geometric_presentation_representation_items
```

```
TYPE name_attribute_select = SELECT (
```

```

    derived_unit,
    product_definition,
    property_definition_representation);
END_TYPE; -- name_attribute_select

TYPE non_negative_length_measure = length_measure;
WHERE
    WR1 : SELF >= 0.0;
END_TYPE; -- non_negative_length_measure

TYPE numeric_measure = NUMBER;
END_TYPE; -- numeric_measure

TYPE parameter_value = REAL;
END_TYPE; -- parameter_value

TYPE pcurve_or_surface = SELECT (
    pcurve,
    surface);
END_TYPE; -- pcurve_or_surface

TYPE plane_angle_measure = REAL;
END_TYPE; -- plane_angle_measure

TYPE positive_length_measure = non_negative_length_measure;
WHERE
    WR1 : SELF > 0.0;
END_TYPE; -- positive_length_measure

TYPE positive_plane_angle_measure = plane_angle_measure;
WHERE
    WR1 : SELF > 0.0;
END_TYPE; -- positive_plane_angle_measure

TYPE positive_ratio_measure = ratio_measure;
WHERE
    WR1 : SELF > 0.0;
END_TYPE; -- positive_ratio_measure

TYPE power_measure = REAL;
END_TYPE; -- power_measure

TYPE preferred_surface_curve_representation = ENUMERATION OF
    ( curve_3d,
      pcurve_s1,
      pcurve_s2 );
END_TYPE; -- preferred_surface_curve_representation

TYPE pressure_measure = REAL;
END_TYPE; -- pressure_measure

TYPE product_definition_or_reference = SELECT (
    product_definition,
    generic_product_definition_reference);
END_TYPE; -- product_definition_or_reference

TYPE radioactivity_measure = REAL;
END_TYPE; -- radioactivity_measure

TYPE ratio_measure = REAL;
END_TYPE; -- ratio_measure

TYPE represented_definition = SELECT (
    general_property,
    property_definition,
    shape_aspect,
    shape_aspect_relationship);

```

```

END_TYPE; -- represented_definition

TYPE resistance_measure = REAL;
END_TYPE; -- resistance_measure

TYPE reversible_topology = SELECT (
    reversible_topology_item,
    list_of_reversible_topology_item,
    set_of_reversible_topology_item);
END_TYPE; -- reversible_topology

TYPE reversible_topology_item = SELECT (
    edge,
    path,
    face,
    face_bound,
    closed_shell,
    open_shell);
END_TYPE; -- reversible_topology_item

TYPE role_select = SELECT (
    group_assignment);
END_TYPE; -- role_select

TYPE set_of_reversible_topology_item = SET [0:?] OF reversible_topology_item;
END_TYPE; -- set_of_reversible_topology_item

TYPE shape_definition = SELECT (
    product_definition_shape,
    shape_aspect,
    shape_aspect_relationship);
END_TYPE; -- shape_definition

TYPE shape_model = SELECT (
    shape_representation);
END_TYPE; -- shape_model

TYPE shell = SELECT (
    open_shell,
    closed_shell);
END_TYPE; -- shell

TYPE si_prefix = ENUMERATION OF
(
    exa,
    peta,
    tera,
    giga,
    mega,
    kilo,
    hecto,
    deca,
    deci,
    centi,
    milli,
    micro,
    nano,
    pico,
    femto,
    atto );
END_TYPE; -- si_prefix

TYPE si_unit_name = ENUMERATION OF
(
    metre,
    gram,
    second,
    ampere,
    kelvin,

```

```

    mole,
    candela,
    radian,
    steradian,
    hertz,
    newton,
    pascal,
    joule,
    watt,
    coulomb,
    volt,
    farad,
    ohm,
    siemens,
    weber,
    tesla,
    henry,
    degree_celsius,
    lumen,
    lux,
    becquerel,
    gray,
    sievert );
END_TYPE; -- si_unit_name

TYPE solid_angle_measure = REAL;
END_TYPE; -- solid_angle_measure

TYPE source_item = SELECT (
    identifier);
END_TYPE; -- source_item

TYPE text = STRING;
END_TYPE; -- text

TYPE thermodynamic_temperature_measure = REAL;
END_TYPE; -- thermodynamic_temperature_measure

TYPE time_measure = REAL;
END_TYPE; -- time_measure

TYPE transition_code = ENUMERATION OF
    ( discontinuous,
      continuous,
      cont_same_gradient,
      cont_same_gradient_same_curvature );
END_TYPE; -- transition_code

TYPE trimming_preference = ENUMERATION OF
    ( cartesian,
      parameter,
      unspecified );
END_TYPE; -- trimming_preference

TYPE trimming_select = SELECT (
    cartesian_point,
    parameter_value);
END_TYPE; -- trimming_select

TYPE unit = SELECT (
    derived_unit,
    named_unit);
END_TYPE; -- unit

TYPE vector_or_direction = SELECT (
    direction,
    vector);

```

```

END_TYPE; -- vector_or_direction

TYPE velocity_measure = REAL;
END_TYPE; -- velocity_measure

TYPE volume_measure = REAL;
END_TYPE; -- volume_measure

ENTITY absorbed_dose_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.ABSORBED_DOSE_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- absorbed_dose_measure_with_unit

ENTITY absorbed_dose_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.gray);
END_ENTITY; -- absorbed_dose_unit

ENTITY acceleration_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.ACCELERATION_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- acceleration_measure_with_unit

ENTITY acceleration_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensional_exponents(1.0, 0.0, -2.0, 0.0, 0.0, 0.0, 0.0);
END_ENTITY; -- acceleration_unit

ENTITY advanced_brep_shape_representation
    SUBTYPE OF (shape_representation);
    WHERE
        WR1 : SIZEOF(QUERY(it <* SELF.items |
            NOT (SIZEOF(['ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP',
                'ISO_14306_SCHEMA.FACETED_BREP',
                'ISO_14306_SCHEMA.MAPPED_ITEM',
                'ISO_14306_SCHEMA.AXIS2_PLACEMENT_3D'] *
                TYPEOF(it)) =
                1))) =
            0;
        WR2 : SIZEOF(QUERY(it <* SELF.items |
            (SIZEOF(['ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP',
                'ISO_14306_SCHEMA.MAPPED_ITEM'] *
                TYPEOF(it)) =
                1))) >
            0;
        WR3 : SIZEOF(QUERY(msb <*
            QUERY(it <* SELF.items |
                ('ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
                TYPEOF(it))) |
            NOT (SIZEOF(QUERY(csh <* msb_shells(msb) |
                NOT (SIZEOF(QUERY(fcs <*
                    csh\connected_face_set.cfs_faces |
                    NOT
                    ('ISO_14306_SCHEMA.ADVANCED_FACE' IN
                    TYPEOF(fcs)))) =
                    0))) =
                    0))) =
            0;

```

```

WR4 : SIZEOF(QUERY(msb <*
    QUERY(it <* items |
        ('ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
        TYPEOF(it))) |
    ('ISO_14306_SCHEMA.ORIENTED_CLOSED_SHELL' IN
    TYPEOF(msb\manifold_solid_brep.outer)))) =
0;
WR5 : SIZEOF(QUERY(brv <*
    QUERY(it <* items |
        ('ISO_14306_SCHEMA.BREP_WITH_VOIDS' IN
        TYPEOF(it))) |
    NOT (SIZEOF(QUERY(csh <* brv\brep_with_voids.voids |
        csh\oriented_closed_shell.orientation))) =
    0))) =
0;
WR6 : SIZEOF(QUERY(mi <*
    QUERY(it <* items |
        ('ISO_14306_SCHEMA.MAPPED_ITEM' IN
        TYPEOF(it))) |
    NOT ('ISO_14306_SCHEMA.ADVANCED_BREP_SHAPE_REPRESENTATION' IN
    TYPEOF(mi\mapped_item.mapping_source.mapped_representation)))) =
0;
END_ENTITY; -- advanced_brep_shape_representation

ENTITY advanced_face
    SUBTYPE OF (face_surface);
WHERE
    WR1 : SIZEOF(['ISO_14306_SCHEMA.ELEMENTARY_SURFACE',
        'ISO_14306_SCHEMA.B_SPLINE_SURFACE',
        'ISO_14306_SCHEMA.SWEPT_SURFACE'] *
        TYPEOF(face_geometry)) =
        1;
    WR2 : SIZEOF(QUERY(elp_fbnds <*
        QUERY(bnds <* bounds |
            ('ISO_14306_SCHEMA.EDGE_LOOP' IN
            TYPEOF(bnds.bound))) |
        NOT (SIZEOF(QUERY(oe <* elp_fbnds.bound\path.edge_list |
            NOT ('ISO_14306_SCHEMA.EDGE_CURVE' IN
            TYPEOF(oe\oriented_edge.edge_element)))) =
            0))) =
        0;
    WR3 : SIZEOF(QUERY(elp_fbnds <*
        QUERY(bnds <* bounds |
            ('ISO_14306_SCHEMA.EDGE_LOOP' IN
            TYPEOF(bnds.bound))) |
        NOT (SIZEOF(QUERY(oe <* elp_fbnds.bound\path.edge_list |
            NOT (SIZEOF(['ISO_14306_SCHEMA.LINE',
                'ISO_14306_SCHEMA.CONIC',
                'ISO_14306_SCHEMA.POLYLINE',
                'ISO_14306_SCHEMA.SURFACE_CURVE',
                'ISO_14306_SCHEMA.B_SPLINE_CURVE'] *
                TYPEOF(oe.edge_element\edge_curve.edge_geometry)) =
                1)))) =
            0))) =
        0;
    WR4 : SIZEOF(QUERY(elp_fbnds <*
        QUERY(bnds <* bounds |
            ('ISO_14306_SCHEMA.EDGE_LOOP' IN
            TYPEOF(bnds.bound))) |
        NOT (SIZEOF(QUERY(oe <* elp_fbnds.bound\path.edge_list |
            NOT (('ISO_14306_SCHEMA.VERTEX_POINT' IN
            TYPEOF(oe\edge.edge_start)) AND

```



```

('ISO_14306_SCHEMA.CARTESIAN_POINT'
IN
TYPEOF(oe\edge.edge_start\vertex_point.vertex_geometry))) AND
('ISO_14306_SCHEMA.VERTEX_POINT' IN
TYPEOF(oe\edge.edge_end)) AND
('ISO_14306_SCHEMA.CARTESIAN_POINT'
IN
TYPEOF(oe\edge.edge_end\vertex_point.vertex_geometry)))))) =
0))) =
0;
WR5 : SIZEOF(QUERY(elp_fbnds <*
QUERY(bnds <* bounds |
('ISO_14306_SCHEMA.EDGE_LOOP' IN
TYPEOF(bnds.bound))) |
('ISO_14306_SCHEMA.ORIENTED_PATH' IN
TYPEOF(elp_fbnds.bound)))) =
0;
WR6 : NOT ('ISO_14306_SCHEMA.SWEPT_SURFACE' IN
TYPEOF(face_geometry)) OR
(SIZEOF(['ISO_14306_SCHEMA.LINE',
'ISO_14306_SCHEMA.CONIC',
'ISO_14306_SCHEMA.POLYLINE',
'ISO_14306_SCHEMA.B_SPLINE_CURVE'] *
TYPEOF(face_geometry\swept_surface.swept_curve)) =
1);
WR7 : SIZEOF(QUERY(vlp_fbnds <*
QUERY(bnds <* bounds |
('ISO_14306_SCHEMA.VERTEX_LOOP' IN
TYPEOF(bnds.bound))) |
NOT (('ISO_14306_SCHEMA.VERTEX_POINT' IN
TYPEOF(vlp_fbnds\face_bound.bound\vertex_loop.loop_vertex)) AND
('ISO_14306_SCHEMA.CARTESIAN_POINT' IN
TYPEOF(vlp_fbnds\face_bound.bound\vertex_loop.loop_vertex\vertex_point.vertex_geome
try)))))) =
0;
WR8 : SIZEOF(QUERY(bnd <* bounds |
NOT (SIZEOF(['ISO_14306_SCHEMA.EDGE_LOOP',
'ISO_14306_SCHEMA.VERTEX_LOOP'] *
TYPEOF(bnd.bound)) =
1))) =
0;
WR9 : SIZEOF(QUERY(elp_fbnds <*
QUERY(bnds <* bounds |
('ISO_14306_SCHEMA.EDGE_LOOP' IN
TYPEOF(bnds.bound))) |
NOT (SIZEOF(QUERY(oe <* elp_fbnds.bound\path.edge_list |
('ISO_14306_SCHEMA.SURFACE_CURVE' IN
TYPEOF(oe\oriented_edge.edge_element\edge_curve.edge_geometry)) AND
NOT (SIZEOF(QUERY(sc_ag <*
oe.edge_element\edge_curve.edge_geometry\surface_curve.associated_geometry |
NOT
('ISO_14306_SCHEMA.PCURVE' IN
TYPEOF(sc_ag)))))) =
0))) =
0;
WR10 : (NOT ('ISO_14306_SCHEMA.SWEPT_SURFACE' IN
TYPEOF(face_geometry)) OR
(NOT ('ISO_14306_SCHEMA.POLYLINE' IN
TYPEOF(face_geometry\swept_surface.swept_curve)) OR

```

```

(SIZEOF(face_geometry\swept_surface.swept_curve\polyline.points) >=
  3))) AND
  (SIZEOF(QUERY(elp_fbnds <*
    QUERY(bnds <* bounds |
      ('ISO_14306_SCHEMA.EDGE_LOOP' IN
        TYPEOF(bnds.bound))) |
    NOT (SIZEOF(QUERY(oe <*
      elp_fbnds.bound\path.edge_list |
        ('ISO_14306_SCHEMA.POLYLINE' IN
          TYPEOF(oe\oriented_edge.edge_element\edge_curve.edge_geometry)) AND
            NOT
              (SIZEOF(oe\oriented_edge.edge_element\edge_curve.edge_geometry\polyline.points) >=
                3))) =
                0))) =
  0);
END_ENTITY; -- advanced_face

ENTITY amount_of_substance_measure_with_unit
  SUBTYPE OF (measure_with_unit);
WHERE
  WR1 : 'ISO_14306_SCHEMA.AMOUNT_OF_SUBSTANCE_UNIT' IN
    TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- amount_of_substance_measure_with_unit

ENTITY amount_of_substance_unit
  SUBTYPE OF (named_unit);
WHERE
  WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
    (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
    (SELF\named_unit.dimensions.time_exponent = 0.0) AND
    (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
    (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
      0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
      1.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
      0.0);
END_ENTITY; -- amount_of_substance_unit

ENTITY application_context;
  application : label;
  DERIVE
    description : text := get_description_value(SELF);
    id : identifier := get_id_value(SELF);
  INVERSE
    context_elements : SET [1:?] OF application_context_element FOR
frame_of_reference:
WHERE
  WR1 : SIZEOF(USEDIN(SELF,
    'ISO_14306_SCHEMA.DESCRPTION_ATTRIBUTE.DESCRIBED_ITEM')) <=
    1;
  WR2 : SIZEOF(USEDIN(SELF,
    'ISO_14306_SCHEMA.ID_ATTRIBUTE.IDENTIFIED_ITEM')) <=
    1;
END_ENTITY; -- application_context

ENTITY application_context_element
  SUPERTYPE OF (ONEOF(product_context, product_definition_context));
  name : label;
  frame_of_reference : application_context;
END_ENTITY; -- application_context_element

ENTITY applied_external_identification_assignment
  SUBTYPE OF (external_identification_assignment);
  items : SET [1:?] OF external_identification_item;
END_ENTITY; -- applied_external_identification_assignment

```

```

ENTITY applied_group_assignment
  SUBTYPE OF (group_assignment);
  items : SET [1:?] OF groupable_item;
END_ENTITY; -- applied_group_assignment

ENTITY applied_identification_assignment
  SUBTYPE OF (identification_assignment);
  items : SET [1:?] OF identification_item;
END_ENTITY; -- applied_identification_assignment

ENTITY area_measure_with_unit
  SUBTYPE OF (measure_with_unit);
  WHERE
    WR1 : 'ISO_14306_SCHEMA.AREA_UNIT' IN
      TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- area_measure_with_unit

ENTITY area_unit
  SUBTYPE OF (derived_unit);
  WHERE
    WR1 : derive_dimensional_exponents(SELF) =
      dimensional_exponents(2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
END_ENTITY; -- area_unit

ENTITY assembly_component_usage
  SUPERTYPE OF (quantified_assembly_component_usage ANDOR
    ONEOF(multi_level_reference_designator,
      next_assembly_usage_occurrence,
      promissory_usage_occurrence,
      specified_higher_usage_occurrence))
  SUBTYPE OF (product_definition_usage);
  reference_designator : OPTIONAL identifier;
END_ENTITY; -- assembly_component_usage

ENTITY axis1_placement
  SUBTYPE OF (placement);
  axis : OPTIONAL direction;
  DERIVE
    z : direction := NVL(normalise(axis),
      dummy_gri || direction([0.0, 0.0, 1.0]));
  WHERE
    WR1 : SELF\geometric_representation_item.dim = 3;
END_ENTITY; -- axis1_placement

ENTITY axis2_placement_2d
  SUBTYPE OF (placement);
  ref_direction : OPTIONAL direction;
  DERIVE
    p : LIST [2:2] OF direction := build_2axes(ref_direction);
  WHERE
    WR1 : SELF\geometric_representation_item.dim = 2;
END_ENTITY; -- axis2_placement_2d

ENTITY axis2_placement_3d
  SUBTYPE OF (placement);
  axis : OPTIONAL direction;
  ref_direction : OPTIONAL direction;
  DERIVE
    p : LIST [3:3] OF direction := build_axes(axis, ref_direction);
  WHERE
    WR1 : SELF\placement.location.dim = 3;
    WR2 : (NOT (EXISTS(axis))) OR (axis.dim = 3);
    WR3 : (NOT (EXISTS(ref_direction))) OR (ref_direction.dim = 3);
    WR4 : (NOT (EXISTS(axis))) OR (NOT (EXISTS(ref_direction))) OR
      (cross_product(axis, ref_direction).magnitude > 0.0);
END_ENTITY; -- axis2_placement_3d

```

```

ENTITY bezier_curve
    SUBTYPE OF (b_spline_curve);
END_ENTITY; -- bezier_curve

ENTITY bezier_surface
    SUBTYPE OF (b_spline_surface);
END_ENTITY; -- bezier_surface

ENTITY binary_generic_expression
    ABSTRACT SUPERTYPE
    SUBTYPE OF (generic_expression);
    operands : LIST [2:2] OF generic_expression;
END_ENTITY; -- binary_generic_expression

ENTITY bounded_curve
    SUPERTYPE OF (ONEOF(polyline, b_spline_curve, trimmed_curve,
                        composite_curve))
    SUBTYPE OF (curve);
END_ENTITY; -- bounded_curve

ENTITY bounded_surface
    SUBTYPE OF (surface);
END_ENTITY; -- bounded_surface

ENTITY brep_with_voids
    SUBTYPE OF (manifold_solid_brep);
    voids : SET [1:?] OF oriented_closed_shell;
END_ENTITY; -- brep_with_voids

ENTITY b_spline_curve
    SUPERTYPE OF (ONEOF(uniform_curve, b_spline_curve_with_knots,
                        quasi_uniform_curve, bezier_curve) ANDOR
                        rational_b_spline_curve)
    SUBTYPE OF (bounded_curve);
    degree : INTEGER;
    control_points_list : LIST [2:?] OF cartesian_point;
    curve_form : b_spline_curve_form;
    closed_curve : LOGICAL;
    self_intersect : LOGICAL;
    DERIVE
        upper_index_on_control_points : INTEGER := (SIZEOF(control_points_list) - 1);
        control_points : ARRAY [0.upper_index_on_control_points] OF cartesian_point :=
list_to_array(control_points_list, 0, upper_index_on_control_points);
    WHERE
        WR1 : ('ISO_14306_SCHEMA.UNIFORM_CURVE' IN TYPEOF(SELF)) OR
              ('ISO_14306_SCHEMA.QUASI_UNIFORM_CURVE' IN TYPEOF(SELF)) OR
              ('ISO_14306_SCHEMA.BEZIER_CURVE' IN TYPEOF(SELF)) OR
              ('ISO_14306_SCHEMA.B_SPLINE_CURVE_WITH_KNOTS' IN TYPEOF(SELF));
END_ENTITY; -- b_spline_curve

ENTITY b_spline_curve_with_knots
    SUBTYPE OF (b_spline_curve);
    knot_multiplicities : LIST [2:?] OF INTEGER;
    knots : LIST [2:?] OF parameter_value;
    knot_spec : knot_type;
    DERIVE
        upper_index_on_knots : INTEGER := SIZEOF(knots);
    WHERE
        WR1 : constraints_param_b_spline(degree, upper_index_on_knots,
                                         upper_index_on_control_points,
                                         knot_multiplicities, knots);
        WR2 : SIZEOF(knot_multiplicities) = upper_index_on_knots;
END_ENTITY; -- b_spline_curve_with_knots

ENTITY b_spline_surface
    SUPERTYPE OF (ONEOF(b_spline_surface_with_knots, uniform_surface,
                        quasi_uniform_surface, bezier_surface) ANDOR

```

```

        rational_b_spline_surface)
    SUBTYPE OF (bounded_surface);
    u_degree : INTEGER;
    v_degree : INTEGER;
    control_points_list : LIST [2:?] OF LIST [2:?] OF cartesian_point;
    surface_form : b_spline_surface_form;
    u_closed : LOGICAL;
    v_closed : LOGICAL;
    self_intersect : LOGICAL;
DERIVE
    u_upper : INTEGER := SIZEOF(control_points_list) - 1;
    v_upper : INTEGER := SIZEOF(control_points_list[1]) - 1;
    control_points : ARRAY [0:u_upper] OF ARRAY [0:v_upper] OF cartesian_point :=
make_array_of_array(control_points_list,0,u_upper,0,v_upper);
    WHERE
        WR1 : ('ISO_14306_SCHEMA.UNIFORM_SURFACE' IN TYPEOF(SELFF)) OR
        ('ISO_14306_SCHEMA.QUASI_UNIFORM_SURFACE' IN TYPEOF(SELFF)) OR
        ('ISO_14306_SCHEMA.BEZIER_SURFACE' IN TYPEOF(SELFF)) OR
        ('ISO_14306_SCHEMA.B_SPLINE_SURFACE_WITH_KNOTS' IN
        TYPEOF(SELFF));
END_ENTITY; -- b_spline_surface

ENTITY b_spline_surface_with_knots
    SUBTYPE OF (b_spline_surface);
    u_multiplicities : LIST [2:?] OF INTEGER;
    v_multiplicities : LIST [2:?] OF INTEGER;
    u_knots : LIST [2:?] OF parameter_value;
    v_knots : LIST [2:?] OF parameter_value;
    knot_spec : knot_type;
DERIVE
    knot_u_upper : INTEGER := SIZEOF(u_knots);
    knot_v_upper : INTEGER := SIZEOF(v_knots);
    WHERE
        WR1 : constraints_param_b_spline(SELFF\b_spline_surface.u_degree,
        knot_u_upper,
        SELFF\b_spline_surface.u_upper,
        u_multiplicities, u_knots);
        WR2 : constraints_param_b_spline(SELFF\b_spline_surface.v_degree,
        knot_v_upper,
        SELFF\b_spline_surface.v_upper,
        v_multiplicities, v_knots);
        WR3 : SIZEOF(u_multiplicities) = knot_u_upper;
        WR4 : SIZEOF(v_multiplicities) = knot_v_upper;
END_ENTITY; -- b_spline_surface_with_knots

ENTITY capacitance_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.CAPACITANCE_UNIT' IN
        TYPEOF(SELFF\measure_with_unit.unit_component);
END_ENTITY; -- capacitance_measure_with_unit

ENTITY capacitance_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELFF) =
        dimensions_for_si_unit(si_unit_name.farad);
END_ENTITY; -- capacitance_unit

ENTITY cartesian_point
    SUBTYPE OF (point);
    coordinates : LIST [1:3] OF length_measure;
END_ENTITY; -- cartesian_point

ENTITY cartesian_transformation_operator
    SUPERTYPE OF (ONEOF(cartesian_transformation_operator_2d,
        cartesian_transformation_operator_3d))

```

```

        SUBTYPE OF (geometric_representation_item,
                    functionally_defined_transformation);
axis1 : OPTIONAL direction;
axis2 : OPTIONAL direction;
local_origin : cartesian_point;
scale : OPTIONAL REAL;
DERIVE
    scl : REAL := NVL(scale, 1.0);
WHERE
    WR1 : scl > 0.0;
END_ENTITY; -- cartesian_transformation_operator

ENTITY cartesian_transformation_operator_2d
    SUBTYPE OF (cartesian_transformation_operator);
DERIVE
    u : LIST [2:2] OF direction := base_axis(2,
SELF\cartesian_transformation_operator.axis1,
SELF\cartesian_transformation_operator.axis2,
    ?);
WHERE
    WR1 : SELF\geometric_representation_item.dim = 2;
END_ENTITY; -- cartesian_transformation_operator_2d

ENTITY cartesian_transformation_operator_3d
    SUBTYPE OF (cartesian_transformation_operator);
axis3 : OPTIONAL direction;
DERIVE
    u : LIST [3:3] OF direction := base_axis(3,
SELF\cartesian_transformation_operator.axis1,
SELF\cartesian_transformation_operator.axis2,
    axis3);
WHERE
    WR1 : SELF\geometric_representation_item.dim = 3;
END_ENTITY; -- cartesian_transformation_operator_3d

ENTITY celsius_temperature_measure_with_unit
    SUBTYPE OF (measure_with_unit);
WHERE
    WR1 : 'ISO_14306_SCHEMA.THERMODYNAMIC_TEMPERATURE_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- celsius_temperature_measure_with_unit

ENTITY chain_based_geometric_item_specific_usage
    SUBTYPE OF (geometric_item_specific_usage,
                chain_based_item_identified_representation_usage);
END_ENTITY; -- chain_based_geometric_item_specific_usage

ENTITY chain_based_item_identified_representation_usage
    SUBTYPE OF (item_identified_representation_usage);
nodes : LIST [2:?] OF UNIQUE representation;
undirected_link : LIST [1:?] OF chained_representation_link;
DERIVE
    root : representation := nodes[1];
    directed_link : LIST [1:?] OF representation_relationship :=
get_directed_link(nodes,undirected_link);
    leaf : representation :=
SELF\item_identified_representation_usage.used_representation;
WHERE
    WR1 : EXISTS(directed_link);
END_ENTITY; -- chain_based_item_identified_representation_usage

ENTITY characterized_object;
name : label;

```

```

    description : OPTIONAL text;
END_ENTITY; -- characterized_object

ENTITY circle
    SUBTYPE OF (conic);
    radius : positive_length_measure;
END_ENTITY; -- circle

ENTITY closed_shell
    SUBTYPE OF (connected_face_set);
END_ENTITY; -- closed_shell

ENTITY complex_number_literal
    SUBTYPE OF (generic_literal);
    real_part : REAL;
    imag_part : REAL;
END_ENTITY; -- complex_number_literal

ENTITY complex_number_literal_polar
    SUBTYPE OF (complex_number_literal);
    modulus : REAL;
    argument : REAL;
    DERIVE
        SELF\complex_number_literal.real_part : REAL := modulus * COS(argument);
        SELF\complex_number_literal.imag_part : REAL := modulus * SIN(argument);
    WHERE
        WR1 : modulus >= 0;
        WR2 : {0 <= argument <= 2 * PI};
END_ENTITY; -- complex_number_literal_polar

ENTITY composite_curve
    SUBTYPE OF (bounded_curve);
    segments : LIST [1:?] OF composite_curve_segment;
    self_intersect : LOGICAL;
    DERIVE
        n_segments : INTEGER := SIZEOF(segments);
        closed_curve : LOGICAL := segments[n_segments].transition <>
        discontinuous;
    WHERE
        WR1 : ((NOT closed_curve) AND
            (SIZEOF(QUERY(temp <* segments | temp.transition = discontinuous)) =
            1)) OR
            ((closed_curve) AND
            (SIZEOF(QUERY(temp <* segments | temp.transition = discontinuous)) =
            0));
END_ENTITY; -- composite_curve

ENTITY composite_curve_on_surface
    SUBTYPE OF (composite_curve);
    DERIVE
        basis_surface : SET [0:2] OF surface := get_basis_surface(SELF);
    WHERE
        WR1 : SIZEOF(basis_surface) > 0;
        WR2 : constraints_composite_curve_on_surface(SELF);
END_ENTITY; -- composite_curve_on_surface

ENTITY composite_curve_segment
    SUBTYPE OF (founded_item);
    transition : transition_code;
    same_sense : BOOLEAN;
    parent_curve : curve;
    INVERSE
        using_curves : BAG [1:?] OF composite_curve FOR segments;
    WHERE
        WR1 : ('ISO_14306_SCHEMA.BOUNDED_CURVE' IN TYPEOF(parent_curve));
END_ENTITY; -- composite_curve_segment

```

```

ENTITY conductance_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.CONDUCTANCE_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- conductance_measure_with_unit

ENTITY conductance_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.siemens);
END_ENTITY; -- conductance_unit

ENTITY conic
    SUPERTYPE OF (ONEOF(circle, ellipse, hyperbola, parabola))
    SUBTYPE OF (curve);
    position : axis2_placement;
END_ENTITY; -- conic

ENTITY conical_surface
    SUBTYPE OF (elementary_surface);
    radius : length_measure;
    semi_angle : plane_angle_measure;
    WHERE
        WR1 : radius >= 0.0;
END_ENTITY; -- conical_surface

ENTITY connected_edge_set
    SUBTYPE OF (topological_representation_item);
    ces_edges : SET [1:?] OF edge;
END_ENTITY; -- connected_edge_set

ENTITY connected_face_set
    SUPERTYPE OF (ONEOF(closed_shell, open_shell))
    SUBTYPE OF (topological_representation_item);
    cfs_faces : SET [1:?] OF face;
END_ENTITY; -- connected_face_set

ENTITY context_dependent_unit
    SUBTYPE OF (named_unit);
    name : label;
END_ENTITY; -- context_dependent_unit

ENTITY conversion_based_unit
    SUBTYPE OF (named_unit);
    name : label;
    conversion_factor : measure_with_unit;
    WHERE
        WR1 : SELF\named_unit.dimensions =
            derive_dimensional_exponents(conversion_factor\measure_with_unit.unit_component);
END_ENTITY; -- conversion_based_unit

ENTITY curve
    SUPERTYPE OF (ONEOF(line, conic, pcurve, surface_curve))
    SUBTYPE OF (geometric_representation_item);
END_ENTITY; -- curve

ENTITY cylindrical_surface
    SUBTYPE OF (elementary_surface);
    radius : positive_length_measure;
END_ENTITY; -- cylindrical_surface

ENTITY definitional_representation
    SUBTYPE OF (representation);
    WHERE

```



```

    WR1 : 'ISO_14306_SCHEMA.PARAMETRIC_REPRESENTATION_CONTEXT' IN
        TYPEOF(SELF\representation.context_of_items);
END_ENTITY; -- definitional_representation

ENTITY definitional_representation_relationship
    SUBTYPE OF (representation_relationship);
WHERE
    WR1 : acyclic_representation_relationship(SELF,

[SELF\representation_relationship.rep_2],
        'ISO_14306_SCHEMA.REPRESENTATION');
END_ENTITY; -- definitional_representation_relationship

ENTITY degenerate_toroidal_surface
    SUBTYPE OF (toroidal_surface);
    select_outer : BOOLEAN;
WHERE
    WR1 : major_radius < minor_radius;
END_ENTITY; -- degenerate_toroidal_surface

ENTITY derived_unit
    SUPERTYPE OF (ONEOF(absorbed_dose_unit, acceleration_unit,
        radioactivity_unit, area_unit, capacitance_unit,
        dose_equivalent_unit, electric_charge_unit,
        conductance_unit, electric_potential_unit,
        energy_unit, magnetic_flux_density_unit, force_unit,
        frequency_unit, illuminance_unit, inductance_unit,
        magnetic_flux_unit, power_unit, pressure_unit,
        resistance_unit, velocity_unit, volume_unit));
    elements : SET [1:?] OF derived_unit_element;
DERIVE
    name : label := get_name_value(SELF);
WHERE
    WR1 : (SIZEOF(elements) > 1) OR
        ((SIZEOF(elements) = 1) AND (elements[1].exponent <> 1.0));
    WR2 : SIZEOF(USEDIN(SELF,
        'ISO_14306_SCHEMA.NAME_ATTRIBUTE.NAMED_ITEM')) <= 1;
END_ENTITY; -- derived_unit

ENTITY derived_unit_element;
    unit : named_unit;
    exponent : REAL;
END_ENTITY; -- derived_unit_element

ENTITY description_attribute;
    attribute_value : text;
    described_item : description_attribute_select;
END_ENTITY; -- description_attribute

ENTITY descriptive_representation_item
    SUBTYPE OF (representation_item);
    description : text;
END_ENTITY; -- descriptive_representation_item

ENTITY dielectric_constant_measure_with_unit
    SUBTYPE OF (ratio_measure_with_unit);
END_ENTITY; -- dielectric_constant_measure_with_unit

ENTITY dimensional_exponents;
    length_exponent : REAL;
    mass_exponent : REAL;
    time_exponent : REAL;
    electric_current_exponent : REAL;
    thermodynamic_temperature_exponent : REAL;
    amount_of_substance_exponent : REAL;
    luminous_intensity_exponent : REAL;
END_ENTITY; -- dimensional_exponents

```

```

ENTITY direction
    SUBTYPE OF (geometric_representation_item);
    direction_ratios : LIST [2:3] OF REAL;
    WHERE
        WR1 : SIZEOF(QUERY(tmp <* direction_ratios | tmp <> 0.0)) > 0;
END_ENTITY; -- direction

ENTITY document;
    id : identifier;
    name : label;
    description : OPTIONAL text;
    kind : document_type;
    INVERSE
        representation_types : SET [0:?] OF document_representation_type FOR
        represented_document;
END_ENTITY; -- document

ENTITY document_file
    SUBTYPE OF (document, characterized_object);
    WHERE
        WR1 : SELF\characterized_object.name = '';
        WR2 : NOT EXISTS(SELF\characterized_object.description);
        WR3 : SIZEOF(QUERY(drt <* SELF\document.representation_types | drt.name IN
            ['digital', 'physical'])) = 1;
END_ENTITY; -- document_file

ENTITY document_representation_type;
    name : label;
    represented_document : document;
END_ENTITY; -- document_representation_type

ENTITY document_type;
    product_data_type : label;
END_ENTITY; -- document_type

ENTITY dose_equivalent_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.DOSE_EQUIVALENT_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- dose_equivalent_measure_with_unit

ENTITY dose_equivalent_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.sievert);
END_ENTITY; -- dose_equivalent_unit

ENTITY edge
    SUPERTYPE OF (ONEOF(edge_curve, oriented_edge))
    SUBTYPE OF (topological_representation_item);
    edge_start : vertex;
    edge_end : vertex;
END_ENTITY; -- edge

ENTITY edge_curve
    SUBTYPE OF (edge, geometric_representation_item);
    edge_geometry : curve;
    same_sense : BOOLEAN;
END_ENTITY; -- edge_curve

ENTITY edge_loop
    SUBTYPE OF (loop, path);
    DERIVE
        ne : INTEGER := SIZEOF(SELF\path.edge_list);

```

```

WHERE
  WR1 : (SELF\path.edge_list[1].edge_start) :=:
        (SELF\path.edge_list[ne].edge_end);
END_ENTITY; -- edge_loop

ENTITY electric_charge_measure_with_unit
  SUBTYPE OF (measure_with_unit);
WHERE
  WR1 : 'ISO_14306_SCHEMA.ELECTRIC_CHARGE_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- electric_charge_measure_with_unit

ENTITY electric_charge_unit
  SUBTYPE OF (derived_unit);
WHERE
  WR1 : derive_dimensional_exponents(SELF) =
        dimensions_for_si_unit(si_unit_name.coulomb);
END_ENTITY; -- electric_charge_unit

ENTITY electric_current_measure_with_unit
  SUBTYPE OF (measure_with_unit);
WHERE
  WR1 : 'ISO_14306_SCHEMA.ELECTRIC_CURRENT_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- electric_current_measure_with_unit

ENTITY electric_current_unit
  SUBTYPE OF (named_unit);
WHERE
  WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
        (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
        (SELF\named_unit.dimensions.time_exponent = 0.0) AND
        (SELF\named_unit.dimensions.electric_current_exponent = 1.0) AND
        (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
          0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
          0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
          0.0);
END_ENTITY; -- electric_current_unit

ENTITY electric_potential_measure_with_unit
  SUBTYPE OF (measure_with_unit);
WHERE
  WR1 : 'ISO_14306_SCHEMA.ELECTRIC_POTENTIAL_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- electric_potential_measure_with_unit

ENTITY electric_potential_unit
  SUBTYPE OF (derived_unit);
WHERE
  WR1 : derive_dimensional_exponents(SELF) =
        dimensions_for_si_unit(si_unit_name.volt);
END_ENTITY; -- electric_potential_unit

ENTITY elementary_brep_shape_representation
  SUBTYPE OF (shape_representation);
WHERE
  WR1 : SIZEOF(QUERY(it <* SELF.items |
                    NOT (SIZEOF(['ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP',
                                'ISO_14306_SCHEMA.FACETED_BREP',
                                'ISO_14306_SCHEMA.MAPPED_ITEM',
                                'ISO_14306_SCHEMA.AXIS2_PLACEMENT_3D'] *
                                TYPEOF(it)) =
                    1))) =
        0;
  WR2 : SIZEOF(QUERY(it <* SELF.items |
                    SIZEOF(['ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP',
                            'ISO_14306_SCHEMA.MAPPED_ITEM'] *

```

```

                                TYPEOF(it)) =
                                1)) >
                                0;
WR3 : SIZEOF(QUERY(msb <*
                                QUERY(it <* SELF.items |
                                    'ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
                                    TYPEOF(it)) |
                                NOT (SIZEOF(QUERY(csh <* msb_shells(msb) |
                                NOT (SIZEOF(QUERY(fcs <*
                                    csh.cfs_faces |
                                NOT ('ISO_14306_SCHEMA.FACE_SURFACE' IN
                                    TYPEOF(fcs)))) = 0))) = 0))) = 0;
WR4 : SIZEOF(QUERY(msb <*
                                QUERY(it <* SELF.items |
                                    'ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
                                    TYPEOF(it)) |
                                NOT (SIZEOF(QUERY(csh <* msb_shells(msb) |
                                NOT (SIZEOF(QUERY(fcs <* csh\connected_face_set.cfs_faces |
                                NOT (('ISO_14306_SCHEMA.ELEMENTARY_SURFACE' IN
                                    TYPEOF(fcs\face_surface.face_geometry)))) = 0))) = 0))) = 0;
WR5 : SIZEOF(QUERY(msb <*
                                QUERY(it <* SELF.items |
                                    'ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
                                    TYPEOF(it)) |
                                NOT (SIZEOF(QUERY(csh <* msb_shells(msb) |
                                NOT (SIZEOF(QUERY(fcs <* csh\connected_face_set.cfs_faces |
                                NOT (SIZEOF(QUERY(elp_fbnds <* QUERY(bnds <* fcs.bounds |
                                    'ISO_14306_SCHEMA.EDGE_LOOP' IN TYPEOF(bnds.bound)) |
                                NOT (SIZEOF(QUERY(oe <* elp_fbnds.bound\path.edge_list |
                                    NOT ('ISO_14306_SCHEMA.EDGE_CURVE' IN
                                        TYPEOF(oe.edge_element)))) = 0))) = 0))) = 0))) = 0;
WR6 : SIZEOF(QUERY(msb <*
                                QUERY(it <* SELF.items |
                                    'ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
                                    TYPEOF(it)) |
                                NOT (SIZEOF(QUERY(csh <* msb_shells(msb) |
                                NOT (SIZEOF(QUERY(fcs <* csh\connected_face_set.cfs_faces |
                                NOT (SIZEOF(QUERY(elp_fbnds <* QUERY(bnds <* fcs.bounds |
                                    'ISO_14306_SCHEMA.EDGE_LOOP' IN TYPEOF(bnds.bound)) |
                                NOT (SIZEOF(QUERY(oe <* elp_fbnds.bound\path.edge_list |
                                NOT (SIZEOF(['ISO_14306_SCHEMA.LINE',
                                    'ISO_14306_SCHEMA.CONIC',
                                    'ISO_14306_SCHEMA.POLYLINE'] *
                                        TYPEOF(oe.edge_element\edge_curve.edge_geometry)) = 1))) =
                                0))) = 0))) = 0))) = 0;
WR7 : SIZEOF(QUERY(msb <*
                                QUERY(it <* SELF.items |
                                    'ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
                                    TYPEOF(it)) |
                                NOT (SIZEOF(QUERY(csh <* msb_shells(msb) |
                                NOT (SIZEOF(QUERY(fcs <* csh\connected_face_set.cfs_faces |
                                NOT (SIZEOF(QUERY(elp_fbnds <* QUERY(bnds <* fcs.bounds |
                                    'ISO_14306_SCHEMA.EDGE_LOOP' IN TYPEOF(bnds.bound)) |
                                NOT (SIZEOF(QUERY(oe <* elp_fbnds.bound\path.edge_list |
                                NOT (('ISO_14306_SCHEMA.VERTEX_POINT' IN
                                    TYPEOF(oe.edge_start)) AND ('ISO_14306_SCHEMA.VERTEX_POINT'
IN
                                TYPEOF(oe.edge_end)))))) = 0))) = 0))) = 0))) = 0;
WR8 : SIZEOF(QUERY(msb <*
                                QUERY(it <* SELF.items |
                                    'ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
                                    TYPEOF(it)) |
                                NOT (SIZEOF(QUERY(csh <* msb_shells(msb) |
                                NOT (SIZEOF(QUERY(fcs <* csh\connected_face_set.cfs_faces |
                                NOT (SIZEOF(QUERY(elp_fbnds <* QUERY(bnds <* fcs.bounds |
                                    'ISO_14306_SCHEMA.EDGE_LOOP' IN TYPEOF(bnds.bound)) |
                                NOT (SIZEOF(QUERY(oe <* elp_fbnds.bound\path.edge_list |

```

```

        ('ISO_14306_SCHEMA.POLYLINE' IN
        TYPEOF(oe.edge_element\edge_curve.edge_geometry)) AND
        (NOT
        (SIZEOF(oe\oriented_edge.edge_element\edge_curve.edge_geometry\polyline.points) >=
        3))) = 0))) = 0))) = 0))) = 0))) = 0;
    WR9 : SIZEOF(QUERY(msb <*
        QUERY(it <* items |
        'ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
        TYPEOF(it)) |
        'ISO_14306_SCHEMA.ORIENTED_CLOSED_SHELL' IN
        TYPEOF(msb\manifold_solid_brep.outer))) =
    0;
    WR10 : SIZEOF(QUERY(brv <*
        QUERY(it <* items |
        'ISO_14306_SCHEMA.BREP_WITH_VOIDS' IN
        TYPEOF(it)) |
        NOT (SIZEOF(QUERY(csh <* brv\brep_with_voids.voids
        csh\oriented_closed_shell.orientation)) =
        0))) =
    0;
    WR11 : SIZEOF(QUERY(mi <*
        QUERY(it <* items |
        'ISO_14306_SCHEMA.MAPPED_ITEM' IN
        TYPEOF(it)) |
        NOT ('ISO_14306_SCHEMA.ELEMENTARY_BREP_SHAPE_REPRESENTATION'
IN
        TYPEOF(mi\mapped_item.mapping_source.mapped_representation)))) =
    0;
    WR12 : SIZEOF(QUERY(msb <*
        QUERY(it <* SELF.items
        'ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
        TYPEOF(it)) |
        NOT (SIZEOF(QUERY(csh <* msb_shells(msb) |
        NOT (SIZEOF(QUERY(fcs <* csh\connected_face_set.cfs_faces |
        NOT (SIZEOF(QUERY(vlp_fbnds <* QUERY(bnds <* fcs.bounds |
        'ISO_14306_SCHEMA.VERTEX_LOOP' IN TYPEOF(bnds.bound)) |
        NOT (('ISO_14306_SCHEMA.VERTEX_POINT' IN
        TYPEOF(vlp_fbnds\face_bound.bound\vertex_loop.loop_vertex))
AND
        ('ISO_14306_SCHEMA.CARTESIAN_POINT' IN
        TYPEOF(vlp_fbnds\face_bound.bound\vertex_loop.loop_vertex\vertex_point.vertex_geome
try)))) = 0))) = 0))) = 0))) = 0;
END_ENTITY; -- elementary_brep_shape_representation

ENTITY elementary_surface
    SUPERTYPE OF (ONEOF(plane, cylindrical_surface, conical_surface,
        spherical_surface, toroidal_surface))
    SUBTYPE OF (surface);
    position : axis2_placement_3d;
END_ENTITY; -- elementary_surface

ENTITY ellipse
    SUBTYPE OF (conic);
    semi_axis_1 : positive_length_measure;
    semi_axis_2 : positive_length_measure;
END_ENTITY; -- ellipse

ENTITY energy_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
    WR1 : 'ISO_14306_SCHEMA.ENERGY_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- energy_measure_with_unit

ENTITY energy_unit

```

```

        SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.joule);
    END_ENTITY; -- energy_unit

ENTITY environment;
    syntactic_representation : generic_variable;
    semantics : variable_semantics;
    END_ENTITY; -- environment

ENTITY expression_conversion_based_unit
    SUBTYPE OF (context_dependent_unit, variable_semantics);
    INVERSE
        associated_variable_environment : environment FOR semantics;
    END_ENTITY; -- expression_conversion_based_unit

ENTITY externally_defined_hatch_style
    SUBTYPE OF (externally_defined_item, geometric_representation_item);
    END_ENTITY; -- externally_defined_hatch_style

ENTITY externally_defined_item;
    item_id : source_item;
    source : external_source;
    END_ENTITY; -- externally_defined_item

ENTITY external_identification_assignment
    ABSTRACT SUPERTYPE
    SUBTYPE OF (identification_assignment);
    source : external_source;
    END_ENTITY; -- external_identification_assignment

ENTITY external_source;
    source_id : source_item;
    DERIVE
        description : text := get_description_value(SELF);
    WHERE
        WR1 : SIZEOF(USEDIN(SELF,
            'ISO_14306_SCHEMA.DESCRPTION_ATTRIBUTE.DESCRIBED_ITEM')) <=
            1;
    END_ENTITY; -- external_source

ENTITY face
    SUPERTYPE OF (ONEOF(face_surface, oriented_face))
    SUBTYPE OF (topological_representation_item);
    bounds : SET [1..?] OF face_bound;
    WHERE
        WR1 : NOT (mixed_loop_type_set(list_to_set(list_face_loops(SELF))));
        WR2 : SIZEOF(QUERY(temp <* bounds |
            'ISO_14306_SCHEMA.FACE_OUTER_BOUND' IN
            TYPEOF(temp))) <=
            1;
    END_ENTITY; -- face

ENTITY faceted_brep
    SUBTYPE OF (manifold_solid_brep);
    END_ENTITY; -- faceted_brep

ENTITY faceted_brep_shape_representation
    SUBTYPE OF (shape_representation);
    WHERE
        WR1 : SIZEOF(QUERY(it <* items |
            NOT (SIZEOF(['ISO_14306_SCHEMA.FACETED_BREP',
                'ISO_14306_SCHEMA.MAPPED_ITEM',
                'ISO_14306_SCHEMA.AXIS2_PLACEMENT_3D'] *
                TYPEOF(it)) =
            1))) =

```

```

0;
WR2 : SIZEOF(QUERY(it <* items |
    (SIZEOF(['ISO_14306_SCHEMA.FACETED_BREP',
        'ISO_14306_SCHEMA.MAPPED_ITEM'] *
        TYPEOF(it)) =
    1))) >

0;
WR3 : SIZEOF(QUERY(fbrep <*
    QUERY(it <* items |
        ('ISO_14306_SCHEMA.FACETED_BREP' IN
        TYPEOF(it))) |
    NOT (SIZEOF(QUERY(csh <* msb_shells(fbrep) |
    NOT (SIZEOF(QUERY(fcs <* csh\connected_face_set.cfs_faces |
    NOT (('ISO_14306_SCHEMA.FACE_SURFACE' IN
    TYPEOF(fcs)) AND (('ISO_14306_SCHEMA.PLANE' IN
    TYPEOF(fcs\face_surface.face_geometry)) AND
        ('ISO_14306_SCHEMA.CARTESIAN_POINT' IN
    TYPEOF(fcs\face_surface.face_geometry\elementary_surface.position.location)))))) =
    0))) = 0))) = 0;
WR4 : SIZEOF(QUERY(fbrep <*
    QUERY(it <* items |
        ('ISO_14306_SCHEMA.FACETED_BREP' IN
        TYPEOF(it))) |
    NOT (SIZEOF(QUERY(csh <* msb_shells(fbrep) |
    NOT (SIZEOF(QUERY(fcs <* csh\connected_face_set.cfs_faces |
    NOT (SIZEOF(QUERY(bnds <* fcs.bounds |
        ('ISO_14306_SCHEMA.FACE_OUTER_BOUND' IN
        TYPEOF(bnds)))) = 1))) = 0))) = 0))) = 0;
WR5 : SIZEOF(QUERY(msb <*
    QUERY(it <* items |
        ('ISO_14306_SCHEMA.MANIFOLD_SOLID_BREP' IN
        TYPEOF(it))) |
        ('ISO_14306_SCHEMA.ORIENTED_CLOSED_SHELL' IN
        TYPEOF(msb\manifold_solid_brep.outer)))) =
0;
WR6 : SIZEOF(QUERY(brv <*
    QUERY(it <* items |
        ('ISO_14306_SCHEMA.BREP_WITH_VOIDS' IN
        TYPEOF(it))) |
    NOT (SIZEOF(QUERY(csh <* brv\brep_with_voids.voids |
        csh\oriented_closed_shell.orientation)) =
    0))) =
0;
WR7 : SIZEOF(QUERY(mi <*
    QUERY(it <* items |
        ('ISO_14306_SCHEMA.MAPPED_ITEM' IN
        TYPEOF(it))) |
    NOT ('ISO_14306_SCHEMA.FACETED_BREP_SHAPE_REPRESENTATION' IN
    TYPEOF(mi\mapped_item.mapping_source.mapped_representation)))) =
0;
END_ENTITY; -- faceted_brep_shape_representation

ENTITY face_bound
    SUBTYPE OF (topological_representation_item);
    bound : loop;
    orientation : BOOLEAN;
END_ENTITY; -- face_bound

ENTITY face_outer_bound
    SUBTYPE OF (face_bound);
END_ENTITY; -- face_outer_bound

ENTITY face_surface
    SUBTYPE OF (face, geometric_representation_item);
    face_geometry : surface;

```

```

    same_sense : BOOLEAN;
WHERE
    WR1 : NOT ('ISO_14306_SCHEMA.ORIENTED_SURFACE' IN
        TYPEOF(face_geometry));
END_ENTITY; -- face_surface

ENTITY force_measure_with_unit
    SUBTYPE OF (measure_with_unit);
WHERE
    WR1 : 'ISO_14306_SCHEMA.FORCE_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- force_measure_with_unit

ENTITY force_unit
    SUBTYPE OF (derived_unit);
WHERE
    WR1 : derive_dimensional_exponents(SELF) =
        dimensions_for_si_unit(si_unit_name.newton);
END_ENTITY; -- force_unit

ENTITY founded_item;
DERIVE
    users : SET [0:?] OF founded_item_select := using_items(SELF, []);
WHERE
    WR1 : SIZEOF(users) > 0;
    WR2 : NOT (SELF IN users);
END_ENTITY; -- founded_item

ENTITY frequency_measure_with_unit
    SUBTYPE OF (measure_with_unit);
WHERE
    WR1 : 'ISO_14306_SCHEMA.FREQUENCY_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- frequency_measure_with_unit

ENTITY frequency_unit
    SUBTYPE OF (derived_unit);
WHERE
    WR1 : derive_dimensional_exponents(SELF) =
        dimensions_for_si_unit(si_unit_name.hertz);
END_ENTITY; -- frequency_unit

ENTITY functionally_defined_transformation;
    name : label;
    description : OPTIONAL text;
END_ENTITY; -- functionally_defined_transformation

ENTITY general_property;
    id : identifier;
    name : label;
    description : OPTIONAL text;
END_ENTITY; -- general_property

ENTITY general_property_association;
    name : label;
    description : OPTIONAL text;
    base_definition : general_property;
    derived_definition : derived_property_select;
WHERE
    WR1 : SIZEOF(USEDIN(derived_definition,
        'ISO_14306_SCHEMA.GENERAL_PROPERTY_ASSOCIATION.DERIVED_DEFINITION')) =
        1;
    WR2 : derived_definition.name = base_definition.name;
END_ENTITY; -- general_property_association

ENTITY general_property_relationship;

```



```

name : label;
description : OPTIONAL text;
relating_property : general_property;
related_property : general_property;
END_ENTITY; -- general_property_relationship

ENTITY generic_expression
    ABSTRACT SUPERTYPE OF (ONEOF(simple_generic_expression,
                                   unary_generic_expression,
                                   binary_generic_expression,
                                   multiple_arity_generic_expression));
    WHERE
        WR1 : is_acyclic(SELF);
END_ENTITY; -- generic_expression

ENTITY generic_literal
    ABSTRACT SUPERTYPE
        SUBTYPE OF (simple_generic_expression);
END_ENTITY; -- generic_literal

ENTITY generic_product_definition_reference
    ABSTRACT SUPERTYPE OF (ONEOF(product_definition_reference,
                                   product_definition_reference_with_local_representation));
    source : external_source;
END_ENTITY; -- generic_product_definition_reference

ENTITY generic_variable
    ABSTRACT SUPERTYPE
        SUBTYPE OF (simple_generic_expression);
    INVERSE
        interpretation : environment FOR syntactic representation;
END_ENTITY; -- generic_variable

ENTITY geometric_item_specific_usage
    SUBTYPE OF (item_identified_representation_usage);
    SELF\item_identified_representation_usage.definition :
geometric_item_specific_usage_select;
    SELF\item_identified_representation_usage.used_representation : shape_model;
    SELF\item_identified_representation_usage.identified_item : geometric_model_item;
    UNIQUE
        UR1: definition;
END_ENTITY; -- geometric_item_specific_usage

ENTITY geometric_model_element_relationship
    SUBTYPE OF (geometric_representation_item,
                representation_item_relationship);
    SELF\representation_item_relationship.relying_representation_item :
geometric_representation_item;
    SELF\representation_item_relationship.related_representation_item :
geometric_representation_item;
    UNIQUE
        UR1: relying_representation_item, related_representation_item;
    WHERE
        WR1 : SELF\representation_item_relationship.relying_representation_item :<>:
            SELF\representation_item_relationship.related_representation_item;
END_ENTITY; -- geometric_model_element_relationship

ENTITY geometric_representation_context
    SUBTYPE OF (representation_context);
    coordinate_space_dimension : dimension_count;
END_ENTITY; -- geometric_representation_context

ENTITY geometric_representation_item
    SUPERTYPE OF (ONEOF(point, direction, vector, placement,
                           cartesian_transformation_operator, curve, surface,
                           edge_curve, face_surface, poly_loop, vertex_point,

```

```

        solid_model))
    SUBTYPE OF (representation_item);
DERIVE
    dim : dimension_count := dimension_of(SELF);
WHERE
    WR1 : SIZEOF(QUERY(using_rep <* using_representations(SELF) |
        NOT ('ISO_14306_SCHEMA.GEOMETRIC_REPRESENTATION_CONTEXT' IN
        TYPEOF(using_rep.context_of_items)))) =
        0;
END_ENTITY; -- geometric_representation_item

ENTITY global_uncertainty_assigned_context
    SUBTYPE OF (representation_context);
    uncertainty : SET [1:?] OF uncertainty_measure_with_unit;
END_ENTITY; -- global_uncertainty_assigned_context

ENTITY global_unit_assigned_context
    SUBTYPE OF (representation_context);
    units : SET [1:?] OF unit;
END_ENTITY; -- global_unit_assigned_context

ENTITY group;
    name : label;
    description : OPTIONAL text;
DERIVE
    id : identifier := get_id_value(SELF);
WHERE
    WR1 : SIZEOF(USEDIN(SELF,
        'ISO_14306_SCHEMA.ID_ATTRIBUTE.IDENTIFIED_ITEM')) <=
        1;
END_ENTITY; -- group

ENTITY group_assignment
    ABSTRACT SUPERTYPE;
    assigned_group : group;
DERIVE
    role : object_role := get_role(SELF);
WHERE
    WR1 : SIZEOF(USEDIN(SELF,
        'ISO_14306_SCHEMA.ROLE_ASSOCIATION.ITEM_WITH_ROLE')) <=
        1;
END_ENTITY; -- group_assignment

ENTITY group_relationship;
    name : label;
    description : OPTIONAL text;
    relating_group : group;
    related_group : group;
END_ENTITY; -- group_relationship

ENTITY hyperbola
    SUBTYPE OF (conic);
    semi_axis : positive_length_measure;
    semi_imag_axis : positive_length_measure;
END_ENTITY; -- hyperbola

ENTITY identification_assignment
    ABSTRACT SUPERTYPE;
    assigned_id : identifier;
    role : identification_role;
END_ENTITY; -- identification_assignment

ENTITY identification_role;
    name : label;
    description : OPTIONAL text;
END_ENTITY; -- identification_role

```

```

ENTITY id_attribute;
    attribute_value : identifier;
    identified_item : id_attribute_select;
END_ENTITY; -- id_attribute

ENTITY illuminance_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.ILLUMINANCE_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- illuminance_measure_with_unit

ENTITY illuminance_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.lux);
END_ENTITY; -- illuminance_unit

ENTITY inductance_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.INDUCTANCE_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- inductance_measure_with_unit

ENTITY inductance_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.henry);
END_ENTITY; -- inductance_unit

ENTITY item_defined_transformation;
    name : label;
    description : OPTIONAL text;
    transform_item_1 : representation_item;
    transform_item_2 : representation_item;
END_ENTITY; -- item_defined_transformation

ENTITY item_identified_representation_usage;
    name : label;
    description : OPTIONAL text;
    definition : item_identified_representation_usage_definition;
    used_representation : representation;
    identified_item : item_identified_representation_usage_select;
    WHERE
        WR1 : SELF.used_representation IN
            using_representations(SELF.identified_item);
END_ENTITY; -- item_identified_representation_usage

ENTITY length_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.LENGTH_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- length_measure_with_unit

ENTITY length_unit
    SUBTYPE OF (named_unit);
    WHERE
        WR1 : (SELF\named_unit.dimensions.length_exponent = 1.0) AND
            (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
            (SELF\named_unit.dimensions.time_exponent = 0.0) AND
            (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
            (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
                0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =

```

```

        0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
        0.0);
END_ENTITY; -- length_unit

ENTITY line
    SUBTYPE OF (curve);
    pnt : cartesian_point;
    dir : vector;
    WHERE
        WR1 : dir.dim = pnt.dim;
END_ENTITY; -- line

ENTITY loop
    SUPERTYPE OF (ONEOF(vertex_loop, edge_loop, poly_loop))
    SUBTYPE OF (topological_representation_item);
END_ENTITY; -- loop

ENTITY loss_tangent_measure_with_unit
    SUBTYPE OF (ratio_measure_with_unit);
END_ENTITY; -- loss_tangent_measure_with_unit

ENTITY luminous_flux_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.LUMINOUS_FLUX_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- luminous_flux_measure_with_unit

ENTITY luminous_flux_unit
    SUBTYPE OF (named_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.lumen);
END_ENTITY; -- luminous_flux_unit

ENTITY luminous_intensity_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.LUMINOUS_INTENSITY_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- luminous_intensity_measure_with_unit

ENTITY luminous_intensity_unit
    SUBTYPE OF (named_unit);
    WHERE
        WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
            (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
            (SELF\named_unit.dimensions.time_exponent = 0.0) AND
            (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
            (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
            0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
            0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
            1.0);
END_ENTITY; -- luminous_intensity_unit

ENTITY magnetic_flux_density_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.MAGNETIC_FLUX_DENSITY_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- magnetic_flux_density_measure_with_unit

ENTITY magnetic_flux_density_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.tesla);

```

```

END_ENTITY; -- magnetic_flux_density_unit

ENTITY magnetic_flux_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.MAGNETIC_FLUX_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- magnetic_flux_measure_with_unit

ENTITY magnetic_flux_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.weber);
END_ENTITY; -- magnetic_flux_unit

ENTITY manifold_solid_brep
    SUBTYPE OF (solid_model);
    outer : closed_shell;
END_ENTITY; -- manifold_solid_brep

ENTITY mapped_item
    SUBTYPE OF (representation_item);
    mapping_source : representation_map;
    mapping_target : representation_item;
    WHERE
        WR1 : acyclic_mapped_representation(SELF);
END_ENTITY; -- mapped_item

ENTITY mass_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.MASS_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- mass_measure_with_unit

ENTITY mass_unit
    SUBTYPE OF (named_unit);
    WHERE
        WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
            (SELF\named_unit.dimensions.mass_exponent = 1.0) AND
            (SELF\named_unit.dimensions.time_exponent = 0.0) AND
            (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
            (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
                0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
                0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
                0.0);
END_ENTITY; -- mass_unit

ENTITY measure_representation_item
    SUBTYPE OF (representation_item, measure_with_unit);
END_ENTITY; -- measure_representation_item

ENTITY measure_with_unit
    SUPERTYPE OF (ONEOF(length_measure_with_unit, mass_measure_with_unit,
        time_measure_with_unit,
        electric_current_measure_with_unit,
        thermodynamic_temperature_measure_with_unit,
        celsius_temperature_measure_with_unit,
        amount_of_substance_measure_with_unit,
        luminous_intensity_measure_with_unit,
        plane_angle_measure_with_unit,
        solid_angle_measure_with_unit, area_measure_with_unit,
        volume_measure_with_unit, ratio_measure_with_unit,
        acceleration_measure_with_unit,
        capacitance_measure_with_unit,
        electric_charge_measure_with_unit,

```

```

        conductance_measure_with_unit,
        electric_potential_measure_with_unit,
        energy_measure_with_unit,
        magnetic_flux_density_measure_with_unit,
        force_measure_with_unit, frequency_measure_with_unit,
        illuminance_measure_with_unit,
        inductance_measure_with_unit,
        luminous_flux_measure_with_unit,
        magnetic_flux_measure_with_unit,
        power_measure_with_unit, pressure_measure_with_unit,
        resistance_measure_with_unit,
        velocity_measure_with_unit,
        absorbed_dose_measure_with_unit,
        radioactivity_measure_with_unit,
        dose_equivalent_measure_with_unit));
    value_component : measure_value;
    unit_component : unit;
WHERE
    WR1 : valid_units(SELF);
END_ENTITY; -- measure_with_unit

ENTITY multiple_arity_generic_expression
    ABSTRACT SUPERTYPE
        SUBTYPE OF (generic_expression);
    operands : LIST [2:?] OF generic_expression;
END_ENTITY; -- multiple_arity_generic_expression

ENTITY multi_level_reference_designator
    SUBTYPE OF (assembly_component_usage);
    location : LIST [1:?] OF UNIQUE next_assembly_usage_occurrence;
    DERIVE
        leaf : product_definition_or_reference :=
SELF\product_definition_relationship.related_product_definition;
        root : product_definition :=
SELF\product_definition_relationship.relatng_product_definition;
    WHERE
        WR1 : unambiguously_specified_multi_level_reference_designator(location);
        WR2 : SIZEOF(QUERY(cp <* location |
            NOT
            (EXISTS(cp\assembly_component_usage.reference_designator)))) =
            0;
END_ENTITY; -- multi_level_reference_designator

ENTITY named_unit
    SUPERTYPE OF (ONEOF(si_unit, conversion_based_unit,
        context_dependent_unit) ANDOR
        ONEOF(length_unit, mass_unit, time_unit,
        electric_current_unit, thermodynamic_temperature_unit,
        amount_of_substance_unit, luminous_flux_unit,
        luminous_intensity_unit, plane_angle_unit,
        solid_angle_unit, ratio_unit));
    dimensions : dimensional_exponents;
END_ENTITY; -- named_unit

ENTITY name_attribute;
    attribute_value : label;
    named_item : name_attribute_select;
END_ENTITY; -- name_attribute

ENTITY next_assembly_usage_occurrence
    SUBTYPE OF (assembly_component_usage);
    UNIQUE
        UR1: reference_designator, relating_product_definition;
END_ENTITY; -- next_assembly_usage_occurrence

ENTITY object_role;
    name : label;

```

```

    description : OPTIONAL text;
END_ENTITY; -- object_role

ENTITY open_shell
    SUBTYPE OF (connected_face_set);
END_ENTITY; -- open_shell

ENTITY oriented_closed_shell
    SUBTYPE OF (closed_shell);
    closed_shell_element : closed_shell;
    orientation : BOOLEAN;
    DERIVE
        SELF\connected_face_set.cfs_faces : SET [1:?] OF face :=
conditional_reverse(SELF.orientation,

SELF.closed_shell_element.cfs_faces);
    WHERE
        WR1 : NOT ('ISO_14306_SCHEMA.ORIENTED_CLOSED_SHELL' IN
            TYPEOF(SELF.closed_shell_element));
END_ENTITY; -- oriented_closed_shell

ENTITY oriented_edge
    SUBTYPE OF (edge);
    edge_element : edge;
    orientation : BOOLEAN;
    DERIVE
        SELF\edge.edge_start : vertex := boolean_choose(SELF.orientation,
            SELF.edge_element.edge_start,
            SELF.edge_element.edge_end);
        SELF\edge.edge_end : vertex := boolean_choose(SELF.orientation,
            SELF.edge_element.edge_end,
            SELF.edge_element.edge_start);
    WHERE
        WR1 : NOT ('ISO_14306_SCHEMA.ORIENTED_EDGE' IN
            TYPEOF(SELF.edge_element));
END_ENTITY; -- oriented_edge

ENTITY oriented_face
    SUBTYPE OF (face);
    face_element : face;
    orientation : BOOLEAN;
    DERIVE
        SELF\face.boundaries : SET [1:?] OF face_bound :=
conditional_reverse(SELF.orientation,

SELF.face_element.boundaries);
    WHERE
        WR1 : NOT ('ISO_14306_SCHEMA.ORIENTED_FACE' IN
            TYPEOF(SELF.face_element));
END_ENTITY; -- oriented_face

ENTITY oriented_open_shell
    SUBTYPE OF (open_shell);
    open_shell_element : open_shell;
    orientation : BOOLEAN;
    DERIVE
        SELF\connected_face_set.cfs_faces : SET [1:?] OF face :=
conditional_reverse(SELF.orientation,

SELF.open_shell_element.cfs_faces);
    WHERE
        WR1 : NOT ('ISO_14306_SCHEMA.ORIENTED_OPEN_SHELL' IN
            TYPEOF(SELF.open_shell_element));
END_ENTITY; -- oriented_open_shell

ENTITY oriented_path
    SUBTYPE OF (path);

```

```

    path_element : path;
    orientation : BOOLEAN;
    DERIVE
        SELF\path.edge_list : LIST [1:?] OF UNIQUE oriented_edge :=
conditional_reverse(SELF.orientation,

SELF.path_element.edge_list);
    WHERE
        WR1 : NOT ('ISO_14306_SCHEMA.ORIENTED_PATH' IN
            TYPEOF(SELF.path_element));
    END_ENTITY; -- oriented_path

ENTITY oriented_surface
    SUBTYPE OF (surface);
    orientation : BOOLEAN;
    END_ENTITY; -- oriented_surface

ENTITY parabola
    SUBTYPE OF (conic);
    focal_dist : length_measure;
    WHERE
        WR1 : focal_dist <> 0.0;
    END_ENTITY; -- parabola

ENTITY parametric_representation_context
    SUBTYPE OF (representation_context);
    END_ENTITY; -- parametric_representation_context

ENTITY path
    SUPERTYPE OF (ONEOF(edge_loop, oriented_path));
    SUBTYPE OF (topological_representation_item);
    edge_list : LIST [1:?] OF UNIQUE oriented_edge;
    WHERE
        WR1 : path_head_to_tail(SELF);
    END_ENTITY; -- path

ENTITY pcurve
    SUBTYPE OF (curve);
    basis_surface : surface;
    reference_to_curve : definitional_representation;
    WHERE
        WR1 : SIZEOF(reference_to_curve\representation.items) = 1;
        WR2 : 'ISO_14306_SCHEMA.CURVE' IN
            TYPEOF(reference_to_curve\representation.items[1]);
        WR3 :
reference_to_curve\representation.items[1]\geometric_representation_item.dim =
2;
    END_ENTITY; -- pcurve

ENTITY placement
    SUPERTYPE OF (ONEOF(axis1_placement, axis2_placement_2d,
axis2_placement_3d))
    SUBTYPE OF (geometric_representation_item);
    location : cartesian_point;
    END_ENTITY; -- placement

ENTITY plane
    SUBTYPE OF (elementary_surface);
    END_ENTITY; -- plane

ENTITY plane_angle_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.PLANE_ANGLE_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
    END_ENTITY; -- plane_angle_measure_with_unit

```



```

ENTITY plane_angle_unit
  SUBTYPE OF (named_unit);
  WHERE
    WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
          (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
          (SELF\named_unit.dimensions.time_exponent = 0.0) AND
          (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
          (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
            0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
            0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
            0.0);
END_ENTITY; -- plane_angle_unit

ENTITY point
  SUPERTYPE OF (ONEOF(cartesian_point, point_on_curve, point_on_surface));
  SUBTYPE OF (geometric_representation_item);
END_ENTITY; -- point

ENTITY point_on_curve
  SUBTYPE OF (point);
  basis_curve : curve;
  point_parameter : parameter_value;
END_ENTITY; -- point_on_curve

ENTITY point_on_surface
  SUBTYPE OF (point);
  basis_surface : surface;
  point_parameter_u : parameter_value;
  point_parameter_v : parameter_value;
END_ENTITY; -- point_on_surface

ENTITY polyline
  SUBTYPE OF (bounded_curve);
  points : LIST [2:?] OF cartesian_point;
END_ENTITY; -- polyline

ENTITY poly_loop
  SUBTYPE OF (loop, geometric_representation_item);
  polygon : LIST [3:?] OF UNIQUE cartesian_point;
END_ENTITY; -- poly_loop

ENTITY power_measure_with_unit
  SUBTYPE OF (measure_with_unit);
  WHERE
    WR1 : 'ISO_14306_SCHEMA.POWER_UNIT' IN
          TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- power_measure_with_unit

ENTITY power_unit
  SUBTYPE OF (derived_unit);
  WHERE
    WR1 : derive_dimensional_exponents(SELF) =
          dimensions_for_si_unit(si_unit_name.watt);
END_ENTITY; -- power_unit

ENTITY pressure_measure_with_unit
  SUBTYPE OF (measure_with_unit);
  WHERE
    WR1 : 'ISO_14306_SCHEMA.PRESSURE_UNIT' IN
          TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- pressure_measure_with_unit

ENTITY pressure_unit
  SUBTYPE OF (derived_unit);
  WHERE
    WR1 : derive_dimensional_exponents(SELF) =
          dimensions_for_si_unit(si_unit_name.pascal);

```

```

END_ENTITY; -- pressure_unit

ENTITY pre_defined_item;
    name : label;
END_ENTITY; -- pre_defined_item

ENTITY product;
    id : identifier;
    name : label;
    description : OPTIONAL text;
    frame_of_reference : SET [1:?] OF product_context;
END_ENTITY; -- product

ENTITY product_context
    SUBTYPE OF (application_context_element);
    discipline_type : label;
END_ENTITY; -- product_context

ENTITY product_definition;
    id : identifier;
    description : OPTIONAL text;
    formation : product_definition_formation;
    frame_of_reference : product_definition_context;
    DERIVE
        name : label := get_name_value(SELF);
    WHERE
        WR1 : SIZEOF(USEDIN(SELF,
                        'ISO_14306_SCHEMA.NAME_ATTRIBUTE.NAMED_ITEM')) <=
            1;
END_ENTITY; -- product_definition

ENTITY product_definition_context
    SUBTYPE OF (application_context_element);
    life_cycle_stage : label;
END_ENTITY; -- product_definition_context

ENTITY product_definition_context_association;
    definition : product_definition;
    frame_of_reference : product_definition_context;
    role : product_definition_context_role;
END_ENTITY; -- product_definition_context_association

ENTITY product_definition_context_role;
    name : label;
    description : OPTIONAL text;
END_ENTITY; -- product_definition_context_role

ENTITY product_definition_formation;
    id : identifier;
    description : OPTIONAL text;
    of_product : product;
    UNIQUE
        UR1 : id, of_product;
END_ENTITY; -- product_definition_formation

ENTITY product_definition_reference
    SUBTYPE OF (generic_product_definition_reference);
    product_id : identifier;
    product_definition_formation_id : identifier;
    product_definition_id : identifier;
    id_owning_organization_name : OPTIONAL label;
END_ENTITY; -- product_definition_reference

ENTITY product_definition_reference_with_local_representation
    SUBTYPE OF (generic_product_definition_reference, product_definition);
END_ENTITY; -- product_definition_reference_with_local_representation

```

```

ENTITY product_definition_relationship;
  id : identifier;
  name : label;
  description : OPTIONAL text;
  relating_product_definition : product_definition_or_reference;
  related_product_definition : product_definition_or_reference;
END_ENTITY; -- product_definition_relationship

ENTITY product_definition_shape
  SUBTYPE OF (property_definition);
  UNIQUE
    UR1: definition;
  WHERE
    WR1 : SIZEOF(['ISO_14306_SCHEMA.CHARACTERIZED_PRODUCT_DEFINITION',
                  'ISO_14306_SCHEMA.CHARACTERIZED_OBJECT'] *
                  TYPEOF(SELF\property_definition.definition)) >
      0;
END_ENTITY; -- product_definition_shape

ENTITY product_definition_usage
  SUBTYPE OF (product_definition_relationship);
  UNIQUE
    UR1: id, relating_product_definition, related_product_definition;
  WHERE
    WR1 : acyclic_product_definition_relationship(SELF,
[SELF\product_definition_relationship.related_product_definition],
'ISO_14306_SCHEMA.PRODUCT_DEFINITION_USAGE');
END_ENTITY; -- product_definition_usage

ENTITY promissory_usage_occurrence
  SUBTYPE OF (assembly_component_usage);
END_ENTITY; -- promissory_usage_occurrence

ENTITY property_definition;
  name : label;
  description : OPTIONAL text;
  definition : characterized_definition;
  DERIVE
    id : identifier := get_id_value(SELF);
  WHERE
    WR1 : SIZEOF(USEDIN(SELF,
                        'ISO_14306_SCHEMA.ID_ATTRIBUTE.IDENTIFIED_ITEM')) <=
      1;
END_ENTITY; -- property_definition

ENTITY property_definition_representation;
  definition : represented_definition;
  used_representation : representation;
  DERIVE
    description : text := get_description_value(SELF);
    name : label := get_name_value(SELF);
  WHERE
    WR1 : SIZEOF(USEDIN(SELF,
                        'ISO_14306_SCHEMA.DESCRPTION_ATTRIBUTE.DESCRIBED_ITEM')) <=
      1;
    WR2 : SIZEOF(USEDIN(SELF,
                        'ISO_14306_SCHEMA.NAME_ATTRIBUTE.NAMED_ITEM')) <=
      1;
END_ENTITY; -- property_definition_representation

ENTITY quantified_assembly_component_usage
  SUBTYPE OF (assembly_component_usage);
  quantity : measure_with_unit;
  WHERE
    WR1 : (NOT ('NUMBER' IN TYPEOF(quantity.value_component))) OR

```

```

        (quantity.value_component > 0);
END_ENTITY; -- quantified_assembly_component_usage

ENTITY quasi_uniform_curve
    SUBTYPE OF (b_spline_curve);
END_ENTITY; -- quasi_uniform_curve

ENTITY quasi_uniform_surface
    SUBTYPE OF (b_spline_surface);
END_ENTITY; -- quasi_uniform_surface

ENTITY radioactivity_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.RADIOACTIVITY_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- radioactivity_measure_with_unit

ENTITY radioactivity_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.becquerel);
END_ENTITY; -- radioactivity_unit

ENTITY rational_b_spline_curve
    SUBTYPE OF (b_spline_curve);
    weights_data : LIST [2:?] OF REAL;
    DERIVE
        weights : ARRAY [0:upper_index_on_control_points] OF REAL :=
list_to_array(weights_data,
    0,upper_index_on_control_points);
    WHERE
        WR1 : SIZEOF(weights_data) = SIZEOF(SELF\b_spline_curve.control_points_list);
        WR2 : curve_weights_positive(SELF);
END_ENTITY; -- rational_b_spline_curve

ENTITY rational_b_spline_surface
    SUBTYPE OF (b_spline_surface);
    weights_data : LIST [2:?] OF LIST [2:?] OF REAL;
    DERIVE
        weights : ARRAY [0:u_upper] OF ARRAY [0:v_upper] OF REAL :=
make_array_of_array(weights_data,
    0,u_upper,0,v_upper);
    WHERE
        WR1 : (SIZEOF(weights_data) =
            SIZEOF(SELF\b_spline_surface.control_points_list)) AND
            (SIZEOF(weights_data[1]) =
            SIZEOF(SELF\b_spline_surface.control_points_list[1]));
        WR2 : surface_weights_positive(SELF);
END_ENTITY; -- rational_b_spline_surface

ENTITY ratio_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.RATIO_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- ratio_measure_with_unit

ENTITY ratio_unit
    SUBTYPE OF (named_unit);
    WHERE
        WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
            (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
            (SELF\named_unit.dimensions.time_exponent = 0.0) AND
            (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
            (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =

```

```

        0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
        0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
        0.0);
END_ENTITY; -- ratio_unit

ENTITY representation;
    name : label;
    items : SET [1:?] OF representation_item;
    context_of_items : representation_context;
    DERIVE
        id : identifier := get_id_value(SELF);
        description : text := get_description_value(SELF);
    WHERE
        WR1 : SIZEOF(USEDIN(SELF,
                            'ISO_14306_SCHEMA.ID_ATTRIBUTE.IDENTIFIED_ITEM')) <=
            1;
        WR2 : SIZEOF(USEDIN(SELF,
                            'ISO_14306_SCHEMA.DESCRPTION_ATTRIBUTE.DESCRIBED_ITEM')) <=
            1;
END_ENTITY; -- representation

ENTITY representation_context;
    context_identifier : identifier;
    context_type : text;
    INVERSE
        representations_in_context : SET [1:?] OF representation FOR context_of_items;
END_ENTITY; -- representation_context

ENTITY representation_item
    SUPERTYPE OF (ONEOF(mapped_item, value_representation_item));
    name : label;
    WHERE
        WR1 : SIZEOF(using_representations(SELF)) > 0;
END_ENTITY; -- representation_item

ENTITY representation_item_relationship;
    name : label;
    description : OPTIONAL text;
    relating_representation_item : representation_item;
    related_representation_item : representation_item;
END_ENTITY; -- representation_item_relationship

ENTITY representation_map;
    mapping_origin : representation_item;
    mapped_representation : representation;
    INVERSE
        map_usage : SET [1:?] OF mapped_item FOR mapping_source;
    WHERE
        WR1 : item_in_context(SELF.mapping_origin,
                             SELF.mapped_representation.context_of_items);
END_ENTITY; -- representation_map

ENTITY representation_relationship;
    name : label;
    description : OPTIONAL text;
    rep_1 : representation;
    rep_2 : representation;
END_ENTITY; -- representation_relationship

ENTITY resistance_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.RESISTANCE_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- resistance_measure_with_unit

ENTITY resistance_unit

```

```

        SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensions_for_si_unit(si_unit_name.ohm);
    END_ENTITY; -- resistance_unit

    ENTITY role_association;
        role : object_role;
        item_with_role : role_select;
    END_ENTITY; -- role_association

    ENTITY shape_aspect;
        name : label;
        description : OPTIONAL text;
        of_shape : product_definition_shape;
        product_definitional : LOGICAL;
    DERIVE
        id : identifier := get_id_value(SELF);
    UNIQUE
        UR1: id, of_shape;
    WHERE
        WR1 : SIZEOF(USEDIN(SELF,
            'ISO_14306_SCHEMA.ID_ATTRIBUTE.IDENTIFIED_ITEM')) <=
            1;
    END_ENTITY; -- shape_aspect

    ENTITY shape_aspect_relationship;
        name : label;
        description : OPTIONAL text;
        relating_shape_aspect : shape_aspect;
        related_shape_aspect : shape_aspect;
    DERIVE
        id : identifier := get_id_value(SELF);
    WHERE
        WR1 : SIZEOF(USEDIN(SELF,
            'ISO_14306_SCHEMA.ID_ATTRIBUTE.IDENTIFIED_ITEM')) <=
            1;
    END_ENTITY; -- shape_aspect_relationship

    ENTITY shape_definition_representation
        SUBTYPE OF (property_definition_representation);
        SELF\property_definition_representation.definition : property_definition;
        SELF\property_definition_representation.used_representation :
    shape_representation;
    WHERE
        WR1 : ('ISO_14306_OPEN_JT_SHAPE.PRODUCT_DEFINITION_SHAPE' IN
            TYPEOF(definition)) OR ('ISO_14306_OPEN_JT_SHAPE.SHAPE_DEFINITION' IN
            TYPEOF(definition\property_definition.definition));
    END_ENTITY; -- shape_definition_representation

    ENTITY shape_representation
        SUBTYPE OF (representation);
    END_ENTITY; -- shape_representation

    ENTITY shape_representation_relationship
        SUBTYPE OF (representation_relationship);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.SHAPE_REPRESENTATION' IN
            (TYPEOF(SELF\representation_relationship.rep_1) +
            TYPEOF(SELF\representation_relationship.rep_2));
    END_ENTITY; -- shape_representation_relationship

    ENTITY simple_generic_expression
        ABSTRACT SUPERTYPE OF (ONEOF(generic_literal, generic_variable))
        SUBTYPE OF (generic_expression);
    END_ENTITY; -- simple_generic_expression

```

```

ENTITY si_absorbed_dose_unit
    SUBTYPE OF (absorbed_dose_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.gray;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_absorbed_dose_unit

ENTITY si_capacitance_unit
    SUBTYPE OF (capacitance_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.farad;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_capacitance_unit

ENTITY si_conductance_unit
    SUBTYPE OF (conductance_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.siemens;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_conductance_unit

ENTITY si_dose_equivalent_unit
    SUBTYPE OF (dose_equivalent_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.sievert;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_dose_equivalent_unit

ENTITY si_electric_charge_unit
    SUBTYPE OF (electric_charge_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.coulomb;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_electric_charge_unit

ENTITY si_electric_potential_unit
    SUBTYPE OF (electric_potential_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.volt;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_electric_potential_unit

ENTITY si_energy_unit
    SUBTYPE OF (energy_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.joule;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_energy_unit

ENTITY si_force_unit
    SUBTYPE OF (force_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.newton;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_force_unit

ENTITY si_frequency_unit
    SUBTYPE OF (frequency_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.hertz;
        WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_frequency_unit

ENTITY si_illuminance_unit
    SUBTYPE OF (illuminance_unit, si_unit);
    WHERE
        WR1 : SELF\si_unit.name = si_unit_name.lux;

```

```

    WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_illuminance_unit

ENTITY si_inductance_unit
    SUBTYPE OF (inductance_unit, si_unit);
WHERE
    WR1 : SELF\si_unit.name = si_unit_name.henry;
    WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_inductance_unit

ENTITY si_magnetic_flux_density_unit
    SUBTYPE OF (magnetic_flux_density_unit, si_unit);
WHERE
    WR1 : SELF\si_unit.name = si_unit_name.tesla;
    WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_magnetic_flux_density_unit

ENTITY si_magnetic_flux_unit
    SUBTYPE OF (magnetic_flux_unit, si_unit);
WHERE
    WR1 : SELF\si_unit.name = si_unit_name.weber;
    WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_magnetic_flux_unit

ENTITY si_power_unit
    SUBTYPE OF (power_unit, si_unit);
WHERE
    WR1 : SELF\si_unit.name = si_unit_name.watt;
    WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_power_unit

ENTITY si_pressure_unit
    SUBTYPE OF (pressure_unit, si_unit);
WHERE
    WR1 : SELF\si_unit.name = si_unit_name.pascal;
    WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_pressure_unit

ENTITY si_radioactivity_unit
    SUBTYPE OF (radioactivity_unit, si_unit);
WHERE
    WR1 : SELF\si_unit.name = si_unit_name.becquerel;
    WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_radioactivity_unit

ENTITY si_resistance_unit
    SUBTYPE OF (resistance_unit, si_unit);
WHERE
    WR1 : SELF\si_unit.name = si_unit_name.ohm;
    WR2 : NOT EXISTS(SELF\derived_unit.name);
END_ENTITY; -- si_resistance_unit

ENTITY si_unit
    SUBTYPE OF (named_unit);
    prefix : OPTIONAL si_prefix;
    name : si_unit_name;
    DERIVE
        SELF\named_unit.dimensions : dimensional_exponents :=
dimensions_for_si_unit(name);
WHERE
    WR1 : NOT (('ISO_14306_SCHEMA.MASS_UNIT' IN TYPEOF(SELF)) AND
        (SIZEOF(USEDIN(SELF,
            'ISO_14306_SCHEMA.DERIVED_UNIT_ELEMENT.UNIT')) >
            0)) OR
        (prefix = si_prefix.kilo);
END_ENTITY; -- si_unit

```



```

ENTITY solid_angle_measure_with_unit
  SUBTYPE OF (measure_with_unit);
  WHERE
    WR1 : 'ISO_14306_SCHEMA.SOLID_ANGLE_UNIT' IN
      TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- solid_angle_measure_with_unit

ENTITY solid_angle_unit
  SUBTYPE OF (named_unit);
  WHERE
    WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
      (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
      (SELF\named_unit.dimensions.time_exponent = 0.0) AND
      (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
      (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
        0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
        0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
        0.0);
END_ENTITY; -- solid_angle_unit

ENTITY solid_model
  SUBTYPE OF (geometric_representation_item);
END_ENTITY; -- solid_model

ENTITY specified_higher_usage_occurrence
  SUBTYPE OF (assembly_component_usage);
  upper_usage : assembly_component_usage;
  next_usage : next_assembly_usage_occurrence;
  UNIQUE
    UR1: upper_usage, next_usage;
  WHERE
    WR1 : SELF <>: upper_usage;
    WR2 : SELF\product_definition_relationship.relatng_product_definition ==:
      upper_usage.relatng_product_definition;
    WR3 : SELF\product_definition_relationship.related_product_definition ==:
      next_usage.related_product_definition;
    WR4 : (upper_usage.related_product_definition ==:
      next_usage.relatng_product_definition) OR
      (SIZEOF(QUERY(pdr < *
        USEDIN(upper_usage.related_product_definition,
'ISO_14306_SCHEMA.PRODUCT_DEFINITION_RELATIONSHIP.RELATED_PRODUCT_DEFINITION') |
      pdr.relatng_product_definition ==:
      next_usage.relatng_product_definition)) =
      1);
    WR5 : SIZEOF(['ISO_14306_SCHEMA.NEXT_ASSEMBLY_USAGE_OCCURRENCE',
      'ISO_14306_SCHEMA.SPECIFIED_HIGHER_USAGE_OCCURRENCE'] *
      TYPEOF(upper_usage)) =
      1;
END_ENTITY; -- specified_higher_usage_occurrence

ENTITY spherical_surface
  SUBTYPE OF (elementary_surface);
  radius : positive_length_measure;
END_ENTITY; -- spherical_surface

ENTITY surface
  SUPERTYPE OF (ONEOF(elementary_surface, swept_surface, bounded_surface))
  SUBTYPE OF (geometric_representation_item);
END_ENTITY; -- surface

ENTITY surface_curve
  SUBTYPE OF (curve);
  curve_3d : curve;
  associated_geometry : LIST [1:2] OF pcurve_or_surface;
  master_representation : preferred_surface_curve_representation;
  DERIVE

```

```

    basis_surface : SET [1:2] OF surface := get_basis_surface(SELF);
WHERE
    WR1 : curve_3d.dim = 3;
    WR2 : ('ISO_14306_SCHEMA.PCURVE' IN
        TYPEOF(associated_geometry[1])) OR (master_representation <>
        pcurve_s1);
    WR3 : ('ISO_14306_SCHEMA.PCURVE' IN
        TYPEOF(associated_geometry[2])) OR (master_representation <>
        pcurve_s2);
    WR4 : NOT ('ISO_14306_SCHEMA.PCURVE' IN TYPEOF(curve_3d));
END_ENTITY; -- surface_curve

ENTITY surface_of_linear_extrusion
    SUBTYPE OF (swept_surface);
    extrusion_axis : vector;
END_ENTITY; -- surface_of_linear_extrusion

ENTITY surface_of_revolution
    SUBTYPE OF (swept_surface);
    axis_position : axis1_placement;
    DERIVE
        axis_line : line := representation_item('') ||
            geometric_representation_item() || curve() ||
            line(axis_position.location,
                representation_item('') ||
                geometric_representation_item() ||
                vector(axis_position.z, 1.0));
END_ENTITY; -- surface_of_revolution

ENTITY swept_surface
    SUPERTYPE OF (ONEOF(surface_of_linear_extrusion, surface_of_revolution))
    SUBTYPE OF (surface);
    swept_curve : curve;
END_ENTITY; -- swept_surface

ENTITY thermal_resistance_measure_with_unit
    SUBTYPE OF (measure_with_unit);
WHERE
    WR1 : 'ISO_14306_SCHEMA.THERMAL_RESISTANCE_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- thermal_resistance_measure_with_unit

ENTITY thermal_resistance_unit
    SUBTYPE OF (derived_unit);
WHERE
    WR1 : derive_dimensional_exponents(SELF) =
        dimensional_exponents(-1.0, -1.0, -3.0, 0.0, 1.0, 0.0, 0.0);
END_ENTITY; -- thermal_resistance_unit

ENTITY thermodynamic_temperature_measure_with_unit
    SUBTYPE OF (measure_with_unit);
WHERE
    WR1 : 'ISO_14306_SCHEMA.THERMODYNAMIC_TEMPERATURE_UNIT' IN
        TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- thermodynamic_temperature_measure_with_unit

ENTITY thermodynamic_temperature_unit
    SUBTYPE OF (named_unit);
WHERE
    WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
        (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
        (SELF\named_unit.dimensions.time_exponent = 0.0) AND
        (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
        (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
            1.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
            0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
            0.0);

```

```

END_ENTITY; -- thermodynamic_temperature_unit

ENTITY time_measure_with_unit
  SUBTYPE OF (measure_with_unit);
  WHERE
    WR1 : 'ISO_14306_SCHEMA.TIME_UNIT' IN
      TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- time_measure_with_unit

ENTITY time_unit
  SUBTYPE OF (named_unit);
  WHERE
    WR1 : (SELF\named_unit.dimensions.length_exponent = 0.0) AND
      (SELF\named_unit.dimensions.mass_exponent = 0.0) AND
      (SELF\named_unit.dimensions.time_exponent = 1.0) AND
      (SELF\named_unit.dimensions.electric_current_exponent = 0.0) AND
      (SELF\named_unit.dimensions.thermodynamic_temperature_exponent =
        0.0) AND (SELF\named_unit.dimensions.amount_of_substance_exponent =
        0.0) AND (SELF\named_unit.dimensions.luminous_intensity_exponent =
        0.0);
END_ENTITY; -- time_unit

ENTITY topological_representation_item
  SUPERTYPE OF (ONEOF(vertex, edge, face_bound, face, connected_edge_set,
    connected_face_set, loop ANDOR path))
  SUBTYPE OF (representation_item);
END_ENTITY; -- topological_representation_item

ENTITY toroidal_surface
  SUBTYPE OF (elementary_surface);
  major_radius : positive_length_measure;
  minor_radius : positive_length_measure;
END_ENTITY; -- toroidal_surface

ENTITY trimmed_curve
  SUBTYPE OF (bounded_curve);
  basis_curve : curve;
  trim_1 : SET [1:2] OF trimming_select;
  trim_2 : SET [1:2] OF trimming_select;
  sense_agreement : BOOLEAN;
  master_representation : trimming_preference;
  WHERE
    WR1 : (HIINDEX(trim_1) = 1) OR (TYPEOF(trim_1[1]) <> TYPEOF(trim_1[2]));
    WR2 : (HIINDEX(trim_2) = 1) OR (TYPEOF(trim_2[1]) <> TYPEOF(trim_2[2]));
END_ENTITY; -- trimmed_curve

ENTITY unary_generic_expression
  ABSTRACT SUPERTYPE
  SUBTYPE OF (generic_expression);
  operand : generic_expression;
END_ENTITY; -- unary_generic_expression

ENTITY uncertainty_measure_with_unit
  SUBTYPE OF (measure_with_unit);
  name : label;
  description : OPTIONAL text;
  WHERE
    WR1 : valid_measure_value(SELF\measure_with_unit.value_component);
END_ENTITY; -- uncertainty_measure_with_unit

ENTITY uniform_curve
  SUBTYPE OF (b_spline_curve);
END_ENTITY; -- uniform_curve

ENTITY uniform_surface
  SUBTYPE OF (b_spline_surface);
END_ENTITY; -- uniform_surface

```

```

ENTITY value_representation_item
    SUBTYPE OF (representation_item);
    value_component : measure_value;
    WHERE
        WR1 : SIZEOF(QUERY(rep <* using representations(SELF) |
                            NOT ('ISO_14306_SCHEMA.GLOBAL_UNIT_ASSIGNED_CONTEXT' IN
                                TYPEOF(rep.context_of_items)))) = 0;
END_ENTITY; -- value_representation_item

ENTITY variable_semantics
    ABSTRACT SUPERTYPE;
END_ENTITY; -- variable_semantics

ENTITY vector
    SUBTYPE OF (geometric_representation_item);
    orientation : direction;
    magnitude : length_measure;
    WHERE
        WR1 : magnitude >= 0.0;
END_ENTITY; -- vector

ENTITY velocity_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.VELOCITY_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- velocity_measure_with_unit

ENTITY velocity_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensional_exponents(1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0);
END_ENTITY; -- velocity_unit

ENTITY vertex
    SUBTYPE OF (topological_representation_item);
END_ENTITY; -- vertex

ENTITY vertex_loop
    SUBTYPE OF (loop);
    loop_vertex : vertex;
END_ENTITY; -- vertex_loop

ENTITY vertex_point
    SUBTYPE OF (vertex, geometric_representation_item);
    vertex_geometry : point;
END_ENTITY; -- vertex_point

ENTITY volume_measure_with_unit
    SUBTYPE OF (measure_with_unit);
    WHERE
        WR1 : 'ISO_14306_SCHEMA.VOLUME_UNIT' IN
            TYPEOF(SELF\measure_with_unit.unit_component);
END_ENTITY; -- volume_measure_with_unit

ENTITY volume_unit
    SUBTYPE OF (derived_unit);
    WHERE
        WR1 : derive_dimensional_exponents(SELF) =
            dimensional_exponents(3.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
END_ENTITY; -- volume_unit

FUNCTION acyclic(arg1 : generic_expression; arg2 : SET OF generic_expression) :
    BOOLEAN;
    LOCAL

```

```

    result : BOOLEAN := true;
END_LOCAL;
IF ('ISO_14306_SCHEMA.SIMPLE_GENERIC_EXPRESSION' IN TYPEOF(arg1))
THEN
    RETURN (true);
END_IF;
IF arg1 IN arg2
THEN
    RETURN (false);
END_IF;
IF 'ISO_14306_SCHEMA.UNARY_GENERIC_EXPRESSION' IN TYPEOF(arg1)
THEN
    RETURN (acyclic(arg1\unary_generic_expression.operand, arg2 + [arg1]));
END_IF;
IF 'ISO_14306_SCHEMA.BINARY_GENERIC_EXPRESSION' IN TYPEOF(arg1)
THEN
    RETURN (acyclic(arg1\binary_generic_expression.operands[1],
                    arg2 + [arg1]) AND
            acyclic(arg1\binary_generic_expression.operands[2], arg2 + [arg1]));
END_IF;
IF 'ISO_14306_SCHEMA.MULTIPLE_ARITY_GENERIC_EXPRESSION' IN
    TYPEOF(arg1)
THEN
    result := true;
    REPEAT
        i := 1 TO SIZEOF(arg1\multiple_arity_generic_expression.operands);
        result :=
            result AND
            acyclic(arg1\multiple_arity_generic_expression.operands[i],
                    arg2 + [arg1]);
    END_REPEAT;
    RETURN (result);
END_IF;
RETURN (result);
END_FUNCTION; -- acyclic

FUNCTION acyclic_mapped_representation(mi : mapped_item) : BOOLEAN;
LOCAL
    rms : SET OF representation_map;
    mis : SET OF mapped_item;
    rs1 : SET OF representation;
    rs2 : SET OF representation;
END_LOCAL;
rs1 := using_representations(mi);
rs2 := [];
REPEAT
    WHILE SIZEOF(rs1) > 0;
    REPEAT
        i := 1 TO HIINDEX(rs1);
        rms :=
            bag_to_set(USEDIN(rs1[i],
                'ISO_14306_SCHEMA.REPRESENTATION_MAP.MAPPED_REPRESENTATION'));
        IF SIZEOF(rms) > 0
        THEN
            REPEAT
                j := 1 TO HIINDEX(rms);
                mis :=
                    bag_to_set(USEDIN(rms[j],
                        'ISO_14306_SCHEMA.MAPPED_ITEM.MAPPING_SOURCE'));
                IF SIZEOF(mis) > 0
                THEN
                    REPEAT
                        j := 1 TO HIINDEX(mis);
                        IF mis[j] := mi
                        THEN
                            RETURN (false);
                        END_REPEAT;
                    END_REPEAT;
                END_IF;
            END_REPEAT;
        END_IF;
    END_REPEAT;
    rs1 := rs2;
    rs2 := [];
END_REPEAT;
RETURN (true);
END_FUNCTION;

```

```

        END_IF;
        rs2 := rs2 + using_representations(mis[i]);
    END_REPEAT;
    END_IF;
    END_REPEAT;
    END_IF;
    rs1 := rs2;
    rs2 := [];
    END_REPEAT;
    RETURN (true);
END_FUNCTION; -- acyclic_mapped_representation

FUNCTION acyclic_product_definition_relationship(relation :
product_definition_relationship;
                                                    relatives : SET [1:?] OF
product_definition_or_reference;
                                                    specific_relation : STRING) :
BOOLEAN;
LOCAL
    x : SET OF product_definition_relationship;
END_LOCAL;
IF relation.relating_product_definition IN relatives
THEN
    RETURN (false);
END_IF;
x :=
    QUERY(pd <*
        bag_to_set(USEDIN(relation.relating_product_definition,
'ISO_14306_SCHEMA.PRODUCT_DEFINITION_RELATIONSHIP.RELATED_PRODUCT_DEFINITION')) |
        specific_relation IN TYPEOF(pd));
REPEAT
    i := 1 TO HIINDEX(x);
    IF NOT acyclic_product_definition_relationship(x[i],
                                                    relatives +
relation.relating_product_definition,
                                                    specific_relation)
    THEN
        RETURN (false);
    END_IF;
END_REPEAT;
RETURN (true);
END_FUNCTION; -- acyclic_product_definition_relationship

FUNCTION acyclic_representation_relationship(relation :
representation_relationship;
                                                    relatives : SET [1:?] OF
representation;
                                                    specific_relation : STRING) :
BOOLEAN;
LOCAL
    x : SET OF representation_relationship;
END_LOCAL;
IF relation.rep_1 IN relatives
THEN
    RETURN (false);
END_IF;
x :=
    QUERY(r <*
        bag_to_set(USEDIN(relation.rep_1,
'ISO_14306_SCHEMA.REPRESENTATION_RELATIONSHIP.REP_2')) |
        specific_relation IN TYPEOF(r));
REPEAT
    i := 1 TO HIINDEX(x);
    IF NOT acyclic_representation_relationship(x[i],

```

```

relatives + relation.rep_1,
specific_relation)

THEN
    RETURN (false);
END_IF;
END_REPEAT;
RETURN (true);
END_FUNCTION; -- acyclic_representation_relationship

FUNCTION associated_surface(arg : pcurve_or_surface) : surface;
LOCAL
    surf : surface;
END_LOCAL;
IF 'ISO_14306_SCHEMA.PCURVE' IN TYPEOF(arg)
THEN
    surf := arg\pcurve.basis_surface;
ELSE
    surf := arg;
END_IF;
RETURN (surf);
END_FUNCTION; -- associated_surface

FUNCTION bag_to_set(the_bag : BAG OF GENERIC:intype) : SET OF GENERIC:intype;
LOCAL
    the_set : SET OF GENERIC:intype := [];
END_LOCAL;
IF SIZEOF(the_bag) > 0
THEN
    REPEAT
        i := 1 TO HIINDEX(the_bag);
        the_set := the_set + the_bag[i];
    END_REPEAT;
END_IF;
RETURN (the_set);
END_FUNCTION; -- bag_to_set

FUNCTION base_axis(dim : INTEGER; axis1 : direction; axis2 : direction;
                  axis3 : direction) :
LIST [2:3] OF direction;
LOCAL
    u : LIST [2:3] OF direction;
    factor : REAL;
    d1 : direction;
    d2 : direction;
END_LOCAL;
IF (dim = 3)
THEN
    d1 := NVL(normalise(axis3), dummy_gri || direction([0.0, 0.0, 1.0]));
    d2 := first_proj_axis(d1, axis1);
    u := [d2, second_proj_axis(d1, d2, axis2), d1];
ELSE
    IF EXISTS(axis1)
    THEN
        d1 := normalise(axis1);
        u := [d1, orthogonal_complement(d1)];
        IF EXISTS(axis2)
        THEN
            factor := dot_product(axis2, u[2]);
            IF (factor < 0.0)
            THEN
                u[2].direction_ratios[1] := -u[2].direction_ratios[1];
                u[2].direction_ratios[2] := -u[2].direction_ratios[2];
            END_IF;
        END_IF;
    ELSE
        IF EXISTS(axis2)
        THEN

```

```

        d1 := normalise(axis2);
        u := [orthogonal_complement(d1), d1];
        u[1].direction_ratios[1] := -u[1].direction_ratios[1];
        u[1].direction_ratios[2] := -u[1].direction_ratios[2];
    ELSE
        u :=
            [dummy_gri || direction([1.0, 0.0]),
             dummy_gri || direction([0.0, 1.0])];
    END_IF;
END_IF;
RETURN (u);
END_FUNCTION; -- base_axis

FUNCTION boolean_choose(b : BOOLEAN; choice1 : GENERIC:item;
                        choice2 : GENERIC:item) :
GENERIC:item;
    IF b
    THEN
        RETURN (choice1);
    ELSE
        RETURN (choice2);
    END_IF;
END_FUNCTION; -- boolean_choose

FUNCTION build_2axes(ref_direction : direction) : LIST [2:2] OF direction;
    LOCAL
        d : direction :=
            NVL(normalise(ref_direction), dummy_gri || direction([1.0, 0.0]));
    END_LOCAL;
    RETURN ([d, orthogonal_complement(d)]);
END_FUNCTION; -- build_2axes

FUNCTION build_axes(axis : direction; ref_direction : direction) :
LIST [3:3] OF direction;
    LOCAL
        d1 : direction;
        d2 : direction;
    END_LOCAL;
    d1 := NVL(normalise(axis), dummy_gri || direction([0.0, 0.0, 1.0]));
    d2 := first_proj_axis(d1, ref_direction);
    RETURN ([d2, normalise(cross_product(d1, d2))\vector.orientation, d1]);
END_FUNCTION; -- build_axes

FUNCTION closed_shell_reversed(a_shell : closed_shell) : oriented_closed_shell;
    LOCAL
        the_reverse : oriented_closed_shell;
    END_LOCAL;
    IF ('ISO_14306_SCHEMA.ORIENTED_CLOSED_SHELL' IN TYPEOF(a_shell))
    THEN
        the_reverse :=
            dummy_tri || connected_face_set(a_shell\connected_face_set.cfs_faces) ||
            closed_shell() ||
            oriented_closed_shell(a_shell\oriented_closed_shell.closed_shell_element,
                                NOT (a_shell\oriented_closed_shell.orientation));
    ELSE
        the_reverse :=
            dummy_tri || connected_face_set(a_shell\connected_face_set.cfs_faces) ||
            closed_shell() || oriented_closed_shell(a_shell, false);
    END_IF;
    RETURN (the_reverse);
END_FUNCTION; -- closed_shell_reversed

FUNCTION conditional_reverse(p : BOOLEAN; an_item : reversible_topology) :
reversible_topology;
    IF p
    THEN

```



```

    RETURN (an_item);
ELSE
    RETURN (topology_reversed(an_item));
END_IF;
END_FUNCTION; -- conditional_reverse

FUNCTION constraints_composite_curve_on_surface(c : composite_curve_on_surface) :
BOOLEAN;
LOCAL
    n_segments : INTEGER := SIZEOF(c.segments);
END_LOCAL;
REPEAT
    k := 1 TO n_segments;
    IF (NOT ('ISO_14306_SCHEMA.PCURVE' IN
        TYPEOF(c\composite_curve.segments[k].parent_curve))) AND
        (NOT ('ISO_14306_SCHEMA.SURFACE_CURVE' IN
        TYPEOF(c\composite_curve.segments[k].parent_curve))) AND
        (NOT ('ISO_14306_SCHEMA.COMPOSITE_CURVE_ON_SURFACE' IN
        TYPEOF(c\composite_curve.segments[k].parent_curve)))
    THEN
        RETURN (false);
    END IF;
END_REPEAT;
RETURN (true);
END_FUNCTION; -- constraints_composite_curve_on_surface

FUNCTION constraints_param_b_spline(degree : INTEGER; up_knots : INTEGER;
    up_cp : INTEGER;
    knot_mult : LIST [0:?] OF INTEGER;
    knots : LIST [0:?] OF parameter_value) :
BOOLEAN;
LOCAL
    result : BOOLEAN := true;
    k : INTEGER;
    sum : INTEGER;
END_LOCAL;
sum := knot_mult[1];
REPEAT
    i := 2 TO up_knots;
    sum := sum + knot_mult[i];
END_REPEAT;
IF (degree < 1) OR (up_knots < 2) OR (up_cp < degree) OR (sum <>
    (degree + up_cp + 2))
THEN
    result := false;
    RETURN (result);
END_IF;
k := knot_mult[1];
IF (k < 1) OR (k > degree + 1)
THEN
    result := false;
    RETURN (result);
END_IF;
REPEAT
    i := 2 TO up_knots;
    IF (knot_mult[i] < 1) OR (knots[i] <= knots[i - 1])
    THEN
        result := false;
        RETURN (result);
    END_IF;
    k := knot_mult[i];
    IF (i < up_knots) AND (k > degree)
    THEN
        result := false;
        RETURN (result);
    END_IF;
    IF (i = up_knots) AND (k > degree + 1)

```

```

    THEN
        result := false;
        RETURN (result);
    END_IF;
END_REPEAT;
RETURN (result);
END_FUNCTION; -- constraints_param_b_spline

FUNCTION cross_product(arg1 : direction; arg2 : direction) : vector;
LOCAL
    mag : REAL;
    res : direction;
    v1 : LIST [3:3] OF REAL;
    v2 : LIST [3:3] OF REAL;
    result : vector;
END_LOCAL;
IF (NOT EXISTS(arg1) OR (arg1.dim = 2)) OR
   (NOT EXISTS(arg2) OR (arg2.dim = 2))
THEN
    RETURN (?);
ELSE
    BEGIN
        v1 := normalise(arg1).direction_ratios;
        v2 := normalise(arg2).direction_ratios;
        res :=
            dummy_gri ||
            direction([(v1[2] * v2[3] - v1[3] * v2[2]),
                       (v1[3] * v2[1] - v1[1] * v2[3]),
                       (v1[1] * v2[2] - v1[2] * v2[1])]);
        mag := 0.0;
        REPEAT
            i := 1 TO 3;
            mag := mag + res.direction_ratios[i] * res.direction_ratios[i];
        END_REPEAT;
        IF (mag > 0.0)
        THEN
            result := dummy_gri || vector(res, SQRT(mag));
        ELSE
            result := dummy_gri || vector(arg1, 0.0);
        END_IF;
        RETURN (result);
    END;
END_IF;
END_FUNCTION; -- cross_product

FUNCTION curve_weights_positive(b : rational_b_spline_curve) : BOOLEAN;
LOCAL
    result : BOOLEAN := true;
END_LOCAL;
REPEAT
    i := 0 TO b.upper_index_on_control_points;
    IF b.weights[i] <= 0.0
    THEN
        result := false;
        RETURN (result);
    END_IF;
END_REPEAT;
RETURN (result);
END_FUNCTION; -- curve_weights_positive

FUNCTION derive_dimensional_exponents(x : unit) : dimensional_exponents;
LOCAL
    result : dimensional_exponents :=
        dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
END_LOCAL;
IF 'ISO_14306_SCHEMA.DERIVED_UNIT' IN TYPEOF(x)
THEN

```

```

REPEAT
    i := LOINDEX(x\derived_unit.elements) TO HIINDEX(x\derived_unit.elements);
    result.length_exponent :=
    result.length_exponent +
    (x\derived_unit.elements[i]\derived_unit_element.exponent *
x\derived_unit.elements[i]\derived_unit_element.unit\named_unit.dimensions.length_e
xponent);
    result.mass_exponent :=
    result.mass_exponent +
    (x\derived_unit.elements[i]\derived_unit_element.exponent *
x\derived_unit.elements[i]\derived_unit_element.unit\named_unit.dimensions.mass_exp
onent);
    result.time_exponent :=
    result.time_exponent +
    (x\derived_unit.elements[i]\derived_unit_element.exponent *
x\derived_unit.elements[i]\derived_unit_element.unit\named_unit.dimensions.time_exp
onent);
    result.electric_current_exponent :=
    result.electric_current_exponent +
    (x\derived_unit.elements[i]\derived_unit_element.exponent *
x\derived_unit.elements[i]\derived_unit_element.unit\named_unit.dimensions.electric
_current_exponent);
    result.thermodynamic_temperature_exponent :=
    result.thermodynamic_temperature_exponent +
    (x\derived_unit.elements[i]\derived_unit_element.exponent *
x\derived_unit.elements[i]\derived_unit_element.unit\named_unit.dimensions.thermody
namic_temperature_exponent);
    result.amount_of_substance_exponent :=
    result.amount_of_substance_exponent +
    (x\derived_unit.elements[i]\derived_unit_element.exponent *
x\derived_unit.elements[i]\derived_unit_element.unit\named_unit.dimensions.amount_o
f_substance_exponent);
    result.luminous_intensity_exponent :=
    result.luminous_intensity_exponent +
    (x\derived_unit.elements[i]\derived_unit_element.exponent *
x\derived_unit.elements[i]\derived_unit_element.unit\named_unit.dimensions.luminous
_intensity_exponent);
END_REPEAT;
ELSE
    result := x\named_unit.dimensions;
END_IF;
RETURN (result);
END_FUNCTION; -- derive_dimensional_exponents

FUNCTION dimensions_for_si_unit(n : si_unit_name) : dimensional_exponents;
CASE n OF
    metre : RETURN (dimensional_exponents(1.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0));
    gram : RETURN (dimensional_exponents(0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
0.0));
    second : RETURN (dimensional_exponents(0.0, 0.0, 1.0, 0.0, 0.0,
0.0, 0.0));
    ampere : RETURN (dimensional_exponents(0.0, 0.0, 0.0, 1.0, 0.0,
0.0, 0.0));
    kelvin : RETURN (dimensional_exponents(0.0, 0.0, 0.0, 0.0, 1.0,
0.0, 0.0));
    mole : RETURN (dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
0.0));
    candela : RETURN (dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 1.0));

```

```

radian : RETURN (dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0,
                                          0.0, 0.0));
steradian : RETURN (dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0,
                                           0.0, 0.0));
hertz : RETURN (dimensional_exponents(0.0, 0.0, -1.0, 0.0, 0.0,
                                       0.0, 0.0));
newton : RETURN (dimensional_exponents(1.0, 1.0, -2.0, 0.0, 0.0,
                                       0.0, 0.0));
pascal : RETURN (dimensional_exponents(-1.0, 1.0, -2.0, 0.0, 0.0,
                                       0.0, 0.0));
joule : RETURN (dimensional_exponents(2.0, 1.0, -2.0, 0.0, 0.0,
                                       0.0, 0.0));
watt : RETURN (dimensional_exponents(2.0, 1.0, -3.0, 0.0, 0.0,
                                       0.0, 0.0));
coulomb : RETURN (dimensional_exponents(0.0, 0.0, 1.0, 1.0, 0.0,
                                         0.0, 0.0));
volt : RETURN (dimensional_exponents(2.0, 1.0, -3.0, -1.0, 0.0,
                                       0.0, 0.0));
farad : RETURN (dimensional_exponents(-2.0, -1.0, 4.0, 2.0, 0.0,
                                       0.0, 0.0));
ohm : RETURN (dimensional_exponents(2.0, 1.0, -3.0, -2.0, 0.0,
                                       0.0, 0.0));
siemens : RETURN (dimensional_exponents(-2.0, -1.0, 3.0, 2.0, 0.0,
                                         0.0, 0.0));
weber : RETURN (dimensional_exponents(2.0, 1.0, -2.0, -1.0, 0.0,
                                       0.0, 0.0));
tesla : RETURN (dimensional_exponents(0.0, 1.0, -2.0, -1.0, 0.0,
                                       0.0, 0.0));
henry : RETURN (dimensional_exponents(2.0, 1.0, -2.0, -2.0, 0.0,
                                       0.0, 0.0));
degree_celsius : RETURN (dimensional_exponents(0.0, 0.0, 0.0, 0.0,
                                                1.0, 0.0, 0.0));
lumen : RETURN (dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0,
                                       0.0, 1.0));
lux : RETURN (dimensional_exponents(-2.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                                     1.0));
becquerel : RETURN (dimensional_exponents(0.0, 0.0, -1.0, 0.0,
                                           0.0, 0.0, 0.0));
gray : RETURN (dimensional_exponents(2.0, 0.0, -2.0, 0.0, 0.0,
                                       0.0, 0.0));
sievert : RETURN (dimensional_exponents(2.0, 0.0, -2.0, 0.0, 0.0,
                                         0.0, 0.0));

OTHERWISE : RETURN (?);
END_CASE;
END_FUNCTION; -- dimensions_for_si_unit

FUNCTION dimension_of(item : geometric_representation_item) : dimension_count;
LOCAL
  x : SET OF representation;
  y : representation_context;
  dim : dimension_count;
END_LOCAL;
IF 'ISO_14306_SCHEMA.CARTESIAN_POINT' IN TYPEOF(item)
THEN
  dim := SIZEOF(item\cartesian_point.coordinates);
  RETURN (dim);
END_IF;
IF 'ISO_14306_SCHEMA.DIRECTION' IN TYPEOF(item)
THEN
  dim := SIZEOF(item\direction.direction_ratios);
  RETURN (dim);
END_IF;
IF 'ISO_14306_SCHEMA.VECTOR' IN TYPEOF(item)
THEN
  dim := SIZEOF(item\vector.orientation.direction.direction_ratios);
  RETURN (dim);
END_IF;

```

```

x := using_representations(item);
IF (SIZEOF(x) > 0)
THEN
  y := x[1].context_of_items;
  dim := y\geometric_representation_context.coordinate_space_dimension;
  RETURN (dim);
ELSE
  RETURN (?);
END_IF;
END_FUNCTION; -- dimension_of

FUNCTION dot_product(arg1 : direction; arg2 : direction) : REAL;
LOCAL
  scalar : REAL;
  vec1 : direction;
  vec2 : direction;
  ndim : INTEGER;
END_LOCAL;
IF NOT EXISTS(arg1) OR NOT EXISTS(arg2)
THEN
  scalar := ?;
ELSE
  IF (arg1.dim <> arg2.dim)
  THEN
    scalar := ?;
  ELSE
    BEGIN
      vec1 := normalise(arg1);
      vec2 := normalise(arg2);
      ndim := arg1.dim;
      scalar := 0.0;
      REPEAT
        i := 1 TO ndim;
        scalar :=
          scalar + vec1.direction_ratios[i] * vec2.direction_ratios[i];
      END_REPEAT;
    END;
  END_IF;
END_IF;
RETURN (scalar);
END_FUNCTION; -- dot_product

FUNCTION edge_reversed(an_edge : edge) : oriented_edge;
LOCAL
  the_reverse : oriented_edge;
END_LOCAL;
IF ('ISO_14306_SCHEMA.ORIENTED_EDGE' IN TYPEOF(an_edge))
THEN
  the_reverse :=
    dummy_tri || edge(an_edge.edge_end, an_edge.edge_start) ||
    oriented_edge(an_edge\oriented_edge.edge_element,
      NOT (an_edge\oriented_edge.orientation));
ELSE
  the_reverse :=
    dummy_tri || edge(an_edge.edge_end, an_edge.edge_start) ||
    oriented_edge(an_edge, false);
END_IF;
RETURN (the_reverse);
END_FUNCTION; -- edge_reversed

FUNCTION face_bound_reversed(a_face_bound : face_bound) : face_bound;
LOCAL
  the_reverse : face_bound;
END_LOCAL;
IF ('ISO_14306_SCHEMA.FACE_OUTER_BOUND' IN TYPEOF(a_face_bound))
THEN
  the_reverse :=

```

```

dummy_tri ||
face_bound(a_face_bound\face_bound.bound,
            NOT (a_face_bound\face_bound.orientation)) ||
face_outer_bound();
ELSE
the_reverse :=
dummy_tri || face_bound(a_face_bound.bound, NOT (a_face_bound.orientation));
END_IF;
RETURN (the_reverse);
END_FUNCTION; -- face_bound_reversed

FUNCTION face_reversed(a_face : face) : oriented_face;
LOCAL
the_reverse : oriented_face;
END_LOCAL;
IF ('ISO_14306_SCHEMA.ORIENTED_FACE' IN TYPEOF(a_face))
THEN
the_reverse :=
dummy_tri || face(set_of_topology_reversed(a_face.bounds)) ||
oriented_face(a_face\oriented_face.face_element,
              NOT (a_face\oriented_face.orientation));
ELSE
the_reverse :=
dummy_tri || face(set_of_topology_reversed(a_face.bounds)) ||
oriented_face(a_face, false);
END_IF;
RETURN (the_reverse);
END_FUNCTION; -- face_reversed

FUNCTION first_proj_axis(z_axis : direction; arg : direction) : direction;
LOCAL
x_axis : direction;
v : direction;
z : direction;
x_vec : vector;
END_LOCAL;
IF (NOT EXISTS(z_axis))
THEN
RETURN (?);
ELSE
z := normalise(z_axis);
IF NOT EXISTS(arg)
THEN
IF ((z.direction_ratios <> [1.0, 0.0, 0.0]) AND (z.direction_ratios <>
[-1.0, 0.0, 0.0]))
THEN
v := dummy_tri || direction([1.0, 0.0, 0.0]);
ELSE
v := dummy_tri || direction([0.0, 1.0, 0.0]);
END_IF;
ELSE
IF (arg.dim <> 3)
THEN
RETURN (?);
END_IF;
IF ((cross_product(arg, z).magnitude) = 0.0)
THEN
RETURN (?);
ELSE
v := normalise(arg);
END_IF;
END_IF;
x_vec := scalar_times_vector(dot_product(v, z), z);
x_axis := vector_difference(v, x_vec).orientation;
x_axis := normalise(x_axis);
END_IF;
RETURN (x_axis);

```

```

END_FUNCTION; -- first_proj_axis

FUNCTION get_basis_surface(c : curve_on_surface) : SET [0:2] OF surface;
LOCAL
  surfs : SET [0:2] OF surface;
  n : INTEGER;
END_LOCAL;
surfs := [];
IF 'ISO_14306_SCHEMA.PCURVE' IN TYPEOF(c)
THEN
  surfs := [c\pcurve.basis_surface];
ELSE
  IF 'ISO_14306_SCHEMA.SURFACE_CURVE' IN TYPEOF(c)
  THEN
    n := SIZEOF(c\surface_curve.associated_geometry);
    REPEAT
      i := 1 TO n;
      surfs :=
        surfs + associated_surface(c\surface_curve.associated_geometry[i]);
    END_REPEAT;
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.COMPOSITE_CURVE_ON_SURFACE' IN TYPEOF(c)
THEN
  n := SIZEOF(c\composite_curve.segments);
  surfs := get_basis_surface(c\composite_curve.segments[1].parent_curve);
  IF n > 1
  THEN
    REPEAT
      i := 2 TO n;
      surfs :=
        surfs * get_basis_surface(c\composite_curve.segments[i].parent_curve);
    END_REPEAT;
  END_IF;
END_IF;
RETURN (surfs);
END_FUNCTION; -- get_basis_surface

FUNCTION get_description_value(obj : description_attribute_select) : text;
LOCAL
  description_bag : BAG OF description_attribute :=
    (USEDIN(obj,
      'ISO_14306_SCHEMA.DESRIPTION_ATTRIBUTE.DESCRIBED_ITEM'));
END_LOCAL;
IF SIZEOF(description_bag) = 1
THEN
  RETURN (description_bag[1].attribute_value);
ELSE
  RETURN (?);
END_IF;
END_FUNCTION; -- get_description_value

FUNCTION get_directed_link(nodes : LIST OF representation;
  undirected_link : LIST OF chained_representation_link) :
LIST OF representation_relationship;
LOCAL
  directed_link : LIST OF representation_relationship := [];
END_LOCAL;
IF (SIZEOF(nodes) <> SIZEOF(undirected_link) + 1) OR (VALUE_UNIQUE(nodes) =
  false)
THEN
  RETURN (?);
END_IF;
REPEAT
  i := 1 TO SIZEOF(undirected_link);
  CASE true OF
    ('ISO_14306_SCHEMA.REPRESENTATION_CONTEXT' IN

```

```

        TYPEOF(undirected_link[i])) : BEGIN
        IF ((nodes[i]\representation.context_of_items :=:
undirected_link[i]) AND (nodes[i +
1]\representation.context_of_items :=:
undirected_link[i]))
        THEN
        INSERT(directed_link,representation_relationship('',' ',nodes[i +
1], nodes[i]), (i - 1));
        ELSE
        RETURN (?);
        END_IF;
        END;
        ('ISO_14306_SCHEMA.REPRESENTATION_RELATIONSHIP' IN
        TYPEOF(undirected_link[i])) : BEGIN
        IF ((nodes[i] :=:
undirected_link[i]\representation_relationship.rep_1) AND
(nodes[i + 1] :=:
undirected_link[i]\representation_relationship.rep_2) OR
((nodes[i] :=:
undirected_link[i]\representation_relationship.rep_2) AND
(nodes[i + 1] :=:
undirected_link[i]\representation_relationship.rep_1)))
        THEN
        INSERT(directed_link,representation_relationship('',' ',nodes[i
+ 1],nodes[i]),(i - 1));
        ELSE
        RETURN (?);
        END_IF;
        END;
        ('ISO_14306_SCHEMA.MAPPED_ITEM' IN
        TYPEOF(undirected_link[i])) : BEGIN
        IF ((nodes[i] IN
        using_representations(undirected_link[i])) AND
        (nodes[i + 1] :=:
undirected_link[i]\mapped_item.mapping_source\representation_map.mapped_representat
ion))
        THEN
        INSERT(directed_link,representation_relationship('',' ',nodes[i
+ 1],nodes[i]),(i - 1));
        ELSE
        RETURN (?);
        END_IF;
        END;
        OTHERWISE : RETURN (?);
        END_CASE;
    END_REPEAT;
    IF (VALUE_UNIQUE(directed_link) = false)
    THEN
        RETURN (?);
    END_IF;
    RETURN (directed_link);
END_FUNCTION; -- get_directed_link

FUNCTION get_id_value(obj : id_attribute_select) : identifier;
LOCAL
    id_bag : BAG OF id_attribute :=
        (USEDIN(obj, 'ISO_14306_SCHEMA.ID_ATTRIBUTE.IDENTIFIED_ITEM'));
END_LOCAL;
    IF SIZEOF(id_bag) = 1
    THEN
        RETURN (id_bag[1].attribute_value);
    ELSE
        RETURN (?);
    END_IF;
END_FUNCTION; -- get_id_value

```



```

FUNCTION get_name_value(obj : name_attribute_select) : label;
LOCAL
  name_bag : BAG OF name_attribute :=
    (USEDIN(obj, 'ISO_14306_SCHEMA.NAME_ATTRIBUTE.NAMED_ITEM'));
END_LOCAL;
IF SIZEOF(name_bag) = 1
THEN
  RETURN (name_bag[1].attribute_value);
ELSE
  RETURN (?);
END_IF;
END_FUNCTION; -- get_name_value

FUNCTION get_role(obj : role_select) : object_role;
LOCAL
  role_bag : BAG OF role_association :=
    (USEDIN(obj, 'ISO_14306_SCHEMA.ROLE_ASSOCIATION.ITEM_WITH_ROLE'));
END_LOCAL;
IF SIZEOF(role_bag) = 1
THEN
  RETURN (role_bag[1].role);
ELSE
  RETURN (?);
END_IF;
END_FUNCTION; -- get_role

FUNCTION is_acyclic(arg : generic_expression) : BOOLEAN;
RETURN (acyclic(arg, []));
END_FUNCTION; -- is_acyclic

FUNCTION item_in_context(item : representation_item;
                        cntxt : representation_context) :
BOOLEAN;
LOCAL
  y : BAG OF representation_item;
END_LOCAL;
IF SIZEOF(USEDIN(item, 'ISO_14306_SCHEMA.REPRESENTATION.ITEMS')) *
  cntxt.representations_in_context >
  0
THEN
  RETURN (true);
ELSE
  y :=
    QUERY(z <* USEDIN(item, '') |
      'ISO_14306_SCHEMA.REPRESENTATION_ITEM' IN TYPEOF(z));
  IF SIZEOF(y) > 0
  THEN
    REPEAT
      i := 1 TO HIINDEX(y);
      IF item_in_context(y[i], cntxt)
      THEN
        RETURN (true);
      END_IF;
    END_REPEAT;
  END_IF;
  RETURN (false);
END_FUNCTION; -- item_in_context

FUNCTION list_face_loops(f : face) : LIST [0:?] OF loop;
LOCAL
  loops : LIST [0:?] OF loop := [];
END_LOCAL;
REPEAT
  i := 1 TO SIZEOF(f.bounds);
  loops := loops + (f.bounds[i].bound);
END_REPEAT;

```

```

    RETURN (loops);
END_FUNCTION; -- list_face_loops

FUNCTION list_of_topology_reversed(a_list : list_of_reversible_topology_item) :
list_of_reversible_topology_item;
LOCAL
    the_reverse : list_of_reversible_topology_item;
END_LOCAL;
the_reverse := [];
REPEAT
    i := 1 TO SIZEOF(a_list);
    the_reverse := topology_reversed(a_list[i]) + the_reverse;
END_REPEAT;
RETURN (the_reverse);
END_FUNCTION; -- list_of_topology_reversed

FUNCTION list_to_array(lis : LIST [0:?] OF GENERIC:t; low : INTEGER;
                        u : INTEGER) :
ARRAY [low:u] OF GENERIC:t;
LOCAL
    n : INTEGER;
    res : ARRAY [low:u] OF GENERIC:t;
END_LOCAL;
n := SIZEOF(lis);
IF (n <> (u - low + 1))
THEN
    RETURN (?);
ELSE
    res := [lis[1]:n];
    REPEAT
        i := 2 TO n;
        res[low + i - 1] := lis[i];
    END_REPEAT;
    RETURN (res);
END_IF;
END_FUNCTION; -- list_to_array

FUNCTION list_to_set(l : LIST [0:?] OF GENERIC:t) : SET OF GENERIC:t;
LOCAL
    s : SET OF GENERIC:t := [];
END_LOCAL;
REPEAT
    i := 1 TO SIZEOF(l);
    s := s + l[i];
END_REPEAT;
RETURN (s);
END_FUNCTION; -- list_to_set

FUNCTION make_array_of_array(lis : LIST [1:?] OF LIST [1:?] OF GENERIC:t;
                             low1 : INTEGER; u1 : INTEGER; low2 : INTEGER;
                             u2 : INTEGER) :
ARRAY [low1:u1] OF ARRAY [low2:u2] OF GENERIC:t;
LOCAL
    res : ARRAY [low1:u1] OF ARRAY [low2:u2] OF GENERIC:t;
END_LOCAL;
IF (u1 - low1 + 1) <> SIZEOF(lis)
THEN
    RETURN (?);
END_IF;
IF (u2 - low2 + 1) <> SIZEOF(lis[1])
THEN
    RETURN (?);
END_IF;
res := [list_to_array(lis[1], low2, u2):(u1 - low1 + 1)];
REPEAT
    i := 2 TO HIINDEX(lis);
    IF (u2 - low2 + 1) <> SIZEOF(lis[i])

```

```

    THEN
        RETURN (?);
    END_IF;
    res[low1 + i - 1] := list_to_array(lis[i], low2, u2);
END_REPEAT;
RETURN (res);
END_FUNCTION; -- make_array_of_array

FUNCTION mixed_loop_type_set(l : SET [0:?] OF loop) : LOGICAL;
LOCAL
    poly_loop_type : LOGICAL;
END_LOCAL;
IF (SIZEOF(l) <= 1)
    THEN
        RETURN (false);
    END_IF;
poly_loop_type := ('ISO_14306_SCHEMA.POLY_LOOP' IN TYPEOF(l[1]));
REPEAT
    i := 2 TO SIZEOF(l);
    IF (('ISO_14306_SCHEMA.POLY_LOOP' IN TYPEOF(l[i])) <>
        poly_loop_type)
    THEN
        RETURN (true);
    END_IF;
END_REPEAT;
RETURN (false);
END_FUNCTION; -- mixed_loop_type_set

FUNCTION msb_shells(brep : manifold_solid_brep) : SET [1:?] OF closed_shell;
LOCAL
    return_set : SET [1:?] OF closed_shell := [brep.outer];
END_LOCAL;
IF SIZEOF(QUERY(msbtype <* TYPEOF(brep) | msbtype LIKE
    '*BREP_WITH_VOIDS')) >=
    1
    THEN
        return_set := return_set + brep\brep_with_voids.voids;
    END_IF;
RETURN (return_set);
END_FUNCTION; -- msb_shells

FUNCTION normalise(arg : vector_or_direction) : vector_or_direction;
LOCAL
    ndim : INTEGER;
    v : direction :=
        dummy_gri || direction([1.0, 0.0, 0.0]);
    result : vector_or_direction;
    vec : vector :=
        dummy_gri || vector(v, 1.0);
    mag : REAL;
END_LOCAL;
IF NOT EXISTS(arg)
    THEN
        result := ?;
    ELSE
        ndim := arg.dim;
        IF 'ISO_14306_SCHEMA.VECTOR' IN TYPEOF(arg)
            THEN
                BEGIN
                    v :=
                        dummy_gri || direction(arg\vector.orientation.direction_ratios);
                    IF arg\vector.magnitude = 0.0
                        THEN
                            RETURN (?);
                        ELSE
                            vec := dummy_gri || vector(v, 1.0);
                        END_IF;
                END
            END
        END
    END

```

```

        END;
    ELSE
        v := dummy_gri || direction(arg.direction_ratios);
    END_IF;
    mag := 0.0;
    REPEAT
        i := 1 TO ndim;
        mag := mag + v.direction_ratios[i] * v.direction_ratios[i];
    END_REPEAT;
    IF mag > 0.0
    THEN
        mag := SQRT(mag);
        REPEAT
            i := 1 TO ndim;
            v.direction_ratios[i] := v.direction_ratios[i] / mag;
        END_REPEAT;
        IF 'ISO_14306_SCHEMA.VECTOR' IN TYPEOF(arg)
        THEN
            vec.orientation := v;
            result := vec;
        ELSE
            result := v;
        END_IF;
    ELSE
        RETURN (?);
    END_IF;
END_IF;
RETURN (result);
END_FUNCTION; -- normalise

FUNCTION open_shell_reversed(a_shell : open_shell) : oriented_open_shell;
LOCAL
    the_reverse : oriented_open_shell;
END_LOCAL;
IF ('ISO_14306_SCHEMA.ORIENTED_OPEN_SHELL' IN TYPEOF(a_shell))
THEN
    the_reverse :=
        dummy_tri || connected_face_set(a_shell\connected_face_set.cfs_faces) ||
        open_shell() ||
        oriented_open_shell(a_shell\oriented_open_shell.open_shell_element,
            (NOT (a_shell\oriented_open_shell.orientation)));
ELSE
    the_reverse :=
        dummy_tri || connected_face_set(a_shell\connected_face_set.cfs_faces) ||
        open_shell() || oriented_open_shell(a_shell, false);
END_IF;
RETURN (the_reverse);
END_FUNCTION; -- open_shell_reversed

FUNCTION orthogonal_complement(vec : direction) : direction;
LOCAL
    result : direction;
END_LOCAL;
IF (vec.dim <> 2) OR NOT EXISTS(vec)
THEN
    RETURN (?);
ELSE
    result :=
        dummy_gri || direction([-vec.direction_ratios[2], vec.direction_ratios[1]]);
    RETURN (result);
END_IF;
END_FUNCTION; -- orthogonal_complement

FUNCTION path_head_to_tail(a_path : path) : LOGICAL;
LOCAL
    n : INTEGER;
    p : LOGICAL :=

```

```

    true;
END_LOCAL;
n := SIZEOF(a_path.edge_list);
REPEAT
    i := 2 TO n;
    p :=
    p AND (a_path.edge_list[i - 1].edge_end := a_path.edge_list[i].edge_start);
END_REPEAT;
RETURN (p);
END_FUNCTION; -- path_head_to_tail

FUNCTION path_reversed(a_path : path) : oriented_path;
LOCAL
    the_reverse : oriented_path;
END_LOCAL;
IF ('ISO_14306_SCHEMA.ORIENTED_PATH' IN TYPEOF(a_path))
THEN
    the_reverse :=
    dummy_tri || path(list_of_topology_reversed(a_path.edge_list)) ||
    oriented_path(a_path\oriented_path.path_element,
        NOT (a_path\oriented_path.orientation));
ELSE
    the_reverse :=
    dummy_tri || path(list_of_topology_reversed(a_path.edge_list)) ||
    oriented_path(a_path, false);
END_IF;
RETURN (the_reverse);
END_FUNCTION; -- path_reversed

FUNCTION scalar_times_vector(scalar : REAL; vec : vector_or_direction) : vector;
LOCAL
    v : direction;
    mag : REAL;
    result : vector;
END_LOCAL;
IF NOT EXISTS(scalar) OR NOT EXISTS(vec)
THEN
    RETURN (?);
ELSE
    IF 'ISO_14306_SCHEMA.VECTOR' IN TYPEOF(vec)
    THEN
        v := dummy_gri || direction(vec\vector.orientation.direction_ratios);
        mag := scalar * vec\vector.magnitude;
    ELSE
        v := dummy_gri || direction(vec.direction_ratios);
        mag := scalar;
    END_IF;
    IF (mag < 0.0)
    THEN
        REPEAT
            i := 1 TO SIZEOF(v.direction_ratios);
            v.direction_ratios[i] := -v.direction_ratios[i];
        END_REPEAT;
        mag := -mag;
    END_IF;
    result := dummy_gri || vector(normalise(v), mag);
END_IF;
RETURN (result);
END_FUNCTION; -- scalar_times_vector

FUNCTION second_proj_axis(z_axis : direction; x_axis : direction;
    arg : direction) :
direction;
LOCAL
    y_axis : vector;
    v : direction;
    temp : vector;

```

```

END_LOCAL;
IF NOT EXISTS(arg)
THEN
  v := dummy_gri || direction([0.0, 1.0, 0.0]);
ELSE
  v := arg;
END_IF;
temp := scalar_times_vector(dot_product(v, z_axis), z_axis);
y_axis := vector_difference(v, temp);
temp := scalar_times_vector(dot_product(v, x_axis), x_axis);
y_axis := vector_difference(y_axis, temp);
y_axis := normalise(y_axis);
RETURN (y_axis.orientation);
END_FUNCTION; -- second_proj_axis

FUNCTION set_of_topology_reversed(a_set : set_of_reversible_topology_item) :
set_of_reversible_topology_item;
LOCAL
  the_reverse : set_of_reversible_topology_item;
END_LOCAL;
the_reverse := [];
REPEAT
  i := 1 TO SIZEOF(a_set);
  the_reverse := the_reverse + topology_reversed(a_set[i]);
END_REPEAT;
RETURN (the_reverse);
END_FUNCTION; -- set_of_topology_reversed

FUNCTION shell_reversed(a_shell : shell) : shell;
IF ('ISO_14306_SCHEMA.OPEN_SHELL' IN TYPEOF(a_shell))
THEN
  RETURN (open_shell_reversed(a_shell));
ELSE
  IF ('ISO_14306_SCHEMA.CLOSED_SHELL' IN TYPEOF(a_shell))
  THEN
    RETURN (closed_shell_reversed(a_shell));
  ELSE
    RETURN (?);
  END_IF;
END_IF;
END_FUNCTION; -- shell_reversed

FUNCTION surface_weights_positive(b : rational_b_spline_surface) : BOOLEAN;
LOCAL
  result : BOOLEAN := true;
END_LOCAL;
REPEAT
  i := 0 TO b.u_upper;
  REPEAT
    j := 0 TO b.v_upper;
    IF (b.weights[i][j] <= 0.0)
    THEN
      result := false;
      RETURN (result);
    END_IF;
  END_REPEAT;
END_REPEAT;
RETURN (result);
END_FUNCTION; -- surface_weights_positive

FUNCTION topology_reversed(an_item : reversible_topology) : reversible_topology;
IF ('ISO_14306_SCHEMA.EDGE' IN TYPEOF(an_item))
THEN
  RETURN (edge_reversed(an_item));
END_IF;
IF ('ISO_14306_SCHEMA.PATH' IN TYPEOF(an_item))
THEN

```

```

    RETURN (path_reversed(an_item));
END_IF;
IF ('ISO_14306_SCHEMA.FACE_BOUND' IN TYPEOF(an_item))
THEN
    RETURN (face_bound_reversed(an_item));
END_IF;
IF ('ISO_14306_SCHEMA.FACE' IN TYPEOF(an_item))
THEN
    RETURN (face_reversed(an_item));
END_IF;
IF ('ISO_14306_SCHEMA.SHELL' IN TYPEOF(an_item))
THEN
    RETURN (shell_reversed(an_item));
END_IF;
IF ('SET' IN TYPEOF(an_item))
THEN
    RETURN (set_of_topology_reversed(an_item));
END_IF;
IF ('LIST' IN TYPEOF(an_item))
THEN
    RETURN (list_of_topology_reversed(an_item));
END_IF;
RETURN (?);
END_FUNCTION; -- topology_reversed

FUNCTION unambiguously_specified_multi_level_reference_designator(links : LIST
[1:?] OF next_assembly_usage_occurrence) :
BOOLEAN;
REPEAT
    i := 1 TO SIZEOF(links) - 1;
    CASE true OF
        ('ISO_14306_SCHEMA.PRODUCT_DEFINITION' IN
        TYPEOF(links[i]\product_definition_relationship.relating_product_definition)) AND
        ('ISO_14306_SCHEMA.PRODUCT_DEFINITION' IN
        TYPEOF(links[i]\product_definition_relationship.related_product_definition)) AND
        ('ISO_14306_SCHEMA.PRODUCT_DEFINITION' IN
        TYPEOF(links[i
+1]\product_definition_relationship.relating_product_definition)) : BEGIN
            IF NOT
            ((links[i]\product_definition_relationship.related_product_definition :=:
            links[i
+1]\product_definition_relationship.relating_product_definition) OR
            ((SIZEOF(QUERY(pdr < *
USEDIN(links[i]\product_definition_relationship.related_product_definition,
'ISO_14306_SCHEMA.PRODUCT_DEFINITION_RELATIONSHIP.RELATED_PRODUCT_DEFINITION') |
pdr\product_definition_relationship.relating_product_definition :=:
            links[i +
1]\product_definition_relationship.relating_product_definition)) = 1) AND
            (links[i]\product_definition_relationship.related_product_definition\product_defini
tion.formation\product_definition_formation.of_product :=:
            links[i +
1]\product_definition_relationship.relating_product_definition\product_definition.f
ormation\product_definition_formation.of_product)))
            THEN
                RETURN (false);
            END_IF;
        END;
        ('ISO_14306_SCHEMA.PRODUCT_DEFINITION_REFERENCE' IN
        TYPEOF(links[i]\product_definition_relationship.related_product_definition)) AND
        ('ISO_14306_SCHEMA.PRODUCT_DEFINITION_REFERENCE' IN
        TYPEOF(links[i +
1]\product_definition_relationship.relating_product_definition)) : BEGIN

```

```

        IF NOT
        ((links[i]\product_definition_relationship.related_product_definition :=:
            links[i +
1]\product_definition_relationship.relateing_product_definition) OR
            ((SIZEOF(QUERY (pdr <*
USEDIN(links[i]\product_definition_relationship.related_product_definition,
'ISO_14306_SCHEMA.PRODUCT_DEFINITION_RELATIONSHIP.RELATED_PRODUCT_DEFINITION') ) |
pdr\product_definition_relationship.relateing_product_definition :=:
            links[i +
1]\product_definition_relationship.relateing_product_definition)) = 1) AND
        (links[i]\product_definition_relationship.related_product_definition
            \product_definition_reference.product_id =
            links[i +
1]\product_definition_relationship.relateing_product_definition
            \product_definition_reference.product_id) AND
        (links[i]\product_definition_relationship.related_product_definition
            \product_definition_reference.id_owning_organization_name =
            links[i
+1]\product_definition_relationship.relateing_product_definition
            \product_definition_reference.id_owning_organization_name) AND
        (links[i]\product_definition_relationship.related_product_definition
            \product_definition_reference.product_definition_id <>
            links[i +
1]\product_definition_relationship.relateing_product_definition
            \product_definition_reference.product_definition_id))
        THEN
            RETURN (false);
        END_IF;
    END;
    ('ISO_14306_SCHEMA.PRODUCT_DEFINITION_REFERENCE' IN
    TYPEOF(links[i]\product_definition_relationship.relateing_product_definition)) AND
    ('ISO_14306_SCHEMA.PRODUCT_DEFINITION' IN
    TYPEOF(links[i]\product_definition_relationship.related_product_definition)) :
    RETURN (false);
    OTHERWISE : RETURN (false);
    END_CASE;
    END_REPEAT;
    RETURN (true);
END_FUNCTION; -- unambiguously_specified_multi_level_reference_designator

FUNCTION using_items(item : founded_item_select;
                    checked_items : SET OF founded_item_select) :
SET OF founded_item_select;
LOCAL
    new_check_items : SET OF founded_item_select;
    result_items : SET OF founded_item_select;
    next_items : SET OF founded_item_select;
END_LOCAL;
result_items := [];
new_check_items := checked_items + item;
next_items :=
    QUERY(z <* bag_to_set(USEDIN(item, '')) |
        ('ISO_14306_SCHEMA.REPRESENTATION_ITEM' IN TYPEOF(z)) OR
        ('ISO_14306_SCHEMA.FOUNDED_ITEM' IN TYPEOF(z)));
IF SIZEOF(next_items) > 0

```



```

THEN
  REPEAT
    i := 1 TO HIINDEX(next_items);
    IF NOT (next_items[i] IN new_check_items)
    THEN
      result_items :=
        result_items + next_items[i] +
        using_items(next_items[i], new_check_items);
    END_IF;
  END_REPEAT;
END_IF;
RETURN (result_items);
END_FUNCTION; -- using_items

FUNCTION using_representations(item : founded_item_select) :
SET OF representation;
LOCAL
  results : SET OF representation;
  result_bag : BAG OF representation;
  intermediate_items : SET OF founded_item_select;
END_LOCAL;
results := [];
result_bag := USEDIN(item, 'ISO_14306_SCHEMA.REPRESENTATION.ITEMS');
IF SIZEOF(result_bag) > 0
THEN
  REPEAT
    i := 1 TO HIINDEX(result_bag);
    results := results + result_bag[i];
  END_REPEAT;
END_IF;
intermediate_items := using_items(item, []);
IF SIZEOF(intermediate_items) > 0
THEN
  REPEAT
    i := 1 TO HIINDEX(intermediate_items);
    result_bag :=
      USEDIN(intermediate_items[i],
        'ISO_14306_SCHEMA.REPRESENTATION.ITEMS');
    IF SIZEOF(result_bag) > 0
    THEN
      REPEAT
        j := 1 TO HIINDEX(result_bag);
        results := results + result_bag[j];
      END_REPEAT;
    END_IF;
  END_REPEAT;
END_IF;
RETURN (results);
END_FUNCTION; -- using_representations

FUNCTION valid_measure_value(m : measure_value) : BOOLEAN;
IF ('REAL' IN TYPEOF(m))
THEN
  RETURN (m > 0.0);
ELSE
  IF ('INTEGER' IN TYPEOF(m))
  THEN
    RETURN (m > 0);
  ELSE
    RETURN (true);
  END_IF;
END_IF;
END_FUNCTION; -- valid_measure_value

FUNCTION valid_units(m : measure_with_unit) : BOOLEAN;
IF 'ISO_14306_SCHEMA.LENGTH_MEASURE' IN TYPEOF(m.value_component)
THEN

```

```

    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.MASS_MEASURE' IN TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.TIME_MEASURE' IN TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.ELECTRIC_CURRENT_MEASURE' IN
    TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.THERMODYNAMIC_TEMPERATURE_MEASURE' IN
    TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.CELSIUS_TEMPERATURE_MEASURE' IN
    TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.AMOUNT_OF_SUBSTANCE_MEASURE' IN
    TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.LUMINOUS_INTENSITY_MEASURE' IN
    TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;

```

```

IF 'ISO_14306_SCHEMA.PLANE_ANGLE_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.SOLID_ANGLE_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.AREA_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.VOLUME_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(3.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.RATIO_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.POSITIVE_LENGTH_MEASURE' IN
  TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.POSITIVE_PLANE_ANGLE_MEASURE' IN
  TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.ACCELERATION_MEASURE' IN
  TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(1.0, 0.0, -2.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;

```

```

IF 'ISO_14306_SCHEMA.CAPACITANCE_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(-2.0, -1.0, 4.0, 2.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.ELECTRIC_CHARGE_MEASURE' IN
  TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.CONDUCTANCE_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(-2.0, -1.0, 3.0, 2.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.ELECTRIC_POTENTIAL_MEASURE' IN
  TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(2.0, 1.0, -3.0, -1.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.ENERGY_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(2.0, 1.0, -2.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.FORCE_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(1.0, 1.0, -2.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.FREQUENCY_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(0.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.ILLUMINANCE_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(-2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.INDUCTANCE_MEASURE' IN TYPEOF(m.value_component)

```

```

THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(2.0, 1.0, -2.0, -2.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.LUMINOUS_FLUX_MEASURE' IN
  TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.MAGNETIC_FLUX_MEASURE' IN
  TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(2.0, 1.0, -2.0, -1.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.MAGNETIC_FLUX_DENSITY_MEASURE' IN
  TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(0.0, 1.0, -2.0, -1.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.POWER_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(2.0, 1.0, -3.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.PRESSURE_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(-1.0, 1.0, -2.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.RESISTANCE_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(2.0, 1.0, -3.0, -2.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.VELOCITY_MEASURE' IN TYPEOF(m.value_component)
THEN
  IF derive_dimensional_exponents(m.unit_component) <>
    dimensional_exponents(1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0)
  THEN
    RETURN (false);
  END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.RADIOACTIVITY_MEASURE' IN

```

```

        TYPEOF(m.value_component)
    THEN
        IF derive_dimensional_exponents(m.unit_component) <>
            dimensional_exponents(0.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0)
        THEN
            RETURN (false);
        END_IF;
    END_IF;
IF 'ISO_14306_SCHEMA.ABSORBED_DOSE_MEASURE' IN
    TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(2.0, 0.0, -2.0, 0.0, 0.0, 0.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
IF 'ISO_14306_SCHEMA.DOSE_EQUIVALENT_MEASURE' IN
    TYPEOF(m.value_component)
THEN
    IF derive_dimensional_exponents(m.unit_component) <>
        dimensional_exponents(2.0, 0.0, -2.0, 0.0, 0.0, 0.0, 0.0)
    THEN
        RETURN (false);
    END_IF;
END_IF;
RETURN (true);
END_FUNCTION; -- valid_units

FUNCTION vector_difference(arg1 : vector_or_direction;
                           arg2 : vector_or_direction) :
vector;
LOCAL
    result : vector;
    res : direction;
    vec1 : direction;
    vec2 : direction;
    mag : REAL;
    mag1 : REAL;
    mag2 : REAL;
    ndim : INTEGER;
END_LOCAL;
IF ((NOT EXISTS(arg1)) OR (NOT EXISTS(arg2))) OR (arg1.dim <> arg2.dim)
THEN
    RETURN (?);
ELSE
    BEGIN
        IF 'ISO_14306_SCHEMA.VECTOR' IN TYPEOF(arg1)
        THEN
            mag1 := arg1\vector.magnitude;
            vec1 := arg1\vector.orientation;
        ELSE
            mag1 := 1.0;
            vec1 := arg1;
        END_IF;
        IF 'ISO_14306_SCHEMA.VECTOR' IN TYPEOF(arg2)
        THEN
            mag2 := arg2\vector.magnitude;
            vec2 := arg2\vector.orientation;
        ELSE
            mag2 := 1.0;
            vec2 := arg2;
        END_IF;
        vec1 := normalise(vec1);
        vec2 := normalise(vec2);
        ndim := SIZEOF(vec1.direction_ratios);
        mag := 0.0;
    
```

```

    res := dummy_gri || direction(vec1.direction_ratios);
  REPEAT
    i := 1 TO ndim;
    res.direction_ratios[i] :=
      mag1 * vec1.direction_ratios[i] - mag2 * vec2.direction_ratios[i];
    mag := mag + (res.direction_ratios[i] * res.direction_ratios[i]);
  END_REPEAT;
  IF (mag > 0.0)
  THEN
    result := dummy_gri || vector(res, SQRT(mag));
  ELSE
    result := dummy_gri || vector(vec1, 0.0);
  END_IF;
END;
END_IF;
RETURN (result);
END_FUNCTION; -- vector_difference

RULE compatible_dimension FOR
(cartesian_point, direction, representation_context,
geometric_representation_context);
WHERE
  WR1 : SIZEOF(QUERY(x <* cartesian_point |
    SIZEOF(QUERY(y <* geometric_representation_context |
      item_in_context(x, y) AND
      (HIINDEX(x.coordinates) <>
        y.coordinate_space_dimension))) >
    0)) =
    0;
  WR2 : SIZEOF(QUERY(x <* direction |
    SIZEOF(QUERY(y <* geometric_representation_context |
      item_in_context(x, y) AND
      (HIINDEX(x.direction_ratios) <>
        y.coordinate_space_dimension))) >
    0)) =
    0;
END_RULE; -- compatible_dimension
END_SCHEMA; -- iso_14306_schema

```

Annex D (informative) Coding Algorithms – An Implementation

This Appendix provides a sample C++ implementation for the encoding and decoding portion of the various compression CODECs (as detailed in [10.2 Encoding Algorithms](#)) used in the ISO 14306 format. This sample code is not intended to be fully functional encoder/decoder class implementations, but is instead intended to demonstrate the fundamentals of implementing the encoding/decoding portion of the CODEC algorithms used in the ISO 14306 format.

D.1 Common classes

The following sub-sections define some general classes used by all the coding algorithms.

D.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 in this symbol seen
};
```

D.1.2 CntxEntryBase2 class

```
//
// Type used to build probability context 2 tables.
// Used by ProbabilityContext 2 class.
//
class CntxEntryBase2
{
public:

    // ----- Housekeeping -----
    typedef enum { CEBUninitialized = -1,
                  CEBEscape        = -2,
                  CEBDeleted        = -3
    } CEBSpecialSym;
    CntxEntryBase2() : _iSym(CEBUninitialized), _cCount(-1), _cCumCount(-1) {};
    CntxEntryBase2(Int32 iSym, Int32 cCount)
        : _iSym(iSym), _cCount(cCount), _cCumCount(-1) {};
    CntxEntryBase2(const CntxEntryBase2 &rhs) { *this = rhs; }
    ~CntxEntryBase2() {};
    CntxEntryBase2 &operator=(const CntxEntryBase2 &rhs)
    {
        _iSym      = rhs._iSym;
        _cCount     = rhs._cCount;
        _cCumCount = rhs._cCumCount;
        return *this;
    }

    // ----- Operations Interface -----
    Bool isEscape() const
```



```

    { return (_iSym == CEBEscape); }
    Bool makeEscape(Bool bEsc = True)
    { _iSym = (bEsc ? CEBEscape : CEBUninitialized); return True; }
    Int32 operator==(const CntxEntryBase2 &rhs) const
    { return (_iSym == rhs._iSym); };

public:

    // ----- Member Variables -----
    Int32      _iSym;      // Symbol
    Int32      _cCount;    // Number of occurrences
    Int32      _cCumCount; // Cumulative number of occurrences
};

class CntxEntry2 : public CntxEntryBase2
{
public:

    // ----- Housekeeping -----
    CntxEntry2() : CntxEntryBase2(), _val(0) {};

    CntxEntry2( Int32 iSym, Int32 cCount, const Int32 &val ):
        CntxEntryBase2(iSym, cCount), _val(val) {};

    CntxEntry2( const CntxEntry2 &rhs ) { *this = rhs; }
    ~CntxEntry2() {};
    CntxEntry2 &operator=(const CntxEntry2 &rhs)
    { _val = rhs._val;
      CntxEntryBase2::operator= (rhs);
      return *this;
    }
    Int32 operator==(const CntxEntry2 &rhs) const
    { return (_iSym == rhs._iSym); };

public:

    // ----- Member Variables -----
    Int32      _val;      // Associated value
};

```

D.1.3 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);
};

```

D.1.4 ProbContext2 class

```

//
// Type used to build probability context 2 tables.
// Used by CodecDriver2 class.
//
class ProbContext2
{
public:
    // ----- Housekeeping -----
    ProbContext2();
    ProbContext2(const ProbContext2 &rhs);
    ~ProbContext2();
    ProbContext2 &operator=(const ProbContext2 &rhs);
    Bool operator==(const ProbContext2 &rhs) const;
    enum { cMaxCntxCnt = 8192 };

    // ----- Accessor Interface -----
    Int32 totalCount() const
    { return _cTotalCount; }
    Int32 numEntries() const
    { return _vEntries.length(); }
    Bool getEntry(Int32 iEntry, const CntxEntryBase2 *&rpEntry) const
    { const CntxEntry2 *aEntries = _vEntries.ptr();
      rpEntry = &aEntries[iEntry]; return True; }
    Bool getEntryV(Int32 iEntry, const CntxEntry2 *&rpEntry) const
    { const CntxEntry2 *aEntries = _vEntries.ptr();
      rpEntry = &aEntries[iEntry]; return True; }
    Bool getEntryV(Int32 iEntry, CntxEntry2 *&rpEntry)
    { CntxEntry2 *aEntries = _vEntries.ptr();
      rpEntry = &aEntries[iEntry]; return True; }

    // ----- Lookup Interface -----
    Bool lookupValue(const Int32 &rValue, const CntxEntry2 *&opCntxEntry) const;
    Bool lookupEntryByCumCount(Int32 iCount, const CntxEntry2 *&opCntxEntry) const;

    // ----- Reorganizing Interface -----
    Bool accumulateCounts();
    Bool sortByValue();

protected:
    Vec< CntxEntry2 > _vEntries;
    Int32 _cTotalCount;
    Int32 _iEscPosCache;

    static int _compareCounts(const void *pVal1,
                             const void *pVal2,
                             const void *uData);
};

Bool
ProbContext2::lookupValue( const Int32      &rValue,
                          const CntxEntry2 *&opCntxEntry ) const
{
    // If we do not find the value, then return NULL for the entry
    opCntxEntry = NULL;

    // If the escape position is not cached, sort the context by value
    // and then set it. Then, we can binary search for values. We do
    // this because translateValuesToSymbols() will call this method in
    // a tight loop. Anything is better than linear search!

```

```

    ProbContext2 *pThis = (ProbContext2*) this;
    CntxEntry2 *pEntries = pThis->_vEntries.ptr();
    Int32 nEntries = _vEntries.length();
    if (_iEscPosCache == -1) {
        // Search for the escape symbol
        Bool bFoundEsc = False;
        for (Int32 i = 0 ; i < nEntries ; i++) {
            if (pEntries[i]._iSym == CntxEntryBase2::CEBEscape) {
                // Move the escape symbol to context slot 0
                ::swap(pEntries[0], pEntries[i]);
                bFoundEsc = True;
                break;
            }
        }
        // Sort by value
        if (bFoundEsc) {
            // Sort by value _except_ leave the escape symbol in slot 0
            ::sort(&pEntries[1], nEntries-1, FtorCntxValue());
            pThis->accumulateCounts();
            pThis->_iEscPosCache = 0;
        }
        else {
            pThis->sortByValue();
            pThis->_iEscPosCache = -2;
        }
    }

    // Binary search for rValue!
    Int32 l = (_iEscPosCache == 0),
        h = nEntries - 1,
        m, d;
    while (l <= h) {
        m = (l + h) >> 1;
        d = pEntries[m]._val - rValue;
        if (d == 0) {
            opCntxEntry = &pEntries[m];
            return True;
        }
        else if (d < 0)
            l = m + 1;
        else
            h = m - 1;
    }

    // If we don't find the value, then we return the position of
    // the escape symbol.
    if (_iEscPosCache >= 0)
        opCntxEntry = &pEntries[_iEscPosCache];

    return True;
}

Bool ProbContext2::lookupEntryByCumCount(Int32 iCount, const CntxEntry2
*&opCntxEntry ) const
{
    const CntxEntry2 *aEntries = _vEntries.ptr();
    const Int32 nEntries = _vEntries.length();

    const Int32 seqSearchLen = 4;
    Int32 ii=0;
    opCntxEntry = NULL;

    // For short lists, do sequential search
    if ( nEntries <= (seqSearchLen*2) ) {
        ii = 0;
        while ( (iCount >= (aEntries[ii]._cCumCount + aEntries[ii]._cCount)) &&
(ii<nEntries) ) {

```

```

        ii++;
    }

    if ( ii >= nEntries ) {
        Assert( 0 && "Bad probability table" );
    }
    opCntxEntry = &aEntries[ii];
}

// For long lists, do a short sequential searches through most likely
// elements, then do a binary search through the rest.
else {
    for (ii=0; ii<seqSearchLen; ii++) {
        if (iCount < (aEntries[ii]._cCumCount + aEntries[ii]._cCount)) {
            opCntxEntry = &aEntries[ii];
            return True;
        }
    }

    Int32 low=ii, high=nEntries-1, mid;
    while(1) {
        if ( high < low ) {
            break;
        }
        mid = low + ((high-low)>>1);

        if ( iCount < aEntries[mid]._cCumCount ) {
            high = mid-1;
            continue;
        }
        if ( iCount >= (aEntries[mid]._cCumCount + aEntries[mid]._cCount) ) {
            low = mid+1;
            continue;
        }

        opCntxEntry = &aEntries[mid];
        return True;
    }
    Assert( 0 && "Bad probability table" );
}

return True;
}

Bool ProbContext2::accumulateCounts()
{
    // Check for zero length context
    CntxEntry2 *aEntries = _vEntries.ptr();
    Int32 nEntries = _vEntries.length();
    if ( nEntries == 0 ) {
        _cTotalCount = 0;
        return True;
    }

    // Accumulate counts in _cCumCount for entries 1 and higher
    aEntries[0]._cCumCount = 0;
    Int32 ii;
    for ( ii=1 ; ii<nEntries ; ii++ ) {
        aEntries[ii]._cCumCount = aEntries[ii-1]._cCount + aEntries[ii-1]._cCumCount;
    }

    // Set the total count for the context
    _cTotalCount = aEntries[ii-1]._cCount + aEntries[ii-1]._cCumCount;

    return True;
}

```

```

struct FtorCntxValue
{
    Bool operator () (const CntxEntry2& l, const CntxEntry2& r) const
    { return (l._val < r._val); }
};

Bool ProbContext2::sortByValue()
{
    // Sort the entries in order of values from smallest to largest
    ProbContext2 *pThis = (ProbContext2*) this;
    sort( _vEntries.ptr(), (size_t)( _vEntries.length()), FtorCntxValue() );

    pThis->_iEscPosCache = -1;
    pThis->accumulateCounts();

    return True;
}

```

D.1.5 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 32 bits of CodeText
    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
    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>& rvVal,
                                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 PredStrid1:
            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 PredStridel:
            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;
}

```

D.1.6 CodecDriver2 class

```

//
// A class that deals with the conversions from SYMBOL to VALUE and
// provides end-consumer APIs for using the codecs.
//
class CodecDriver2
{
public:

    // ----- Internal Types -----

```



```

typedef enum {
    CodecNull      = 0,    // Null Codec
    CodecBitLength = 1,    // Bitlength Codec
    CodecArithmetic = 3,   // Arithmetic Codec
    CodecChopper   = 4,    // Chopper Pseudo-codec
} CodecType;
// Type of value predictor
typedef enum {
    PredLag1      = 0, // Predicts as last values
    PredLag2      = 1, // Predicts as value before last
    PredStride1    = 2, // Predict using stride from last two values
    PredStride2    = 3, // Predict using stride from values 2 and 4 back.
    PredStripIndex = 4, // Domain-specific trisrip index list predictor
    PredRamp       = 5, // Predict value i as i
    PredXor1       = 6, // Predict as last, but use xor instead of subtract
    PredXor2       = 7, // Predict as last, but use xor instead of subtract
    PredNULL       = 8, // No prediction.
    PredMean2      = 9, // Predict as mean of last two values
    PredMean3      = 10, // Predict as mean of last three values
    PredMean4      = 11, // Predict as mean of last four values
} PredictorType;

static Bool unpackResiduals(const Veci    &rvResidual,
                           Veci          &rvVals,
                           PredictorType ePredType);
static Bool unpackResiduals(const Vecu    &rvResidual,
                           Vecu          &rvVals,
                           PredictorType ePredType);

static Float64 log2( Float64 x ) { return (log(x) / 0.6931471805599453); }

////////////////////
// Convenience Methods
////////////////////

////////////////////
//
// NOTE: These methods "CANNOT" be used with the 9.x serialization of
//       Coordinates, Normals, Colours, and Tex Coordinate do the
//       way the hash value is calculated.
//
/////

void assemble( const Vecu    &vMant,    // Output vector of mantissae
*
*      const Vecu    &vExp,    // Output vector of exponents
*
*      UInt32        iComp,    // Active data componenet per record
*
*      UInt32        nComp,    // Number of component per data record
*
*      UInt32        nIdx,    // Length of index list
*
*      UInt32        &uHash,   // Hash value
*
*      Float32       *opData ) // Ptr to data
*
{
    __FloatInt32Union fi;
    UInt32 *pExp = (UInt32*)vExp.ptr(),
            *pMant = (UInt32*)vMant.ptr();
    for (UInt32 i = 0 ; i < nIdx ; i++) {
        fi.ui = pExp[i] << 23
              | pMant[i];
        opData[i * nComp + iComp] = fi.f;
    }
    uHash = ::hash32(vMant.ptr(), nIdx, uHash);
}

```

```

        uHash = ::hash32(vExp.ptr(), nIdx, uHash);
    }
    void assemble( const Vecu      &vMant0,      // Output vector of mantissae
*
*                  const Vecu      &vMant1,      // Output vector of mantissae
*
*                  const Vecu      &vExp,        // Output vector of exponents
*
*                  UInt32          iComp,        // Active data componenet per record
*
*                  UInt32          nComp,        // Number of component per data record
*
*                  UInt32          nIdx,        // Length of index list
*
*                  UInt32          &uHash,      // Hash value
*
*                  Float64         *opData )    // Ptr to data
*
    {
        __FloatInt64Union fi;
        UInt32 *pExp = (UInt32*)vExp.ptr(),
                *pMant1 = (UInt32*)vMant1.ptr(),
                *pMant0 = (UInt32*)vMant0.ptr();
        for (UInt32 i = 0 ; i < nIdx ; i++) {
            fi.ui = (UInt64(pExp[i]) << 52) | (UInt64(pMant1[i]) << 31) |
pMant0[i];
            opData[i * nComp + iComp] = fi.f;
        }
        uHash = ::hash32(vMant0.ptr(), nIdx, uHash);
        uHash = ::hash32(vMant1.ptr(), nIdx, uHash);
        uHash = ::hash32(vExp.ptr(), nIdx, uHash);
    }

protected:
    static Int32 _predictValue( const Int32 *vVal, Int32 iIndex,
                                PredictorType ePredType );
};

Bool
CodecDriver2::unpackResiduals( const Veci      &rvResidual,
                               Veci           &rvVals,
                               PredictorType   ePredType )
{
    const Int32 len = rvResidual.length();
    Int32 iPredicted;
    rvVals.setLength(len);
    Int32 *aVals = rvVals.ptr();
    const Int32 *aResidual = rvResidual.ptr();
    for ( Int32 i = 0 ; i < len ; i++ ) {
        if (i < 4) {
            // The first four values are just primers
            aVals[i] = aResidual[i];
        } else {
            // Get a predicted value
            iPredicted = _predictValue(rvVals.ptr(), i, ePredType);

            if (ePredType == PredXor1 || ePredType == PredXor2) {
                // Encode the residual as the current value XOR predicted
                aVals[i] = aResidual[i] ^ iPredicted;
            } else {
                // Encode the residual as the current value plus predicted
                aVals[i] = aResidual[i] + iPredicted;
            }
        }
    }
}

```

```

    return True;
}

Bool
CodecDriver2::unpackResiduals( const Vecu      &rvResidual,
                               Vecu           &rvVals,
                               PredictorType   ePredType )
{
    return unpackResiduals(*((const Veci*)&rvResidual),
                           *((Veci*)&rvVals),
                           ePredType);
}

Int32
CodecDriver2::_predictValue( const Int32*   paVals,
                             Int32         iIndex,
                             PredictorType   ePredType )
{
    Int32 iPredicted;
    switch (ePredType) {

        default:
            case PredLag1:      // Predicts as last value
            case PredXor1:      // Predicts as last value
                iPredicted = paVals[iIndex-1];
                break;
            case PredLag2:      // Predicts as value before last
            case PredXor2:      // Predicts as value before last
                iPredicted = paVals[iIndex-2];
                break;
            case PredStride1:   // Predict using stride from last two values
                iPredicted = paVals[iIndex-1] + (paVals[iIndex-1] - paVals[iIndex-2]);
                break;
            case PredStride2:   // Predict using stride from values 2 and 4 back.
                iPredicted = paVals[iIndex-2] + (paVals[iIndex-2] - paVals[iIndex-4]);
                break;
            case PredStripIndex: // Domain-specific trisrip index list predictor
                // Predict using stride from last two values 2 and 4 back.
                if (paVals[iIndex-2] - paVals[iIndex-4] < 8 && paVals[iIndex-2] -
paVals[iIndex-4] > -8)
                    iPredicted = paVals[iIndex-2] + (paVals[iIndex-2] - paVals[iIndex-
4]);
                else
                    iPredicted = paVals[iIndex-2] + 2;
                break;
            case PredRamp:      // Predict value i as i
                iPredicted = iIndex;
                break;
            case PredMean2:     // Predict as mean of last two values
                iPredicted = (paVals[iIndex-1] + paVals[iIndex-2]) >> 1;
                break;
            case PredMean3:     // Predict as mean of last three values
                iPredicted = (paVals[iIndex-1] + paVals[iIndex-2] + paVals[iIndex-3]) /
3;
                break;
            case PredMean4:     // Predict as mean of last four values
                iPredicted = ( paVals[iIndex-1] + paVals[iIndex-2] +
paVals[iIndex-3] + paVals[iIndex-4] ) >> 2;
                break;
    }

    return iPredicted;
}

```

D.2 Bitlength decoding classes

The following sub-sections contain a sample implementation of the decoding portion of the Bitlength CODEC algorithm. A summary technical explanation of the Bitlength CODEC can be found in 10.2.2 Bitlength CODEC.

D.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);
    Bool encode(CodecDriver* pDriver);

    Int32 _nBitsInSymbol(Int32 iSymbol) const;
    Int32 cStepBits = 2;
};

Bool BitLengthCodec::encode( CodecDriverBase *pDriver )
{
    JtInt32  iSymbol,                // Symbol value to encode
             iNextSymbol;           // Lookahead symbol value
    JtInt32  cNumCurBits = 0,       // Current field width
             cNumSymBits = 0,       // Number of bits in iSymbol
             cNumSymBitsNext = 0,   // Number of bits in iNextSymbol
             nSymbols;              // Number of symbols

    // Reset driver's symbol iterator and get number of symbols.
    pDriver->resetInputSymbolIter();
    nSymbols = pDriver->numInputSymbols();

    // Short circuit for null array of symbols
    if (nSymbols <= 0)
        return JtTrue;

    // Get list of input symbols/counts/contexts from driver
    const Int32 *aiSymbol, *acSymCount, *acCumCount, *acTotalCount, *aiSymContext;
    pDriver->getInputSymbols(aiSymbol, acSymCount, acCumCount, acTotalCount,
aiSymContext);

    // Prime the symbol pump
    iNextSymbol = aiSymbol[0];
    cNumSymBitsNext = _nBitsInSymbol(iNextSymbol);

    // Iterate over the remaining symbols
    UInt32 uCodeText = 0;
    Int32 nBits = 0;
    for (Int32 i = 0; i < nSymbols; i++) {
        // Get the next symbol
        iSymbol = iNextSymbol;
        cNumSymBits = cNumSymBitsNext;
        if (i < nSymbols - 1)
            iNextSymbol = aiSymbol[i + 1];

        // Find out how many bits it takes to represent the symbol
        cNumSymBitsNext = _nBitsInSymbol(iNextSymbol);

        // Look ahead
        if (cNumCurBits > cNumSymBits && cNumSymBitsNext > cNumSymBits) {
            // Case 0: cur > next
            if (cNumCurBits > cNumSymBitsNext) {
                // Number of bits it will cost to adjust down

```

```

        // and then back up to NextSymBits from here.
        Int32 cCost = ((cNumCurBits - cNumSymBits + cStepBits - 1) /
cStepBits
                    + (cNumSymBitsNext - cNumSymBits + cStepBits - 1) /
cStepBits)
                    * 2;
        // Number of bits that will be saved by stepping
        // down the number of encoding bits.
        Int32 cSave = ((cNumCurBits - cNumSymBits) / cStepBits)
                    * cStepBits;
        if (cCost > cSave)
            cNumSymBits = cNumSymBitsNext;
    }
    // Case 1: next > cur
    else {
        // Number of bits it will cost to adjust down
        // and then back up to here from here.
        Int32 cCost = (cNumCurBits - cNumSymBits + cStepBits - 1)
                    / cStepBits * 4;
        // Number of bits that will be saved by stepping
        // down the number of encoding bits.
        Int32 cSave = (cNumCurBits - cNumSymBits) / cStepBits
                    * cStepBits;
        if (cCost > cSave)
            cNumSymBits = cNumCurBits;
    }
}

// Emit bits to adjust the code length to match
// Note: We could look ahead and see if it's worth
// bothering to adjust...
UInt32 uPrefix = 0;
Int32 nPrefixBits = 0;

// Stay at the same number of bits
if (cNumCurBits >= cNumSymBits &&
    cNumCurBits <= cNumSymBits + cStepBits - 1)
{
    // A 0 bit indicates that the bit length stays the same
    uPrefix = 0;
    nPrefixBits = 1;
}
// Go up or down some multiple of cStepBits
else {
    // Start with a 1 bit to indicate that we are
    // changing the number of bits.
    uPrefix = 1;
    nPrefixBits = 1;
    // Accumulate a 1 for every step to increase
    while (cNumCurBits < cNumSymBits) {
        uPrefix <<= 1;
        uPrefix |= 0x1;
        nPrefixBits++;
        cNumCurBits += cStepBits;
    }
    // Accumulate a 0 for every step to decrease
    // Note that we never both increase and decrease!
    while (cNumCurBits > cNumSymBits + cStepBits - 1) {
        uPrefix <<= 1;
        nPrefixBits++;
        cNumCurBits -= cStepBits;
    }
    // Terminator bit is opposite of last bit
    uPrefix = (uPrefix << 1) | ((uPrefix & 0x1) ^ 0x1);
    nPrefixBits++;
}
}

```

```

// Turn the symbol value itself into bitcode
// Equivalent: pDriver->addCodeText(uPrefix, nPrefixBits);
// Equivalent: pDriver->addCodeText(iSymbol, cNumCurBits);

// Output prefix code
if (nBits + nPrefixBits <= 32) {
    uCodeText <= nPrefixBits;
    uCodeText |= uPrefix;
    nBits += nPrefixBits;
}
else {
    // Flush the local bit cache and start a new one
    pDriver->addCodeText(uCodeText, nBits);
    uCodeText = uPrefix;
    nBits = nPrefixBits;
}
// Output symbol
if (nBits + cNumCurBits <= 32) {
    if (cNumCurBits == 32) {
        uCodeText = iSymbol;
    } else {
        uCodeText <= cNumCurBits;
        uCodeText |= iSymbol & ((1<<cNumCurBits)-1);
    }
    nBits += cNumCurBits;
}
else {
    // Flush the local bit cache and start a new one
    pDriver->addCodeText(uCodeText, nBits);

    //uCodeText = the lowest order cNumCurBits of iSymbol
    uCodeText = iSymbol & (((cNumCurBits>>5)^1)<<cNumCurBits)-1);
    nBits = cNumCurBits;
}

}

// Flush the local bit cache
pDriver->addCodeText(uCodeText, nBits);

return True;
}

Int32 BitLengthCodec::nBitsInSymbol(Int32 iSymbol) const
{
    // Zero is the only number that can be encoded in a
    // single bit with this scheme, so this method returns
    // 0 bits for a symbol value of zero! A side-effect of
    // this is that this method never says there are any
    // 1-bit symbols (See below).
    if (iSymbol == 0)
        return 0;

    // Note: This calculation assumes that iSymbol can be
    // either positive or negative! Therefore, the number
    // of bits, say for 5 is given as 4 instead of 3 so
    // that -5 can also be encoded.
    Int32 cMaxCodeSpan = ::jtabs(iSymbol);
    Int32 i, nBits;
    for (i = 1, nBits = 0 ; i <= cMaxCodeSpan && nBits < 31 ; i += i, nBits++);
    return nBits + 1;
}

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-matching 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;
}

```

D.2.2 BitLengthCodec2 class

```

class BitLengthCodec2 : public Codec2
{
public:

    Bool encode( const Veci    &vValues,
                 Veci          &ovOOBValues,
                 Vecu          &ovCodeText,
                 Int32         &onBitsCodeText,
                 ProbContext2 *pProbCntx );

    Bool decode( Int32         nValues,
                 const Vecu    &vCodeText,

```



```

        Int32      nBitsCodeText,
        Veci       &ovValues );

protected:
    Int32 _nBitsInSymbol(Int32 iSymbol) const;
    Bool getNextCodeText (UInt32 &uCodeText, Int32 &nBits);

    Vecu      *_pvCodeText;
    Int32      *_pcCodeTextLen;
    Int32      _iCurCodeText;
    Vecus      _vnValBits;
};

#define GetSignedBits(iOut, n)           \
    GetUnsignedBits(iOut,n)             \
    iOut <= (32 - n);                   \
    iOut >= (32 - n);

#define GetUnsignedBits(uOut, n)         \
    if (n == 0) uOut = 0;                \
    else if (nValBits >= n) {            \
        uOut = uVal >> (32 - n);         \
        uVal <= n;                      \
        uVal &= (n==32)-1;              \
        nValBits -= n;                  \
        nBits += n;                     \
    }                                    \
    else {                               \
        Int32 _nLBits = nValBits;        \
        uOut = uVal >> (32 - n);         \
        nBits += _nLBits;               \
        getNextCodeText (uVal, nValBits); \
        Int32 _nRBits = (n - _nLBits);   \
        uOut |= uVal >> (32 - _nRBits);   \
        uVal <= _nRBits;                 \
        uVal &= (_nRBits==32)-1;         \
        nValBits -= _nRBits;             \
        nBits += _nRBits;                \
    }

Bool
BitLengthCodec2::encode(const Veci &vValues,
                        Veci &ovOOBValues,
                        Vecu &ovCodeText,
                        Int32 &onBitsCodeText,
                        ProbContext2 *pProbCntx )
{
    Int32 i, j, k;
    Int32 iSymbol; // Symbol value to encode
    Int32 cSymBits = 0; // Number of bits in iSymbol
    Int32 nValues; // Number of values to encode
    // Initialize output state
    ovOOBValues.setLength(0);
    ovCodeText.setLength(0);
    onBitsCodeText = 0;
    _pvCodeText = &ovCodeText;
    _pcCodeTextLen = &onBitsCodeText;

    // Short circuit for null array of values
    nValues = vValues.length();
    if (nValues <= 0)
        return True;

    _vnValBits.setLength(nValues);
    UInt16 *paiSymBits = _vnValBits.ptr();
    const Int32 *paiValues = vValues.ptr();
    // Find the minimum value and compute how many bits each value takes

```

```

Int32 iMinSymbol = 0x7fffffff;
Int32 iMaxSymbol = 0x80000000;
Float32 fMean = 0.0f;
for (i = 0 ; i < nValues ; i++) {
    iMinSymbol = ::min(iMinSymbol, paiValues[i]);
    iMaxSymbol = ::max(iMaxSymbol, paiValues[i]);
    fMean += Float32(paiValues[i]);
}
fMean /= nValues;
Int32 iMean = Int32(fMean);
for (i = 0 ; i < nValues ; i++) {
    paiSymBits[i] = _nBitsInSymbol(::abs(paiValues[i] - iMean)) + 1;
}
// A "block" is: 3 bits of number of bits (repeats while value is either 3 or -
4 for
larger width changes)
// 4 bits of block length
Int32 cBlkLenBits = 4; // Number of bits used to express a block length. 0
means 0.
Int32 cBlkValBits = 4; // *Delta* number of bits to express the current field
width
Int32 cBlkHdrBits = cBlkLenBits + cBlkValBits;
Bool bMerged;
// Block-forming: Merge Down/Up blocks
do {
    bMerged = JtFalse;
    Int32 iPrevRunPos = 0,
        iCurRunPos = 0,
        iNextRunPos = 0;
    while (iCurRunPos < nValues - 1) {
        // Advance the next run
        iNextRunPos++;
        while (iNextRunPos < nValues - 1 &&
            paiSymBits[iNextRunPos] == paiSymBits[iNextRunPos-1])
        {
            iNextRunPos++;
        }
        if (iNextRunPos >= nValues)
            break;
        Int32 wab = (iCurRunPos - iPrevRunPos),
            wbc = (iNextRunPos - iCurRunPos);
        if (wab == 0) {
            iCurRunPos = iNextRunPos;
            continue;
        }
        else if (wbc == 0) {
            continue;
        }
        // If we've bitten off more than one block's worth of data
        // we shall start a new block. Length 0 is allowed because
        // we may need to insert multiple consecutive block headers
        // in order to change the field width more bits than can be
        // represented in cBlkValBits.
        if (wab > (1 << cBlkLenBits)) {
            iPrevRunPos += (1 << cBlkLenBits);
        }
        else if (wbc == (1 << cBlkLenBits)) {
            iPrevRunPos = iCurRunPos;
            iCurRunPos = iNextRunPos;
            continue;
        }
        UInt16 &ua = paiSymBits[iPrevRunPos],
            &ub = paiSymBits[iCurRunPos],
            &uc = paiSymBits[iNextRunPos];
        // If the runs go "down-up"
        if (ua > ub && ub < uc) {
            // Test if we should increase ub to the lesser of ua and uc

```

```

        if (ua < uc) {
            // Test if we should increase ub to ua
            if (wbc * (ua - ub) <= cBlkHdrBits) {
                for (j = iCurRunPos ; j < iNextRunPos ; j++)
                    paiSymBits[j] = ua;
                iCurRunPos = iNextRunPos;
                continue;
            }
        }
        else if (ua > uc) {
            // Test if we should increase ub to uc
            if (wbc * (uc - ub) <= cBlkHdrBits) {
                for (j = iCurRunPos ; j < iNextRunPos ; j++)
                    paiSymBits[j] = uc;
                iNextRunPos = iCurRunPos;
                bMerged = True;
                continue;
            }
        }
        else { // ua == uc
            // Test if we should increase ub to ua/uc
            if (wbc * (ua - ub) <= 2 * cBlkHdrBits) {
                for (j = iCurRunPos ; j < iNextRunPos ; j++)
                    paiSymBits[j] = ua;
                iCurRunPos = iPrevRunPos;
                iNextRunPos = iPrevRunPos;
                bMerged = True;
                continue;
            }
        }
    }

    // Shift down the runs
    iPrevRunPos = iCurRunPos;
    iCurRunPos = iNextRunPos;
}
} while (bMerged);
// Block forming: Merge down/down and up/up runs
do {
    bMerged = JtFalse;
    Int32 iPrevRunPos = 0;
    iCurRunPos = 0;
    iNextRunPos = 0;
    while (iCurRunPos < nValues - 1) {
        // Advance the next run
        iNextRunPos++;
        while (iNextRunPos < nValues - 1 &&
            paiSymBits[iNextRunPos] == paiSymBits[iNextRunPos-1])
        {
            iNextRunPos++;
        }
        if (iNextRunPos >= nValues)
            break;
        Int32 wab = (iCurRunPos - iPrevRunPos),
            wbc = (iNextRunPos - iCurRunPos);
        if (wab == 0) {
            iCurRunPos = iNextRunPos;
            continue;
        }
        else if (wbc == 0) {
            continue;
        }
        // If we've bitten off more than one block's worth of data
        // we shall start a new block. Length 0 is allowed because
        // we may need to insert multiple consecutive block headers
        // in order to change the field width more bits than can be
        // represented in cBlkValBits.
    }
} while (bMerged);

```

```

        if (wab > (1 << cBlkLenBits)) {
            iPrevRunPos += (1 << cBlkLenBits);
        }
        else if (wab == (1 << cBlkLenBits)) {
            iPrevRunPos = iCurRunPos;
            iCurRunPos = iNextRunPos;
            continue;
        }
        UInt16 &ua = paiSymBits[iPrevRunPos],
            &ub = paiSymBits[iCurRunPos],
            &uc = paiSymBits[iNextRunPos];
        // If the runs go "up-down"
        if (ua < ub && ub > uc) {
            /* Empty */
        }
        // If the runs go "up-up"
        else if (ua < ub && ub < uc) {
            // Test if we should increase ua to ub
            if (wab * (ub - ua) < cBlkHdrBits) {
                for (j = iPrevRunPos ; j < iCurRunPos ; j++)
                    paiSymBits[j] = ub;
                iCurRunPos = iNextRunPos;
                bMerged = True;
                continue;
            }
            // Test if we should increase ub to uc
            else if (wbc * (uc - ub) < cBlkHdrBits) {
                for (j = iCurRunPos ; j < iNextRunPos ; j++)
                    paiSymBits[j] = uc;
                iNextRunPos = iCurRunPos;
                bMerged = True;
                continue;
            }
        }
        // If the runs go "down-down"
        else if (ua > ub && ub > uc) {
            // Test if we should increase ub to ua
            if (wbc * (ua - ub) < cBlkHdrBits) {
                for (j = iCurRunPos ; j < iNextRunPos ; j++)
                    paiSymBits[j] = ua;
                iCurRunPos = iNextRunPos;
                continue;
            }
        }

        // Shift down the runs
        iPrevRunPos = iCurRunPos;
        iCurRunPos = iNextRunPos;
    }
} while (bMerged);

// Compute the total bits
Int32 cTotalBits = paiSymBits[0] + cBlkHdrBits;
for (i = 1 ; i < nValues ; i++) {
    cTotalBits += paiSymBits[i];
    if (paiSymBits[i] != paiSymBits[i-1])
        cTotalBits += cBlkHdrBits;
}
Int32 cFixedWidBits = nValues * _nBitsInSymbol(iMaxSymbol - iMinSymbol);

/////
// If the fixed-width total bits are better, then write out the values
// in a single fixed-width format.
/////
if (cFixedWidBits < cTotalBits) {
    // Write the fixed-width tag
    addCodeText(0, 1);
}

```

```

// Find the minimum symbol, and encode all further symbols as
// positive number excess of this minimum value. This will save
// us at least half a bit on average because _nBitsInSymbol()
// doesn't have to add a bit to account for negative numbers.
Int32 iMinSymbol = 0x7fffffff;
Int32 iMaxSymbol = 0x80000000;
for (Int32 i = 0 ; i < nValues ; i++) {
    iMinSymbol = ::min(iMinSymbol, paiValues[i]);
    iMaxSymbol = ::max(iMaxSymbol, paiValues[i]);
}
Int32 cBitsInMinSymbol;
Int32 cBitsInMaxSymbol;
// If some symbols are negative, then we shall include a sign bit
cBitsInMinSymbol = _nBitsInSymbol(iMinSymbol) + 1;
cBitsInMaxSymbol = _nBitsInSymbol(iMaxSymbol) + 1;
// Write the min and max symbols into the stream
addCodeText(cBitsInMinSymbol, 6);
addCodeText(cBitsInMaxSymbol, 6);
addCodeText((UInt32) iMinSymbol, cBitsInMinSymbol);
addCodeText((UInt32) iMaxSymbol, cBitsInMaxSymbol);

// Use a fixed number of bits for all symbols
cSymBits = _nBitsInSymbol(iMaxSymbol - iMinSymbol);

// Iterate over the remaining symbols
UInt32 uCodeText = 0;
for (Int32 i = 0; i < nValues; i++ ) {

    // Get the next symbol
    iSymbol = paiValues[i];

    // Write it
    uCodeText = iSymbol - iMinSymbol;
    addCodeText(uCodeText, cSymBits);
}
}
/////
// Otherwise, encode with variable-length fields
/////
else {
    // Write the variable-width tag
    addCodeText(1, 1);

    // Write out the mean value
    addCodeText(iMean, 32);
    addCodeText(cBlkValBits, 3);
    addCodeText(cBlkLenBits, 3);

    // Set the initial field-width
    Int32 cMaxFieldDecr = -(1 << (cBlkValBits - 1)), // -ve number
    cMaxFieldIncr = (1 << (cBlkValBits - 1)) - 1; // +ve number
    Int32 cCurFieldWidth = 0;
    Int32 cTargFieldWidth;
    for (Int32 ii = 0 ; ii < nValues ;) {
        // Adjust the current field width to the target field width
        cTargFieldWidth = paiSymBits[ii];
        {
            if (cCurFieldWidth <= cTargFieldWidth) {
                while (cTargFieldWidth - cCurFieldWidth >= cMaxFieldIncr) {
                    addCodeText(cMaxFieldIncr, cBlkValBits);
                    cCurFieldWidth += cMaxFieldIncr;
                }
            }
            else {
                while (cTargFieldWidth - cCurFieldWidth <= cMaxFieldDecr) {
                    addCodeText(cMaxFieldDecr, cBlkValBits);

```

```

        cCurFieldWidth += cMaxFieldDecr;
    }
    }
    addCodeText(cTargFieldWidth - cCurFieldWidth, cBlkValBits);
    cCurFieldWidth = cTargFieldWidth;
}

// Write out the run length
for (j = ii+1 ; j < ii + (1 << cBlkLenBits) - 1 && j < nValues ; j++)
    if (paiSymBits[ii] != paiSymBits[j])
        break;
addCodeText(j - ii, cBlkLenBits);

// Write out the data bits for the run
for (k = ii ; k < j ; k++)
    addCodeText(paiValues[k] - iMean, cCurFieldWidth);

// Advance to the end of the run
ii = j;
}
}

return True;
}

Bool
BitLengthCodec2::decode( Int32      nValues,
                        const Vecu  &vCodeText,
                        Int32      nBitsCodeText,
                        Veci       &ovValues )
{
    Int32  nBits = 0,          // Number of codetext bits decoded so far
    nTotalBits = 0,          // Total number of codetext bits expected
    nValBits = 0,            // Number of accumulated value bits
    iSymbol;                // Decoded symbol value
    UInt32 uVal = 0;          // Current chunk of codetext bits
    Int32  cNumCurBits = 0;   // Current field width in bits
    Int32  cBitsInMinSymbol;   // Number of bits in the minimum symbol
    Int32  cBitsInMaxSymbol;   // Number of bits in the maximum symbol
    Int32  iMinSymbol = 0;     // The minimum symbol value. Used as bias.
    Int32  iMaxSymbol = 0;     // The maximum symbol value. Used as bias.
    Int32  nSyms = 0;          // Number of symbols read so far
    Int32  *paiValues;         // Pointer into ovValues where we write decoded
    values

    // Get codetext from the driver and loop over it until it's gone!
    ovValues.setLength(nValues);
    paiValues = ovValues.ptr();

    _iCurCodeText = 0;
    _pvCodeText = (Vecu*) &vCodeText;
    _pcCodeTextLen = &nBitsCodeText;

    /////
    // If the fixed-width total bits are better, then write out the values
    // in a single fixed-width format.
    /////
    // Read the variable-width tag
    Int32 iTmp;
    GetUnsignedBits(iTmp, 1); // 0 = Fixed-width, 1 = Variable width
    if (iTmp == 0) {
        // Read the min and max symbols from the stream
        GetUnsignedBits(cBitsInMinSymbol, 6);
        GetUnsignedBits(cBitsInMaxSymbol, 6);
        GetSignedBits (iMinSymbol, cBitsInMinSymbol);
        GetSignedBits (iMaxSymbol, cBitsInMaxSymbol);
        cNumCurBits = _nBitsInSymbol(iMaxSymbol - iMinSymbol);
    }
}

```

```

// Read each fixed-width field and output the value
while (nBits < nTotalBits || nSyms < nValues) {
    GetUnsignedBits(iSymbol, cNumCurBits);
    iSymbol += iMinSymbol;
    *paiValues++ = iSymbol;
    nSyms++;
}
}
/////
// Otherwise, encode with variable-length fields
/////
else {
    // Write out the mean value
    Int32 iMean, cBlkValBits, cBlkLenBits;
    GetSignedBits(iMean, 32);
    GetUnsignedBits(cBlkValBits, 3);
    GetUnsignedBits(cBlkLenBits, 3);

    // Set the initial field-width
    Int32 cMaxFieldDecr = -(1 << (cBlkValBits - 1)), // -ve number
          cMaxFieldIncr = (1 << (cBlkValBits - 1)) - 1; // +ve number
    Int32 cCurFieldWidth = 0, cDeltaFieldWidth, cRunLen, k;
    for (Int32 ii = 0 ; ii < nValues ;) {
        // Adjust the current field width to the target field width
        do {
            GetSignedBits(cDeltaFieldWidth, cBlkValBits);
            cCurFieldWidth += cDeltaFieldWidth;
        } while (cDeltaFieldWidth == cMaxFieldDecr || cDeltaFieldWidth ==
cMaxFieldIncr);

        // Read in the run length
        GetUnsignedBits(cRunLen, cBlkLenBits);

        // Read in the data bits for the run
        for (k = ii ; k < ii + cRunLen ; k++) {
            GetSignedBits(iTmp, cCurFieldWidth);
            *paiValues++ = iTmp + iMean;
        }

        // Advance to the end of the run
        ii += cRunLen;
    }

    // Assert that we have consumed exactly all of the bits
    Assert(nValBits == 0);
    Assert(uVal == 0);

    return True;
}

Int32 BitLengthCodec2::_nBitsInSymbol(Int32 iSymbol) const
{
    // Zero is the only number that can be encoded in a
    // single bit with this scheme, so this method returns
    // 0 bits for a symbol value of zero!
    if (iSymbol == 0)
        return 0;

    // Note: This calculation assumes that iSymbol is
    // positive!
    Int32 cMaxCodeSpan = ::abs(iSymbol);
    Int32 i, nBits;
    for (i = 1, nBits = 0 ; i <= cMaxCodeSpan && nBits < 31 ; i += i, nBits++);
    return nBits;
}

```

```

}

Bool
BitLengthCodec2::getNextCodeText (UInt32 &uCodeText, Int32 &nBits)
{
    uCodeText = _pvCodeText->value(_iCurCodeText);
    nBits = ::min(32, *_pcCodeTextLen - 32 * _iCurCodeText);
    _iCurCodeText++;
    return True;
}

```

D.3 Arithmetic decoding classes

The following sub-sections contain a sample implementation of the decoding portion of the Arithmetic CODEC algorithm. A summary technical explanation of the Arithmetic CODEC can be found in [10.2.3 Arithmetic CODEC](#).

D.3.1 ArithmeticProbabilityRange class

```

class ArithmeticProbabilityRange
{
public:
    UInt16 low_count;
    UInt16 high_count;
    UInt16 scale;
};

```

D.3.2 ArithmeticCodec class

```

class ArithmeticCodec
{
public:
    ArithmeticCodec() :
        code = 0x0000,
        low = 0x0000,
        high = 0xffff,
        nUnderflowBits = 0,
        bitBuffer = 0x00000000,
        nBits = 0 {}

    // Encodes a list of symbols.
    Bool encode( CodecDriver* pDriver );

    // 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:
    void encodeSymbol(ArithmeticProbabilityRange &sym );
    // Remove the most recently decoded symbol from the front of the
    // list of encoded symbols.
    Bool removeSymbolFromStream( ArithmeticProbabilityRange& sym,
                                CodecDriver* pDriver);

    void flushEncoder();
    void writeBit(UInt16 bit );

    //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::encode( CodecDriver *pDriver )
{
    _pDriver = pDriver;

    // Initialize the encoder state
    _iCurContext = 0;
    _iCurSymbol = 0;
    _low = 0x0000;
    _high = 0xffff;
    _nUnderflowBits = 0;

    _bitBuffer = 0x00000000;
    _nBits = 0;
    const Int32 *aiSymbol;
    const Int32 *acSymCount;
    const Int32 *acCumCount;
    const Int32 *acTotalCount;
    const Int32 *aiSymContext;

    pDriver->getInputSymbols(aiSymbol, acSymCount, acCumCount, acTotalCount,
aiSymContext);

    ArithmeticProbabilityRange newSymbolRange;
    Int32 nSymbols = pDriver->numInputSymbols();
    for (Int32 i = 0; i < nSymbols; i++ ) {
#ifdef DEBUG
        Assert( acCumCount[i] <= 0xffff );
        Assert( acSymCount[i] + acCumCount[i] <= 0xffff );
        Assert( acTotalCount[i] <= 0xffff );
#endif
        newSymbolRange.low_count = acCumCount[i];
        newSymbolRange.high_count = acSymCount[i] + acCumCount[i];
        newSymbolRange.scale = acTotalCount[i];
        Assert( acSymCount[i] > 0 );

        encodeSymbol(newSymbolRange);
    }

    flushEncoder();

    _pDriver = 0;
    return True;
}

void ArithmeticCodec::encodeSymbol(ArithmeticProbabilityRange &sym )
{
    // These three lines rescale _high and _low for the new symbol.
    UInt32 range = UInt32(_high - _low) + 1;
    _high = _low + (range * sym.high_count) / sym.scale - 1;
    _low = _low + (range * sym.low_count) / sym.scale;

    // This loop turns out new bits until _high and _low are far enough
    // apart to have stabilized.
    for (;;)
    {
        // If this test passes, it means that the most signif digits match,
        // and can be sent to the output stream.
        if ( (_high & 0x8000) == (_low & 0x8000) )
        {
            UInt16 bitToWrite = _high>>15;
            writeBit(bitToWrite);
            bitToWrite = ~bitToWrite;
            while ( _nUnderflowBits > 0 ) {
                writeBit(bitToWrite);
                _nUnderflowBits--;
            }
        }
    }
}

```

```

    }
}

// If this test passes, the numbers are in danger of underflow, because
// the most signif digits don't match, and the 2nd digits are just one
apart.
//
// _low = 01... and _high = 10...
else if ( ( _low & 0x4000 ) && !( _high & 0x4000 ) )
{
    _nUnderflowBits += 1;
    _low  &= 0x3fff;
    _high |= 0x4000;
}
else
    break;

// Shift all bits left.  Move 0 into _low and 1 into _high.
_low  <<= 1;
_high <<= 1;
_high |= 1;
}
}

void ArithmeticCodec::flushEncoder()
{
    // Write out some underflow bits and misc.
    writeBit( ( _low & 0x4000 ) >> 14 );

    _nUnderflowBits++;

    while ( _nUnderflowBits-- > 0 ) {
        writeBit( ( ~_low & 0x4000 ) >> 14 );
    }

    //Need 16 zeros at the end, for this decoding algorithm
    UInt32 zeroBit = 0x0000;
    for (int ii=0; ii<16; ii++) {
        writeBit(zeroBit);
    }

    //Flush out the local buffer
    if ( _nBits > 0 ) {
        _pDriver->addCodeText( _bitBuffer<<(32-_nBits), 32 );
    }
}

void ArithmeticCodec::writeBit(UInt16 bit )
{
    _bitBuffer <<= 1;
    _bitBuffer |= ((UInt32)bit & 0x1);
    _nBits++;

    if ( _nBits==32 ) {
        _pDriver->addCodeText(_bitBuffer, 32);
        _nBits = 0;
    }
}

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 ) {
            // Do nothing
        } 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 ) {
            // The returned nBits here will always be 32
            pDriver->getNextCodeText(bitBuffer, nBits);
        }

        code |= (UInt16)(bitBuffer >> 31);
        bitBuffer <<= 1;
        nBits--;
    }
}

```

D.3.3 ArithmeticCodec2 class

```

class ArithmeticCodec2: public Codec2
{
public:
    Bool encode( const Veci    &vValues,
                 Veci          &ovOOBValues,
                 Vecu          &ovCodeText,
                 Int32         &onBitsCodeText,

                 ProbContext2 *pProbCntx    );

    Bool decode( Int32         nValues,
                 const Veci    &vOOBValues,
                 const Vecu    &vCodeText,
                 Int32         nBitsCodeText,
                 Veci          &ovValues,
                 ProbContext2 *pProbCntx    );

protected:
    Bool _encodeSymbol(UInt16 uLowCt, UInt16 uHighCt, UInt16 uScale );
    Bool _flushEncoder();
    Bool _removeSymbolFromStream( UInt16 uLowCt, UInt16 uHighCt, UInt16 uScale );
    Bool _flushDecoder();

    Bool getNextCodeText (UInt32 &uCodeText, Int32 &nBits);

    UInt16 _code;           // Present input code value, for decoding only
    UInt16 _low;            // Start of the current code range
    UInt16 _high;           // End of the current code range
    Int32  _nUnderflowBits; // Number of underflow bits pending

    Vecu    * _pvCodeText;
    Int32    * _pCodeTextLen;
    Int32    _iCurCodeText;

    UInt32    _uBitBuff;
    Int32     _nBitBuff;
};

// Reads a bit and places it into ouBit
#define ReadBit(ouBit)
    if (_nBitBuff==0) {
        getNextCodeText(_uBitBuff, _nBitBuff);
    }
    ouBit = (_uBitBuff >> 31);
    _uBitBuff <<= 1;
    _nBitBuff--;

// Reads a bit and ORs it into bit 0 of ouBit
#define ReadBit0(ouBit)
    if (_nBitBuff==0) {

```

```

        getNextCodeText(_uBitBuff, _nBitBuff); \
    } \
    ouBit |= (_uBitBuff >> 31); \
    _uBitBuff <<= 1; \
    _nBitBuff--; \
// Writes bit 0 of uBit
#define WriteBit(uBit) \
    if (_nBitBuff==32) { \
        addCodeText(_uBitBuff, 32); \
        _uBitBuff = _nBitBuff = 0; \
    } \
    _uBitBuff <<= 1; \
    _uBitBuff |= (UInt32(uBit) & 0x1); \
    _nBitBuff++; \

Bool ArithmeticCodec2::encode(const Veci &vValues,
                             Veci &ovOOBValues,
                             Vecu &ovCodeText,
                             Int32 &onBitsCodeText,
                             ProbContext2 *pProbCntx )
{
    // Initialize output state
    ovOOBValues.setLength(0);
    ovCodeText.setLength(0);
    onBitsCodeText = 0;
    _pvCodeText = &ovCodeText;
    _pcCodeTextLen = &onBitsCodeText;

    // Initialize the encoder state
    _low = 0x0000;
    _high = 0xffff;
    _nUnderflowBits = 0;

    // Prime the bit buffer
    _uBitBuff = 0;
    _nBitBuff = 0;

    const JtInt32 *paiValues = vValues.ptr();
    const JtCntxEntry2 *pEntry;
    Int32 nValues = vValues.length();
    Int32 cTotalCount = pProbCntx->totalCount();
    for (Int32 i = 0; i < nValues; i++) {
        // Look up the value in the prob context
        pProbCntx->lookupValue(paiValues[i], pEntry );

        // If this is not the null context, then we emit an escape symbol,
        // move the context it specifies, and restart the translation of
        // the same value. Thus, a value may emit more than one symbol.
        if (pEntry->_iSym == CntxEntryBase2::CEBEscape) {
            ovOOBValues.append(paiValues[i]);
        }
        _encodeSymbol(pEntry->_cCumCount,
                     pEntry->_cCumCount + pEntry->_cCount,
                     cTotalCount));
    }

    _flushEncoder();

    return True;
}

Bool ArithmeticCodec2::_encodeSymbol(UInt16 uLowCt, UInt16 uHighCt, UInt16 uScale )
{
    // These three lines rescale _high and _low for the new symbol.
    UInt32 uRange = UInt32(_high - _low) + 1;
    _high = _low + (uRange * uHighCt) / uScale - 1;

```

```

_low = _low + (uRange * uLowCt) / uScale;

// This loop turns out new bits until _high and _low are far enough
// apart to have stabilized.
for (;;) {
    // If this test passes, it means that the most signif digits match,
    // and can be sent to the output stream.
    if ( (_high & 0x8000) == (_low & 0x8000) )
    {
        // Flush the bit buff if the MSB and underflow bits
        // won't all fit in what's left
        if (1+_nUnderflowBits > 32 - _nBitBuff) {
            addCodeText(_uBitBuff, _nBitBuff);
            _uBitBuff = 0;
            _nBitBuff = 0;
        }
        // Write the MSB and all uflow bits at once
        if (1+_nUnderflowBits <= 32 - _nBitBuff) {
            _uBitBuff <<= (1 + _nUnderflowBits);
            _uBitBuff |= (1 << _nUnderflowBits
                + (Int32(JtInt16(~_high)) >> 15);
            _nBitBuff += 1 + _nUnderflowBits;
            _nUnderflowBits = 0;
        }
        else {
            // We're writing more than 32 bits!
            _uBitBuff = (1 << 31)
                + (Int32(JtInt16(~_high)) >> 15);
            addCodeText(_uBitBuff, 32);
            _nBitBuff = 0;
            _nUnderflowBits -= 31;
            // Emit the rest of the underflow bits
            _uBitBuff = (_uBitBuff << 1) | (_uBitBuff & 1);
            while (_nUnderflowBits >= 32) {
                addCodeText(_uBitBuff, 32);
                _nUnderflowBits -= 32;
            }
            addCodeText(_uBitBuff, _nUnderflowBits);
            _nUnderflowBits = 0;
            _uBitBuff = 0;
        }
    }

    // If this test passes, the numbers are in danger of underflow, because
    // the most signif digits don't match, and the 2nd digits are just one
    apart.
    //
    // _low = 01... and _high = 10...
    else if ( (_low & 0x4000) && !(_high & 0x4000) )
    {
        _nUnderflowBits++;
        _low &= 0x3fff;
        _high |= 0x4000;
    }
    else
        break;

    //Shift all bits left. Move 0 into _low and 1 into _high.
    _low <<= 1;
    _high <<= 1;
    _high |= 1;
}

return True;
}

Bool ArithmeticCodec2::_flushEncoder()

```

```

{
    // Write out some underflow bits and misc.
    WriteBit((_low & 0x4000)>>14)
    _nUnderflowBits++;
    while (_nUnderflowBits-- > 0)
        WriteBit((~_low & 0x4000)>>14)

    //Need 16 zeros at the end, for this decoding algorithm
    UInt32 zeroBit = 0x0000;
    for (Int32 ii=0; ii<16; ii++) {
        WriteBit(zeroBit)
    }

    // Flush out the local buffer
    addCodeText(_uBitBuff, _nBitBuff);

    return True;
}

Bool ArithmeticCodec2::decode( Int32          nValues,
                               const Veci    &vOOBValues,
                               const Vecu    &vCodeText,
                               Int32         nBitsCodeText,
                               Veci          &ovValues,
                               ProbContext2 *pProbCntx )
{
    ovValues.setLength(0);
    const Int32 *paiOOBValues = vOOBValues.ptr();
    ovValues.setLength(nValues);
    Int32 *paiValues = ovValues.ptr();
    Int32 cSymbolsCurrCtx = pProbCntx->totalCount();
    const CntxEntry2 *pCntxEntry = 0;

    // Initialize the arithmetic decoder state
    _iCurCodeText = 0;
    _pvCodeText = (Vecu*) &vCodeText;
    _pcCodeTextLen = &nBitsCodeText;
    getNextCodeText(_uBitBuff, _nBitBuff);
    _low = 0;
    _high = 0xffff;
    _code = (_uBitBuff >> 16);
    _uBitBuff <<= 16;
    _nBitBuff -= 16;

    // Decode each symbol
    for (Int32 i = 0; i < nValues; i++) {
        // Scale the current "code" into the range of counts presented by
        // the probcontext so we can look up the code.
        UInt16 rescaledCode = (((UInt32)(_code - _low) + 1) *
(UInt32)cSymbolsCurrCtx -
1)
        / ((UInt32)(_high - _low) + 1);
        pProbCntx->lookupEntryByCumCount( (Int32)rescaledCode, pCntxEntry );

        // Emit the value corresponding to the symbol we just decoded
        if (pCntxEntry->_iSym != CntxEntryBase2::CEBEscape)
            *paiValues++ = pCntxEntry->_val;
        else
            *paiValues++ = *paiOOBValues++;

        // Set up the symbol's range and adjust the decoder state
        // to "remove" it.
        _removeSymbolFromStream( pCntxEntry->_cCumCount,
                                pCntxEntry->_cCumCount + pCntxEntry->_cCount,
                                cSymbolsCurrCtx );
    }
}

```

```

    _flushDecoder();

    return True;
}

Bool ArithmeticCodec2::_flushDecoder()
{
    UInt32 dummyBit;
    ReadBit(dummyBit)
    ReadBit(dummyBit)

    Assert( _uBitBuff == 0 );
    _nBitBuff = 0;

    return True;
}

Bool ArithmeticCodec2::_removeSymbolFromStream( UInt16 uLowCt, UInt16 uHighCt,
UInt16
uScale )
{
    // First, the range is expanded to account for the symbol removal.
    UInt32 uRange = UInt32(_high - _low) + 1;
    _high = _low + (UInt32)((uRange * uHighCt) / uScale - 1);
    _low = _low + (UInt32)((uRange * uLowCt) / uScale);

    //Next, any possible bits are shipped out.
    for (;;) {
        // If the most signif digits match, the bits will be shifted out.
        if (UInt16(~(_high ^ _low)) >> 15){
        }
        // Else, if underflow is threatening, shift out the 2nd most signif digit.
        //else if ((_low & 0x4000) && !(_high & 0x4000))
        // If high=10xx and low=01xx
        else if (((_low >> 14) == 1) & ((_high >> 14) == 2)) {
            _code ^= 0x4000;
            _low  &= 0x3fff;
            _high |= 0x4000;
        }
        // Otherwise, nothing can be shifted out, so return.
        else {
            return True;
        }

        _low <<= 1;
        _high <<= 1;
        _high |= 1;
        _code <<= 1;

        ReadBit0( _code )
    }
}

```

D.4 Deering Normal decoding classes

The following sub-sections contain a sample implementation of the decoding portion of the Deering Normal CODEC algorithm. A summary technical explanation of the Deering Normal CODEC can be found in [10.2.4 Deering Normal CODEC](#).

D.4.1 DeeringNormalLookupTable class

The DeeringNormalLookupTable class represents a lookup table used by the DeeringNormalCodec class for faster conversion from the compressed normal representation to the standard 3-float representation. The tables hold precomputed results of the trig functions called during conversion.


```

class DeeringNormalLookupTable
{
public:
    DeeringNormalLookupTable();

    // Lookup and return the result of converting iTheta and iPsi to
    // real angles and taking the sine and cosine of both. This gives
    // a slight speedup for normal decoding.
    Bool lookupThetaPsi(Int32 iTheta,
                        Int32 iPsi,
                        UInt32 numberBits,
                        Float32 outCosTheta,
                        Float32 outSinTheta,
                        Float32 outCosPsi,
                        Float32 outSinPsi );

    UInt32 numBitsPerAngle() {return nBits;}

private:
    UInt32 nBits;
    Vector vCosTheta;
    Vector vSinTheta;
    Vector vCosPsi;
    Vector vSinPsi;
};

DeeringNormalLookupTable::DeeringNormalLookupTable()
{
    UInt32 numberbits = 8;
    nBits = min(numberbits, (UInt32)31);

    Int32 tableSize = (1 << nBits);

    vCosTheta.setLength(tableSize+1);
    vSinTheta.setLength(tableSize+1);
    vCosPsi.setLength(tableSize+1);
    vSinPsi.setLength(tableSize+1);

    Float32 fPsiMax = 0.615479709;
    Float32 fTableSize = (Float32)tableSize;

    for( Int32 ii = 0; ii <= tableSize; ii++ )
    {
        Float32 fTheta =
            asin(tan(fPsiMax * Float32(tableSize - ii) / fTableSize));

        Float32 fPsi = fPsiMax * (((Float32)ii) / fTableSize);
        vCosTheta[ii] = cos(fTheta);
        vSinTheta[ii] = sin(fTheta);
        vCosPsi[ii] = cos(fPsi);
        vSinPsi[ii] = sin(fPsi);
    }
}

Bool DeeringNormalLookupTable::lookupThetaPsi(Int32 iTheta,
                                                Int32 iPsi,
                                                UInt32 numberBits,

                                                Float32 outCosTheta,
                                                Float32 outSinTheta,
                                                Float32 outCosPsi,
                                                Float32 outSinPsi)
{
    Int32 offset = nBits - numberBits;

    outCosTheta = vCosTheta[iTheta << offset];
    outSinTheta = vSinTheta[iTheta << offset];
    outCosPsi = vCosPsi[iPsi << offset];

```

```

    outSinPsi    = vSinPsi[iPsi << offset];

    return True;
}

```

D.4.2 DeeringNormalCodec class

The DeeringNormalCodec class converts a normal vector to and from the standard 3-float representation and a lower-precision representation. The precision can be adjusted using the nbits parameter.

```

class DeeringNormalCodec
{
public:
    DeeringNormalCodec(Int32 numberbits = 6)
    {
        numBits = numberbits;
    }

    // Converts a compressed normal into a vector.
    Bool convertCodeToVec(UInt32 code, Vector& outVec);

    // Converts a compressed normal into a vector.
    Bool convertCodeToVec(UInt32 iSextant,
        UInt32 iOctant,
        UInt32 iTheta,
        UInt32 iPsi,
        Vector& outVec);

    // Separates an encoded normal into its 4 pieces
    Bool unpackCode(UInt32 code,
        UInt32& outSextant,
        UInt32& outOctant,
        UInt32& outTheta,
        UInt32& outPsi );

private:
    Int32 numBits;
}

Bool DeeringNormalCodec::convertCodeToVec(UInt32 code, Vector& outVec)
{
    UInt32 s=0, o=0, t=0, p=0;
    unpackCode(code, s, o, t, p);

    convertCodeToVec(s, o, t, p, outVec);

    return True;
}

Bool DeeringNormalCode::convertCodeToVec(UInt32 iSextant,
                                           UInt32 iOctant,
                                           UInt32 iTheta,
                                           UInt32 iPsi,
                                           Vector& outVec)
{
    // Size of code = 6+2*numBits, and max code size is 32 bits,
    // so numBits shall be <= 13.

    // Code layout: [sextant:3][octant:3][theta:numBits][psi:numBits]

    outVec.setValues(0,0,0);
    Float32 fPsiMax = 0.615479709;

    UInt32 iBitRange = 1<<numBits;
    Float32 fBitRange = Float32(iBitRange);

```

```

// For sextants 1, 3, and 5, iTheta needs to be incremented
iTheta += (iSextant & 1);

Float32 fCosTheta, fSinTheta, fCosPsi, fSinPsi;

DeeringNormalLookupTable LookupTable;

if( (LookupTable.numBitsPerAngle() < (UInt32)numBits) ||
    !LookupTable.lookupThetaPsi(iTheta, iPsi, numBits,
                                fCosTheta, fSinTheta,
                                fCosPsi, fSinPsi) )
{
    Float32 fTheta = asin(tan(fPsiMax * Float32(iBitRange - iTheta) /
                                                fBitRange));

    Float32 fPsi = fPsiMax * (iPsi / fBitRange);
    fCosTheta = cos(fTheta);
    fSinTheta = sin(fTheta);
    fCosPsi   = cos(fPsi);
    fSinPsi   = sin(fPsi);
}

Float32 x,y,z;
Float32 xx = x = fCosTheta * fCosPsi;
Float32 yy = y = fSinPsi;
Float32 zz = z = fSinTheta * fCosPsi;

//Change coordinates based on the sextant
switch( iSextant )
{
    case 0:      // No op
        break;

    case 1:      // Mirror about x=z plane
        z = xx;
        x = zz;
        break;

    case 2:      // Rotate CW
        z = xx;
        x = yy;
        y = zz;
        break;

    case 3:      // Mirror about x=y plane
        y = xx;
        x = yy;
        break;

    case 4:      // Rotate CCW
        y = xx;
        z = yy;
        x = zz;
        break;

    case 5:      // Mirror about y=z plane
        z = yy;
        y = zz;
        break;
};

//Change some more based on the octant

//if first bit is 0, negate x component
if( !(iOctant & 0x4) )
    x = -x;

```

```

    //if second bit is 0, negate y component
    if( !(iOctant & 0x2) )
        y = -y;

    //if third bit is 0, negate z component
    if( !(iOctant & 0x1) )
        z = -z;

    outVec.setValues(x,y,z);

    return True;
}

Bool DeeringNormalCodec::unpackCode(UInt32 code,
                                     UInt32& outSextant,
                                     UInt32& outOctant,
                                     UInt32& outTheta,
                                     UInt32& outPsi)
{
    UInt32 mask = (1<<numBits)-1;

    outSextant = (code >> (numBits+numBits+3)) & 0x7;
    outOctant = (code >> (numBits+numBits)) & 0x7;
    outTheta = (code >> (numBits)) & mask;
    outPsi = (code) & mask;

    return True;
}

```

Annex E (informative)

Hashing – An Implementation

This Appendix provides a sample C++ implementation for the creation of hash values (as detailed in 10.2 Encoding Algorithms) used in the ISO 14306 format.

```

unsigned int hash32( const unsigned int *pWords,
                    int nWords,
                    unsigned int uSeedHashValue )
{ return hash2(pWords, nWords, uSeedHashValue); }

unsigned int jthash16(const unsigned short *pBytes,
                    int nShort,
                    unsigned int uSeedHashValue)
{ return hash3(pBytes, nShort, uSeedHashValue); }

//-----
// mix -- mix 3 32-bit values reversibly.
// For every delta with one or two bit set, and the deltas of all three
// high bits or all three low bits, whether the original value of a,b,c
// is almost all zero or is uniformly distributed,
// * If mix() is run forward or backward, at least 32 bits in a,b,c
// have at least 1/4 probability of changing.
// * If mix() is run forward, every bit of c will change between 1/3 and
// 2/3 of the time. (Well, 22/100 and 78/100 for some 2-bit deltas.)
// mix() was built out of 36 single-cycle latency instructions in a
// structure that could supported 2x parallelism, like so:
//     a -= b;
//     a -= c; x = (c>>13);
//     b -= c; a ^= x;
//     b -= a; x = (a<<8);
//     c -= a; b ^= x;
//     c -= b; x = (b>>13);
//     ...
// Unfortunately, superscalar Pentiums and Sparcs can't take advantage
// of that parallelism. They've also turned some of those single-cycle
// latency instructions into multi-cycle latency instructions. Still,
// this is the fastest good hash I could find. There were about 2^68
// to choose from. I only looked at a billion or so.
//-----

#define mix(a,b,c) \
{ \
    a -= b; a -= c; a ^= (c>>13); \
    b -= c; b -= a; b ^= (a<<8); \
    c -= a; c -= b; c ^= (b>>13); \
    a -= b; a -= c; a ^= (c>>12); \
    b -= c; b -= a; b ^= (a<<16); \
    c -= a; c -= b; c ^= (b>>5); \
    a -= b; a -= c; a ^= (c>>3); \
    b -= c; b -= a; b ^= (a<<10); \
    c -= a; c -= b; c ^= (b>>15); \
}

//-----
// hash() -- hash a variable-length key into a 32-bit value
// k      : the key (the unaligned variable-length array of bytes)
// len    : the length of the key, counting by bytes
// level  : can be any 4-byte value
// Returns a 32-bit value. Every bit of the key affects every bit of

```

```

// the return value. Every 1-bit and 2-bit delta achieves avalanche.
// About 36+6len instructions.

// The best hash table sizes are powers of 2. There is no need to do
// mod a prime (mod is sooo slow!). If you need less than 32 bits,
// use a bitmask. For example, if you need only 10 bits, do
//   h = (h & hashmask(10));
// In which case, the hash table should have hashsize(10) elements.
//
// If you are hashing n strings (JtUInt8 **)k, do it like this:
//   for (i=0, h=0; i<n; ++i) h = hash( k[i], len[i], h);
//
// By Bob Jenkins, 1996. bob_jenkins@burtleburtle.net. You may use this
// code any way you wish, private, educational, or commercial. It's free.
//
// See http://burtleburtle.net/bob/ // 2010/02/12
// See http://burtleburtle.net/bob/hash/doobs.html // 2010/02/12
//
// Use for hash table lookup, or anything where one collision in 2^32 is
// acceptable. Do NOT use for cryptographic purposes.
//-----

//-----
// This works on all machines. hash2() is identical to hash() on
// little-endian machines, except that the length has to be measured
// in ub4s instead of bytes. It is much faster than hash(). It
// requires
// -- that the key be an array of UInt32's, and
// -- that all your machines have the same endianness, and
// -- that the length be the number of UInt32's in the key
// -----
unsigned int hash(const unsigned char *k, // key
                 unsigned int length, // length of the key
                 unsigned int initval) // prev hash, or an arbitrary value
{
    register unsigned int a,b,c,len;

    /* Set up the internal state */
    len = length;
    a = b = 0x9e3779b9; /* the golden ratio; an arbitrary value */
    c = initval; /* the previous hash value */
    /*----- handle most of the key */
    while (len >= 12) {
        a += (k[0] + ((UInt32)k[1]<<8) + ((UInt32)k[2]<<16) + ((UInt32)k[3]<<24));
        b += (k[4] + ((UInt32)k[5]<<8) + ((UInt32)k[6]<<16) + ((UInt32)k[7]<<24));
        c += (k[8] + ((UInt32)k[9]<<8) + ((UInt32)k[10]<<16) + ((UInt32)k[11]<<24));
        mix(a,b,c);
        k += 12; len -= 12;
    }
    /*----- handle the last 11 bytes */
    c += length;
    switch(len) { /* all the case statements fall through */
        case 11: c+=((UInt32)k[10]<<24);
        case 10: c+=((UInt32)k[9]<<16);
        case 9 : c+=((UInt32)k[8]<<8);
        /* the first byte of c is reserved for the length */
        case 8 : b+=((UInt32)k[7]<<24);
        case 7 : b+=((UInt32)k[6]<<16);
        case 6 : b+=((UInt32)k[5]<<8);
        case 5 : b+=k[4];
        case 4 : a+=((UInt32)k[3]<<24);
        case 3 : a+=((UInt32)k[2]<<16);
        case 2 : a+=((UInt32)k[1]<<8);
        case 1 : a+=k[0];
        /* case 0: nothing left to add */
    }
    mix(a,b,c);
}

```

```

/*----- report the result */
return c;
}

unsigned int hash3(const unsigned short *k,      /* the key */
                  unsigned int      length,    /* the length of the key */
                  unsigned int      initval) /* the previous hash, or an
arbitrary value */
{
    unsigned int a,b,c,len;

    /* Set up the internal state */
    len = length;
    a = b = 0x9e3779b9; /* the golden ratio; an arbitrary value */
    c = initval;        /* the previous hash value */

    /*----- handle most of the key */
    while (len >= 6)
    {
        a += (k[0] + (UInt32(k[1]) << 16));
        b += (k[2] + (UInt32(k[3]) << 16));
        c += (k[4] + (UInt32(k[5]) << 16));
        mix(a,b,c);
        k += 6; len -= 6;
    }

    /*----- handle the last 2 uint32s */
    c += length;
    switch(len) /* all the case statements fall through */
    {
        case 5 : c+=(UInt32(k[4]) << 16);
        /* c is reserved for the length */
        case 4 : b+=(UInt32(k[3]) << 16);
        case 3 : b+=k[2];
        case 2 : a+=(UInt32(k[1]) << 16);
        case 1 : a+=k[0];
        /* case 0: nothing left to add */
    }
    mix(a,b,c);
    /*----- report the result */
    return c;
}

```

Annex F (informative) Polygon Mesh Topology Coder

The topology coding algorithm described here is used to code the *dual* of the desired mesh. Thus, for example, the reader will need to take the dual of the decoded mesh in order to obtain the original primal mesh. Presented below are classes suitable for representing the dual of a polygon mesh and the dual topology decoding algorithm.

At a high level, the topology coder works by traversing the dual mesh to be encoded one vertex and one face at a time. The coder maintains a queue of faces to be processed; the initial queue is created using the valence of an arbitrary vertex of the mesh followed by the degrees of the faces adjacent to that vertex, and adds the adjacent faces to the face queue. Each time it visits a face, it encodes the *degree* of that face and emits any incident vertices that have not yet been visited. Each time the coder visits a vertex, it encodes the *valence* of the vertex (usually 3 in the current case), and emits any incident faces that have not yet been visited. It works its way through the mesh in this fashion until all vertices and faces have been encoded. Thus, the primary output from the topology coder is a list of vertex valences and face degrees. These two fields plus two more encoding so-called *split* faces, coupled with the exact coder implementation completely encode the mesh topology in a very compact manner¹.

In addition to these two basic fields are added a number of other fields that organize the dual vertices into *vertex groups*, and also encode the *vertex attributes* (e.g. normals, colours, and texture coordinates) around each dual face's *degree ring*.

The topological coder can only encode *closed, manifold* meshes. It cannot encode *boundaries*; it can only encode edges with exactly two incident faces. But, as we know, real-world data is chock full of meshes with boundaries. In order to encode these types of meshes, it is necessary to add *cover faces* incident to all boundary loops whose sole job is to turn the mesh into a *closed* mesh. It is the dual of this closed, manifold mesh that is actually encoded. Thus, most meshes encoded in JT files contain a few cover faces. These faces may be of arbitrarily high degree, and they represent the only exceptions to the general rule that the numbers in the dual vertex valence array are usually three. It is necessary to flag all such artificially introduced cover faces so that they can be removed by the loader. These flags are encoded below in the Face Flags array. Primal faces are flagged with zero, while cover faces are flagged with one.

Now, let us make the connection between topological vertices and how vertex attributes relate to them. Several faces may be incident on the same topological mesh vertex. While this topological vertex has only a single 3D coordinate, it may have a different set of *vertex attributes* for each incident face. Vertex attributes include colour, normal, and texture coordinates. An important observation in real-world data is that adjacent faces tend to share the same vertex attributes. Thus, a natural way to encode which vertex attributes map to which faces within a given valence ring (the counter-clockwise ordered set of faces incident on a given vertex) is by way of a bit vector. The bit vector begins at the first face the coder encounters that is incident to the vertex, and proceeds counter clockwise around the vertex, allocating one bit per incident face. A value of 0 is assigned to the bit if all vertex attributes for the face are the same as the face immediately clockwise. A value of 1 is assigned if the vertex attributes for the face are different. Recall that these bits from the original primal mesh are encoded as face attributes in the dual mesh.

Thus, at the end of the coding process, there will be one such bit vector per topological vertex in the mesh. These bit vectors will be of disparate lengths because all vertex valences are not the same.

¹ Similar methods of topology coding are described in [12] and US patent # 7,098,916. The topology coding algorithm described herein differs from such methods in that while they utilize a queue of active *vertices*, the instant algorithm utilizes a queue of active *faces*. Other differences include the tracking of face group numbers and per-vertex attributes such as normals, colours, and texture coordinates.

Though there is no theoretical limit to the valence of any given vertex, in practice, the vertex valences seldom rise above six, and only rarely rise into the dozens. As a matter of practicality, then, we break this list of bit vectors into those of length 64 and smaller into one group, and all others into a list of so-called “high-valence” bit vectors. The low-valence bit vectors are encoded into three fields of 30, 30, and 4 bits respectively. The high-valence bit vectors are adjoined end-to-end into a single long bit vector, and encoded as a single array of integers. As an additional optimization, the low-valence bit vectors are grouped into 8 “context groups” depending on the valence of the vertex being coded. This is done in order to improve compression performance because the valence bit vectors in each of the most common groups typically share similar statistics. Context group number 8 is the only one that encodes valence rings up to valence 64. Again, recall that these attribute bits from the original primal mesh are encoded as face attribute bits in the dual mesh.

F.1 DualVFMesh

The DualVFMesh (Dual Vertex-Facet Mesh) is a support class paired with the topology decoder itself, and represents a closed two-manifold polygon mesh. The topology decoder reconstructs the encoded dual mesh into a DualVFMesh, building it one vertex and one facet at a time. When the decoder is finished, it will have visited each vertex and each face of the dual mesh exactly once. DualVFMesh is not intended as a work horse in-memory storage container because its way of encoding the topological connections between faces and vertices is memory-intensive.

```
class DualVFMesh
{
public:
    // ===== Housekeeping Interface =====
    DualVFMesh();
    DualVFMesh (const DualVFMesh &rhs);
    DualVFMesh &operator=(const DualVFMesh &rhs);

    // ===== Topology Interface =====

    // Vtx creation
    bool        isValidVtx (Int32  iVtx) const;
    bool        newVtx      (Int32  iVtx,
                             Int32  iValence,
                             UInt16 uFlags = 0);
    bool        setVtxFlags (Int32  iVtx,
                             UInt16 uFlags);
    bool        setVtxGrp   (Int32  iVtx,
                             Int32  iVGrp);
    UInt16      vtxFlags    (Int32  iVtx) const;
    Int32       vtxGrp      (Int32  iVtx) const;

    // Face creation
    bool        isValidFace (Int32  iFace) const;
    bool        newFace     (Int32  iFace,
                             Int32  cDegree,
                             Int32  cFaceAttrs = 0,
                             UInt64 uFaceAttrMask = 0,
                             UInt16 uFlags = 0);
    bool        newFace     (Int32  iFace,
                             Int32  cDegree,
                             Int32  cFaceAttrs,
                             const BitVec *pVbFaceAttrMask,
                             UInt16 uFlags);
    bool        setFaceFlags (Int32  iFace,
                             UInt16 uFlags);
    UInt16      faceFlags    (Int32  iVtx) const;
    bool        setFaceAttr  (Int32  iFace,
                             Int32  iAttrSlot,
                             Int32  iFaceAttr);
    Int32       faceAttr     (Int32  iFace,
                             Int32  iAttrSlot) const;
```

```

// Topology connection
bool      setVtxFace(Int32  iVtx,
                    Int32  iFaceSlot,
                    Int32  iFace);
bool      setFaceVtx(Int32  iFace,
                    Int32  iVtxSlot,
                    Int32  iVtx);

// Queries
Int32      valence      (Int32  iVtx) const
{ return _vVtxEnts[iVtx].cVal; }
Int32      degree      (Int32  iFace ) const
{ return _vFaceEnts[iFace].cDeg; }
Int32      face        (Int32  iVtx,
                    Int32  iFaceSlot) const
{ return _viVtxFaceIndices[(_vVtxEnts[iVtx]).iVFI + iFaceSlot]; }
Int32      vtx          (Int32  iFace,
                    Int32  iVtxSlot) const
{ return _viFaceVtxIndices[_vFaceEnts[iFace].iFVI + iVtxSlot]; }
Int32      numVtxs      () const
{ return _vVtxEnts.length(); }
Int32      numFaces      () const
{ return _vFaceEnts.length(); }
Int32      numAttrs      () const
{ return _viFaceAttrIndices.length(); }
Int32      numAttrs      (Int32  iFace) const
{ return _vFaceEnts[iFace].cFaceAttrs; }
UInt64      attrMask      (Int32  iFace) const
{ return _vFaceEnts[iFace].u.uAttrMask; }
const BitVec *attrMaskV      (Int32  iFace) const
{ return _vFaceEnts[iFace].u.pvbAttrMask; }
Int32      findVtxSlot    (Int32  iFace,
                    Int32  iTargVtx) const;
Int32      findFaceSlot    (Int32  iVtx,
                    Int32  iTargFace) const;
Int32      emptyFaceSlots (Int32  iFace) const
{ return _vFaceEnts[iFace].cEmptyDeg; }

// ===== VFMesh Data Members =====
public:
class VtxEnt {
public:
    VtxEnt() : cVal(0), uFlags(0), iVGrp(-1), iVFI(-1) {}
    UInt16    cVal;    // Vtx valence
    UInt16    uFlags;  // User flags
    Int32     iVGrp;   // Vtx group
    Int32     iVFI;    // Idx into _viVtxFaceIndices of cVal incident faces
};

// Number of optimized mask bits.
static const Int32 cMBits = 64;

class FaceEnt {
public:
    FaceEnt() : cDeg(0), uFlags(0), cEmptyDeg(0),
                cFaceAttrs(0), iFVI(-1), iFAI(-1) { u.uAttrMask = 0; }
    FaceEnt(const FaceEnt &rhs) : cDeg(rhs.cDeg), cEmptyDeg(rhs.cEmptyDeg),
                                cFaceAttrs(rhs.cFaceAttrs), iFVI(rhs.iFVI),
                                iFAI(rhs.iFAI)
    {
        if (cDeg <= cMBits)
            u.uAttrMask = rhs.u.uAttrMask;
        else
            JtWrapNew(u.pvbAttrMask, new BitVec(*rhs.u.pvbAttrMask));
    }
    ~FaceEnt() { if (cDeg > cMBits && u.pvbAttrMask) delete u.pvbAttrMask; }
};

```

```

    UInt16    cDeg;          // Face degree
    UInt16    cEmptyDeg;     // Empty degrees (opt for emptyFaceSlots())
    UInt16    cFaceAttrs;    // Number of face attributes
    UInt16    uFlags;        // User flags
    union {
        UInt64  uAttrMask;    // Degree-ring attr mask as a UInt64
        BitVec  *pVbAttrMask; // Degree-ring attr mask as a BitVec
    } u;
    Int32      iFVI; // Idx into _viFaceVtxIndices of cDeg incident vts
    Int32      iFAI; // Idx into _viFaceAttrIndices of cAttr attributes
};

protected:
// Subscripted by atom number, the entry contains the vtx valence and
// points to the location in _viVtxFaceIndices of valence consecutive
// integers that in turn contain the indices of the incident faces
// in _vFaceRecs to the vtx.
JtVec<VtxEnt> _vVtxEnts;

// Subscripted by unique vertex record number, the entry contains the
// face degree and points to the location in _viFaceVtxIndices of
// cDeg consecutive integers that in turn contain the indices of the
// vertices incident upon the face, in CCW order, in _vVtxRecs.
JtVec<FaceEnt> _vFaceEnts;

// Combined storage for all vtxs.
JtVeci        _viVtxFaceIndices;

// Combined storage for all faces.
JtVeci        _viFaceVtxIndices;

// Combined storage for all face attribute record identifiers
JtVeci        _viFaceAttrIndices;
};

bool
DualVFMesh::isValidVtx(Int32 iVtx) const
{
    bool bRet = JtFalse;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        bRet = (rFE.cVal != 0);
    }
    return bRet;
}

bool
DualVFMesh::newVtx(Int32 iVtx,
                   Int32 iValence,
                   UInt16 uFlags)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    if (rFE.cVal != iValence) {
        rFE.cVal = iValence;
        rFE.uFlags = uFlags;
        rFE.iVFI = _viVtxFaceIndices.length();
        _viVtxFaceIndices.verify(rFE.iVFI + iValence - 1);
        for (Int32 i = rFE.iVFI; i < rFE.iVFI + iValence; i++)
            _viVtxFaceIndices[i] = -1;
    }
    return true;
}

bool
DualVFMesh::setVtxGrp(Int32 iVtx,
                      Int32 iVGrp)
{

```

```

    VtxEnt &rFE = _vVtxEnts[iVtx];
    rFE.iVGrp = iVGrp;
    return true;
}

bool
DualVFMesh::setVtxFlags(Int32 iVtx,
                        UInt16 uFlags)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    rFE.uFlags = uFlags;
    return true;
}

Int32
DualVFMesh::vtxGrp (Int32 iVtx) const
{
    Int32 u = -1;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        u = rFE.iVGrp;
    }
    return u;
}

UInt16
DualVFMesh::vtxFlags (Int32 iVtx) const
{
    UInt16 u = 0;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        u = rFE.uFlags;
    }
    return u;
}

bool
DualVFMesh::isValidFace(Int32 iFace) const
{
    bool bRet = JtFalse;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rFE = _vFaceEnts[iFace];
        bRet = (rFE.cDeg != 0);
    }
    return bRet;
}

bool
DualVFMesh::newFace(Int32 iFace,
                    Int32 cDegree,
                    Int32 cFaceAttrs,
                    UInt64 uFaceAttrMask,
                    UInt16 uFlags)
{
    FaceEnt &rFE = _vFaceEnts[iFace];
    if (rFE.cDeg != cDegree) {
        rFE.cDeg = cDegree;
        rFE.cEmptyDeg = cDegree;
        rFE.cFaceAttrs = cFaceAttrs;
        rFE.uFlags = uFlags;
        rFE.uAttrMask = uFaceAttrMask;
        rFE.iFVI = _viFaceVtxIndices.length();
        rFE.iFAI = _viFaceAttrIndices.length();
        _viFaceVtxIndices.verify(rFE.iFVI + cDegree - 1);
        if (cFaceAttrs > 0)
            _viFaceAttrIndices.verify(rFE.iFAI + cFaceAttrs - 1);
    }
}

```

```

        for (Int32 i = rVE.iFVI ; i < rVE.iFVI + cDegree ; i++)
            _viFaceVtxIndices[i] = -1;
        for (Int32 i = rVE.iFAI ; i < rVE.iFAI + cFaceAttrs ; i++)
            _viFaceAttrIndices[i] = -1;
    }
    return true;
}

bool
DualVFMesh::newFace(Int32 iFace,
                    Int32 cDegree,
                    Int32 cFaceAttrs,
                    const BitVec *pvbFaceAttrMask,
                    UInt16 uFlags)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    if (rVE.cDeg != cDegree) {
        rVE.cDeg = cDegree;
        rVE.cEmptyDeg = cDegree;
        rVE.cFaceAttrs = cFaceAttrs;
        rVE.uFlags = uFlags;
        rVE.u.pvbAttrMask = new BitVec(*pvbFaceAttrMask);
        rVE.iFVI = _viFaceVtxIndices.length();
        rVE.iFAI = _viFaceAttrIndices.length();
        _viFaceVtxIndices.verify(rVE.iFVI + cDegree - 1);
        if (cFaceAttrs > 0)
            _viFaceAttrIndices.verify(rVE.iFAI + cFaceAttrs - 1);
        for (Int32 i = rVE.iFVI ; i < rVE.iFVI + cDegree ; i++)
            _viFaceVtxIndices[i] = -1;
        for (Int32 i = rVE.iFAI ; i < rVE.iFAI + cFaceAttrs ; i++)
            _viFaceAttrIndices[i] = -1;
    }
    return true;
}

bool
DualVFMesh::setFaceFlags(Int32 iFace,
                        UInt16 uFlags)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    rVE.uFlags = uFlags;
    return true;
}

UInt16
DualVFMesh::faceFlags (Int32 iFace) const
{
    UInt16 u = 0;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rVE = _vFaceEnts[iFace];
        u = rVE.uFlags;
    }
    return u;
}

bool
DualVFMesh::setFaceAttr(Int32 iFace,
                       Int32 iAttrSlot,
                       Int32 iFaceAttr)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    Int32 *paiFAI = _viFaceAttrIndices.ptr();
    paiFAI[rVE.iFAI + iAttrSlot] = iFaceAttr;
    return true;
}

Int32

```

```

DualVFMesh::faceAttr(Int32 iFace,
                    Int32 iAttrSlot) const
{
    Int32 u = 0;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rVE = _vFaceEnts[iFace];
        if (iAttrSlot >= 0 && iAttrSlot < rVE.cDeg) {
            const Int32 *paiFAI = _viFaceAttrIndices.ptr();
            u = paiFAI[rVE.iFAI + iAttrSlot];
        }
    }
    return u;
}

// Attaches VF face iFace to VF vertex iVtx in the vertex's
// face slot iFaceSlot
bool
DualVFMesh::setVtxFace(Int32 iVtx,
                      Int32 iFaceSlot,
                      Int32 iFace)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    _viVtxFaceIndices[rFE.iVFI + iFaceSlot] = iFace;
    return true;
}

// Attaches VF vertex iVtx to VF face iFace in the face's
// vertex slot iVtxSlot
bool
DualVFMesh::setFaceVtx(Int32 iFace,
                      Int32 iVtxSlot,
                      Int32 iVtx)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    Int32 *paiFVI = _viFaceVtxIndices.ptr();
    rVE.cEmptyDeg -= (paiFVI[rVE.iFVI + iVtxSlot] != iVtx);
    paiFVI[rVE.iFVI + iVtxSlot] = iVtx;
    return true;
}

// Searches the list of incident vts to face iFace for
// iTargVtx and returns the vtx slot at which it is found
// or -1 if iTargVtx is not found.
Int32
DualVFMesh::findVtxSlot(Int32 iFace,
                      Int32 iTargVtx) const
{
    const FaceEnt &rVE = _vFaceEnts[iFace];
    const Int32 *const pFaceVtxIndices = _viFaceVtxIndices.ptr() + rVE.iFVI;
    Int32 cDeg = rVE.cDeg;
    Int32 iSlot = -1;
    for (Int32 iVtxSlot = 0 ; iVtxSlot < cDeg ; iVtxSlot++) {
        if (pFaceVtxIndices[iVtxSlot] == iTargVtx) {
            iSlot = iVtxSlot;
            break;
        }
    }
    return iSlot;
}

// Searches the list of incident faces to vertex iVtx for
// iTargFace and returns the face slot at which it is found
// or -1 if iTargFace is not found.
Int32
DualVFMesh::findFaceSlot (Int32 iVtx,
                      Int32 iTargFace) const
{

```

```

const VtxEnt &rFE = _vVtxEnts[iVtx];
const Int32 *const pVtxFaceIndices = _viVtxFaceIndices.ptr() + rFE.iVFI;
for (Int32 iFaceSlot = 0 ; iFaceSlot < rFE.cVal ; iFaceSlot++) {
    if (pVtxFaceIndices[iFaceSlot] == iTargFace) {
        return iFaceSlot;
    }
}
return -1;
}

```

F.2 Topology Decoder

Partial implementations of three classes are given here for MeshCoderDriver, MeshCodec, and MeshDecoder. MeshCodec contains the abstract implementation of the topology coder. MeshDecoder implements the functionality needed to *decode* a mesh from the input data read from a JT file (see Topologically Compressed Rep Data). MeshCoderDriver manages the input data, the output VFMesh, and the MeshDecoder itself, providing a simple three-step API.

F.2.1 MeshCoderDriver class

```

// This class serves as a coordinating driver for mesh coding and decoding.
class MeshCoderDriver
{
public:
    MeshCoderDriver ();

    // ===== Operations Interface =====
    void      setInputData(const Veci    vviOutValSyms[/*8*/],
                           const Veci    &viOutDegSyms,
                           const Veci    &viOutFGrpSyms,
                           const Vecus    &vuOutFaceFlags,
                           const Vecclu   vvuOutAttrMasks[/*8*/],
                           const Vecu     &vuOutAttrMasksLrg,
                           const Veci     &viOutSplitVtxSyms,
                           const Veci     &viOutSplitPosSyms)
    { /* Copy into 22 fields below */ }

    void      decode();
    VFMesh    *vfm() const { return _pOutVFM; }

    // ===== Utility Methods =====
    Int32      _nextDegSymbol      (Int32 iCCntx);
    Int32      _nextValSymbol      ();
    Int32      _nextFGrpSymbol     ();
    UInt16     _nextVtxFlagSymbol  ();
    UInt64     _nextAttrMaskSymbol(Int32 iCCntx); // <= 64-bit attrmask
    void        _nextAttrMaskSymbol(BitVec *iopvbAttrMask,
                                     Int32   cDegree); // > 64 bit attrmask

    Int32      _nextSplitFaceSymbol();
    Int32      _nextSplitPosSymbol();
    Int32      _faceCntxt(Int32 iVtx, JtDualVFMesh *pVFM);

    // ===== Member Data =====
protected:
    SharedPtr<MeshCodec>    _pMC; // The mesh coder or decoder being used
    SharedPtr<JtDualVFMesh> _pOutVFM; // Back-end VFMesh built by decoder
    SharedPtr<MeshDecoder> _pMeshDecoder;

    // Coding symbols generated by encoding operation, auxiliary data such as
    // offsets, etc.
    Veci      _vviOutDegSyms[8]; // Face degree + SPLIT symbols for multiple
    contexts
    Veci      _viOutValSyms; // Vtx valence symbols
    Veci      _viOutVGrpSyms; // Vtx group of each encoded vtx
    Vecus     _vuOutVtxFlags; // Vtx flags; parallel to _viOutValSyms.
}

```

```

    Vecclu      _vvuOutAttrMasks[8]; // Attribute bitmasks per face for multiple
contexts.

    Vecu        _vuOutAttrMasksLrg; // > 64-bit attrmasks
Veci          _viOutSplitFaceSyms; // Split face offsets
Veci          _viOutSplitPosSyms; // Split face vtx slots

    // The next symbol to be consumed by _next*Symbol()
Int32         _iValReadPos[8];
Int32         _iDegReadPos;
Int32         _iVGrpReadPos;
Int32         _iFFlagReadPos;
Int32         _iAttrMaskReadPos[8];
Int32         _iAttrMaskLrgReadPos;
Int32         _iSplitFaceReadPos;
Int32         _iSplitPosReadPos;
};

void MeshCoderDriver::decode()
{
    // Allocate a coder
    if (!_pMeshDecoder) {
        _pMeshDecoder = new MeshDecoder(this);
    }
    _pMC = _pMeshDecoder;
    _pMC->setTopoDualMeshCoder(this);

    // Reset the symbol counters
    for (Int32 i = 0 ; i < 8 ; i++) {
        _iValReadPos[i] = 0;
        _iAttrMaskReadPos[i] = 0;
    }
    _iDegReadPos = 0;
    _iVGrpReadPos = 0;
    _iFFlagReadPos = 0;
    _iAttrMaskLrgReadPos = 0;
    _iSplitFaceReadPos = 0;
    _iSplitPosReadPos = 0;

    // Run the decoder
    _pMC->run();

    // Assert that ALL symbols have been consumed
    for (Int32 i = 0 ; i < 8 ; i++) {
        Assert(_iValReadPos[i] == _vviOutDegSyms[i].length());
        Assert(_iAttrMaskReadPos[i] == _vvuOutAttrMasks[i].length());
    }
    Assert(_iDegReadPos == _viOutValSyms.length());
    Assert(_iVGrpReadPos == _viOutVGrpSyms.length());
    Assert(_iFFlagReadPos == _vuOutVtxFlags.length());
    Assert(_iAttrMaskLrgReadPos == _vvuOutAttrMasksLrg.length());
    Assert(_iSplitFaceReadPos == _viOutSplitFaceSyms.length());
    Assert(_iSplitPosReadPos == _viOutSplitPosSyms.length());

    // Set output VFMesh
    _pOutVFM = _pMC->vfm();
}

Int32 MeshCoderDriver::_nextDegSymbol (Int32 iCCntx)
{
    Int32 eSym = -1;
    if (_iValReadPos[iCCntx] < _vviOutDegSyms[iCCntx].length())
        eSym = _vviOutDegSyms[iCCntx].value(_iValReadPos[iCCntx]++);
    return eSym;
}

Int32

```



```

MeshCoderDriver::_nextValSymbol  ()
{
    Int32 eSym = -1;
    if (_iDegReadPos < _viOutValSyms.length())
        eSym = _viOutValSyms.value(_iDegReadPos++);
    return eSym;
}

Int32 MeshCoderDriver::_nextFGrpSymbol()
{
    Int32 eSym = -1;
    if (_iVGrpReadPos < _viOutVGrpSyms.length())
        eSym = _viOutVGrpSyms.value(_iVGrpReadPos++);
    return eSym;
}

UInt16 MeshCoderDriver::_nextVtxFlagSymbol  ()
{
    UInt16 eSym = 0;
    if (_iFFlagReadPos < _vuOutVtxFlags.length())
        eSym = _vuOutVtxFlags.value(_iFFlagReadPos++);
    return eSym;
}

UInt64 MeshCoderDriver::_nextAttrMaskSymbol  (Int32 iCCntx)
{
    UInt64 eSym = 0;
    if (_iAttrMaskReadPos[iCCntx] < _vvuOutAttrMasks[iCCntx].length())
        eSym = _vvuOutAttrMasks[iCCntx].value(_iAttrMaskReadPos[iCCntx]++);
    return eSym;
}

void MeshCoderDriver::_nextAttrMaskSymbol(BitVec *iopvbAttrMask, Int32 cDegree)
{
    if (_iAttrMaskLrgReadPos < _vuOutAttrMasksLrg.length()) {
        iopvbAttrMask->setLength(cDegree);
        UInt32 *pu = iopvbAttrMask->ptr();
        Int32 nWords = (cDegree + BitVec::cWordBits - 1) >> BitVec::cBitsLog2;
        memcpy(pu, &_vuOutAttrMasksLrg.value(_iAttrMaskLrgReadPos), nWords *
sizeof(UInt32));
        _iAttrMaskLrgReadPos += nWords;
    }
    else {
        iopvbAttrMask->setLength(0);
    }
}

Int32 MeshCoderDriver::_nextSplitFaceSymbol  ()
{
    Int32 eSym = -1;
    if (_iSplitFaceReadPos < _viOutSplitFaceSyms.length())
        eSym = _viOutSplitFaceSyms.value(_iSplitFaceReadPos++);
    return eSym;
}

Int32 MeshCoderDriver::_nextSplitPosSymbol  ()
{
    Int32 eSym = -1;
    if (_iSplitPosReadPos < _viOutSplitPosSyms.length())
        eSym = _viOutSplitPosSyms.value(_iSplitPosReadPos++);
    return eSym;
}

// Computes a "compression context" from 0 to 7 inclusive for
// faces on vertex iVtx. The context is based on the vertex's
// valence, and the total _known_ degree of already-coded
// faces on the vertex at the time of the call.

```

```

Int32 MeshCoderDriver::_faceCntxt(JtInt32 iVtx, JtDualVFMesh *pVFM)
{
    // Here, we are going to gather data to be used to determine a
    // compression contest for the face degree.
    JtInt32 cVal = pVFM->valence(iVtx);
    JtInt32 nKnownFaces = 0;
    JtInt32 cKnownTotDeg = 0;
    for (JtInt32 i = 0 ; i < cVal ; i++) {
        JtInt32 iTmpFace = pVFM->face(iVtx, i);
        if (!pVFM->isValidFace(iTmpFace))
            continue;
        nKnownFaces++;
        cKnownTotDeg += pVFM->degree(iTmpFace);
    }
    JtInt32 iCCntxt = 0;
    if (cVal == 3) {
        // Regular tristrip-like meshes tend to have degree 6 faces
        iCCntxt = (cKnownTotDeg < nKnownFaces * 6) ? 0 :
            (cKnownTotDeg == nKnownFaces * 6) ? 1 : 2;
    }
    else if (cVal == 4) {
        // Regular quadstrip-like meshes tend to have degree 4 faces
        iCCntxt = (cKnownTotDeg < nKnownFaces * 4) ? 3 :
            (cKnownTotDeg == nKnownFaces * 4) ? 4 : 5;
    }
    else if (cVal == 5)
        // Pentagons are all lumped into context 6
        iCCntxt = 6;
    else
        // All other polygons are lumped into context 7
        iCCntxt = 7;

    return iCCntxt;
}

```

F.2.2 MeshCodec class

```

// This class serves as the abstract base class from which two concrete classes
// are derived to implement the core operations for a polygonal
// mesh coder or decoder. An instance of this object is used by the
// MeshCoderDriver to encode and decode polygonal meshes.
//
// This class makes extensive use of DualVFMesh objects as the primary source and
// destination mesh topology storage data structures. This mediating data
// structure is necessary because the mesh coding scheme is deeply cooperative
// with and dependent upon such a vertex-facet data structure. Please refer to
// DualVFMesh for more information.
class MeshCodec {
public:
    // ===== Housekeeping Interface =====
    MeshCodec (MeshCoderDriver *pTMC = NULL);
protected:
    virtual ~MeshCodec() {}
public:

    // ===== Setup and Apply Interface =====
    void setMeshCoderDriver(MeshCoderDriver *pTMC) { _pTMC = pTMC; }
    JtDualVFMesh *vfm() const { return _pDstVFM; }
    void run();

    // ===== Generic encode/decode Driver Chain =====
    void clear();
    void runComponent(bool &obFoundComponent);
    void initNewComponent(bool &obFoundComponent);
    void completeV(Int32 iFace);
    Int32 activateV(Int32 iVtx, Int32 iVSlot);
    Int32 activateF(Int32 iFace, Int32 iFSlot);
}

```

```

void completeF(Int32 iVtx, Int32 jFSlot);
void addVtxToFace (Int32 iVtx, Int32 iVSlot,
                  Int32 iFace, Int32 iFSlot);

// Active face list management
void addActiveFace(Int32 iFace);
Int32 nextActiveFace();
void removeActiveFace(Int32 iFace);
Int32 activeFaceOffset(Int32 iFace) const;

private:
// ===== Polymorphic I/O Interface =====
virtual Int32 ioVtxInit    () = 0;
virtual Int32 ioVtx       (Int32 iFace, Int32 jFSlot) = 0;
virtual Int32 ioFace      (Int32 iVtx, Int32 iVSlot) = 0;
virtual Int32 ioSplitFace (Int32 iVtx, Int32 iVSlot) = 0;
virtual Int32 ioSplitPos  (Int32 iVtx, Int32 iVSlot) = 0;

// ===== Member Data =====
protected:
MeshCoderDriver *_pTMC; // TopoDualMeshCoder this codec is
attached to
SharedPtr<JtDualVFMesh> _pSrcVFM; // Input VFMesh
SharedPtr<JtDualVFMesh> _pDstVFM; // Output VFMesh
Veci _viActiveFaces; // Stack of incomplete "active faces"
BitVec _vbRemovedActiveFaces; // Helper bitvec parallel to
above
// Used by decoder to assign running attr indices
Int32 _iFaceAttrCtr;
};

// Runs the mesh encoder/decoder machine.
// If decoding is being performed, it consumes the mesh
// coding symbols from pre-filled member variables to produce
// the output VFMesh _pDstVFM.
void MeshCodec::run()
{
    // Assert state is consistent and ready to co/dec
    if (!_pDstVFM)
        _pDstVFM = new JtDualVFMesh();
    Assert(_pDstVFM);
    _pDstVFM->clear();
    clear();

    // Co/dec connected mesh components one at a time
    bool bFoundComponent = JtTrue;
    while (bFoundComponent) {
        runComponent(bFoundComponent);
    }
}

void MeshCodec::clear()
{
    // Setup
    _viActiveFaces.setLength(0);
    _vbRemovedActiveFaces.setLength(0);
    _iFaceAttrCtr = 0;
}

// Decodes one "connected component" (contiguous group of polygons) into
// _pDstVFM. Because the polygonal model may be formed of multiple
// disconnected mesh components, it may be necessary for run() to call this
// method multiple times. This method returns obFoundComponent = True
// if it actually encoded a new mesh component, and obFoundComponent = False
// if it did not.
void MeshCodec::runComponent(bool &obFoundComponent)

```

```

{
    Int32 iFace;
    initNewComponent(obFoundComponent);
    if (!obFoundComponent)
        return;
    while ((iFace = nextActiveFace()) != -1) {
        completeF(iFace);
        removeActiveFace(iFace);
    }
}

// Locates an unencoded vertex and begins the encoding
// process for the newly-found mesh component.
void MeshCodec::initNewComponent(bool &obFoundComponent)
{
    obFoundComponent = JtTrue;

    // Call ioVtxInit() to start us off with the seed face
    // from a new "connected component" of polygons.
    Int32 iVtx, i;
    if ((iVtx = ioVtxInit()) == -1) {
        obFoundComponent = JtFalse; // All vtxs are processed
        return;
    }
    Int32 cVal = _pDstVFM->valence(iVtx);
    for (i = 0 ; i < cVal ; i++)
        activateF(iVtx, i); // Process all faces
}

// Completes the VFMesh face iFace on _pDstVFM by calling activateV() and
// completeV() for each as-yet inactive incident vertexes in the face's
// degree ring.
void MeshCodec::completeF(Int32 iFace)
{
    // While there is an empty vtx slot on the face
    Int32 jVtxSlot, iVtx;
    Int32 iVSlot = 0;
    while ((jVtxSlot = _pDstVFM->findVtxSlot(iFace, -1)) != -1) {
        // Create and return a vtx iVtx, attaching it to iFace at vtx
        // slot jVtxSlot.
        iVtx = activateV(iFace, jVtxSlot);

        // Assert FV consistency
        Assert(_pDstVFM->vtx (iFace, jVtxSlot) == iVtx &&
            _pDstVFM->face(iVtx, iVSlot) == iFace );

        // Process the faces of iVtx starting from face slot
        // jVtxSlot where iVtx is incident on iFace.
        completeV(iVtx, jVtxSlot);

        // Invariant "VF": vtx(iVtx).face(iVSlot) == iFace &&
        // face(iFace).vtx(jVtxSlot) == iVtx
    }
}

// "Activates" the VFMesh face, on _pDstVFM, at face iFace vertex slot iVSlot
// by calling ioFace() to obtain a new vertex number and hooking it up to the
// topological structure. If the face is a SPLIT face, then call
// ioSplitFace() and ioSplitPos() to get the information necessary to connect
// to an already-active face. Note that we use the term "activate" here to
// mean "read" for mesh decoding.
Int32 MeshCodec::activateF(Int32 iVtx, Int32 iVSlot)
{
    Int32 jFSlot;
    // ioFace might return -2 as an error condition
    Int32 iFace = ioFace(iVtx, iVSlot);
    if (iFace >= 0) { // If a new active face

```

```

        if (!_pDstVFM->setVtxFace(iVtx, iVSlot, iFace) ||
            !_pDstVFM->setFaceVtx(iFace, 0, iVtx) ||
            !addActiveFace(iFace)
        )
        {
            return -2;
        }
    }
    else if (iFace == -1) {
        // Face already exists, so Split
        iFace = ioSplitFace(iVtx, iVSlot); // v's index in ActiveSet, returns v
        jFSlot = ioSplitPos(iVtx, iVSlot); // Position of iVtx in v
        _pDstVFM->setVtxFace(iVtx, iVSlot, iFace);
        addVtxToFace(iVtx, iVSlot, iFace, jFSlot);
    }
    return iFace;
}

// "Activates" the VFMesh vertex, on _pDstVFM, at face iFace vertex slot iVSlot
// by calling ioFace() to obtain a new face number and hooking it up to the
// topological structure. Note that we use the term "activate" here to
// mean "read" for mesh decoding.
Int32 MeshCodec::activateV(Int32 iFace, Int32 iVSlot)
{
    Int32 iVtx = ioVtx(iFace, iVSlot); // I/O valence; create a vtx
    _pDstVFM->setVtxFace(iVtx, 0, iFace);
    addVtxToFace(iVtx, 0, iFace, iVSlot);
    return iVtx;
}

// Completes the vertex iVtx on _pDstVFM by activating all inactive faces
// incident upon it. As an optimization, the user shall also pass in iVSlot
// which is the vertex slot on face 0 of iVtx where iVtx is located. This
// method begins its examination of iVtx's faces at face 0 by working its
// way around the vertex in both CCW and CW directions, checking to see if there
// are any faces that can be hooked into iVtx without calling activateF().
// This can happen when a face is completed by a nearby vertex before coming
// here. The situation can be detected by traversing the topology of the
// _pDstVFM over to the neighboring vertex and checking if it already has a
// face number for the corresponding face entry on iVtx. If so, then
// iVtx and the already completed face are connected together, and the
// next face around iVtx is examined. When the process can go no further,
// this method calls _activateF() on the remaining unresolved span of faces
// around the vertex.
void MeshCodec::completeF(Int32 iVtx, Int32 iVSlot)
{
    JtDualVFMesh *pDstVFM = _pDstVFM;
    Int32 i, vp, vn, jp, jn,
        iVtx2,
        cVal = pDstVFM->valence(iVtx);

    // Walk CCW from face slot 0, attempting to link in as many
    // already-reachable faces as possible until we reach one
    // that is inactive.
    vp = pDstVFM->face(iVtx, 0);
    jp = iVSlot;
    i = 1;
    JtDebugOnly(_assertParallelValRings(vp));
    while ((vn = pDstVFM->face(iVtx, i)) != -1) { // Forces "FV" in the "next"
direction
        DecModN(jp, pDstVFM->degree(vp));
        iVtx2 = pDstVFM->vtx(vp, jp);
        if (iVtx2 == -1)
            break;
        jn = pDstVFM->findVtxSlot(vn, iVtx2);
        Assert(jn > -1);
        DecModN(jn, pDstVFM->degree(vn));
        addVtxToFace(iVtx, i, vn, jn);
        vp = vn;
    }
}

```

```

        jp = jn;
        i++;
        if (i >= cVal)
            return;
    }

    // Walk CW from face slot 0, attempting to link in as many
    // already-reachable faces as possible until we reach one
    // that is inactive.
    Int32 ilast = i;
    vp = pDstVFM->face(iVtx, 0);
    jp = iVSlot;
    i = pDstVFM->valence(iVtx) - 1;
    while ((vn = pDstVFM->face(iVtx, i)) != -1) { // Forces "VF" in "prev"
direction
        IncModN(jp, pDstVFM->degree(vp));
        iVtx2 = pDstVFM->vtx(vp, jp);
        if (iVtx2 == -1)
            break;
        jn = pDstVFM->findVtxSlot(vn, iVtx2);
        Assert(jn > -1);
        IncModN(jn, pDstVFM->degree(vn));
        addVtxToFace(iVtx, i, vn, jn);
        vp = vn;
        jp = jn;
        i--;
        if (i < ilast)
            return;
    }

    // Activate the remaining faces on iVtx that cannot be deduced from
    // the already-assembled topology in the destination VFMesh.
    for (; ilast <= i ; ilast++) {
        Int32 iFace = activateV(iVtx, ilast);
        JtDemandState(iFace >= -1);
    }
}

// This method connects vertex iVtx into the topology of
// _pDstVFM at and around iFace. First, it connects iVtx
// to iFace's degree ring at position iVSlot. Next, it
// will connect iVtx into the faces at the other ends of
// the shared edges between iVtx and the next vertices CS and
// CCW about iFace if necessary.
void MeshCodec::addVtxToFace (Int32 iVtx, Int32 jFSlot,
                             Int32 iFace, Int32 iVSlot)
{
    Int32 iVSlotCW = iVSlot,
        iVSlotCCW = iVSlot,
        fp, ip,
        fn, in;
    JtDualVFMesh *pDstVFM = _pDstVFM;
    IncModN(iVSlotCCW, pDstVFM->degree(iFace));
    DecModN(iVSlotCW, pDstVFM->degree(iFace));

    // Connect iVtx to iFace/iVSlot
    JtRethrow(pDstVFM->setFaceVtx(iFace, iVSlot, iVtx));

    // Connect iVtx across the shared edge between iVtx and the vtx CW
    // from iVtx at iFace. Connect iVtx into the face at the other
    // end of this edge if it is not already connected there.
    if ((fp = pDstVFM->vtx(iFace, iVSlotCW)) != -1) {
        ip = pDstVFM->findFaceSlot(fp, iFace);
        Int32 iVSlotCCW = jFSlot;
        IncModN(iVSlotCCW, pDstVFM->valence(iVtx));
        if (pDstVFM->face(iVtx, iVSlotCCW) == -1) {
            DecModN(ip, pDstVFM->valence(fp));

```

```

        pDstVFM->setVtxFace(iVtx, iVSlotCCW, pDstVFM->face(fp, ip));
    }
}

// Connect iVtx across the shared edge between iVtx and the vtx CCW
// from iVtx at iFace. Connect iVtx into the face at the other
// end of this edge if it is not already connected there.
if ((fn = pDstVFM->vtx(iFace, iVSlotCCW)) != -1) {
    in = pDstVFM->findFaceSlot(fn, iFace);
    Int32 iVSlotCW = jFSlot;
    DecModN(iVSlotCW, pDstVFM->valence(iVtx));
    if (pDstVFM->face(iVtx, iVSlotCW) == -1) {
        IncModN(in, pDstVFM->valence(fn));
        pDstVFM->setVtxFace(iVtx, iVSlotCW, pDstVFM->face(fn, in));
    }
}
}

void MeshCodec::addActiveFace(Int32 iFace)
{
    JtRethrow(_viActiveFaces.pushBack(iFace));
}

// Returns a face from the active queue to be completed. This needn't be the
// one at the end of the queue, because the choice of the next active face
// can affect how many SPLIT symbols are produced. This method employs a
// fairly simple scheme of searching the most recent 16 active faces for the
// first one with the smallest number of incomplete slots in its degree ring.
Int32 MeshCodec::nextActiveFace()
{
    Int32 iFace = -1;
    // Search the 16 face record at the end of the
    // queue for the one with lowest remaining degree.
    while (_viActiveFaces.length() > 0 &&
        _vbRemovedActiveFaces.test(_viActiveFaces.back()))
        _viActiveFaces.popBack();
    Int32 cLowestEmptyDegree = 9999999;
    Int32 i, iFace0, cEmptyDeg;
    const Int32 cWidth = 16;
    JtDualVFMesh *pDstVFM = pDstVFM;
    for (i = _viActiveFaces.length() - 1;
        i >= ::jtmax(0, _viActiveFaces.length() - cWidth);
        i--)
    {
        iFace0 = _viActiveFaces[i];
        if (_vbRemovedActiveFaces.test(iFace0)) {
            _viActiveFaces.remove(i); // TOXIC: O(N^2)
            continue;
        }
        cEmptyDeg = pDstVFM->emptyFaceSlots(iFace0);
        if (cEmptyDeg < cLowestEmptyDegree) {
            cLowestEmptyDegree = cEmptyDeg;
            iFace = iFace0;
        }
    }

    // Return the selected active face
    return iFace;
}

// Removes iFace from the active face queue.
void MeshCodec::removeActiveFace(Int32 iFace)
{
    _vbRemovedActiveFaces.set(iFace);
}

// Searches the active face queue for iFace and returns

```

```

// its index position from the _end_ of the queue. This is
// needed by the ioFace() method when encoding a SPLIT
// symbol.
Int32 MeshCodec::activeFaceOffset(Int32 iFace) const
{
    Int32 iOffset = -1;
    Int32 i, cLen = _viActiveFaces.length();
    const Int32 *paiActiveFaces = _viActiveFaces.ptr();
    for (i = cLen - 1 ; i >= 0 ; i--) {
        if (paiActiveFaces[i] == iFace) {
            // The offset is how far FROM THE END of the active
            // face list we found iFace. This serves to make
            // the iOffset a much smaller number, which is better
            // for compression!
            iOffset = cLen - i;
            break;
        }
    }
    return iOffset;
}

```

F.2.3 MeshDecoder class

```

// This class implements the five abstract methods from
// MeshCodec to realize a mesh decoder.
class MeshDecoder : public MeshCodec {
public:
    // ===== Housekeeping Interface =====
    MeshDecoder (MeshCoderDriver *pTMC = NULL);
protected:
    virtual ~MeshDecoder() {}

private:
    // ===== Polymorphic I/O Interface =====
    virtual Int32 ioVtxInit () ;
    virtual Int32 ioVtx (Int32 iFace, Int32 iVSlot);
    virtual Int32 ioFace (Int32 iVtx , Int32 jFSlot);
    virtual Int32 ioSplitFace(Int32 iVtx , Int32 jFSlot);
    virtual Int32 ioSplitPos (Int32 iVtx , Int32 jFSlot);
};

// Begins decoding a new connected mesh component by calling
// ioVtx() to read the next vertex from the symbol stream.
Int32 MeshDecoder::ioVtxInit()
{
    return ioVtx(-1, -1);
}

// Read a vertex valence symbol, vertex group number, and vertex
// flags from the input symbols stream. Create a new vertex
// on _pDstVFM with this data, and return the new vertex number.
// It is this method's responsibility to detect the end of
// the input symbol stream by returning -1 when that happens.
Int32 MeshDecoder::ioVtx (Int32 /*iFace*/ , Int32 /*iVSlot*/)
{
    // Obtain a VERTEX VALENCE symbol
    Int32 eSym = _pTMC->_nextValSymbol();
    Int32 iVtxVal, iVtx = -1;
    if (eSym > -1) {
        // Create a new vtxt on the VFMesh
        iVtx = _pDstVFM->numVtxs();
        iVtxVal = eSym;
        _pDstVFM->newVtx (iVtx, iVtxVal);
        _pDstVFM->setVtxGrp (iVtx, _pTMC->_nextFGrpSymbol());
        _pDstVFM->setVtxFlags(iVtx, _pTMC->_nextVtxFlagSymbol());
    }
}

```



```

    return iVtx;
}

// Read a face degree symbol, and attribute mask bit
// vector, create a new DualVFMesh face, initialize the
// face attribute record numbers from a running counter,
// and return the new face number. If the degree symbol
// read from the input symbol stream is 0, signify this by
// returning -1.
Int32
MeshDecoder::ioFace      (Int32 iVtx, Int32 /*jFSlot*/)
{
    // Obtain a FACE DEGREE symbol
    Int32 iCntxt = _pTMC->_faceCntxt(iVtx, _pDstVFM);
    Int32 eSym = _pTMC->_nextDegSymbol(iCntxt);
    Int32 cDeg, iFace = -1;
    if (eSym != 0) {
        // Create a new face on the VFMesh
        iFace = _pDstVFM->numFaces();
        cDeg = eSym;
        Int32 nFaceAttrs = 0;
        if (cDeg <= JtDualVFMesh::cMbits) {
            UInt64 uAttrMask = _pTMC-
>_nextAttrMaskSymbol(/*iCntxt*/::jtmmin(7,::jtmmax(0,cDeg-2)));
            for (UInt64 uMask = uAttrMask ; uMask ; nFaceAttrs += (uMask & 1),
uMask >= 1);
            _pDstVFM->newFace(iFace, cDeg, nFaceAttrs, uAttrMask);
        }
        else {
            BitVec vbAttrMask;
            _pTMC->_nextAttrMaskSymbol(&vbAttrMask, cDeg);
            for (Int32 i = 0 ; i < cDeg ; i++) {
                if (vbAttrMask.test(i))
                    nFaceAttrs++;
            }
            _pDstVFM->newFace(iFace, cDeg, nFaceAttrs, &vbAttrMask, 0);
        }

        // Error check for a corrupt degree or attrmask
        if (nFaceAttrs > cDeg) {
            Assert (nFaceAttrs <= cDeg);
            return -2;
        }

        // Set up the face attributes
        for (Int32 iAttrSlot = 0 ; iAttrSlot < nFaceAttrs ; iAttrSlot++) {
            _pDstVFM->setFaceAttr(iFace, iAttrSlot, _iFaceAttrCtr++);
        }
    }
}

// Consumes a split offset symbol from the SPLIT offset
// symbol stream, and determines the face number referenced
// by the offset. Returns the referenced face number.
Int32 MeshDecoder::ioSplitFace(Int32 /*iVtx*/, Int32 /*jFSlot*/)
{
    // Obtain a SPLITFACE symbol
    Int32 eSym = _pTMC->_nextSplitFaceSymbol();
    Assert(eSym >= -1);
    Int32 iOffset = -1, iFace = -1;
    if (eSym > -1) {
        // Use the offset to index into the active face queue
        // to determine the actual face number.
        iOffset = eSym;
        Int32 cLen = _viActiveFaces.length();
        Assert(iOffset > 0 && iOffset <= cLen);
    }
}

```

```

        iFace = _viActiveFaces[cLen - iOffset];
    }

    return iFace;
}

// Consumes a split position symbol from the associated symbol
// stream, and returns the vertex slot number on the current
// split face at which the topological split/merge occurred.
Int32 MeshDecoder::ioSplitPos (Int32 /*iVtx*/, Int32 /*jFSlot*/)
{
    // Obtain a SPLITVTX symbol
    Int32 eSym = _pTMC->_nextSplitPosSymbol();
    Assert(eSym >= -1);
    Int32 iVSlot = -1;
    if (eSym > -1) {
        // Return the vtx slot number
        iVSlot = eSym;
    }

    return iVSlot;
}

```

STANDARDSISO.COM : Click to view the full PDF of ISO 14306:2017

Annex G (informative)

XT B- Rep segment

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

XT B-Rep Segments are typically referenced by Part Node Elements (see Part Node Element) using Late Loaded Property Atom Elements (see Late 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 XT B-Rep Segment form of element header data.

G.1 XT B-Rep Element

Object Type ID: 0x873a70e0, 0x2ac9, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97

XT B-Rep Segment represents a particular part's precise data in XT boundary representation format.

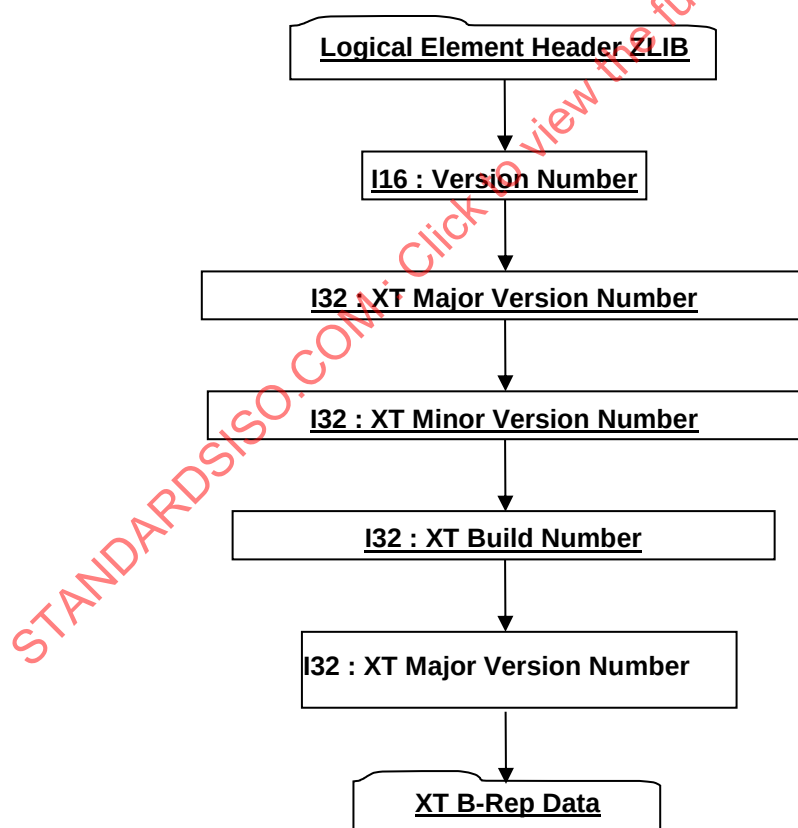


Figure 180 — XT B-Rep Element data collection

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

I32 : Version Number

Version Number is the version identifier for this XT B-Rep Element. Version number is an informative field which can be set to 0 without negative impact to implementation.

I32 : XT Major Version Number

XT Major Version Number specifies the major version number for the for the XT B-Rep data in the JT File. Major version number is an informative field which can be set to 0 without negative impact to implementation.

I32 : XT Minor Version Number

XT Minor Version Number specifies the minor version number for the XT B-Rep data in the JT File. Minor version number is an informative field which can be set to 0 without negative impact to implementation.

I32 : XT Build Number

XT Build Number specifies the build number for the XT B-Rep data in the JT File. XT build number is an informative field which can be set to 0 without negative impact to implementation.

I32 : XT B-Rep Data Length

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

G.1.1 XT B-Rep Data

The XT B-Rep Data collection specifies the raw stream of bytes used to represent a Part's XT B-Rep Body(s) Complete documentation for the XT "Neutral Binary" encoding format can be found in XT B-rep Data Segment.

G.2 Procedural Geometry

G.2.1 Background

Geometric elements can be defined in two fundamentally different ways. The descriptions below discuss the case of curves, but surfaces are analogous. The fundamental difference lies in the methods used to calculate points on the curve or surface:

- **Explicit Definition:** Here, a closed-form formula can be used to compute a point on the curve at a given parameter value. For example, a simple formula is used to compute a point on a circle, given the circle's center, orientation, and radius. Similarly, a point on a NURBS curve can be calculated if the control points, weights, and knots are known. Based on this, it is also possible to calculate derivatives (tangents, curvature)
- **Implicit Definition:** Here, a point on the curve cannot be calculated by using a closed-form formula, but only by using a procedure. For this reason, the geometry is often referred to as "procedural" geometry. A procedural intersection curve is an example. The curve is well-defined and unambiguous.
Procedural geometry is useful because it is concise (requiring less storage space) and does not involve any approximation.

G.2.2 Issue

In the context of the JT Implementor Forum, the following procedural geometry elements have been discussed in particular:

- Intersection curves (JT File Spec)

- Blend surfaces (JT File Spec)

The Annex H of the ISO File describes the Parasolid XT-Brep data segment with its classes and structures.

In addition, the ISO File Specification contains an Annex H with technical discussions on the XT data description. In particular, sections 1 (intersection curves), 2 (rolling ball blend surface) and 3 (procedural geometry – general) are of interest. These give a general introduction to the calculation of these types of geometry, including references to literature.

Looking at the potential usage for an Intersection Curve or Blend Surface implementation based on ISO, it will be mainly the reader (reading and interpreting an ISO file) who is looking for a proper handling. Without having access to the Parasolid kernel, it is very unlikely that somebody would like to create an Intersection Curve or Blend Surface.

Intersection Curve

For an Intersection Curve (icurve, for short), the reader has two possibilities to handle the curve:

(1) Evaluate points (and tangents) directly on the icurve, that is, evaluate the vector $p(t)$ for an arbitrary curve parameter t . This could be done based on the information provided in Annex H of the ISO document.

An own implementation is required to find the intersection of two surfaces and a plane (the plane being given by so-called chart points). This algorithm is not defined in Annex H of the ISO document. Typically, the algorithm would be based on Newton iteration. The iteration can be done with sufficient accuracy, but is more time consuming than the evaluation of an explicit curve.

Although the curve contains only a set of points (charts), the evaluation is possible for any curve parameter t according to the description in Annex H.

(2) Approximation of the curve by an explicit curve, typically a spline curve. Once this explicit curve approximation has been constructed, it can be used in place of the original icurve. No further iterative computations are needed; therefore better performance can be expected.

Rolling Ball Blend Surface

The Rolling Ball Blend Surface (“Blend Surface” for short) describes a fillet surface with a fixed radius. The Blend Surface is defined by the Spine (Guide) Curve, the radius, and the 2 adjacent surfaces – see “BLENDED_EDGE (Rolling Ball Blend)” and “rolling ball blend surface” in the ISO file specification.

Similar to the icurve, the surface calculation can be done by either:

(1) Evaluating points (and tangents) directly on the surface. The Annex F of the ISO Specification provides some details on the calculation, but due to the higher complexity compared to the icurve, further information might be needed for an implementation.

(2) Approximation of the surface by an explicit surface type, typically a NURBS surface.

For both types of elements, an approximation is required when an icurve or a Blend Surface should be converted from JT ISO into a CAD system not supporting the same kind of definition. Furthermore, it will allow faster computation of geometrical tasks (projections, evaluation, etc.), once the approximation is done.

On the other hand, a NURBS approximation will be less accurate than the original procedural definition and will occupy more space. Depending on the specific use case, either (1 - evaluation based on the procedural definition) or (2 – approximation) is the better choice.

The explicit representation of the geometry can be constructed by approximation algorithms. A recommendation for an approximation algorithm for both types of procedural elements is provided in Annex D of the JT industrialization implementation guidelines v1.4.2.

G.2.3 Recommended Solution

It was originally agreed that providing only the implicit definition of procedural elements is not sufficient to guarantee process stability or precise interoperability of the geometry. Correct interpretation of procedural geometry was thought to be too difficult to implement, based on the information available.

Consequently, the original targeted solution was the following:

Therefore, an explicit definition of the geometry – e.g. as NURBS curve or surface – shall be given instead of, or at least in addition to, the procedural definition.

This agreement was confirmed at the JT-IF Meeting on June 19, 2013, and shall be passed on to ISO as a requirement for ISO JT version 2.

This is common practice in other neutral geometry formats such as ISO 10303 (STEP): there, one of the attributes of the entity type intersection_curve is a reference to the explicit representation of the 3D curve.

There are two major concerns with this:

- There is already a lot of legacy data containing only the procedural definition. It is absolutely necessary to handle this data in a correct way.
- Providing (additionally) the explicit definition will may increase the amount of data considerably.

The recommended solution is to provide more detailed documentation about procedural geometry, including pseudo code for the algorithms. Annex D of the JT Industrialization Implementation guidelines contains a detailed description of the mathematical background and a recommended approximation algorithm for the icurve and the Blend Surface.

Based on this information, software vendors can implement both approaches, the evaluation based on the procedural definition and / or the approximation by a NURBS element.

G.3 Non-Manifold Bodies

G.3.1 Background

Non-Manifold bodies are present if single cells of the topological body - face or edges - are “overused”. More precisely: if a face is used by two different regions within the same body, or an edge is used by more than two faces.

Some CAD systems do support Non-Manifold bodies; others do not support this special kind of topological body.

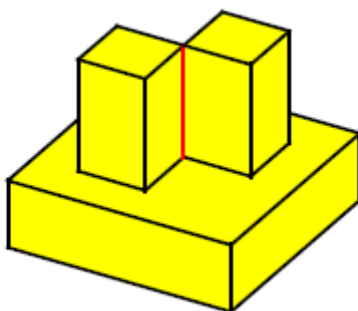


Figure 181 — Non-manifold body

G.3.2 Issue

In the JT ISO, the XT format generally allows usage of non-manifold bodies, but only for a specific type of body.

The XT topology provides the following types of bodies (see Annex H of ISO specification):

- 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. 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.
- 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).

Work is required in case a non-manifold XT body of type general is converted into a CAD format which does not support non-manifold bodies. This is a common topic of data conversion and not JT ISO specific.

G.3.3 Recommended Solution

The recommendation is to avoid the creation of bodies of type “general”. In case of importing a non-manifold body from another CAD system, the body should be separated at the non-manifold location and be created as two separate volumes. Some topological cells (edges, faces, and vertices) shall be duplicated for this.

A general body may raise problems also for other reasons. Also, the possible multi dimensionality of a general body should be avoided when creating a JT ISO file.

In order to avoid problems, XT bodies should be of type “solid”, “sheet” or “wire”.

Annex H (informative)

XT B- Rep data segment

H.1 Introduction to the XT B- Rep data segment

The XT B- Rep data segment in a JT file is a neutral binary definition of the solid body. A proprietary kernel is not required to read the XT B-Rep data segment. The model definition in the XT B-Rep data segment of a JT file is a fully described geometric and topological representation.

H.2 Logical Layout

The logical layout of a XT B- Rep data segment is:

- a short flag sequence describing the data format, followed by modeller identification information and user field size;
- The various flag sequences (mixtures of text and numbers) are documented under 'Physical layout'.
- The content of the modeller identification information is:

The version of the Parasolid Kernel (if appropriate) used to write the data, as a text string of the form:

: TRANSMIT FILE created by modeller version 1200123The schema version describing the field sequences of the part nodes as a text string of the form:

SCH_1200123_12006

This example above denotes XT data written by the Parasolid Kernel V12.0.123 using schema number 12006.

Applications writing XT data segments without using the Parasolid Kernel should use version 1200000 and schema number 12 (i.e.):

: TRANSMIT FILE created by modeller version 1200000

SCH_1200000_12

- The user field size is a simple integer.
- the objects (known as 'nodes') in the XT data in an unordered sequence, followed by a terminator;
- Every node in the XT data is assigned an integer index from 1 upwards (some indices may not be used). Pointer fields are output as these indices, or as zero for a null pointer.
- Each node entry begins with the node type. If the node is of variable length (see below), this is followed by the length of the variable field. The index of the node is then output, followed by the fields of the node.
- The terminator which follows the sequence of nodes is a two-byte integer with value 1, followed by an index with value 0. The index is output as a 2-byte integer with value 1 in binary XT data.
- The node with index 1 is the root node of the data as follows:

Table 98 — Object Nodes

Contents of XT data	Type of root node
Body	BODY
Assembly	ASSEMBLY
Array of parts	POINTER_LIS_BLOCK
Partition	WORLD

H.2.1 Schema

XT permanent structures are defined in a language akin to C which generates the appropriate files for a C compiler, the runtime information used by the Parasolid Kernel, along with an optional schema file that can be used during transmit and receive. The schema file for version 12.0 is named sch_12006 and is distributed with the Parasolid kernel. However, it is not necessary for applications reading and writing XT data directly to have a copy of this schema file to understand the XT.

H.2.2 Embedded schemas

XT parts, partitions and deltas can be transmitted with extra information that is intended to replace the schema used to describe the data layout. This information contains the differences between its schema and a defined base schema. The only fields that are included in this information are those which can be referenced in a cut-down version of the schema pertaining only to the XT part data that is present. Specifically, a full schema definition can contain fields that are not relevant in the context of the transmitted data and these fields are excluded.

Fields that are included are referred to as effective fields, and are either transmittable (`xmt_code == 1`) or have variable-length (`n_elts == 1`)

H.2.2.1 Physical layout

Most of the XT data are composed of integers, logical flags, and strings, but are of restricted ranges and so transmitted specially in binary format. The binary representation is given in bold type, such as “integer (**byte**)”. This is relevant to applications that attempt to read or write XT data directly. Two important elements are:

- short strings.

These are transmitted as an integer length (byte) followed by the characters (without trailing zero).

- positive integers.

These are transmitted similarly to the pointer indices which link individual objects together, i.e., small values 0..32766 are transmitted as a single **short** integer, larger ones encoded into two.

H.2.2.2 XT format

Presence of the new format is indicated by a change to the standard header: the archive name is extended by the number of the base schema, e.g., SCH_1400068_14000_13006, and then the maximum number of node types is inserted (short).

Transmission then continues as normal, except that when transmitting the first node of any particular type, extra information is inserted between the nodetype and the variable-length, index data as follows:

- the arrays of effective fields in the base schema node and the current schema node are assembled;
- if the nodetype does not exist in the base schema then it is output as follows:

- number of fields (byte);
- name and description (short strings);
- fields one by one as shown in the table F.2.

Table 99 — Field types in order one by one

name	short string	
ptr_class	Short	
n_elts	Positive integer	
type	short string	The field type. Allowed values are described in “Field types”, below. Omitted if ptr_class non-zero
xmt_code	logical (byte)	Omitted for fixed-length ($n\_elts \neq 1$)

- 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.

H.2.3 Field types

The XT format is not itself a binary protocol, and so does not define data sizes; the only requirement is that a runtime implementation has sufficient room for the information. The available implementations run with 8bit ASCII characters, 8bit unsigned bytes (0..255), 16bit short integers (0..65535 or -32768..32767), 32bit integers (0..4G-1, -2G..2G-1) and IEEE reals. The implementation used in the given binary XT data is specified by the "PS<code>" at the start of the XT data. See the chapter on “Physical Layout” for more information.

The full list of field types used in XT segment data is as follows:

- u unsigned byte 0-255
- c char
- l unsigned byte 0-1 (i.e. logical)

```
typedef char logical;
```

n short int

w unicode character, output as a short int

d Int

p pointer-index

Small indices (less than 32767) are treated specially in binary XT data to save space. See the section below on binary format.

f double

i These correspond to a region of the real line:

```
typedef struct { double low, high; } interval;
```

v array [3] of doubles

These correspond to a 3-space position or direction:

```
typedef struct { double x,y,z; } vector;
```

b array [6] of doubles

These correspond to a 3-spce region:

```
typedef struct { interval x,y,z; } box;
```

h array [3] of doubles

These represent points of intersection between two surfaces; only the position vector is written to the XT data. The structure is documented further in the section on intersection curves.

H.2.4 Variable-length nodes

Variable-length nodes differ from fixed-length nodes in that their last field is of variable length, i.e. different nodes of the same type may have different lengths.

The number of entries in each such node is indicated by an integer in the XT data between its nodetype and index, so an example might be:

```
83 3 15 1 2 3
```

H.2.5 Unresolved indices

In some cases a node will contain an index field which does not correspond to a node in the XT data, in this case the index is to be interpreted as zero.

H.3 Physical Layout

H.3.1 Binary

The flag sequence is followed by the length of the modeller version as a 2-byte integer, the characters of the modeller version, the length of the schema version as a 4-byte integer, the characters of the schema version, and finally the userfield size as a 4-byte integer.

There are two special numeric values (-32764 for integral values, -3.14158e13 for floating point) which are used to mark an 'unset' or 'null' value.

H.3.1.1 Neutral Binary

In the XT Segment neutral binary, data is represented in big-endian format, with IEEE floating point numbers and ASCII characters. The flag sequence is the 4-byte sequence "PS" followed by two zero bytes, i.e., 'P' 'S' '\0' '\0'. The initial letters are ASCII, thus '\120' '\123'. The nodetype at the start of a node is a 2-byte integer, the variable length which may follow it is a 4-byte integer.

Logical values (0,1) are represented as themselves in 1 byte.

Small pointer indices (in the range 0-32766) are implemented as a 2-byte integer, larger indices are represented as a pair, thus:

```

if (index < 32767)
{
    // case: small index
    // offset so is > 0
    op_short( index + 1 );
}
else
{
    // case: big index
    // remainder: add 1 so > 0
    op_short( -(index % 32767 + 1) );
    // nonzero quotient
    op_short( index / 32767 );
}

```

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.

H.4 Model Structure

H.4.1 Topology

This section describes the XT Topology model, it gives an overview of how the nodes in the XT data are joined together.

The topological representation allows for:

- non-manifold solids;
- solids with internal partitions;

- bodies of mixed dimension (i.e. with wire, sheet, and solid 'bits');
- pure wire-frame bodies;
- disconnected bodies.

Each entity is described, and its properties and links to other entities given.

H.4.2 General points

In this section a set is called finite if it can be enclosed in a ball of finite radius - not that it has a finite number of members.

A set of points in 3-dimensional space is called open if it does not contain its boundary.

Back-pointers, next and previous pointers in a chain, and derived pointers are not described explicitly here. For information on this see the following description of the schema-level model.

H.4.3 Entity definitions

H.4.3.1 Assembly

An assembly is a collection of instances of bodies or assemblies. It may also contain construction geometry. An assembly has the following fields:

- a set of instances;
- a set of geometry (surfaces, curves and points).

H.4.3.2 Instance

An instance is a reference to a body or an assembly, with an optional transform:

- Body or assembly;
- Transform. If null, the identity transform is assumed.

H.4.3.3 Body

A body is a collection of faces, edges and vertices, together with the 3-dimensional connected regions into which space is divided by these entities. Each region is either **solid** or **void** (indicating whether it represents material or not).

The point-set represented by the body is the disjoint union of the point-sets represented by its solid regions, faces, edges, and vertices. This point-set need not be connected, but it shall be finite.

A body has the following fields:

- a set of regions.

A body has one or more regions. These, together with their boundaries, make up the whole of 3-space, and do not overlap, except at their boundaries. One region in the body is distinguished as the exterior region, which shall be infinite; all other regions in the body shall be finite.

- a set of geometry (surfaces, curve and/or points).
- a body-type. This may be wire, sheet, solid or general.

H.4.3.4 Region

A region is an open connected subset of 3-dimensional space whose boundary is a collection of vertices, edges, and oriented faces.

Regions are either solid or void, and they may be non-manifold. A solid region contributes to the point-set of its owning body; a void region does not (although its boundary will).

Two regions may share a face, one on each side.

A region may be infinite, but a body shall have exactly one infinite region. The infinite region of a body shall be void.

A region has the following fields:

- a logical indicating whether the region is solid;
- a set of shells. The positive shell of a region, if it has one, is not distinguished.

The shells of a region do not overlap or share faces, edges or vertices.

A region may have no shells, in which case it represents all space (and will be the only region in its body, which will have no faces, edges or vertices).

H.4.3.5 Shell

A shell is a connected component of the boundary of a region. As such it will be defined by a collection of faces, each used by the shell on one 'side', or on both sides; and some edges and vertices.

A shell has the following fields:

- a set of (face, logical) pairs;

Each pair represents one side of a face (where true indicates the front of the face, i.e. the side towards which the face normal points), and means that the region to which the shell belongs lies on that side of the face. The same face may appear twice in the shell (once with each orientation), in which case the face is a 2-dimensional cut subtracted from the region which owns the shell.

- a set of wireframe edges;

Edges are called **wireframe** if they do not bound any faces, and so represent 1-dimensional cuts in the shell's region. These edges are not shared by other shells.

- a vertex.

This is only non-null if the shell is an **acorn** shell, i.e. it represents a 0-dimensional hole in its region, and has one vertex, no edges and no faces.

A shell shall contain at least one vertex, edge, or face.

H.4.3.6 Face

A face is an open finite connected subset of a surface, whose boundary is a collection of edges and vertices. It is the 2-dimensional analogy of a region.

A face has the following fields:

- a set of loops. A face may have zero loops (e.g. a full spherical face), or any number;
- Surface. This may be null, and may be used by other faces;
- Sense. This logical indicates whether the normal to the face is aligned with or opposed to that of the surface.

H.4.3.7 Loop

A loop is a connected component of the boundary of a face. It is the 2-dimensional analogy of a shell. As such it will be defined by a collection of fins and a collection of vertices.

A loop has the following fields:

- an ordered ring of fins:

Each fin represents the oriented use of an edge by a loop. The sense of the fin indicates whether the loop direction and the edge direction agree or disagree. A loop may not contain the same edge more than once in each direction.

The ordering of the fins represents the way in which their owning edges are connected to each other via common vertices in the loop (i.e. nose to tail, taking the sense of each fin into account).

The loop direction is such that the face is locally on the left of the loop, as seen from above the face and looking in the direction of the loop.

- a vertex:

This is only non-null if the loop is an isolated loop, i.e. has no fins and represents a 0-dimensional hole in the face.

Consequently, a loop shall consist either of:

- a single fin whose owning ring edge has no vertices, or
- at least one fin and at least one vertex, or
- a single vertex.

H.4.3.8 Fin

A fin represents the oriented use of an edge by a loop.

A fin has the following fields:

- a logical **sense** indicating whether the fin's orientation (and thus the orientation of its owning loop) is the same as that of its owning edge, or different;
- a curve. This is only non-null if the fin's edge is tolerant, in which case every fin of that edge will reference a trimmed SP-curve. The underlying surface of the SP-curve shall be the same as that of the corresponding face. The curve shall not deviate by more than the edge tolerance from curves on other fins of the edge, and its ends shall be within vertex tolerance of the corresponding vertices.

H.4.3.9 Edge

An edge is an open finite connected subset of a curve; its boundary is a collection of zero, one or two vertices. It is the 1-dimensional analogy of a region.

An edge has the following fields:

- start vertex;
- end vertex. If one vertex is null, then so is the other; the edge will then be called a **ring** edge;
- an ordered ring of distinct fins;
- the ordering of the fins represents the spatial ordering of their owning faces about the edge (with a right-hand screw rule, i.e. looking in the direction of the edge the fin ordering is clockwise). The edge may have zero or any number of fins; if it has none, it is called a **wireframe** edge;
- a curve. This will be null if the edge has a tolerance. Otherwise, the vertices shall lie within vertex tolerance of this curve, and if it is a Trimmed Curve, they shall lie within vertex tolerance of the corresponding ends of the curve. The curve shall also lie in the surfaces of the faces of the edge, to within modeller resolution;
- Sense. This logical indicates whether the direction of the edge (start to end) is the same as that of the curve;
- a tolerance. If this is null-double, the edge is **accurate** and is regarded as having a tolerance of half the modeller linear resolution, otherwise the edge is called **tolerant**.

H.4.3.10 Vertex

A vertex represents a point in space. It is the 0-dimensional analogy of a region.

A vertex has the following fields:

- a geometric point;
- a tolerance. If this is null-double, the vertex is **accurate** and is regarded as having a tolerance of half the modeller linear resolution.

H.4.3.11 Attributes

An attribute is an entity which contains data, and which can be attached to any other entity except attributes, fins, lists, transforms or attribute definitions. An attribute has the following fields:

- definition. An attribute definition is an entity which defines the number and type of the data fields in a specific type of attribute, which entities may have such an attribute attached, and what happens to the attribute when its owning entity is changed. XT data shall not contain duplicate attribute definitions. Each attribute of a given type should reference the same instance of the attribute definition for that type. It is incorrect, for example, to create a copy of an attribute definition for each instance of the attribute of that type. Only those attribute definitions referenced by attributes in the part occur in the data;
- owner;
- fields. These are data fields consisting of one or more integers, doubles, vectors etc.

There are a number of system attribute definitions which may be present in the XT data. These are documented in the section 'System Attribute Definitions'. User attribute definitions can also be created. These are included in the XT data along with any attributes that use them.

H.4.3.12 Groups

A group is a collection of entities in the same part. Groups in assemblies may contain instances, surfaces, curves and points. Groups in bodies may contain regions, faces, edges, vertices, surfaces, curves and points. Groups have:

- owning part;
- a set of member entities;
- type. The type of the group specifies the allowed type of its members, e.g. a 'face' group in a body may only contain faces, whereas a 'mixed' group may have any valid members.

H.4.3.13 Node-ids

All entities in a part, other than fins, have a non-zero integer node-id which is unique within a part. This is intended to enable the entity to be identified within the XT data.

H.4.4 Entity matrix

Thus the relations between entities can be represented in matrix form as follows in Table F.3. The numbers represent the number of distinct entities connected (either directly or indirectly) to the given one.

Table 100 — Entity Matrix relations

	Body	Region	Shell	Face	Loop	Fin	Edge	Vertex
Body	-	>0	any	any	any	any	any	any
Region	1	-	any	any	any	any	any	any
Shell	1	1	-	any	any	any	any	any
Face	1	1-2	1-2	-	any	any	any	any
Loop	1	1-2	1-2	1	-	any	any	any
Fin	1	1-2	1-2	1	1	-	1	0-2
Edge	1	any	any	any	any	any	-	0-2
Vertex	1	any	any	any	any	any	any	-

H.4.5 Representation of manifold bodies

H.4.5.1 Body types

XT bodies have a field `body_type` which takes values from an enumeration indicating whether the body is:

- **solid**, representing a manifold 3-dimensional volume, possibly with internal voids. It need not be connected;
- **sheet**, representing a 2-dimensional subset of 3-space which is either manifold or manifold with boundary (certain cases are not strictly manifold – see below for details). It need not be connected;
- **wire**, representing a 1-dimensional subset of 3-space which is either manifold or manifold with boundary, and which need not be connected. An **acorn** body, which represents a single 0-dimensional point in space, also has body-type wire;

- **general** - none of the above.

A general body is not necessarily non-manifold, but at the same time it is not constrained to be manifold, connected, or of a particular dimensionality (indeed, it may be of mixed dimensionality).

Restrictions on entity relationships for manifold body types

Solid, sheet, and wire bodies are best regarded as special cases of the topological model; for convenience we call them the manifold body types (although as stated above, a general body may also be manifold).

In particular, bodies of these manifold types shall obey the following constraints:

- an acorn body shall consist of a single void region with a single shell consisting of a single vertex;
- a wire body shall consist of a single void region, with one or more shells, consisting of one or more wireframe edges and zero or more vertices (and no faces). Every vertex in the body shall be used by exactly one or two of the edges (so, in particular, there are no acorn vertices);
- so each connected component will be either: closed, where every vertex has exactly two edges; or open, where all but two vertices have exactly two edges each;
- a wire is called open if all its components are open, and closed if all its components are closed;
- solid and sheet bodies shall each contain at least one face; they may not contain any wireframe edges or acorn vertices;
- a solid body shall consist of at least two regions; at least one of its regions shall be solid. Every face in a solid body shall have a solid region on its negative side and a void region on its positive side (in other words, every face forms part of the boundary of the solid, and the face normals always point away from the solid);
- every edge in a solid body shall have exactly two fins, which will have opposite senses. Every vertex in a solid body shall either belong to a single isolated loop, or belong to one or more edges; in the latter case, the faces which use those edges shall form a single edgewise-connected set (when considering only connections via the edges which meet at the vertex);
- these constraints ensure that the solid is manifold;
- all the regions of a sheet body shall be void. It is known as an open sheet if it has one region, and a closed sheet if it has no boundary;
- every edge in a sheet body shall have exactly one or two fins; if it has two, these shall have opposite senses. In a closed sheet body, all the edges will have exactly two fins. Every vertex in a sheet body shall either belong to a single isolated loop, or belong to one or more edges; in the latter case, the faces which use those edges shall either form a single edgewise-connected set where all the edges involved have exactly two fins, or any number of edgewise-connected sets, each of which shall involve exactly two edges with one fin each (again, considering only connections via the edges which meet at the vertex).

Note that, although the constraints on edges and vertices in a sheet body are very similar to those which apply to a solid, in this case they do not guarantee that the body will be manifold; indeed, the rather complicated rules about vertices in an open sheet body specifically allow bodies which are non-manifold (such as a body consisting of two square faces which share a single corner vertex).

H.5 Schema Definition

H.5.1 Underlying types

```

union CURVE_OWNER_u
{
    struct EDGE_s          *edge;
    struct FIN_s           *fin;
    struct BODY_s          *body;
    struct ASSEMBLY_s      *assembly;
    struct WORLD_s         *world;
};

union SURFACE_OWNER_u
{
    struct FACE_s          *face;
    struct BODY_s          *body;
    struct ASSEMBLY_s      *assembly;
    struct WORLD_s         *world;
};

union ATTRIB_GROUP_u
{
    struct ATTRIBUTE_s     *attribute;
    struct GROUP_s         *group;
    struct MEMBER_OF_GROUP_s *member_of_group;
};

typedef union ATTRIB_GROUP_u ATTRIB_GROUP;

```

H.5.2 Geometry

```

union CURVE_u
{
    struct LINE_s          *line;
    struct CIRCLE_s        *circle;
    struct ELLIPSE_s       *ellipse;
    struct INTERSECTION_s  *intersection;
    struct TRIMMED_CURVE_s *trimmed_curve;
    struct PE_CURVE_s      *pe_curve;
    struct B_CURVE_s       *b_curve;
    struct SP_CURVE_s      *sp_curve;
};

typedef union CURVE_u CURVE;

union SURFACE_u
{

```

```

        struct PLANE_s          *plane;
        struct CYLINDER_s       *cylinder;
        struct CONE_s           *cone;
        struct SPHERE_s         *sphere;
        struct TORUS_s          *torus;
        struct BLENDED_EDGE_s   *blended_edge;
        struct BLEND_BOUND_s    *blend_bound;
        struct OFFSET_SURF_s    *offset_surf;
        struct SWEPT_SURF_s     *swept_surf;
        struct SPUN_SURF_s      *spun_surf;
        struct PE_SURF_s        *pe_surf;
        struct B_SURFACE_s      *b_surface;
    };

typedef union SURFACE_u        SURFACE;

union GEOMETRY_u
{
    union SURFACE_u            surface;
    union CURVE_u              curve;
    struct POINT_s             *point;
    struct TRANSFORM_s         *transform;
};

typedef union GEOMETRY_u GEOMETRY;

```

H.5.2.1 Curves

In the following field tables, 'pointer0' means a reference to another node which may be null. 'Pointer' means a non-null reference.

All curve nodes share the following common fields:

Table 101 — Curve node common fields

Field name	Data type	Description
node_id	int	Integer value unique to curve in part
attributes_groups	pointer0	Attributes and groups associated with curve
owner	pointer0	topological owner
next	pointer0	next curve in geometry chain
previous	pointer0	previous curve in geometry chain
geometric_owner	pointer0	geometric owner node
sense	char	sense of curve: '+' or '-' (see end of Geometry section)

```

struct ANY_CURVE_s           // Any Curve
{

```

```

        int                node_id;                // $d
        union ATTRIB_GROUP_u attributes_groups;      // $p
        union CURVE_OWNER_u owner;                 // $p
        union CURVE_u      next;                   // $p
        union CURVE_u      previous;               // $p
        struct GEOMETRIC_OWNER_s *geometric_owner; // $p
        char               sense;                  // $c
    };

typedef struct ANY_CURVE_s *ANY_CURVE;

```

H.5.2.1.1 Line

A straight line has a parametric representation of the form:

$$R(t) = P + t D$$

Where:

- P is a point on the line;
- D is its direction.

Table 102 — Line Fields

Field name	Data type	Description
pvec	vector	point on the line
direction	vector	direction of the line (a unit vector)

```

struct LINE_s == ANY_CURVE_s // Straight line
{
    int                node_id;                // $d
    union ATTRIB_GROUP_u attributes_groups;      // $p
    union CURVE_OWNER_u owner;                 // $p
    union CURVE_u      next;                   // $p
    union CURVE_u      previous;               // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char               sense;                  // $c
    vector             pvec;                   // $v
    vector             direction;              // $v
};

typedef struct LINE_s    *LINE;

```

H.5.2.1.2 CIRCLE

A circle has a parametric representation of the form:

$$R(t) = C + r X \cos(t) + r Y \sin(t)$$

Where:

- C is the centre of the circle;
- r is the radius of the circle;
- X and Y are the axes in the plane of the circle.

Table 103 — Circle fields

Field name	Data type	Description
centre	vector	Centre of circle
normal	vector	Normal to the plane containing the circle (a unit vector)
x_axis	vector	X axis in the plane of the circle (a unit vector)
radius	double	Radius of circle

The Y axis in the definition above is the vector cross product of the normal and x_axis.

```

struct CIRCLE_s == ANY_CURVE_s // Circle
{
    int node_id; // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union CURVE_OWNER_u owner; // $p
    union CURVE_u next; // $p
    union CURVE_u previous; // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char sense; // $c
    vector centre; // $v
    vector normal; // $v
    vector x_axis; // $v
    double radius; // $f
};

typedef struct CIRCLE_s *CIRCLE;

```

H.5.2.1.3 ELLIPSE

An ellipse has a parametric representation of the form:

$$R(t) = C + a X \cos(t) + b Y \sin(t)$$

Where:

- C is the centre of the circle;
- X is the major axis;
- r is the major radius;
- Y and b are the minor axis and minor radius respectively.

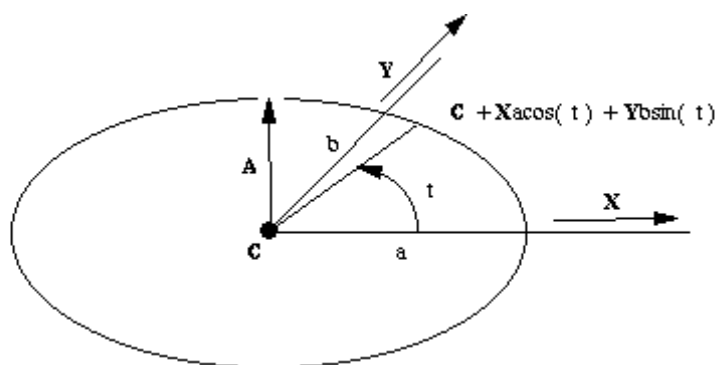


Table 104 — Ellipse fields

Field name	Data type	Description
centre	Vector	Centre of ellipse
normal	Vector	Normal to the plane containing the ellipse (a unit vector)
x_axis	Vector	major axis in the plane of the ellipse (a unit vector)
major_radius	Double	major radius
minor_radius	Double	minor radius

The minor axis (Y) in the definition above is the vector cross product of the normal and x_axis.

```

struct ELLIPSE_s == ANY_CURVE_s      // Ellipse
{
    int                                node_id;                               // $d
    union ATTRIB_GROUP_u               attributes_groups;                     // $p
    union CURVE_OWNER_u               owner;                                // $p
    union CURVE_u                     next;                                 // $p
    union CURVE_u                     previous;                             // $p
    struct GEOMETRIC_OWNER_s          *geometric_owner;                     // $p
    vector                            centre;                               // $v
    char                              sense;                                // $c
    vector                            normal;                               // $v
    vector                            x_axis;                               // $v
    double                            major_radius;                         // $f
    double                            minor_radius;                         // $f
};
typedef struct ELLIPSE_s      *ELLIPSE;

```

H.5.2.1.4 B_CURVE (B-spline curve)

XT supports B spline curves in full NURBS format. The mathematical description of these curves is:

- Non Uniform Rational B-splines as (NURBS), and

$$P(t) = \frac{\sum_{i=0}^{n-1} b_i(t) w_i V_i}{\sum_{i=0}^{n-1} b_i(t) w_i}$$

— the more simple Non Uniform B-spline:

— Where:

n = number of vertices ($n\_vertices$ in the PK standard form)

$V_0 \dots V_{n-1}$ are the B-spline vertices

$w_0 \dots w_{n-1}$ are the weights

$b_i(t)$, $i = 0 \dots n-1$ are the B-spline basis functions

Knot Vectors

The parameter t above is global. The user supplies an ordered set of values of t at specific points. The points are called knots and the set of values of t is called the knot vector. Each successive value in the set shall be greater than or equal to its predecessor. Where two or more such values are the same we say that the knots are coincident, or that the knot has multiplicity greater than 1. In this case it is best to think of the knot set as containing a null or zero length span. The principal use of coincident knots is to allow the curve to have less continuity at that point than is formally required for a spline. A curve with a knot of multiplicity equal to its *degree* can have a discontinuity of first derivative and hence of tangent direction. This is the highest permitted multiplicity except at the first or last knot where it can go as high as $(degree+1)$.

In order to avoid problems associated, for example with rounding errors in the knot set, XT stores an array of distinct values and an array of integer multiplicities. This is reflected in the standard form used by the PK for input and output of B-curve data.

Most algorithms in the literature, and the following discussion refer to the expanded knot set in which a knot of multiplicity n appears explicitly n times.

THE NUMBER OF KNOTS AND VERTICES

The knot set determines a set of basic 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 $\{t_0 \text{ to } t_{n\_knots-1}\}$ it can be seen then that it is only after t_{degree} that sufficient $(degree+1)$ basis functions are present for the curve to be fully defined, and that the B-curve ceases to be fully defined after $t_{n\_knots-1-degree}$.

The first *degree* knots and the last *degree* knots are known as the imaginary knots because their parameter values are outside the defined range of the B-curve.

PERIODIC B-CURVES

When the end of a B-curve meets its start sufficiently smoothly XT allows it to be defined to have periodic parametrization. That is to say that if the valid range were from t_{degree} to $t_{n\_knots - 1 - \text{degree}}$ then the difference between these values is called the period and the curve can continue to be evaluated with the same point reoccurring every period.

The minimal smoothness requirement for periodic curves in XT is tangent continuity, but we strongly recommend $C_{\text{degree}-1}$, or continuity in the $(\text{degree}-1)^{\text{th}}$ derivative. This in turn is best achieved by repeating the first *degree* vertices at the end, and by matching knot intervals so that counting from the start of the defined range, t_{degree} , the first *degree* intervals between knots match the last *degree* intervals, and similarly matching the last *degree* knot intervals before the end of the defined range to the first *degree* intervals.

CLOSED B-CURVES

A periodic B-curve shall also be closed, but is permitted to have a closed Bcurve that is not periodic.

In this case the rules for continuity are relaxed so that only C_0 or positional continuity is required between the start and end. Such closed non-periodic curves are not able to be attached to topology.

RATIONAL B-CURVE

In the rational form of the curve, each vertex is associated with a weight, which increases or decreases the effect of the vertex without changing the curve hull. To ensure that the convex hull property is retained, the curve equation is divided by a denominator which makes the coefficients of the vertices sum to one.

Where $w_0 \dots w_{n-1}$ are weights.

Each weight may take any positive value, and the larger the value, the greater the effect of the associated vertex. However, it is the relative sizes of the weights which is important, as may be seen from the fact that in the equation given above, all the weights may be multiplied by a constant without changing the equation.

In XT the weights are stored with the vertices by treating these as having an extra dimension. In the usual case of a curve in 3-d cartesian space this means that `vertex_dim` is 4, the x, y, z values are multiplied through by the corresponding weight and the 4th value is the weight itself.

```
struct B_CURVE_s == ANY_CURVE_s // B curve
{
    int node_id; // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union CURVE_OWNER_u owner; // $p
    union CURVE_u next; // $p
    union CURVE_u previous; // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char sense; // $c
    struct NURBS_CURVE_s *nurbs; // $p
    struct CURVE_DATA_s *data; // $p
};

typedef struct B_CURVE_s *B_CURVE;
```

The data stored in the XT data for a NURBS_CURVE is

Table 105 — NURB curve fields

Field name	Data type	Description
degree	Short	degree of the curve
n_vertices	Int	number of control vertices ('poles')
vertex_dim	Short	dimension of control vertices
n_knots	Int	number of distinct knots
knot_type	Byte	form of knot vector
periodic	Logical	true if curve is periodic
closed	Logical	true if curve is closed
rational	Logical	true if curve is rational
curve_form	Byte	shape of curve, if special
bspline_vertices	Pointer	control vertices node
knot_mult	Pointer	knot multiplicities node
knots	Pointer	knots node

The knot_type enum is used to describe whether or not the knot vector has a certain regular spacing or other common property:

```
typedef enum
{
    SCH_unset = 1,                // Unknown
    SCH_non_uniform = 2,          // Known to be not special
    SCH_uniform = 3,              // Uniform knot set
    SCH_quasi_uniform = 4,        // Uniform apart from bezier ends
    SCH_piecewise_bezier = 5,     // Internal multiplicity of order-1
    SCH_bezier_ends = 6           // Bezier ends, no other property
}
SCH_knot_type_t;
```

A uniform knot set is one where all the knots are of multiplicity one and are equally spaced. A curve has bezier ends if the first and last knots both have multiplicity 'order'.

The curve_form enum describes the geometric shape of the curve. The parameterization of the curve is not relevant.

```
typedef enum
{
    SCH_unset = 1,                // Form is not known
    SCH_arbitrary = 2,            // Known to be of no particular shape
    SCH_polyline = 3,
    SCH_circular_arc = 4,
    SCH_elliptic_arc = 5,
    SCH_parabolic_arc = 6,
    SCH_hyperbolic_arc = 7
}
SCH_curve_form_t;
```

```

struct NURBS_CURVE_s                                     // NURBS curve
{
    short                degree;                          // $n
    int                  n_vertices;                       // $d
    short                vertex_dim;                       // $n
    int                  n_knots;                          // $d
    SCH_knot_type_t      knot_type;                       // $u
    logical              periodic;                         // $l
    logical              closed;                           // $l
    logical              rational;                         // $l
    SCH_curve_form_t     curve_form;                      // $u
    struct BSPLINE_VERTICES_s *bspline_vertices;          // $p
    struct KNOT_MULT_s   *knot_mult;                      // $p
    struct KNOT_SET_s    *knots;                          // $p
};

```

```
typedef struct NURBS_CURVE_s *NURBS_CURVE;
```

The bspline vertices node is simply an array of doubles; 'vertex_dim' doubles together define one control vertex. Thus the length of the array is n_vertices * vertex_dim.

```

struct BSPLINE_VERTICES_s                               // B-spline vertices
{
    double               vertices[ 1 ];                   // $f[]
};

```

```
typedef struct BSPLINE_VERTICES_s *BSPLINE_VERTICES;
```

The knot vector of the NURBS_CURVE is stored as an array of distinct knots and an array describing the multiplicity of each distinct knot. Hence the two nodes

```

struct KNOT_SET_s                                         // Knot set
{
    double               knots[ 1 ];                      // $f[]
};

```

```
typedef struct KNOT_SET_s *KNOT_SET;
```

and

```

struct KNOT_MULT_s                                       // Knot multiplicities
{
    short                mult[ 1 ];                       // $n[]
};

```

```
typedef struct KNOT_MULT_s *KNOT_MULT;
```

The data stored in the XT data for a CURVE_DATA node is:

```

typedef enum
{
    SCH_unset = 1,                                       // check has not been performed

```

```

        SCH_no_self_intersections = 2,           // passed checks
        SCH_self_intersects = 3,                 // fails checks
        SCH_checked_ok_in_old_version = 4        // see below
    }
    SCH_self_int_t;

struct CURVE_DATA_s           // curve_data
{
    SCH_self_int_t             self_int;          // $u
    Struct HELIX_CU_FORM_s     *analytic_form    // $p
};

typedef struct CURVE_DATA_s *CURVE_DATA;

```

The self-intersection enum describes whether or not the geometry has been checked for self-intersections, and whether such self-intersections were found to exist:

If the analytic_form field is not null, it will point to a HELIX_CU_FORM node, which indicates that the curve has a helical shape, as follows:

```

struct HELIX_CU_FORM_s
{
    vector             axis_pt                    // $v
    vector             axis_dir                  // $v
    vector             point                     // $v
    char               hand                      // $c
    interval            turns                    // $i
    double              pitch                    // $f
    double              tol                      // $f
};

typedef struct HELIX_CU_FORM_s *HELIX_CU_FORM;

```

The axis_pt and axis_dir fields define the axis of the helix. The hand field is '+' for a right-handed and '-' for a left-handed helix. A representative point on the helix is at turn position zero. The turns field gives the extent of the helix relative to the point. For instance, an interval [0 10] indicates a start position at the point and an end 10 turns along the axis. Pitch is the distance travelled along the axis in one turn. Tol is the accuracy to which the owning bcurve fits this specification.

H.5.2.1.5 INTERSECTION

An intersection curve is one of the branches of a surface / surface intersection. XT represents these curves exactly; the information held in an intersection curve node is sufficient to identify the particular intersection branch involved, to identify the behaviour of the curve at its ends, and to evaluate precisely at any point in the curve. Specifically, the data is:

- the two surfaces involved in the intersection;
- the two ends of the intersection curve. These are referred to as the 'limits' of the curve. They identify the particular branch involved;
- an ordered array of points along the curve. This array is referred to as the 'chart' of the curve. It defines the parameterization of the curve, which increases as the array index increases.

The natural tangent to the curve at any point (i.e. in the increasing parameter direction) is given by the vector cross-product of the surface normals at that point, taking into account the senses of the surfaces.

Singular points where the cross-product of the surface normals is zero, or where one of the surfaces is degenerate, are called terminators. Intersection curves do not contain terminators in their interior. At terminators, the tangent to the curve is defined by the limit of the curve tangent as the curve parameter approaches the terminating value.

Table 106 — Curve intersection fields

Field name	Data type	Description
Surface	pointer array [2]	Surfaces of intersection curve
chart	Pointer	array of hvecs on the curve – see below
start	Pointer	start limit of the curve
end	Pointer	end limit of the curve

```

struct INTERSECTION_s == ANY_CURVE_s // Intersection
{
    int node_id; // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union CURVE_OWNER_u owner; // $p
    union CURVE_u next; // $p
    union CURVE_u previous; // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char sense; // $c
    union SURFACE_u surface[ 2 ]; // $p[2]
    struct CHART_s *chart; // $p
    struct LIMIT_s *start; // $p
    struct LIMIT_s *end; // $p
};

typedef struct INTERSECTION_s *INTERSECTION;

```

A point on an intersection curve is stored in a data structure called an 'hvec' (hepta-vec, or 7-vector):

```

typedef struct hvec_s // hepta_vec
{
    vector Pvec; // position
    double u[2]; // surface parameters
    double v[2];
    vector Tangent; // curve tangent
    double t; // curve parameter
} hvec;

```

Where:

- pvec is a point common to both surfaces;
- u[] and v[] are the u and v parameters of the pvec on each of the surfaces;

- tangent is the tangent to the curve at pvec. This will be equal to the (normalized) vector cross product of the surface normals at pvec, when this cross product is non-zero. These surface normals take account of the surface sense fields;
- t is the parameter of the pvec on the curve.

Note that only the pvec part of an hvec is actually transmitted.

The chart data structure essentially describes a piecewise-linear (chordal) approximation to the true curve. As well as containing the ordered array of hvecs defining this approximation, it contains extra information pertaining to the accuracy of the approximation:

```
struct CHART_s                                // Chart
{
    double          Base_parameter;           // $f
    double          Base_scale;               // $f
    int             Chart_count;              // $d
    double          Chordal_error;            // $f
    double          Angular_error;            // $f
    double          Parameter_error[2];       // $f[2]
    hvec            Hvec[ 1 ];                // $h[]
};
```

Where:

- base_parameter is the parameter of the first hvec in the chart;
- base_scale determines the scale of the parameterization (see below);
- chart_count is the length of the hvec array;
- chordal_error is an estimate of the maximum deviation of the curve from the piecewise-linear approximation given by the hvec array. It may be null;
- angular_error is the maximum angle between the tangents of two sequential hvecs. It may be null.
- parameter_error[] is always [null, null];
- hvec[] is the ordered array of hvecs.

The limits of the intersection curve are stored in the following data structure:

```
struct LIMIT_s                                // Limit
{
    char            type;                     // $c
    hvec            hvec[ 1 ];                // $h[]
};
```

The 'type' field may take one of the following values

```
const char SCH_help          = 'H';          // help hvec
const char SCH_terminator    = 'T';          // terminator
```

```

const char SCH_limit           = 'L';           // arbitrary limit
const char SCH_boundary        = 'B';           // spine boundary

```

The length of the hvec array depends on the type of the limit:

- a SCH_help limit is an arbitrary point on a closed intersection curve. There will be one hvec in the hvec array, locating the curve;
- a SCH_terminator limit is a point where one of the surface normals is degenerate, or where their cross-product is zero. Typically, there will be more than one branch of intersection between the two surfaces at these singularities. There will be two values in the hvec array. The first will be the exact position of the singularity, and the second will be a point on the curve a small distance away from the terminator. This 'branch point' identifies which branch relates to the curve in question. The branch point is the one which appears in the chart, at the corresponding end – so the singularity lies just outside the parameter range of the chart;
- a SCH_limit limit is an artificial boundary of an intersection curve on an otherwise potentially infinite branch. The single hvec describes the end of the curve;
- a SCH_boundary limit is used to describe the end of a degenerate rolling-ball blend. It is not relevant to intersection curves.

The parameterization of the curve is given as follows. If the chart points are P_i , $i = 0$ to n , with parameters t_i , and natural tangent vectors T_i , then define:

$$C_i = | P_{i+1} - P_i |;$$

$$\cos(a_i) = T_i \cdot (P_{i+1} - P_i) / C_i;$$

$$\cos(b_i) = T_i \cdot (P_i - P_{i-1}) / C_{i-1};$$

Then at any chart point P_i the angles a_i and b_i are the deviations between the tangent at the chart point and the next and previous chords respectively.

Let $f_0 = \text{base\_scale}$

$$f_i = (\cos(b_i) / \cos(a_i)) f_{i-1}$$

Then $t_0 = \text{base\_parameter}$

$$t_i = t_{i-1} + C_{i-1} f_{i-1}$$

The factors f_i are chosen so that the parameterization is C1. The parameter of a point between two chart points is given by projecting the point onto the chord between the previous and next chart points.

The point on the intersection curve corresponding to a given parameter is defined as follows:

- for a parameter equal to that of a chart point, it is the position of the chart point;
- for a parameter interior to the chart, it is the local point of intersection of three surfaces: the two surfaces of the intersection, and a plane defined by the chart. If the parameter t lies between chart parameters t_i , t_{i+1} , then the chord point corresponding to t lies at:

$$(t_{i+1} - t) / (t_{i+1} - t_i) P_i + (t - t_i) / (t_{i+1} - t_i) P_{i+1}$$

The plane lies through this point and is orthogonal to the chord (P_{i+1}, P_i) .

- for a parameter between a branch chart point and a terminator, it is the local point of intersection of three surfaces: one of the intersection surfaces and two planes. Surface[0] is used unless it is singular at the terminator and surface[1] is not singular at the terminator. The first plane contains the chord between the branch and the terminator, and the normal of the chosen intersection surface at the terminator or the curve tangent at the branch chart point if the surface normal cannot be defined. The second plane is the plane orthogonal to the chord between the branch and terminator points through the chord point as calculated above.

H.5.2.1.6 TRIMMED_CURVE

A trimmed curve is a bounded region of another curve, referred to as its basis curve. It is defined by the basis curve and two points and their corresponding parameters. Trimmed curves are most commonly attached to fins (fins) of tolerant edges in order to specify which portion of the underlying basis curve corresponds to the tolerant edge. They are necessary since the tolerant vertices of the edge do not necessarily lie exactly on the basis curve; the 'point' fields of the trimmed curve lie exactly on the basis curve, and within tolerance of the relevant vertex.

The rules governing the parameter fields and points are:

- point_1 and point_2 correspond to parm_1 and parm_2 respectively;
- If the basis curve has positive sense, $\text{parm}_2 > \text{parm}_1$;
- If the basis curve has negative sense, $\text{parm}_2 < \text{parm}_1$.

In addition, for open basis curves:

- both parm_1 and parm_2 shall be in the parameter range of the basis curve;
- point_1 and point_2 shall not be equal.

For periodic basis curves:

- parm_1 shall lie in the base range of the basis curve;
- if the whole basis curve is required then parm_1 and parm_2 should be a period apart and point_1 = point_2. Equality of parm_1 and parm_2 is not permitted;
- parm_1 and parm_2 shall not be more than a period apart.

For closed but non-periodic basis curves:

- both parm_1 and parm_2 shall be in the parameter range of the basis curve;
- If the whole of the basis curve is required, parm_1 and parm_2 shall lie close enough to each end of the valid parameter range in order that point_1 and point_2 are coincident to XT tolerance (1.0e-8 by default).

The sense of a trimmed curve is positive.

Table 107 — Trimmed curve fields

Field name	Data type	Description
basis_curve	pointer	Basis curve
point_1	vector	start of trimmed portion
point_2	vector	end of trimmed portion
parm_1	double	parameter on basis curve corresponding to point_1

parm_2	double	parameter on basis curve corresponding to point_2

```

struct TRIMMED_CURVE_s == ANY_CURVE_s           // Trimmed Curve
{
    int node_id;                                // $d
    union ATTRIB_GROUP_u attributes_groups;      // $p
    union CURVE_OWNER_u owner;                  // $p
    union CURVE_u next;                         // $p
    union CURVE_u previous;                     // $p
    struct GEOMETRIC_OWNER_s *geometric_owner;  // $p
    char sense;                                 // $c
    union CURVE_u basis_curve;                  // $p
    vector point_1;                             // $v
    vector point_2;                             // $v
    double parm_1;                              // $f
    double parm_2;                              // $f
};

typedef struct TRIMMED_CURVE_s *TRIMMED_CURVE;

```

H.5.2.1.7 SP_CURVE

An SP curve is the 3D curve resulting from embedding a 2D curve in the parameter space of a surface.

The 2D curve shall be a 2D BCURVE; that is it shall either be a rational B curve with a vertex dimensionality of 3, or a non-rational B curve with a vertex dimensionality of 2.

Table 108 — SP curve fields

Field name	Data type	Description
surface	pointer	surface
b_curve	pointer	2D Bcurve
original	pointer0	not used
tolerance_to_original	double	not used

```

struct SP_CURVE_s == ANY_CURVE_s           // SP curve
{
    int node_id;                                // $d
    union ATTRIB_GROUP_u attributes_groups;      // $p
    union CURVE_OWNER_u owner;                  // $p
    union CURVE_u next;                         // $p
    union CURVE_u previous;                     // $p
    struct GEOMETRIC_OWNER_s *geometric_owner;  // $p
    char sense;                                 // $c
    union SURFACE_u surface;                    // $p
    struct B_CURVE_s *b_curve;                  // $p
    union CURVE_u original;                     // $p
    double tolerance_to_original;               // $f
};

```

```
};
typedef struct SP_CURVE_s    *SP_CURVE;
```

H.5.2.2 Surfaces

All surface nodes share the following common fields:

Table 109 — Surface node fields

Field name	Data type	Description
node_id	int	Integer value unique to surface in part
attributes_groups	pointer0	Attributes and groups associated with surface
owner	pointer	topological owner
next	pointer0	next surface in geometry chain
previous	pointer0	previous surface in geometry chain
geometric_owner	pointer0	geometric owner node
sense	char	sense of surface: '+' or '-' (see end of Geometry section)

```
struct ANY_SURF_s                // Any Surface
{
    int                          node_id;                // $d
    union ATTRIB_GROUP_u         attributes_groups;      // $p
    union SURFACE_OWNER_u        owner;                 // $p
    union SURFACE_u              next;                  // $p
    union SURFACE_u              previous;              // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;      // $p
    char                         sense;                 // $c
};

typedef struct ANY_SURF_s    *ANY_SURF;
```

H.5.2.2.1 PLANE

A plane has a parametric representation of the form:

$$R(u, v) = P + uX + vY$$

Where:

- P is a point on the plan;
- X and Y are axes in the plane.

Table 110 — Plane fields

Field name	Data type	Description
pvec	vector	point on the plane
normal	vector	normal to the plane (a unit vector)
x_axis	vector	X axis of the plane (a unit vector)

The Y axis in the definition above is the vector cross product of the normal and x_axis.

```
struct PLANE_s == ANY_SURF_s // Plane
{
    int node_id; // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union SURFACE_OWNER_u owner; // $p
    union SURFACE_u next; // $p
    union SURFACE_u previous; // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char sense; // $c
    vector pvec; // $v
    vector normal; // $v
    vector x_axis; // $v
};

typedef struct PLANE_s *PLANE;
```

H.5.2.2.2 CYLINDER

A cylinder has a parametric representation of the form:

$$R(u,v) = P + rX\cos(u) + rY\sin(u) + vA$$

Where:

- P is a point on the cylinder axis;
- r is the cylinder radius;
- A is the cylinder axis;
- X and Y are unit vectors such that A, X and Y form an orthonormal set.

Table 111 — Cylinder fields

Field name	Data type	Description
pvec	vector	point on the cylinder axis
axis	vector	direction of the cylinder axis (a unit vector)
radius	double	radius of cylinder
x_axis	vector	X axis of the cylinder (a unit vector)

The Y axis in the definition above is the vector cross product of the axis and x_axis.

```
struct CYLINDER_s == ANY_SURF_s // Cylinder
{
    int node_id; // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union SURFACE_OWNER_u owner; // $p
    union SURFACE_u next; // $p
};
```

```

        union SURFACE_u          previous;          // $p
        struct GEOMETRIC_OWNER_s *geometric_owner;  // $p
        char                     sense;              // $c
        vector                   pvec;               // $v
        vector                   axis;               // $v
        double                   radius;             // $f
        vector                   x_axis;             // $v
    };

typedef struct CYLINDER_s *CYLINDER;

```

H.5.2.2.3 CONE

A cone in XT is only half of a mathematical cone. By convention, the cone axis points away from the half of the cone in use. A cone has a parametric representation of the form:

$$R(u, v) = P - vA + (X \cos(u) + Y \sin(u))(r + v \tan(a))$$

Where:

- P is a point on the cone axis;
- r is the cone radius at the point P;
- A is the cone axis;
- X and Y are unit vectors such that A, X and Y form an orthonormal set, i.e. $Y = A \times X$;
- a is the cone half angle.

Table 112 — Cone fields

Field name	Data type	Description
pvec	vector	point on the cone axis
axis	vector	direction of the cone axis (a unit vector)
radius	double	radius of the cone at its pvec
sin_half_angle	double	sine of the cone's half angle
cos_half_angle	double	cosine of the cone's half angle
x_axis	vector	X axis of the cone (a unit vector)

The Y axis in the definition above is the vector cross product of the axis and x_axis.

```

struct CONE_s == ANY_SURF_s          // Cone
{
    int                     node_id;          // $d
    union ATTRIB_GROUP_u    attributes_groups; // $p
    union SURFACE_OWNER_u   owner;           // $p
    union SURFACE_u         next;            // $p
    union SURFACE_u         previous;        // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
}

```

```

        char                sense;                // $c
        vector              pvec;                 // $v
        vector              axis;                 // $v
        double              radius;               // $f
        double              sin_half_angle;       // $f
        double              cos_half_angle;       // $f
        vector              x_axis;               // $v
    };

typedef struct CONE_s      *CONE;

```

H.5.2.2.4 SPHERE

A sphere has a parametric representation of the form:

$$R(u, v) = C + (X \cos(u) + Y \sin(u)) r \cos(v) + r \sin(v)$$

Where:

- C is centre of the sphere;
- r is the sphere radius;
- A, X and Y form an orthonormal axis set.

Table 113 — Sphere fields

Field name	Data type	Description
Centre	vector	centre of the sphere
Radius	double	radius of the sphere
Axis	vector	A axis of the sphere (a unit vector)
x_axis	vector	X axis of the sphere (a unit vector)

The Y axis of the sphere is the vector cross product of its A and X axes.

```

struct SPHERE_s == ANY_SURF_s                // Sphere
{
    int                node_id;                // $d
    union ATTRIB_GROUP_u
    union SURFACE_OWNER_u
    union SURFACE_u
    union SURFACE_u
    struct GEOMETRIC_OWNER_s
    char                sense;                // $c
    vector              centre;               // $v
    double              radius;               // $f
    vector              axis;                 // $v
    vector              x_axis;               // $v
};

```

```
typedef struct SPHERE_s      *SPHERE;
```

H.5.2.2.5 TORUS

A torus has a parametric representation of the form:

$$R(u, v) = C + (X \cos(u) + Y \sin(u))(a + b \cos(v)) + b A \sin(v)$$

Where:

- C is centre of the torus;
- A is the torus axis;
- a is the major radius;
- b is the minor radius;
- X and Y are unit vectors such that A, X and Y form an orthonormal set.

In XT, there are three types of torus:

Doughnut - the torus is not self-intersecting ($a > b$).

Apple - the outer part of a self-intersecting torus ($a \leq b$, $a > 0$).

Lemon - the inner part of a self-intersecting torus ($a < 0$, $|a| \leq b$).

The limiting case $a = b$ is allowed; it is called an 'osculating apple', but there is no 'lemon' surface corresponding to this case.

The limiting case $a = 0$ cannot be represented as a torus; this is a sphere.

Table 114 — Torus fields

Field name	Data type	Description
centre	vector	centre of the torus
axis	vector	axis of the torus (a unit vector)
major_radius	double	major radius
minor_radius	double	minor radius
x_axis	vector	X axis of the torus (a unit vector)

The Y axis in the definition above is the vector cross product of the axis of the torus and the x_axis.

```
struct TORUS_s == ANY_SURF_s      // Torus
{
    int                node_id;                // $d
    union ATTRIB_GROUP_u    attributes_groups;    // $p
    union SURFACE_OWNER_u    owner;              // $p
    union SURFACE_u          next;              // $p
    union SURFACE_u          previous;          // $p
    struct GEOMETRIC_OWNER_s *geometric_owner;    // $p
    char                 sense;                // $c
    vector              centre;                // $v
}
```

```

vector          axis;                      // $v
double          major_radius;              // $f
double          minor_radius;              // $f
vector          x_axis;                    // $v
};

typedef struct TORUS_s      *TORUS;

```

H.5.2.2.6 BLENDED_EDGE (Rolling Ball Blend)

XT supports exact rolling ball blends. They have a parametric representation of the form:

$$R(u, v) = C(u) + rX(u)\cos(v a(u)) + rY(u)\sin(v a(u))$$

Where:

- $C(u)$ is the spine curve;
- r is the blend radius;
- $X(u)$ and $Y(u)$ are unit vectors such that $C'(u) \cdot X(u) = C'(u) \cdot Y(u) = 0$;
- $a(u)$ is the angle subtended by points on the boundary curves at the spine.

X , Y and a are expressed as functions of u , as their values change with u .

The spine of the rolling ball blend is the centre line of the blend; i.e. the path along which the centre of the ball moves.

Table 115 — Blended edge fields

Field name	Data type	Description
type	char	type of blend: 'R' or 'E'
surface	pointer[2]	supporting surfaces (adjacent to original edge)
spine	pointer	spine of blend
range	double[2]	offsets to be applied to surfaces
thumb_weight	double[2]	always [1,1]
boundary	pointer0[2]	always [0, 0]
start	pointer0	Start LIMIT in certain degenerate cases
end	pointer0	End LIMIT in certain degenerate cases

```

struct BLENDED_EDGE_s == ANY_SURF_s      // Blended edge
{
    int          node_id;                  // $d
    union  ATTRIB_GROUP_u          attributes_groups;      // $p
    union  SURFACE_OWNER_u          owner;                  // $p
    union  SURFACE_u          next;                          // $p
    union  SURFACE_u          previous;                      // $p
    struct GEOMETRIC_OWNER_s      *geometric_owner;         // $p
}

```

```

char          sense;                // $c
char          blend_type;           // $c
union SURFACE_u    surface[2];      // $p[2]
union CURVE_u      spine;           // $p
double         range[2];            // $f[2]
double         thumb_weight[2];     // $f[2]
union SURFACE_u    boundary[2];     // $p[2]
struct LIMIT_s     *start;          // $p
struct LIMIT_s     *end;            // $p
};

typedef struct BLENDED_EDGE_s *BLENDED_EDGE;

```

The parameterization of the blend is as follows. The *u* parameter is inherited from the spine, the constant *u* lines being circles perpendicular to the spine curve. The *v* parameter is zero at the blend boundary on the first surface, and one on the blend boundary on the second surface; unless the sense of the spine curve is negative, in which case it is the other way round. The *v* parameter is proportional to the angle around the circle.

XT data can contain blends of the following types:

```

const char SCH_rolling_ball = 'R';    // rolling ball blend
const char SCH_cliff_edge   = 'E';    // cliff edge blend

```

For rolling ball blends, the spine curve will be the intersection of the two surfaces obtained by offsetting the supporting surfaces by an amount given by the respective entry in *range[]*. Note that the offsets to be applied may be positive or negative, and that the sense of the surface is significant; i.e. the offset vector is the natural unit surface normal, times the range, times -1 if the sense is negative.

For cliff edge blends, one of the surfaces will be a *blended_edge* with a range of $[0,0]$; its spine will be the cliff edge curve, and its supporting surfaces will be the surfaces of the faces adjacent to the cliff edge. Its type will be *R*.

The limit fields will only be non-null if the spine curve is periodic but the edge curve being blended has terminators – for example if the spine is elliptical but the blend degenerates. In this case the two *LIMIT* nodes, of type 'L', determine the extent of the spine.

H.5.2.2.7 BLEND_BOUND (Blend boundary surface)

A *blend_bound* surface is a construction surface, used to define the boundary curve where a blend becomes tangential to its supporting surface. It is an implicit surface defined internally so that it intersects one of the supporting surfaces along the boundary curve. It is orthogonal to the blend and the supporting surface along this boundary curve. The supporting surface corresponding to the *blend_bound* is

Blend_bound -> *blend.blended_edge* -> *surface[1-*blend_bound*->boundary]*.

Blend boundary surfaces have no parameterization, but are defined by the distance function

$$f(X) = f_0(X + r_1 * \text{grad}_f_1(X)) - r_0$$

Where:

- *f₀* is the surface distance function of the supporting surface corresponding to the *blend_bound*;
- *r₀* is the blend radius corresponding to that supporting surface;
- *f₁* is the surface distance function of the other supporting surface of the blend;

— r_1 is the blend radius corresponding to the other supporting surface.

Blend boundary surfaces are most commonly referenced by the intersection curve representing the boundary curve of the blend.

The data stored in the XT data for a blend_bound is only that necessary to identify the relevant blend and supporting surface:

Table 116 — Blend boundary surface fields

Field name	Data type	Description
boundary	short	index into supporting surface array
blend	pointer	corresponding blend surface

```

struct BLEND_BOUND_s == ANY_SURF_s           // Blend boundary
{
    int                                     node_id;                               // $d
    union ATTRIB_GROUP_u                   attributes_groups;                     // $p
    union SURFACE_OWNER_u                   owner;                               // $p
    union SURFACE_u                         next;                               // $p
    union SURFACE_u                         previous;                             // $p
    struct GEOMETRIC_OWNER_s                 *geometric_owner;                   // $p
    char                                     sense;                               // $c
    short                                   boundary;                             // $n
    union SURFACE_u                         blend;                               // $p
};

typedef struct BLEND_BOUND_s *BLEND_BOUND;

```

The supporting surface corresponding to the blend_bound is

blend_bound->blend.blended_edge->surface[1 - blend_bound->boundary].

H.5.2.2.8 OFFSET_SURF

An offset surface is the result of offsetting a surface a certain distance along its normal, taking into account the surface sense. It inherits the parameterization of this underlying surface.

Table 117 — Offset surface fields

Field name	Data type	Description
Check	char	check status
true_offset	logical	not used
surface	pointer	underlying surface
offset	double	signed offset distance
scale	double	for internal use only – may be set to null

```

struct OFFSET_SURF_s == ANY_SURF_s           // Offset surface
{
    int                                     node_id;                               // $d
    union ATTRIB_GROUP_u                   attributes_groups;                     // $p

```

```

union SURFACE_OWNER_u      owner;          // $p
union SURFACE_u            next;            // $p
union SURFACE_u            previous;        // $p
struct GEOMETRIC_OWNER_s   *geometric_owner; // $p
char                       sense;           // $c
char                       check;           // $c
logical                    true_offset;     // $l
union SURFACE_u            surface;         // $p
double                    offset;           // $f
double                    scale;           // $f
};

typedef struct OFFSET_SURF_s *OFFSET_SURF;

```

The offset surface is subject to the following restrictions:

- the offset distance shall not be within modeller linear resolution of zero;
- the sense of the offset surface shall be the same as that of the underlying surface;
- Offset surfaces may not share a common underlying surface.

The 'check' field may take one of the following values:

```

const char SCH_valid       = 'V';          // valid
const char SCH_invalid    = 'I';          // invalid
const char SCH_unchecked  = 'U';          // has not been checked

```

H.5.2.2.9 B_SURFACE

XT supports B spline surfaces in full NURBS format.

B-SURFACE DEFINITION

The B-surface definition is best thought of as an extension of the B-curve definition into two parameters, usually called *u* and *v*. Two knot sets are required and the number of control vertices is the product of the number that would be required for a curve using each knot vector. The rules for periodicity and closure given in the B-curve documentation are extended to surfaces in an obvious way.

For attachment to topology a B-surface is required to have G_1 continuity. That is to say that the surface normal direction shall be continuous.

Surfaces that are self-intersecting or contain cusps are not permitted to be attached to topology.

Table 118 — B-Surface fields

Field name	Data type	Description
nurbs	pointer	Geometric definition
data	pointer0	Auxiliary information

```

struct B_SURFACE_s == ANY_SURF_s           // B surface
{
    int node_id;                           // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union SURFACE_OWNER_u owner;           // $p
    union SURFACE_u next;                  // $p
    union SURFACE_u previous;              // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char sense;                            // $c
    struct NURBS_SURF_s *nurbs;            // $p
    struct SURFACE_DATA_s *data;           // $p
};

typedef struct B_SURFACE_s *B_SURFACE;

```

The data stored in the XT data for a NURBS surface is

Table 119 — NURB Surface fields

Field name	Data type	Description
u_periodic	logical	true if surface is periodic in u parameter
v_periodic	logical	true if surface is periodic in v parameter
u_degree	short	u degree of the surface
v_degree	short	v degree of the surface
n_u_vertices	int	number of control vertices ('poles') in u direction
n_v_vertices	int	number of control vertices ('poles') in v direction
u_knot_type	byte	form of u knot vector – see "B curve"
v_knot_type	byte	form of v knot vector
n_u_knots	int	number of distinct u knots
n_v_knots	int	number of distinct v knots
Rational	logical	true if surface is rational
u_closed	logical	true if surface is closed in u
v_closed	logical	true if surface is closed in v
surface_form	byte	shape of surface, if special
vertex_dim	short	dimension of control vertices
bspline_vertices	pointer	control vertices (poles) node
u_knot_mult	pointer	multiplicities of u knot vector
v_knot_mult	pointer	multiplicities of v knot vector
u_knots	pointer	u knot vector
v_knots	pointer	v knot vector

The surface form enum is defined below.

```

typedef enum
{
    SCH_unset = 1,                // Unknown
    SCH_arbitrary = 2,            // No particular shape
    SCH_planar = 3,
    SCH_cylindrical = 4,
    SCH_conical = 5,
    SCH_spherical = 6,
    SCH_toroidal = 7,
    SCH_surf_of_revolution = 8,
    SCH_ruled = 9,
    SCH_quadric = 10,
    SCH_swept = 11
}
SCH_surface_form_t;

struct NURBS_SURF_s                // NURBS surface
{
    logical        u_periodic;      // $l
    logical        v_periodic;      // $l
    short          u_degree;        // $n
    short          v_degree;        // $n
    int            n_u_vertices;     // $d
    int            n_v_vertices;     // $d
    SCH_knot_type_t u_knot_type;     // $u
    SCH_knot_type_t v_knot_type;     // $u
    int            n_u_knots;        // $d
    int            n_v_knots;        // $d
    logical        rational;        // $l
    logical        u_closed;        // $l
    logical        v_closed;        // $l
    SCH_surface_form_t surface_form; // $u
    short          vertex_dim;       // $n
    struct BSPLINE_VERTICES_s *bspline_vertices; // $p
    struct KNOT_MULT_s *u_knot_mult; // $p
    struct KNOT_MULT_s *v_knot_mult; // $p
    struct KNOT_SET_s *u_knots;      // $p
    struct KNOT_SET_s *v_knots;      // $p
};

typedef struct NURBS_SURF_s *NURBS_SURF;

```

The 'bspline_vertices', 'knot_set' and 'knot_mult' nodes and the 'knot_type' enum are described in the documentation for B_CURVE.

The 'surface data' field in a B surface node is a structure designed to hold auxiliary or 'derived' data about the surface: it is not a necessary part of the definition of the B surface. It may be null, or the majority of its individual fields may be null.

```

struct SURFACE_DATA_s          // auxiliary surface data
{
    interval                    original_uint;           // $i
    interval                    original_vint;           // $i
    interval                    extended_uint;           // $i
    interval                    extended_vint;           // $i
    SCH_self_int_t              self_int;               // $u
    char                        original_u_start;         // $c
    char                        original_u_end;           // $c
    char                        original_v_start;         // $c
    char                        original_v_end;           // $c
    char                        extended_u_start;         // $c
    char                        extended_u_end;           // $c
    char                        extended_v_start;         // $c
    char                        extended_v_end;           // $c
    char                        analytic_form_type;       // $c
    char                        swept_form_type;         // $c
    char                        spun_form_type;           // $c
    char                        blend_form_type;          // $c
    void                        *analytic_form;           // $p
    void                        *swept_form;              // $p
    void                        *spun_form;               // $p
    void                        *blend_form;              // $p
};

typedef struct SURFACE_DATA_s *SURFACE_DATA;

```

The 'original_' and 'extended_' parameter intervals and corresponding character fields original_u_start etc. are all connected with the ability to extend B surfaces when necessary – functionality which is commonly exploited in "local operation" algorithms for example. This is done automatically without the need for user intervention.

In cases where the required extension can be performed by adding rows or columns of control points, then the nurbs data will be modified accordingly – this is referred to as an 'explicit' extension. In some rational B surface cases, explicit extension is not possible - in these cases, the surface will be 'implicitly' extended. When a B surface is implicitly extended, the nurbs data is not changed, but it will be treated as being larger by allowing out-of-range evaluations on the surface. Whenever an explicit or implicit extension takes place, it is reflected in the following fields:

- "original_u_int" and "original_v_int" are the original valid parameter ranges for a B surface before it was extended;
- "extended_u_int" and "extended_v_int" are the valid parameter ranges for a B surface once it has been extended.

The character fields 'original_u_start' etc. all refer to the status of the corresponding parameter boundary of the surface before or after an extension has taken place. For B surfaces, the character can have one of the following values:

```
const char SCH_degenerate = 'D';      // Degenerate edge
const char SCH_periodic   = 'P';      // Periodic parameterization
const char SCH_bounded    = 'B';      // Parameterization bounded
const char SCH_closed     = 'C';      // Closed, but not periodic
```

The separate fields original_u_start and extended_u_start etc. are necessary because an extension may cause the corresponding parameter boundary to become degenerate.

If the surface_data node is present, then the original_u_int, original_v_int, original_u_start, original_u_end, original_v_start and original_v_end fields should be set to their appropriate values. If the surface has not been extended, the extended_u_int and extended_v_int fields should contain null, and the extended_u_start etc. fields should contain:

```
const char SCH_unset_char = '?'; // generic uninvestigated value
```

As soon as any parameter boundary of the surface is extended, all the fields should be set, regardless of whether the corresponding boundary has been affected by the extension.

The SCH_self_int_t enum is documented in the corresponding curve_data structure under B curve.

The 'swept_form_type', 'spun_form_type' and 'blend_form_type' characters and the corresponding pointers swept_form, spun_form and blend_form, are not implemented in XT. The character fields should be set to SCH_unset_char ('?') and the pointers should be set to null pointer.

If the analytic_form field is not null, it will point to a HELIX_SU_FORM node, which indicates that the surface has a helical shape. In this case the analytic_form_type field will be set to 'H'.

```
struct HELIX_SU_FORM_s
{
    vector    axis_pt           // $v
    vector    axis_dir         // $v
    char      hand              // $c
    interval  turns             // $i
    double    pitch             // $f
    double    gap               // $f
    double    tol               // $f
};

typedef struct HELIX_SU_FORM_s *HELIX_SU_FORM;
```

The axis_pt and axis_dir fields define the axis of the helix. The hand field is '+' for a right-handed and '-' for a left-handed helix. The turns field gives the extent of the helix relative to the profile curve which was used to generate the surface. For instance, an interval [0 10] indicates a start position at the profile curve and an end 10 turns along the axis. Pitch is the distance travelled along the axis in one turn. Tol is the accuracy to which the owning bsurface fits this specification. Gap is for future expansion and will currently be zero. The v parameter increases in the direction of the axis.

H.5.2.2.10 SWEPT_SURF

A swept surface has a parametric representation of the form: