JtXTBrep: In the case of JtXTBrep, the mapping is based on Parasolid identifier of each XT face that is persisted on disk. The identifier is unique within each Parasolid body, but it is not an index. XTBrep maintains a zero-based contiguous index of XT face based on increasing identifier value within the same XT body. If XTBrep contains multiple XT bodies, then the sequence of those XT bodies are fixed across different Parasolid releases and therefore the index of each XT body is implied. 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

Appendix A: Object Type Identifiers

All objects stored in a JT file are classified by type and thus include an object type identifier as part of their persisted data. The data format for these Object Type identifiers is a GUID. These Object Type identifiers are consistent for all objects, of a particular type, in all Version 8.1 JT files.

Table 11: Object Type Identifiers lists the assigned identifier for each Object Type that can exist in a Version 9.5 JT file.

GUID	Object Type
0xffffffff, 0xffff, 0xffff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff	Identifier to signal End-Of-Elements.
Types Stored Within LSG Segment (Segment Type = 1)	
0x10dd1035, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Node Element
0x10dd101b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Group Node Element
0x10dd102a, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Instance Node Element
0x10dd102c, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	LOD Node Element
0xce357245, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	Meta Data Node Element
0xd239e7b6, 0xdd77, 0x4289, 0xa0, 0x7d, 0xb0, 0xee, 0x79, 0xf7, 0x94, 0x94	NULL Shape Node Element
0xce357244, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	Part Node Element
0x10dd103e, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Partition Node Element
0x10dd104c, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Range LOD Node Element
0x10dd10f3, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Switch Node Element
0x10dd1059, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Shape Node Element
0x98134716, 0x0010, 0x0818, 0x19, 0x98, 0x08, 0x00, 0x09, 0x83, 0x5d, 0x5a	Point Set Shape Node Element
0x10dd1048, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Polygon Set Shape Node Element
0x10dd1046, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Polyline Set Shape Node Element
0xe40373c1, 0x1ad9, 0x11d3, 0x9d, 0xaf, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2	Primitive Set Shape Node Element

GUID	Object Type
0x10dd1077, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Tri-Strip Set Shape Node Element
0x10dd107f, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Vertex Shape Node Element
0x10dd1001, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Attribute Data
0x10dd1014, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Draw Style Attribute Element
0xad8dccc2, 0x7a80, 0x456d, 0xb0, 0xd5, 0xdd, 0x3a, 0xb, 0x8d, 0x21, 0xe7	Fragment Shader Attribute Element
0x10dd1083, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Geometric Transform Attribute Element
0x10dd1028, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Infinite Light Attribute Element
0x10dd1096, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Light Set Attribute Element
0x10dd10c4, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Linestyle Attribute Element
0x10dd1030, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Material Attribute Element
0x10dd1045, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Point Light Attribute Element
0x8d57c010, 0xe5cb, 0x11d4, 0x84, 0xe, 0x00, 0xa0, 0xd2, 0x18, 0x2f, 0x9d	Pointstyle Attribute Element
0xaa1b831d, 0x6e47, 0x4fee, 0xa8, 0x65, 0xcd, 0x7e, 0x1f, 0x2f, 0x39, 0xdb	Shader Effects Attribute Element
0x10dd1073, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Texture Image Attribute Element
0x2798bcad, 0xe409, 0x47ad, 0xbd, 0x46, 0xb, 0x37, 0x1f, 0xd7, 0x5d, 0x61	Vertex Shader Attribute Element
0xad8dccc2, 0x7a80, 0x456d, 0xb0, 0xd5, 0xdd, 0x3a, 0xb, 0x8d, 0x21, 0xe7	Fragment Shader Attribute Element
0xaa1b831d, 0x6e47, 0x4fee, 0xa8, 0x65, 0xcd, 0x7e, 0x1f, 0x2f, 0x39, 0xdc	Texture Coordinate Generator Attribute Element
0xa3cfb921, 0xbdeb, 0x48d7, 0xb3, 0x96, 0x8b, 0x8d, 0xe, 0xf4, 0x85, 0xa0	Mapping Plane Element
0x3e70739d, 0x8cb0, 0x41ef, 0x84, 0x5c, 0xa1, 0x98, 0xd4, 0x0, 0x3b, 0x3f	Mapping Cylinder Element
0x72475fd1, 0x2823, 0x4219, 0xa0, 0x6c, 0xd9, 0xe6, 0xe3, 0x9a, 0x45, 0xc1	Mapping Sphere Element

GUID	Object Type
0x92f5b094, 0x6499, 0x4d2d, 0x92, 0xaa, 0x60, 0xd0, 0x5a, 0x44, 0x32, 0xcf	Mapping TriPlanar Element
0x10dd104b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Base Property Atom Element
0xce357246, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	Date Property Atom Element
0x10dd102b, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Integer Property Atom Element
0x10dd1019, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Floating Point Property Atom Element
0xe0b05be5, 0xfbbd, 0x11d1, 0xa3, 0xa7, 0x00, 0xaa, 0x00, 0xd1, 0x09, 0x54	Late Loaded Property Atom ElementSecond specifies the date Second value. Valid values are [0, 59] inclusive. Late Loaded Property Atom Element
0x10dd1004, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	JT Object Reference Property Atom Element
0x10dd106e, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	String Property Atom Element
Types Stored Within JT B-Rep Segment (Segment Type = 2)	
0x873a70c0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	JT B-Rep Element
Types Stored Within Meta Data Segment (Segment Type = 4)	
0xce357249, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	PMI Manager Meta Data Element
0xce357247, 0x38fb, 0x11d1, 0xa5, 0x6, 0x0, 0x60, 0x97, 0xbd, 0xc6, 0xe1	Property Proxy Meta Data Element
Types Stored Within Shape LOD Segment (Segment Type = 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)	
0x3e637aed, 0x2a89, 0x41f8, 0xa9, 0xfd, 0x55, 0x37, 0x37, 0x3, 0x96, 0x82	Null Shape LOD Element
0x98134716, 0x0011, 0x0818, 0x19, 0x98, 0x08, 0x00, 0x09, 0x83, 0x5d, 0x5a	Point Set Shape LOD Element
0x10dd10a1, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Polyline Set Shape LOD Element
0xe40373c2, 0x1ad9, 0x11d3, 0x9d, 0xaf, 0x0, 0xa0, 0xc9, 0xc7, 0xdd, 0xc2	Primitive Set Shape Element
0x10dd10ab, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Tri-Strip Set Shape LOD Element

GUID	Object Type
0x10dd10b0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Vertex Shape LOD Element
Types Stored Within XT B-Rep Segment (Segment Type = 17)	
0x873a70e0, 0x2ac9, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	XT B-Rep Element
Types Stored Within Wireframe Segment (Segment Type = 18)	
0x873a70d0, 0x2ac8, 0x11d1, 0x9b, 0x6b, 0x00, 0x80, 0xc7, 0xbb, 0x59, 0x97	Wireframe Rep Element
Types Stored Within JT ULP Segment (Segment Type = 20)	
0xf338a4af, 0xd7d2, 0x41c5, 0xbc, 0xf2, 0xc5, 0x5a, 0x88, 0xb2, 0x1e, 0x73	JT ULP Element
Types Stored Within JT LWPA Segment (Segment Type = 24)	
0xd67f8ea8, 0xf524, 0x4879, 0x92, 0x8c, 0x4c, 0x3a, 0x56, 0x1f, 0xb9, 0x3a	JT LWPA Element

Table 11: Object Type Identifiers

Appendix B: Semantic Value Class Shader Parameter Values

7.2.1.1.2.12 Vertex Shader Attribute Element and 7.2.1.1.2.13 Fragment Shader Attribute Element contain shader parameters. These shader parameters can be of a "Semantic" Value Class which indicates that the shader parameter is implicitly tied/bound to a piece of either OpenGL or JT graphics system state. Table 12 below documents all the possible "Semantic" Value Class shader parameter Values (i.e. the graphics system state the parameter is bound to).

Table 12: Semantic Value Class Shader Parameter Values

Value	Description of Semantically Bound Graphics State	Notes
= 0	– Unknown	
Related to Current OpenGL State		
= 30	– View Transform Matrix	Cg only
= 31	– Combined Model-View Transform Matrix	Cg only
= 32	– Projection Transform Matrix	Cg only
= 33	– Texture Transform Matrix	Cg only
= 34	– Combined Model-View-Projection Transform Matrix	Cg only
= 35	– View Matrix Transposed	Cg only
= 36	– Combined Model-View Transform Matrix Transposed	Cg only
= 37	– Projection Transform Matrix Transposed	Cg only
= 38	– Texture Transform Matrix Transposed	Cg only
= 39	– Combined Model-View-Projection Transform Matrix Transposed	Cg only
= 40	– View Transform Matrix Inverse	Cg only
= 41	– Combined Model-View Transform Matrix Inverse	Cg only
= 42	– Projection Transform Matrix Inverse	Cg only
= 43	– Texture Transform Matrix Inverse	Cg only
= 44	– Combined Model-View-Projection Transform Matrix Inverse	Cg only
= 45	– View Transform Matrix Inverse Transposed	Cg only
= 46	– Combined Model-View Transform Matrix Inverse Transposed	Cg only
= 47	– Projection Transform Matrix Inverse Transposed	Cg only
= 48	– Texture Transform Matrix Inverse Transposed	Cg only
= 49	– Combined Model-View-Projection Transform Matrix Inverse Transposed	Cg only
Related to Current JT State		
= 70	– Current Model Transform	
= 71	– Current Model Transform Transposed	
= 72	– Current Model Transform Inverse	
= 73	– Current Model Transform Inverse Transposed	
= 75	– Current Material Emissive Color	
= 76	– Current Material Diffuse Color	
= 77	– Current Material Specular Color	
= 78	– Current Material Ambient Color	
= 79	– Current Material Shininess	
= 80	– Current Fog Color	
= 81	– Separate Specular Color Flag	
= 82	– Global Ambient Light Color	
= 83	– Exposure	
= 84	– Bumpiness	
= 85	– Environment Reflectivity	
= 86	– Depth Peeling Texture 0	

= 87	– Depth Peeling Texture 1	
= 99	– Number of VPCS Lights	
= 101	– VPCS Light-0 Specular Color	
= 102	– VPCS Light-0 Ambient Color	
= 103	– VPCS Light-0 Attenuation	
= 104	– VPCS Light-0 Position	
= 105	– VPCS Light-0 Direction	
= 106	– VPCS Light-0 Spot Direction	
= 107	– VPCS Light-0 Spot Cone Angle	
= 108	– VPCS Light-0 Cosine of Spot Cone Angle	
= 109	– VPCS Light-0 Spot Exponent	
= 110	– VPCS Light-0 Shadow Opacity	
= 120 → 130	– Same as values 100 → 110 except for VPCS Light-1	
= 140 → 150	– Same as values 100 → 110 except for VPCS Light-2	
= 160 → 170	– Same as values 100 → 110 except for VPCS Light-3	
= 180 → 190	– Same as values 100 → 110 except for VPCS Light-4	
= 200 → 210	– Same as values 100 → 110 except for VPCS Light-5	
= 220 → 230	– Same as values 100 → 110 except for VPCS Light-6	
= 240 → 250	– Same as values 100 → 110 except for VPCS Light-7	
= 499	– Number of MCS Lights	
= 500 → 510	– Same as values 100 → 110 except for MCS Light-0	
= 520 → 530	– Same as values 100 → 110 except for MCS Light-1	
= 540 → 550	– Same as values 100 → 110 except for MCS Light-2	
= 560 → 570	– Same as values 100 → 110 except for MCS Light-3	
= 580 → 590	– Same as values 100 → 110 except for MCS Light-4	
= 600 → 610	– Same as values 100 → 110 except for MCS Light-5	
= 620 → 630	– Same as values 100 → 110 except for MCS Light-6	
= 640 → 650	– Same as values 100 → 110 except for MCS Light-7	
= 899	– Number of WCS Lights	
= 900 → 910	– Same as values 100 → 110 except for WCS Light-0	
= 920 → 930	– Same as values 100 → 110 except for WCS Light-1	
= 940 → 950	– Same as values 100 → 110 except for WCS Light-2	
= 960 → 970	– Same as values 100 → 110 except for WCS Light-3	
= 980 → 990	– Same as values 100 → 110 except for WCS Light-4	
= 1000 → 1010	– Same as values 100 → 110 except for WCS Light-5	
= 1020 → 1030	– Same as values 100 → 110 except for WCS Light-6	
= 1040 → 1050	– Same as values 100 → 110 except for WCS Light-7	
= 1500	– Current Texture Object-0	Cg only
= 1501	– Current Texture Object-1	Cg only
= 1502	– Current Texture Object-2	Cg only
= 1503	– Current Texture Object-3	Cg only
= 1504	– Current Texture Object-4	Cg only
= 1505	– Current Texture Object-5	Cg only
= 1506	– Current Texture Object-6	Cg only
= 1507	– Current Texture Object-7	Cg only

= 1600	– Current Texture Unit-0	GLSL only
= 1601	– Current Texture Unit-1	GLSL only
= 1602	– Current Texture Unit-2	GLSL only
= 1603	– Current Texture Unit-3	GLSL only
= 1604	– Current Texture Unit-4	GLSL only
= 1605	– Current Texture Unit-5	GLSL only
= 1606	– Current Texture Unit-6	GLSL only
= 1607	– Current Texture Unit-7	GLSL only
= 1700	– Texture Channel-0 VCS Texture Coordinate Generation S-Plane	
= 1701	– Texture Channel-0 VCS Texture Coordinate Generation T-Plane	
= 1702	– Texture Channel-0 VCS Texture Coordinate Generation R-Plane	
= 1703	– Texture Channel-0 VCS Texture Coordinate Generation Q-Plane	
= 1710 → 1713	– Same as 1700 → 1703 except for Chanel-1 VCS	
= 1720 → 1723	– Same as 1700 → 1703 except for Chanel-2 VCS	
= 1730 → 1733	– Same as 1700 → 1703 except for Chanel-3 VCS	
= 1740 → 1743	– Same as 1700 → 1703 except for Chanel-4 VCS	
= 1750 → 1753	– Same as 1700 → 1703 except for Chanel-5 VCS	
= 1760 → 1763	– Same as 1700 → 1703 except for Chanel-6 VCS	
= 1770 → 1773	– Same as 1700 → 1703 except for Chanel-7 VCS	
= 2000 → 2003	– Same as 1700 → 1703 except for Chanel-0 MCS	
= 2010 → 2013	– Same as 1700 → 1703 except for Chanel-1 MCS	
= 2020 → 2023	– Same as 1700 → 1703 except for Chanel-2 MCS	
= 2030 → 2033	– Same as 1700 → 1703 except for Chanel-3 MCS	
= 2040 → 2043	– Same as 1700 → 1703 except for Chanel-4 MCS	
= 2050 → 2053	– Same as 1700 → 1703 except for Chanel-5 MCS	
= 2060 → 2063	– Same as 1700 → 1703 except for Chanel-6 MCS	
= 2070 → 2073	– Same as 1700 → 1703 except for Chanel-7 MCS	
= 3000	– Texture Channel-0 Matrix	
= 3001	– Texture Channel-1 Matrix	
= 3002	– Texture Channel-2 Matrix	
= 3003	– Texture Channel-3 Matrix	
= 3004	– Texture Channel-4 Matrix	
= 3005	– Texture Channel-5 Matrix	
= 3006	– Texture Channel-6 Matrix	
= 3007	– Texture Channel-7 Matrix	
= 3100	– Texture Channel-0 Map Resolution	
= 3101	– Texture Channel-1 Map Resolution	
= 3102	– Texture Channel-2 Map Resolution	
= 3103	– Texture Channel-3 Map Resolution	
= 3104	– Texture Channel-4 Map Resolution	
= 3105	– Texture Channel-5 Map Resolution	
= 3106	– Texture Channel-6 Map Resolution	
= 3107	– Texture Channel-7 Map Resolution	
= 3200	– Texture Channel-0 Map Resolution Inverses (i.e. 1.0 /"Map Resolution")	
= 3201	– Texture Channel-1 Map Resolution Inverses	

= 3202	– Texture Channel-2 Map Resolution Inverses	
= 3203	– Texture Channel-3 Map Resolution Inverses	
= 3204	– Texture Channel-4 Map Resolution Inverses	
= 3205	– Texture Channel-5 Map Resolution Inverses	
= 3206	– Texture Channel-6 Map Resolution Inverses	
= 3207	– Texture Channel-7 Map Resolution Inverses	
= 3300	– Texture Channel-0 Blend Color	
= 3301	– Texture Channel-1 Blend Color	
= 3302	– Texture Channel-2 Blend Color	
= 3303	– Texture Channel-3 Blend Color	
= 3304	– Texture Channel-4 Blend Color	
= 3305	– Texture Channel-5 Blend Color	
= 3306	– Texture Channel-6 Blend Color	
= 3307	– Texture Channel-7 Blend Color	

Appendix C: Decoding Algorithms – An Implementation

This Appendix provides a sample C++ implementation for the decoding portion of the various compression CODECs (as detailed in 8.2 Encoding Algorithms) used in the JT format. This sample code is not intended to be fully functional decoder class implementations, but is instead intended to demonstrate the fundamentals of implementing the decoding portion of the CODEC algorithms used in the JT format.

1 Common classes

The following sub-sections define some general classes used by all the decoding algorithms.

1.1 CntxEntry class

```
//
// Type used to build probability context tables.
// Used by ProbabilityContext class.
//
class CntxEntry
{
public:

    Int32 iSym;          // Symbol
    Int32 cCount;        // Number of occurrences
    Int32 cCumCount;     // Cumulative number of occurrences
    Int32 iNextCntx = 0; // Next context if this symbol seen
};
```

1.2 ProbabilityContext class

```
//
// Type used to build probability context tables.
// Used by CodecDriver class.
//
class ProbabilityContext
{
public:

    // Returns total cumulative count for all context entries
    Int32 totalCount();

    // Returns number of context entries
    Int32 numEntries();

    // Returns context entry of index iEntry
    Bool getEntry(Int32 iEntry, CntxEntry& rpEntry);

    // Looks up the next context field given by the context entry
    // with input symbol 'iSymbol'
    Bool lookupNextContext(Int32 iSymbol, Int32& iNextContext);

    // Looks up the index of the context entry for the given
    // input symbol 'iSymbol'
    Bool lookupSymbol(Int32 iSymbol, Int32& iOutEntry);

    // Looks up the index of the context entry that falls just above
    // the accumulated count.
    Bool lookupEntryByCumCount(Int32 iCount, Int32& iOutEntry);
};
```

1.3 CodecDriver class

```
//
// A class that deals with the conversions from SYMBOL to VALUE and
// provides end-consumer APIs for using the codecs.
//
class CodecDriver
```

```
{
public:
    // ---------- Codec Decoding Interface ----------
    // Returns the number of symbols to be read
    Int32 numSymbolsToRead();

    // Returns index of the first context entry and total number of bits
    Bool getDecodeData(Int32& iFirstContext, Int32& nTotalBits);

    // Returns the next 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>& vVal,
                                      Int32 iIndex,
                                      PredictorType ePredType)
{
    Int32* aVals = (Int32 *) rvVals;
```

```
    JtInt32 iPredicted,
              v1 = aVals[iIndex-1],
              v2 = aVals[iIndex-2],
              v3 = aVals[iIndex-3],
              v4 = aVals[iIndex-4];

    switch( ePredType )
    {
        default:
        case PredLag1:
        case PredXor1:
            iPredicted = v1;
            break;

        case PredLag2:
        case PredXor2:
            iPredicted = v2;
            break;

        case PredStride1:
            iPredicted = v1 + (v1 - v2);
            break;

        case PredStride2:
            iPredicted = v2 + (v2 - v4);
            break;

        case PredStripIndex:
            if( v2 - v4 < 8 && v2 - v4 > -8 )
                iPredicted = v2 + (v2 - v4);
            else
                iPredicted = v2 + 2;
            break;

        case PredRamp:
            iPredicted = iIndex;
            break;
    }

    return iPredicted;
}


Float64 CodecDriverBase::predictValue(Vector<Float64>& vVal,
                                      Int32 iIndex,
                                      PredictorType ePredType)
{
    Float64* aVals = (Float64 *) rvVals;
    Float64 iPredicted,
              v1 = aVals[iIndex-1],
              v2 = aVals[iIndex-2],
              v3 = aVals[iIndex-3],
              v4 = aVals[iIndex-4];

    switch( ePredType )
    {
        default:
        case PredLag1:
            iPredicted = v1;
            break;

        case PredLag2:
            iPredicted = v2;
            break;

        case PredStride1:
            iPredicted = v1 + (v1 - v2);
            break;

        case PredStride2:
            iPredicted = v2 + (v2 - v4);
```

```
            break;

        case PredStripIndex:
            if( v2 - v4 < 8 && v2 - v4 > -8 )
                iPredicted = v2 + (v2 - v4);
            else
                iPredicted = v2 + 2;
            break;

        case PredRamp:
            iPredicted = iIndex;
            break;
    }

    return iPredicted;
}
```

1.4 CodecDriver2 class

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 8.2.2 Bitlength CODEC.

2.1 BitLengthCodec class

```
class BitLengthCodec
{
public:
    // This method decodes a given stream of symbols into their values.
    // The stream is described by the codec driver
    Bool decode(CodecDriver* pDriver);

    Int32 cStepBits = 2;
};

Bool BitLengthcodec::decode(CodecDriver* pDriver)
{
    Int32 iBit;               // Codetext bit number
    Int32 nBits = 0;          // Number of codetext bits decoded so far
    Int32 nTotalBits = 0;  // Total number of codetext bits expected
    Int32 nValBits = 0;       // Number of accumulated value bits
    Int32 iContext = 0;       // Probability context number
    Int32 iSymbol;            // Decoded symbol value
    UInt32 uVal = 0;          // Current chunk of codetext bits
    UInt32 uAccVal = 0;       // Number of valid bits returned from
                                  // getNextCodeText
    UInt32 uLastIncBit = 0;   // Used to calculate whether terminator bit
                                  // is 0 or 1
    Int32 cNumCurBits = 0; // Current field width in bits
    Int32 nAccBits = 0;       // Number of bits accum'ed in uAccVal
    Int32 iDecodeState = 0;   // State of decoder; see below

    // Get codetext from the driver and loop over it until it's gone!
    pDriver->getDecodeData(iContext, nTotalBits);

    while( nBits < nTotalBits )
    {
        // Get the next 32 bits from the input stream
        pDriver->getNextCodeText(uVal, nValBits);

        // Scan through each bit either walking the Huffman code
        // tree or accumulating escaped bit values.
        Int32 n = min(32, min(nValBits, nTotalBits - nBits));
        for( iBit = 0; iBit < n ; iBit++ )
        {
            // Code-accumulation mode is handled is this block
            // as many bits at a time as possible.
```

```
if( iDecodeState == 2 )
{

    // Slice off as many bits as we can all at once.
    Int32 m = min(n - iBit, cNumCurBits - nAccBits);
    if( m < 32 )
    {
        uAccVal <<= m;
        uAccVal |= ((uVal >> (32 - m)) & ((1 << m) - 1));
        nAccBits += m;
        iBit += m - 1;

        // Advance the bit-marching counters
        uVal <<= m;
        nBits += m;
        nValBits -= m;
    }
    else
    {
        uAccVal = uVal;
        nAccBits += m;
        iBit += m - 1;

        // Advance the bit-marching counters
        uVal = 0;
        nBits += m;
        nValBits -= m;
    }

    if( nAccBits >= cNumCurBits )
    {
        // Convert and sign-extend the symbol
        iSymbol = Int32(uAccVal);
        iSymbol <<= (32 - cNumCurBits);
        iSymbol >>= (32 - cNumCurBits);

        // Output the symbol and restart
        pDriver->addOutputSymbol(iSymbol, iContext);
        iDecodeState = 0;
        uAccVal  = 0;
        nAccBits = 0;
    }
}
else
{
    // All other decode states are handled one bit at a time
    // inside this block.
    // Get the next bit
    uAccVal = (uVal >> 31);

    switch( iDecodeState )
    {

        // DecodeState = 0: Recognize prefix bit (0=Same size
        // code, 1=Different size code).
        case 0:
            // Recognize "same length" prefix code
            if( uAccVal == 0 )
                iDecodeState = 2;
            else
            {
                // Recognize "different length" prefix code
                iDecodeState = 1;
                uLastIncBit = 2;
            }

            uAccVal  = 0;
            break;

        case 1:  // Length adjustment mode
            // Recognize the terminator bit
```

```
                    if( uLastIncBit != 2 && (uAccVal ^ uLastIncBit) )
                    {
                        iDecodeState = 2;
                        uLastIncBit = 2;
                    }
                    else
                    {
                        // Recognize the "increment" prefix code
                        if( uAccVal == 1 )
                        {
                            cNumCurBits += cStepBits;
                        }
                        else
                        {
                            // Recognize the "decrement" prefix code
                            cNumCurBits -= cStepBits;
                        }

                        uLastIncBit = uAccVal;
                    }

                    uAccVal  = 0;
                    break;
                }

                // Advance the bit-marching counters that keep track of the
                // "current codetext bit", and how many bits are left.
                uVal <<= 1;
                nBits++;
                nValBits--;
            }
        }
    }

    // If the last symbol was zero and the current bit length
    // is also zero, then the above loop terminated before
    // actually decoding the last zero-valued symbol.  Test
    // for that condition here and decode it if necessary.
    if( iDecodeState == 2 && cNumCurBits == 0 )
    {
        // Output the symbol and restart
        iSymbol = Int32(0);
        pDriver->addOutputSymbol(iSymbol, iContext);
    }

    return True;
}
```

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 8.2.3 Arithmetic CODEC.

3.1 ArithmeticProbabilityRange class

```
class ArithmeticProbabilityRange
{
public:
    UInt16 low_count;
    UInt16 high_count;
    UInt16 scale;
}
```

3.2 ArithmeticCodec class

ArithmeticCodec class is the class that decodes arithmetic encoded data.

```
class ArithmeticCodec
{
```

```
public:
    ArithmeticCodec() :
        code = 0x0000,
        low = 0x0000,
        high = 0xffff,
        nUnderflowBits = 0,
        bitBuffer =0x00000000,
        nBits = 0
    {
    }

    // Decodes a list of symbols. The codecDriver provides the range
    // info for the symbols to decode.  It also stores the symbols as
    // they are decoded.
    Bool decode(CodecDriver* pDriver);

private:
    // Remove the most recently decoded symbol from the front of the
    // list of encoded symbols.
    Bool removeSymbolFromStream(ArithmeticProbabilityRange& sym,
                                CodecDriver* pDriver);

    //State variables used in decoding.
    UInt16 code;      // Present input code value, for decoding only
    UInt16 low;       // Start of the current code range
    UInt16 high;      // End of the current code range

    UInt32 bitBuffer; // Temporary i/o buffer
    Int32  nBits;     // Number of bits in _bitBuffer
};

Bool ArithmeticCodec::decode(CodecDriver* pDriver )
{
    ArithmeticProbabilityRange newSymbolRange;
    Int32 iCurrContext, nDummyTotalBits, cSymbolsCurrCtx, iCurrEntry;

    Int32 nSymbols = pDriver->numSymbolsToRead();

    ProbabilityContext* pCurrContext = NULL;
    CntxEntry* pCntxEntry = NULL;

    // Initialize decoding process
    Int32 nBitsRead = -1;
    pDriver->getNextCodeText(bitBuffer, nBitsRead);

    low  = 0;
    high = 0xffff;
    code = (bitBuffer >> 16);

    bitBuffer <<= 16;
    nBits = 16;

    // Begin decoding
    pDriver->getDecodeData(iCurrContext, nDummyTotalBits);
    for( Int32 ii = 0; ii < nSymbols; ii++ )
    {
        pDriver->getContext(iCurrContext, pCurrContext);

        cSymbolsCurrCtx = pCurrContext->totalCount();
        UInt16 rescaledCode =
            ((((UInt32)(code - low) + 1) * (UInt32) cSymbolsCurrCtx - 1) /
                ((UInt32)(high - low) + 1));

        pCurrContext->lookupEntryByCumCount((Int32)rescaledCode,
                                            iCurrEntry);

        pCurrContext->getEntry(iCurrEntry, pCntxEntry);

        newSymbolRange.high_count = pCntxEntry->cCumCount +
                                    pCntxEntry.cCount;
        newSymbolRange.low_count  = pCntxEntry->cCumCount;
```

```
        newSymbolRange.scale      = cSymbolsCurrCtx;

        removeSymbolFromStream(newSymbolRange, pDriver);

        pDriver->addOutputSymbol(pCntxEntry);

        iCurrContext = pCntxEntry->iNextCntx;
    }

    return True;
}

Bool ArithmeticCodec::removeSymbolFromStream(
                                ArithmeticProbabilityRange& sym,
                                CodecDriver* pDriver)
{
    // First, the range is expanded to account for the symbol removal.
    UInt32 range = UInt32(high - low)  + 1;
    high = low + (UInt32)((range * sym.high_count) / sym.scale - 1);
    low  = low + (UInt32)((range * sym.low_count ) / sym.scale);

    //Next, any possible bits are shipped out.
    for (;;)
    {
        // If the most signif digits match, the bits will be shifted out.
        if( (~(high^low)) & 0x8000 )
        {
        }
        else if( (low & 0x4000) && !(high & 0x4000) )
        {
            // Underflow is threatening, shift out 2nd most signif digit.
            code ^= 0x4000;
            low  &= 0x3fff;
            high |= 0x4000;
        }
        else
        {
            // Nothing can be shifted out, so return.
            return True;
        }

        low  <<= 1;
        high <<= 1;
        high |=  1;
        code <<= 1;

        if( nBits == 0 )
        {
          // The returned nBits here will always be 32
            pDriver->getNextCodeText(bitBuffer, nBits);
        }

        code |= (UInt16)(bitBuffer >> 31);
        bitBuffer <<= 1;
        nBits--;
    }
}
```

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 8.2.4 Deering Normal CODEC.

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;
}
```

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 must be <= 13.

    // Code layout: [sextant:3][octant:3][theta:numBits][psi:numBits]

    outVec.setValues(0,0,0);
    Float32 fPsiMax = 0.615479709;

    UInt32  iBitRange = 1<<numBits;
    Float32 fBitRange = Float32(iBitRange);

    // For sextants 1, 3, and 5, iTheta needs to be incremented
```

```
iTheta += (iSextant & 1);

Float32 fCosTheta, fSinTheta, fCosPsi, fSinPsi;

DeeringNormalLookupTable LookupTable;

if( (LookupTable.numBitsPerAngle() < (UInt32)numBits) ||
     !LookupTable.lookupThetaPsi(iTheta, iPsi, numBits,
                                      fCosTheta, fSinTheta,
                                      fCosPsi, fSinPsi) )
{
    Float32 fTheta = asin(tan(fPsiMax * Float32(iBitRange - iTheta) /
                                    fBitRange));

    Float32 fPsi = fPsiMax * (iPsi / fBitRange);
    fCosTheta = cos(fTheta);
    fSinTheta = sin(fTheta);
    fCosPsi   = cos(fPsi);
    fSinPsi   = sin(fPsi);
}

Float32 x,y,z;
Float32 xx = x = fCosTheta * fCosPsi;
Float32 yy = y = fSinPsi;
Float32 zz = z = fSinTheta * fCosPsi;

//Change coordinates based on the sextant
switch( iSextant )
{
    case 0:     // No op
        break;

    case 1:     // Mirror about x=z plane
        z = xx;
        x = zz;
        break;

    case 2:     // Rotate CW
        z = xx;
        x = yy;
        y = zz;
        break;

    case 3:     // Mirror about x=y plane
        y = xx;
        x = yy;
        break;

    case 4:     // Rotate CCW
        y = xx;
        z = yy;
        x = zz;
        break;

    case 5:     // Mirror about y=z plane
        z = yy;
        y = zz;
        break;
};

//Change some more based on the octant

//if first bit is 0, negate x component
if( !(iOctant & 0x4) )
    x = -x;

//if second bit is 0, negate y component
if( !(iOctant & 0x2) )
    y = -y;

//if third bit is 0, negate z component
```

```
    if( !(iOctant & 0x1) )
        z = -z;

    outVec.setValues(x,y,z);

    return True;
}

Bool DeeringNormalCodec::unpackCode(UInt32 code,
                                    UInt32& outSextant,
                                    UInt32& outOctant,
                                    UInt32& outTheta,
                                    UInt32& outPsi)
{
    UInt32 mask = (1<<numBits)-1;

    outSextant = (code >> (numBits+numBits+3)) & 0x7;
    outOctant  = (code >> (numBits+numBits))   & 0x7;
    outTheta   = (code >> (numBits))           & mask;
    outPsi     = (code)                        & mask;

    return True;
}
```

Appendix D: Hashing – An Implementation

This Appendix provides a sample C++ implementation for the creation of hash values (as detailed in 8.2 Encoding Algorithms) used in the JT format.

```
unsigned int hash32( const unsigned int *pWords,
                     int nWords,
                     unsigned int uSeedHashValue )
{ return hash2(pWords, nWords, uSeedHashValue); }

unsigned int jthash16(const unsigned short *pBytes,
                      int nShort,
                      unsigned int uSeedHashValue)
{ return hash3(pBytes, nShort, uSeedHashValue); }

//--------------------------------------------------------------------
//  mix -- mix 3 32-bit values reversibly.
// For every delta with one or two bit set, and the deltas of all three
//   high bits or all three low bits, whether the original value of a,b,c
//   is almost all zero or is uniformly distributed,
// * If mix() is run forward or backward, at least 32 bits in a,b,c
//   have at least 1/4 probability of changing.
// * If mix() is run forward, every bit of c will change between 1/3 and
//   2/3 of the time.  (Well, 22/100 and 78/100 for some 2-bit deltas.)
// mix() was built out of 36 single-cycle latency instructions in a
//   structure that could supported 2x parallelism, like so:
//       a -= b;
//       a -= c; x = (c>>13);
//       b -= c; a ^= x;
//       b -= a; x = (a<<8);
//       c -= a; b ^= x;
//       c -= b; x = (b>>13);
//       ...
//   Unfortunately, superscalar Pentiums and Sparcs can't take advantage
//   of that parallelism.  They've also turned some of those single-cycle
//   latency instructions into multi-cycle latency instructions.  Still,
//   this is the fastest good hash I could find.  There were about 2^^68
//   to choose from.  I only looked at a billion or so.
--------------------------------------------------------------------

#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8); \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12);  \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5); \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}

--------------------------------------------------------------------
// hash() -- hash a variable-length key into a 32-bit value
//   k     : the key (the unaligned variable-length array of bytes)
//   len   : the length of the key, counting by bytes
//   level : can be any 4-byte value
// Returns a 32-bit value.  Every bit of the key affects every bit of
// the return value.  Every 1-bit and 2-bit delta achieves avalanche.
// About 36+6len instructions.

// The best hash table sizes are powers of 2.  There is no need to do
// mod a prime (mod is sooo slow!).  If you need less than 32 bits,
// use a bitmask.  For example, if you need only 10 bits, do
//   h = (h & hashmask(10));
// In which case, the hash table should have hashsize(10) elements.
//
```

```
// If you are hashing n strings (JtUInt8 **)k, do it like this:
//   for (i=0, h=0; i<n; ++i) h = hash( k[i], len[i], h);
//
// By Bob Jenkins, 1996.  bob_jenkins@burtleburtle.net.  You may use this
// code any way you wish, private, educational, or commercial.  It's free.
//
// See http://burtleburtle.net/bob/                  // 2010/02/12
// See http://burtleburtle.net/bob/hash/doobs.html   // 2010/02/12
//
// Use for hash table lookup, or anything where one collision in 2^32 is
// acceptable.  Do NOT use for cryptographic purposes.
//--------------------------------------------------------------------

//--------------------------------------------------------------------
// This works on all machines.  hash2() is identical to hash() on
// little-endian machines, except that the length has to be measured
// in ub4s instead of bytes.  It is much faster than hash().  It
// requires
// -- that the key be an array of UInt32's, and
// -- that all your machines have the same endianness, and
// -- that the length be the number of UInt32's in the key
// --------------------------------------------------------------------
unsigned int hash(const usigned char *k,        // key
                  unsigned int        length,   // length of the key
                  unsigned int        initval)  // prev hash, or an arbitrary value
{
   register unsigned int a,b,c,len;

   /* Set up the internal state */
   len = length;
   a = b = 0x9e3779b9;  /* the golden ratio; an arbitrary value */
   c = initval;           /* the previous hash value */
   /*---------------------------------------- handle most of the key */
   while (len >= 12) {
      a += (k[0] +((UInt32)k[1]<<8) +((UInt32)k[2]<<16) +((UInt32)k[3]<<24));
      b += (k[4] +((UInt32)k[5]<<8) +((UInt32)k[6]<<16) +((UInt32)k[7]<<24));
      c += (k[8] +((UInt32)k[9]<<8) +((UInt32)k[10]<<16)+((UInt32)k[11]<<24));
      mix(a,b,c);
      k += 12; len -= 12;
   }
   /*------------------------------------- handle the last 11 bytes */
   c += length;
   switch(len) {            /* all the case statements fall through */
     case 11[239b((UInt32)k[10]<<24);
     case 10[239b((UInt32)k[9]<<16);
     case 9 [239b((UInt32)k[8]<<8);
      /* the first byte of c is reserved for the length */
     case 8 [2b9b((UInt32)k[7]<<24);
     case 7 [2b9b((UInt32)k[6]<<16);
     case 6 [2b9b((UInt32)k[5]<<8);
     case 5 [2b9bk[4];
     case 4 [2a9b((UInt32)k[3]<<24);
     case 3 [2a9b((UInt32)k[2]<<16);
     case 2 [2a9b((UInt32)k[1]<<8);
     case 1 [2a9bk[0];
     /* case 0[2nothing 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;
}
```

Appendix E: Polygon Mesh Topology Coder

The topology coding algorithm described here is used to code the *dual* of the desired mesh. Thus, for example, the reader will need to take the dual of the decoded mesh in order to obtain the original primal mesh. Presented below are classes suitable for representing the dual of a polygon mesh and the dual topology decoding algorithm.

At a high level, the topology coder works by traversing the dual mesh to be encoded one vertex and one face at a time. The coder maintains a queue of faces to be processed; the initial queue is created using the valence of an arbitrary vertex of the mesh followed by the degrees of the faces adjacent to that vertex, and adds the adjacent faces to the face queue. Each time it visits a face, it encodes the *degree* of that face and emits any incident vertices that have not yet been visited. Each time the coder visits a vertex, it encodes the *valence* of the vertex (usually 3 in the current case), and emits any incident faces that have not yet been visited. It works its way through the mesh in this fashion until all vertices and faces have been encoded. Thus, the primary output from the topology coder is a list of vertex valences and face degrees. These two fields plus two more encoding so-called *split faces*, coupled with the exact coder implementation completely encode the mesh topology in a very compact manner[1].

In addition to these two basic fields are added a number of other fields that organize the dual vertices into *vertex groups*, and also encode the *vertex attributes* (e.g. normals, colors, 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 color, normal, and texture coordinates. An important observation in real-world data is that adjacent faces tend to share the same vertex attributes. Thus, a natural way to encode which vertex attributes map to which faces within a given valence ring (the counter-clockwise ordered set of faces incident on a given vertex) is by way of a bit vector. The bit vector begins at the first face the coder encounters that is incident to the vertex, and proceeds counter clockwise around the vertex, allocating one bit per incident face. A value of 0 is assigned to the bit if all vertex attributes for the face are the same as the face immediately clockwise. A value of 1 is assigned if the vertex attributes for the face are different. Recall that these bits from the original primal mesh are encoded as face attributes in the dual mesh.

Thus, at the end of the coding process, there will be one such bit vector per topological vertex in the mesh. These bit vectors will be of disparate lengths because all vertex valences are not the same. Though there is no theoretical limit to the valence of any given vertex, in practice, the vertex valences seldom rise above six, and only rarely rise into the dozens. As a matter of practicality, then, we break this list of bit vectors into those of length 64 and smaller into one group, and all others into a list of so-called "high-valence" bit vectors. The low-valence bit vectors are encoded into 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.

[1] Similar methods of topology coding are described in [18] 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, colors, and texture coordinates.

1 DualVFMesh

The DualVFMesh (Dual Vertex-Facet Mesh) is a support class paired with the topology decoder itself, and represents a closed two-manifold polygon mesh. The topology decoder reconstructs the encoded dual mesh into a DualVFMesh, building it one vertex and one facet at a time. When the decoder is finished, it will have visited each vertex and each face of the dual mesh exactly once. DualVFMesh is not intended as a work horse in-memory storage container because its way of encoding the topological connections between faces and vertices is memory-intensive.

```
class DualVFMesh
{
  public:
    // ========== Housekeeping Interface ==========
    DualVFMesh();
    DualVFMesh (const DualVFMesh &rhs);
    DualVFMesh &operator=(const DualVFMesh &rhs);

    // ========== Topology Interface ==========

    // Vtx creation
    bool          isValidVtx (Int32  iVtx) const;
    bool          newVtx     (Int32  iVtx,
                              Int32  iValence,
                              UInt16 uFlags = 0);
    bool          setVtxFlags(Int32  iVtx,
                              UInt16 uFlags);
    bool          setVtxGrp  (Int32  iVtx,
                              Int32  iVGrp);
    UInt16        vtxFlags   (Int32  iVtx) const;
    Int32         vtxGrp     (Int32  iVtx) const;

    // Face creation
    bool          isValidFace  (Int32  iFace) const;
    bool          newFace      (Int32  iFace,
                                Int32  cDegree,
                                Int32  cFaceAttrs = 0,
                                UInt64 uFaceAttrMask = 0,
                                UInt16 uFlags = 0);
    bool          newFace      (Int32  iFace,
                                Int32  cDegree,
                                Int32  cFaceAttrs,
                                const BitVec *pvbFaceAttrMask,
                                UInt16 uFlags);
    bool          setFaceFlags (Int32  iFace,
                                UInt16 uFlags);
    UInt16        faceFlags    (Int32  iVtx) const;
    bool          setFaceAttr  (Int32  iFace,
                                Int32  iAttrSlot,
                                Int32  iFaceAttr);
    Int32         faceAttr     (Int32  iFace,
                                Int32  iAttrSlot) const;

    // Topology connection
    bool          setVtxFace(Int32  iVtx,
                             Int32  iFaceSlot,
                             Int32  iFace);
    bool          setFaceVtx(Int32  iFace,
                             Int32  iVtxSlot,
                             Int32  iVtx);

    // Queries
    Int32         valence    (Int32  iVtx) const
        { return _vVtxEnts[iVtx].cVal; }
    Int32         degree   (Int32  iFace ) const
        { return _vFaceEnts[iFace].cDeg; }
    Int32         face    (Int32  iVtx,
                           Int32  iFaceSlot) const
          { return _viVtxFaceIndices[(_vVtxEnts[iVtx]).iVFI + iFaceSlot]; }
    Int32         vtx      (Int32  iFace,
                            Int32  iVtxSlot) const
```

```
      { return _viFaceVtxIndices[_vFaceEnts[iFace].iFVI + iVtxSlot]; }
  Int32         numVts       () const
      { return _vVtxEnts.length(); }
  Int32         numFaces     () const
      { return _vFaceEnts.length(); }
  Int32         numAttrs     () const
      { return _viFaceAttrIndices.length(); }
  Int32         numAttrs       (Int32 iFace) const
      { return _vFaceEnts[iFace].cFaceAttrs; }
  UInt64        attrMask       (Int32 iFace) const
      { return _vFaceEnts[iFace].u.uAttrMask; }
  const BitVec *attrMaskV      (Int32 iFace) const
      { return _vFaceEnts[iFace].u.pvbAttrMask; }
  Int32         findVtxSlot   (Int32 iFace,
                               Int32 iTargVtx) const;
  Int32         findFaceSlot    (Int32 iVtx,
                                 Int32 iTargFace) const;
  Int32         emptyFaceSlots  (Int32 iFace) const
      { return _vFaceEnts[iFace].cEmptyDeg; }

  // ========== VFMesh Data Members ==========
public:
  class VtxEnt {
    public:
      VtxEnt() : cVal(0), uFlags(0), iVGrp(-1), iVFI(-1) {}
      UInt16    cVal;   // Vtx valence
      UInt16    uFlags; // User flags
      Int32     iVGrp;  // Vtx group
      Int32     iVFI;   // Idx into _viVtxFaceIndices of cVal incident faces
  };

  // Number of optimized mask bits.
  static const Int32 cMBits = 64;

  class FaceEnt {
    public:
      FaceEnt() : cDeg(0), uFlags(0), cEmptyDeg(0),
                  cFaceAttrs(0), iFVI(-1), iFAI(-1) { u.uAttrMask = 0; }
      FaceEnt(const FaceEnt &rhs) : cDeg(rhs.cDeg), cEmptyDeg(rhs.cEmptyDeg),
                                    cFaceAttrs(rhs.cFaceAttrs), iFVI(rhs.iFVI),
                                    iFAI(rhs.iFAI)
      {
          if (cDeg <= cMBits)
              u.uAttrMask = rhs.u.uAttrMask;
          else
              JtWrapNew(u.pvbAttrMask, new BitVec(*rhs.u.pvbAttrMask));
      }
      ~FaceEnt() { if (cDeg > cMBits && u.pvbAttrMask) delete u.pvbAttrMask; }
      UInt16    cDeg;       // Face degree
      UInt16    cEmptyDeg;  // Empty degrees (opt for emptyFaceSlots())
      UInt16    cFaceAttrs;  // Number of face attributes
      UInt16    uFlags;     // User flags
      union {
          UInt64  uAttrMask;    // Degree-ring attr mask as a UInt64
          BitVec *pvbAttrMask;  // Degree-ring attr mask as a BitVec
      } u;
      Int32     iFVI; // Idx into _viFaceVtxIndices of cDeg incident vts
      Int32     iFAI; // Idx into _viFaceAttrIndices of cAttr attributes
  };

protected:
  // Subscripted by atom number, the entry contains the vtx valence and
  // points to the location in _viVtxFaceIndices of valence consecutive
  // integers that in turn contain the indices of the incident faces
  // in _vFaceRecs to the vtx.
  JtVec<VtxEnt>  _vVtxEnts;

  // Subscripted by unique vertex record number, the entry contains the
  // face degree and points to the location in _viFaceVtxIndices of
  // cDeg consecutive integers that in turn contain the indices of the
  // vertices indicent upon the face, in CCW order, in _vVtxRecs.
```

```
    JtVec<FaceEnt>   _vFaceEnts;

    // Combined storage for all vtxs.
    JtVeci          _viVtxFaceIndices;

    // Combined storage for all faces.
    JtVeci          _viFaceVtxIndices;

    // Combined storage for all face attribute record identifiers
    JtVeci          _viFaceAttrIndices;
};

bool
DualVFMesh::isValidVtx(Int32 iVtx) const
{
    bool bRet = JtFalse;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        bRet = (rFE.cVal != 0);
    }
    return bRet;
}

bool
DualVFMesh::newVtx(Int32 iVtx,
                   Int32 iValence,
                   UInt16 uFlags)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    if (rFE.cVal != iValence) {
        rFE.cVal   = iValence;
        rFE.uFlags = uFlags;
        rFE.iVFI   = _viVtxFaceIndices.length();
        _viVtxFaceIndices.verify(rFE.iVFI + iValence - 1);
        for (Int32 i = rFE.iVFI ; i < rFE.iVFI + iValence ; i++)
            _viVtxFaceIndices[i] = -1;
    }
    return true;
}

bool
DualVFMesh::setVtxGrp(Int32  iVtx,
                      Int32  iVGrp)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    rFE.iVGrp = iVGrp;
    return true;
}

bool
DualVFMesh::setVtxFlags(Int32  iVtx,
                        UInt16 uFlags)
{
    VtxEnt &rFE = _vVtxEnts[iVtx];
    rFE.uFlags = uFlags;
    return true;
}

Int32
DualVFMesh::vtxGrp   (Int32  iVtx) const
{
    Int32 u = -1;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        u = rFE.iVGrp;
    }
    return u;
}

UInt16
DualVFMesh::vtxFlags   (Int32  iVtx) const
```

```
{
    UInt16 u = 0;
    if (iVtx >= 0 && iVtx < _vVtxEnts.length()) {
        const VtxEnt &rFE = _vVtxEnts[iVtx];
        u = rFE.uFlags;
    }
    return u;
}


bool
DualVFMesh::isValidFace(Int32 iFace) const
{
    bool bRet = JtFalse;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rVE = _vFaceEnts[iFace];
        bRet = (rVE.cDeg != 0);
    }
    return bRet;
}

bool
DualVFMesh::newFace(Int32  iFace,
                    Int32  cDegree,
                    Int32  cFaceAttrs,
                    UInt64 uFaceAttrMask,
                    UInt16 uFlags)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    if (rVE.cDeg != cDegree) {
        rVE.cDeg        = cDegree;
        rVE.cEmptyDeg   = cDegree;
        rVE.cFaceAttrs   = cFaceAttrs;
        rVE.uFlags      = uFlags;
        rVE.u.uAttrMask = uFaceAttrMask;
        rVE.iFVI        = _viFaceVtxIndices.length();
        rVE.iFAI        = _viFaceAttrIndices.length();
        _viFaceVtxIndices.verify(rVE.iFVI + cDegree  - 1);
        if (cFaceAttrs > 0)
            _viFaceAttrIndices.verify(rVE.iFAI + cFaceAttrs - 1);
        for (Int32 i = rVE.iFVI ; i < rVE.iFVI + cDegree ; i++)
            _viFaceVtxIndices[i] = -1;
        for (Int32 i = rVE.iFAI ; i < rVE.iFAI + cFaceAttrs ; i++)
            _viFaceAttrIndices[i] = -1;
    }
    return true;
}

bool
DualVFMesh::newFace(Int32  iFace,
                    Int32  cDegree,
                    Int32  cFaceAttrs,
                    const BitVec *pvbFaceAttrMask,
                    UInt16 uFlags)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    if (rVE.cDeg != cDegree) {
        rVE.cDeg        = cDegree;
        rVE.cEmptyDeg   = cDegree;
        rVE.cFaceAttrs   = cFaceAttrs;
        rVE.uFlags      = uFlags;
        rVE.u.pvbAttrMask = new BitVec(*pvbFaceAttrMask);
        rVE.iFVI        = _viFaceVtxIndices.length();
        rVE.iFAI        = _viFaceAttrIndices.length();
        _viFaceVtxIndices.verify(rVE.iFVI + cDegree  - 1);
        if (cFaceAttrs > 0)
            _viFaceAttrIndices.verify(rVE.iFAI + cFaceAttrs - 1);
        for (Int32 i = rVE.iFVI ; i < rVE.iFVI + cDegree ; i++)
            _viFaceVtxIndices[i] = -1;
        for (Int32 i = rVE.iFAI ; i < rVE.iFAI + cFaceAttrs ; i++)
            _viFaceAttrIndices[i] = -1;
```

```
    }
    return true;
}

bool
DualVFMesh::setFaceFlags(Int32  iFace,
                         UInt16 uFlags)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    rVE.uFlags = uFlags;
    return true;
}

UInt16
DualVFMesh::faceFlags   (Int32  iFace) const
{
    UInt16 u = 0;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rVE = _vFaceEnts[iFace];
        u = rVE.uFlags;
    }
    return u;
}

bool
DualVFMesh::setFaceAttr(Int32  iFace,
                        Int32  iAttrSlot,
                        Int32  iFaceAttr)
{
    FaceEnt &rVE = _vFaceEnts[iFace];
    Int32 *paiFAI = _viFaceAttrIndices.ptr();
    paiFAI[rVE.iFAI + iAttrSlot] = iFaceAttr;
    return true;
}

Int32
DualVFMesh::faceAttr(Int32  iFace,
                     Int32  iAttrSlot) const
{
    Int32 u = 0;
    if (iFace >= 0 && iFace < _vFaceEnts.length()) {
        const FaceEnt &rVE = _vFaceEnts[iFace];
        if (iAttrSlot >= 0 && iAttrSlot < rVE.cDeg) {
            const Int32 *paiFAI = _viFaceAttrIndices.ptr();
            u = paiFAI[rVE.iFAI + iAttrSlot];
        }
    }
    return u;
}

// Attaches VF face iFace to VF vertex iVtx in the vertex's
// face slot iFaceSlot
bool
DualVFMesh::setVtxFace(Int32 iVtx,
                       Int32 iFaceSlot,
                       Int32 iFace)
{
    VtxEnt &rFE = _vVtxEntt &107..75 TpAi6he ve/9[995 ol                                    t

                                    Int32 iFaceSlot,
```

```
    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;
}
```

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 7.2.2.1.2.5 Topologically Compressed Rep Data). MeshCoderDriver manages the input data, the output VFMesh, and the MeshDecoder itself, providing a simple three-step API.

2.1 MeshCoderDriver class

```
// This class serves as a coordinating driver for mesh coding and decoding.
class MeshCoderDriver
{
  public:
    MeshCoderDriver ();

    // ========== Operations Interface ==========
    void      setInputData(const Veci   vviOutValSyms[/*8*/],
                           const Veci  &viOutDegSyms,
                           const Veci  &viOutFGrpSyms,
                           const Vecus &vuOutFaceFlags,
                           const Veclu  vvuOutAttrMasks[/*8*/],
                           const Vecu  &vuOutAttrMasksLrg,
                           const Veci  &viOutSplitVtxSyms,
                           const Veci  &viOutSplitPosSyms)
                           { /* Copy into 22 fields below */ }
    void      decode();
    VFMesh   *vfm() const { return _pOutVFM; }
```

```
    // ========== Utility Methods ==========
    Int32         _nextDegSymbol    (Int32 iCCntx);
    Int32         _nextValSymbol    ();
    Int32         _nextFGrpSymbol   ();
    UInt16        _nextVtxFlagSymbol();
    UInt64        _nextAttrMaskSymbol(Int32 iCCntx);     // <= 64-bit attrmask
    void          _nextAttrMaskSymbol(BitVec *iopvbAttrMask,
                                      Int32   cDegree);  // > 64 bit attrmask
    Int32         _nextSplitFaceSymbol();
    Int32         _nextSplitPosSymbol();
    Int32         _faceCntxt(Int32 iVtx, JtDualVFMesh *pVFM);

    // ========== Member Data ==========
  protected:
    SharedPtr<MeshCodec>     _pMC;      // The mesh coder or decoder being used
    SharedPtr<JtDualVFMesh>  _pOutVFM;  // Back-end VFMesh built by decoder
    SharedPtr<MeshDecoder>   _pMeshDecoder;

    // Coding symbols generated by encoding operation, auxiliary data such as
    // offsets, etc.
    Veci          _vviOutDegSyms[8];  // Face degree + SPLIT symbols for multiple contexts
    Veci          _viOutValSyms;      // Vtx valence symbols
    Veci          _viOutVGrpSyms;     // Vtx group of each encoded vtx
    Vecus         _vuOutVtxFlags;     // Vtx flags; parallel to _viOutValSyms.
    Veclu         _vvuOutAttrMasks[8];// Attribute bitmasks per face for multiple contexts.
                                      //  One per non-split entry in _viOutValSyms.
    Vecu          _vuOutAttrMasksLrg; // > 64-bit attrmasks
    Veci          _viOutSplitFaceSyms;// Split face offsets
    Veci          _viOutSplitPosSyms; // Split face vtx slots

    // The next symbol to be consumed by _next*Symbol()
    Int32         _iValReadPos[8];
    Int32         _iDegReadPos;
    Int32         _iVGrpReadPos;
    Int32         _iFFlagReadPos;
    Int32         _iAttrMaskReadPos[8];
    Int32         _iAttrMaskLrgReadPos;
    Int32         _iSplitFaceReadPos;
    Int32         _iSplitPosReadPos;
};

void MeshCoderDriver::decode()
{
    // Allocate a coder
    if (!_pMeshDecoder) {
        _pMeshDecoder = new MeshDecoder(this);
    }
    _pMC = _pMeshDecoder;
    _pMC->setTopoDualMeshCoder(this);

    // Reset the symbol counters
    for (Int32 i = 0 ; i < 8 ; i++) {
        _iValReadPos[i] = 0;
        _iAttrMaskReadPos[i] = 0;
    }
    _iDegReadPos = 0;
    _iVGrpReadPos = 0;
    _iFFlagReadPos = 0;
    _iAttrMaskLrgReadPos = 0;
    _iSplitFaceReadPos = 0;
    _iSplitPosReadPos = 0;

    // Run the decoder
    _pMC->run();

    // Assert that ALL symbols have been consumed
    for (Int32 i = 0 ; i < 8 ; i++) {
        Assert(_iValReadPos[i]         == _vviOutDegSyms[i].length());
        Assert(_iAttrMaskReadPos[i]    == _vvuOutAttrMasks[i].length());
    }
```

```
    Assert(_iDegReadPos         == _viOutValSyms.length());
    Assert(_iVGrpReadPos        == _viOutVGrpSyms.length());
    Assert(_iFFlagReadPos       == _vuOutVtxFlags.length());
    Assert(_iAttrMaskLrgReadPos == _vuOutAttrMasksLrg.length());
    Assert(_iSplitFaceReadPos    == _viOutSplitFaceSyms.length());
    Assert(_iSplitPosReadPos    == _viOutSplitPosSyms.length());

    // Set output VFMesh
    _pOutVFM = _pMC->vfm();
}

Int32 MeshCoderDriver::_nextDegSymbol  (Int32 iCCntx)
{
    Int32 eSym = -1;
    if (_iValReadPos[iCCntx] < _vviOutDegSyms[iCCntx].length())
        eSym = _vviOutDegSyms[iCCntx].value(_iValReadPos[iCCntx]++);
    return eSym;
}

Int32
MeshCoderDriver::_nextValSymbol  ()
{
    Int32 eSym = -1;
    if (_iDegReadPos < _viOutValSyms.length())
        eSym = _viOutValSyms.value(_iDegReadPos++);
    return eSym;
}

Int32 MeshCoderDriver::_nextFGrpSymbol()
{
    Int32 eSym = -1;
    if (_iVGrpReadPos < _viOutVGrpSyms.length())
        eSym = _viOutVGrpSyms.value(_iVGrpReadPos++);
    return eSym;
}

UInt16 MeshCoderDriver::_nextVtxFlagSymbol  ()
{
    UInt16 eSym = 0;
    if (_iFFlagReadPos < _vuOutVtxFlags.length())
        eSym = _vuOutVtxFlags.value(_iFFlagReadPos++);
    return eSym;
}

UInt64 MeshCoderDriver::_nextAttrMaskSymbol  (Int32 iCCntx)
{
    UInt64 eSym = 0;
    if (_iAttrMaskReadPos[iCCntx] < _vvuOutAttrMasks[iCCntx].length())
        eSym = _vvuOutAttrMasks[iCCntx].value(_iAttrMaskReadPos[iCCntx]++);
    return eSym;
}

void MeshCoderDriver::_nextAttrMaskSymbol(BitVec *iopvbAttrMask, Int32 cDegree)
{
    if (_iAttrMaskLrgReadPos < _vuOutAttrMasksLrg.length()) {
        iopvbAttrMask->setLength(cDegree);
        UInt32 *pu = iopvbAttrMask->ptr();
        Int32 nWords = (cDegree + BitVec::cWordBits - 1) >> BitVec::cBitsLog2;
        memcpy(pu, &_vuOutAttrMasksLrg.value(_iAttrMaskLrgReadPos), nWords * sizeof(UInt32));
        _iAttrMaskLrgReadPos += nWords;
    }
    else {
        iopvbAttrMask->setLength(0);
    }
}

Int32 MeshCoderDriver::_nextSplitFaceSymbol  ()
{
    Int32 eSym = -1;
    if (_iSplitFaceReadPos < _viOutSplitFaceSyms.length())
        eSym = _viOutSplitFaceSyms.value(_iSplitFaceReadPos++);
```

```
    return eSym;
}

Int32 MeshCoderDriver::_nextSplitPosSymbol  ()
{
    Int32 eSym = -1;
    if (_iSplitPosReadPos < _viOutSplitPosSyms.length())
        eSym = _viOutSplitPosSyms.value(_iSplitPosReadPos++);
    return eSym;
}

// Computes a "compression context" from 0 to 7 inclusive for

               ))
               Int32 Mes -0.Cntxt(JtTc 8.8iVtx, JtDualVF6641 *pVF6Syms.length())


adPos++);
                                                              i   f      (   _   i   S
   t                      J                                 9
                                                                      ) )    6  ;)++soPdaeRsoPtili ,xtVi i
```

```
    // ========== 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
```

```
            return;
    }

    // Walk CW from face slot 0, attempting to link in as many
    // already-reachable faces as possible until we reach one
    // that is inactive.
    Int32 ilast = i;
    vp = pDstVFM->face(iVtx, 0);
    jp = iVSlot;
    i = pDstVFM->valence(iVtx) - 1;
    while ((vn = pDstVFM->face(iVtx, i)) != -1) { // Forces "VF" in "prev" direction
        IncModN(jp, pDstVFM->degree(vp));
        iVtx2 = pDstVFM->vtx(vp, jp);
        if (iVtx2 == -1)
            break;
        jn = pDstVFM->findVtxSlot(vn, iVtx2);
        Assert(jn > -1);
        IncModN(jn, pDstVFM->degree(vn));
        addVtxToFace(iVtx, i, vn, jn);
        vp = vn;
        jp = jn;
        i--;
        if (i < ilast)
            return;
    }

    // Activate the remaining faces on iVtx that cannot be decuced from
    // the already-assembled topology in the destination VFMesh.
    for (; ilast <= i ; ilast++) {
        Int32 iFace = activateV(iVtx, ilast);
        JtDemandState(iFace >= -1);
    }
}

// This method connects vertex iVtx into the topology of
// _pDstVFM at and around iFace.  First, it connects iVtx
// to iFace's degree ring at position iVSlot.  Next, it
// will connect iVtx into the faces at the other ends of
// the shared edges between iVtx and the next vertices CS and
// CCW about iFace if necessary.
void MeshCodec::addVtxToFace (Int32 iVtx,  Int32 jFSlot,
                              Int32 iFace, Int32 iVSlot)
{
    Int32   iVSlotCW  = iVSlot,
            iVSlotCCW = iVSlot,
            fp, ip,
            fn, in;
    JtDualVFMesh *pDstVFM = _pDstVFM;
    IncModN(iVSlotCCW, pDstVFM->degree(iFace));
    DecModN(iVSlotCW,  pDstVFM->degree(iFace));

    // Connect iVtx to iFace/iVSlot
    JtRethrow(pDstVFM->setFaceVtx(iFace, iVSlot, iVtx));

    // Connect iVtx across the shared edge between iVtx and the vtx CW
    // from iVtx at iFace.  Connect iVtx into the face at the other
    // end of this edge if it is not already connected there.
    if ((fp = pDstVFM->vtx(iFace, iVSlotCW)) != -1) {
        ip = pDstVFM->findFaceSlot(fp, iFace);
        Int32 iVSlotCCW = jFSlot;
        IncModN(iVSlotCCW, pDstVFM->valence (iVtx));
        if (pDstVFM->face(iVtx, iVSlotCCW) == -1) {
            DecModN(ip, pDstVFM->valence(fp));
            pDstVFM->setVtxFace(iVtx, iVSlotCCW, pDstVFM->face(fp, ip));
        }
    }

    // Connect iVtx across the shared edge between iVtx and the vtx CCW
    // from iVtx at iFace.  Connect iVtx into the face at the other
    // end of this edge if it is not already connected there.
    if ((fn = pDstVFM->vtx(iFace, iVSlotCCW)) != -1) {
```

```
        in = pDstVFM->findFaceSlot(fn, iFace);
        Int32 iVSlotCW  = jFSlot;
        DecModN(iVSlotCW,  pDstVFM->valence (iVtx));
        if (pDstVFM->face(iVtx, iVSlotCW) == -1) {
            IncModN(in, pDstVFM->valence(fn));
            pDstVFM->setVtxFace(iVtx, iVSlotCW, pDstVFM->face(fn, in));
        }
    }
}

void MeshCodec::addActiveFace(Int32 iFace)
{
    JtRethrow(_viActiveFaces.pushBack(iFace));
}

// Returns a face from the active queue to be completed. This needn't be the
// one at the end of the queue, because the choice of the next active face
// can affect how many SPLIT symbols are produced.  This method employs a
// fairly simple scheme of searching the most recent 16 active faces for the
// fist one with the smallest number of incomplete slots in its degree ring.
Int32 MeshCodec::nextActiveFace()
{
    Int32 iFace = -1;
    // Search the 16 face record at the end of the
    // queue for the one with lowest remaining degree.
    while (_viActiveFaces.length() > 0 && _vbRemovedActiveFaces.test(_viActiveFaces.back()))
        _viActiveFaces.popBack();
    Int32 cLowestEmptyDegree = 9999999;
    Int32 i, iFace0, cEmptyDeg;
    const Int32 cWidth = 16;
    JtDualVFMesh *pDstVFM = _pDstVFM;
    for (i = _viActiveFaces.length() - 1 ;
         i >= ::jtmax(0, _viActiveFaces.length() - cWidth) ;
         i--)
    {
        iFace0 = _viActiveFaces[i];
        if (_vbRemovedActiveFaces.test(iFace0)) {
            _viActiveFaces.remove(i); // TOXIC: O(N^2)
            continue;
        }
        cEmptyDeg = pDstVFM->emptyFaceSlots(iFace0);
        if (cEmptyDeg < cLowestEmptyDegree) {
            cLowestEmptyDegree = cEmptyDeg;
            iFace = iFace0;
        }
    }

    // Return the selected active face
    return iFace;
}

// Removes iFace from the active face queue.
void MeshCodec::removeActiveFace(Int32 iFace)
{
    _vbRemovedActiveFaces.set(iFace);
}

// Searches the active face queue for iFace and returns
// its index position from the _end_ of the queue.  This is
// needed by the ioFace() method when encoding a SPLIT
// symbol.
Int32 MeshCodec::activeFaceOffset(Int32 iFace) const
{
    Int32 iOffset = -1;
    Int32 i, cLen = _viActiveFaces.length();
    const Int32 *paiActiveFaces = _viActiveFaces.ptr();
    for (i = cLen - 1 ; i >= 0 ; i--) {
        if (paiActiveFaces[i] == iFace) {
            // The offset is how far FROM THE END of the active
            // face list we found iFace.  This serves the make
            // the iOffset a much smaller number, which is better
```

```
            // for compression!
            iOffset = cLen - i;
            break;
        }
    }
    return iOffset;
}
```

2.3 MeshDecoder class

```
// This class implements the five abstract methods from
// MeshCodec to realize a mesh decoder.
class MeshDecoder : public MeshCodec {
  public:
    // ========== Housekeeping Interface ==========
    MeshDecoder (MeshCoderDriver *pTMC = NULL);
  protected:
    virtual ~MeshDecoder() {}

  private:
    // ========== Polymorphic I/O Interface ==========
    virtual Int32 ioVtxInit  ()                          ;
    virtual Int32 ioVtx      (Int32 iFace, Int32 iVSlot);
    virtual Int32 ioFace     (Int32 iVtx , Int32 jFSlot);
    virtual Int32 ioSplitFace(Int32 iVtx , Int32 jFSlot);
    virtual Int32 ioSplitPos (Int32 iVtx , Int32 jFSlot);
};

// Begins decoding a new connected mesh component by calling
// ioVtx() to read the next vertex from the symbol stream.
Int32 MeshDecoder::ioVtxInit()
{
    return ioVtx(-1, -1);
}

// Read a vertex valence symbol, vertex group number, and vertex
// flags from the input symbols stream.  Create a new vertex
// on _pDstVFM with this data, and return the new vertex number.
// It is this method's responsibility to detect the end of
// the input symbol stream by returning -1 when that happens.
Int32 MeshDecoder::ioVtx (Int32 /*iFace*/ , Int32 /*iVSlot*/)
{
    // Obtain a VERTEX VALENCE symbol
    Int32 eSym = _pTMC->_nextValSymbol();
    Int32 iVtxVal, iVtx = -1;
    if (eSym > -1) {
        // Create a new vtxt on the VFMesh
        iVtx = _pDstVFM->numVts();
        iVtxVal = eSym;
        _pDstVFM->newVtx     (iVtx, iVtxVal);
        _pDstVFM->setVtxGrp  (iVtx, _pTMC->_nextFGrpSymbol());
        _pDstVFM->setVtxFlags(iVtx, _pTMC->_nextVtxFlagSymbol());
    }

    return iVtx;
}

// Read a face degree symbol, and attribute mask bit
// vector, create a new DualVFMesh face, initialize the
// face attribute record numbers from a running counter,
// and return the new face number.  If the degree symbol
// read from the input symbol stream is 0, signify this by
// returning -1.
Int32
MeshDecoder::ioFace     (Int32 iVtx, Int32 /*jFSlot*/)
{
    // Obtain a FACE DEGREE symbol
    Int32 iCntxt = _pTMC->_faceCntxt(iVtx, _pDstVFM);
    Int32 eSym = _pTMC->_nextDegSymbol(iCntxt);
    Int32 cDeg, iFace = -1;
```

```
    if (eSym != 0) {
        // Create a new face on the VFMesh
        iFace = _pDstVFM->numFaces();
        cDeg = eSym;
        Int32 nFaceAttrs = 0;
        if (cDeg <= JtDualVFMesh::cMBits) {
            UInt64 uAttrMask = _pTMC->_nextAttrMaskSymbol(/*iCntxt*/::jtmin(7,::jtmax(0,cDeg-2)));
            for (UInt64 uMask = uAttrMask ; uMask ; nFaceAttrs += (uMask & 1), uMask >>= 1);
            _pDstVFM->newFace(iFace, cDeg, nFaceAttrs, uAttrMask);
        }
        else {
            BitVec vbAttrMask;
            _pTMC->_nextAttrMaskSymbol(&vbAttrMask, cDeg);
            for (Int32 i = 0 ; i < cDeg ; i++) {
                if (vbAttrMask.test(i))
                    nFaceAttrs++;
            }
            _pDstVFM->newFace(iFace, cDeg, nFaceAttrs, &vbAttrMask, 0);
        }

        // Error check for a corrupt degree or attrmask
        if (nFaceAttrs > cDeg) {
            Assert (nFaceAttrs <= cDeg);
            return -2;
        }

        // Set up the face attributes
        for (Int32 iAttrSlot = 0 ; iAttrSlot < nFaceAttrs ; iAttrSlot++) {
            _pDstVFM->setFaceAttr(iFace, iAttrSlot, _iFaceAttrCtr++);
        }
    }

}

// Consumes a split offset symbol from the SPLIT offset
// symbol stream, and determines the face number referenced
// by the offset.  Returns the referenced face number.
Int32 MeshDecoder::ioSplitFace(Int32 /*iVtx*/, Int32 /*jFSlot*/)
{
    // Obtain a SPLITFACE symbol
    Int32 eSym = _pTMC->_nextSplitFaceSymbol();
    Assert(eSym >= -1);
    Int32 iOffset = -1, iFace = -1;
    if (eSym > -1) {
        // Use the offset to index into the active face queue
        // to determine the actual face number.
        iOffset = eSym;
        Int32 cLen = _viActiveFaces.length();
        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;
}
```

Appendix F: Parasolid XT Format Reference

November 2008

Table of Contents

Introduction to the Parasolid XT Format

This Parasolid® Transmit File Format manual describes the formats in which Parasolid represents model information in external files. Parasolid is a geometric modeling kernel that can represent wireframe, surface, solid, cellular and general non-manifold models.

Parasolid stores topological and geometric information defining the shape of models in transmit files. These files have a published format so that applications can have access to Parasolid models without necessarily using the Parasolid kernel.

This manual documents the Parasolid transmit file format. This format will change in subsequent Parasolid releases at which time this manual will be updated. As new versions of Parasolid can read and write older transmit file formats these changes will not invalidate applications written based on the information herein.

Types of File Documented

There are a number of different interface routines in Parasolid for writing transmit files. Each of these routines can write slightly different combinations of Parasolid data, the ones that are documented herein are:

- Individual components (or assemblies) written using SAVMOD
- Individual components written using PK_PART_transmit
- Lists of components written using PK_PART_transmit
- Partitions written using PK_PARTITION_transmit

The basic format used to write data in all the above cases is identical; there are a small number of node types that are specific to each of the above file types.

Text and Binary Formats

Parasolid can encode the data it writes out in four different formats:

1. Text (usually ASCII)
2. Neutral binary
3. Bare binary (this is not recommended)
4. Typed binary

In text format all the data is written out as human readable text, they have the advantage that they are readable but they also have a number of disadvantages. They are relatively slow to read and write, converting to and from text forms of real numbers introduces rounding errors that can (in extreme cases) cause problems and finally there are limitations when dealing with multi-byte character sets. Carriage return or line feed characters can appear anywhere in a text transmit file but other unexpected non-printing characters will cause Parasolid to reject the file as corrupt.

Neutral binary is a machine independent binary format.

Bare binary is a machine dependent binary format. It is not a recommended format since the machine type which wrote it must be known before it can be interpreted.

Typed binary is a machine dependent binary format, but it has a machine independent prefix describing the machine type that wrote it and so can be read on all machine types.

Standard File Names and Extensions

Due to changing operation system restrictions on file names over the years Parasolid has used several different file extensions to denote file contents. The recommended set of file extensions is:

- .X_T and .X_B for part files, .P_T and .P_B for partition files.

Extensions that have been used in the past are:

- xmt_txt, xmp_txt - text format files on VMS or Unix platforms
- xmt_bin, xmp_bin - binary format files on VMS or Unix platforms

Logical Layout

The logical layout of a Parasolid transmit file is:

- A human-oriented text header.
  - The initial text header is read and written by applications' Frustrums and is not accessible to Parasolid. Its detailed format is described in the section `Physical layout'.
- A short flag sequence describing the file format, followed by modeller identification information and user field size.
  - The various flag sequences (mixtures of text and numbers) are documented under `Physical layout'; the content of the modeller identification information is:

  the modeller version used to write the file, as a text string of the form:

  : TRANSMIT FILE created by modeller version 1200123

  This information is used by routines such as PK_PART_ask_kernel_version.

  the schema version describing the field sequences of the part nodes as a text string of the form:

  SCH_1200123_12006

  This example denotes a file written by Parasolid V12.0.123 using schema number 12006: there will be a corresponding file sch_12006 in the Parasolid schema distribution.

  Note that applications writing XT files should use version 1200000 and schema number 12006.
  - The user field size is a simple integer.
- The objects (known as ‘nodes’) in the file in an unordered sequence, followed by a terminator.
  - Every node in the file is assigned an integer index from 1 upwards (some indices may not be used). Pointer fields are output as these indices, or as zero for a null pointer.
  - Each node entry begins with the node type. If the node is of variable length (see below), this is followed by the length of the variable field. The index of the node is then output, followed by the fields of the node. If the file contains user fields, and the node is visible at the PK interface, then the fields are followed by the user field, in integers.
  - The terminator which follows the sequence of nodes is a two-byte integer with value 1, followed by an index with value 0. The index is output as ‘0’ in a text file, and as a 2-byte integer with value 1 in a binary file.
  - The node with index 1 is the root node of the transmit file as follows:
  -

Contents of file	**Type of root node**
Body	BODY
Assembly	ASSEMBLY
Array of parts	POINTER_LIS_BLOCK
Partition	WORLD

Schema

Parasolid permanent structures are defined in a special language akin to C which generates the appropriate files for a C compiler, the runtime information used by Parasolid, along with a schema file used during transmit and receive. The schema file for version 12.0 is named sch_12006 and is distributed with Parasolid. It is not necessary to have a copy of this file to understand the XT format.

For each node type, the schema file has a node specifier of the form

<nodetype> <nodename>; <description>; <transmit 1/0> <no. of fields> <variable 1/0>

e.g.

29 POINT; Point; 1 6 0

This is followed by a list of field specifiers which say what fields, and in what order, occur in the transmit file.

Field specifiers have the format:

<fieldname>; <type>; <transmit 1/0> <node class> <n_elements>

e.g.

owner; p; 1 1011 1

Nodes and fields with a transmit flag of zero are ephemeral information not written to a transmit file. Only pointer fields have non-zero node class, in which case it specifies the set of node types to which this field is allowed to point. The element count is interpreted as follows:

0	a scalar, a single value
1	a variable length field (see below)
n > 1	an array of n values

Note that in the schema file, fins are referred to as ‘halfedges’, and groups are referred to as ‘features’. These are internal names not used elsewhere in this document.

Embedded schemas

When reading a part, partition, or delta, Parasolid converts any data that it encounters from older versions of Parasolid to the current format using a mixture of automatic table

Fields that are included are referred to as **effective fields**, and are either transmittable (**`xmt_code`** == 1) or have variable-length (**`n_elts`** == 1)

Physical layout

Most of the data are composed of integers, logical flags, and strings, but are of restricted ranges and so transmitted specially in binary format. The binary representation is given in **bold** type, such as "integer (**byte**)". This is relevant to applications that attempt to read or write Parasolid data directly. Two important elements are

- **short string**s

  These are transmitted as an integer length (**byte**) followed by the characters (without trailing zero).

- **positive integer**s

  These are transmitted similarly to the pointer indices which link individual objects together, i.e., small values 0..32766 are transmitted as a single **short** integer, larger ones encoded into two.

XT format

Presence of the new format is indicated by a change to the standard header: the archive name is extended by the number of the base schema, e.g., **`SCH_1400068_14000_13006`**, and then the maximum number of node types is inserted (**short**).

Transmission then continues as normal, except that when transmitting the first node of any particular type, extra information is inserted between the nodetype and the variable-length, index data as follows:

- The arrays of effective fields in the base schema node and the current schema node are assembled.
- If the nodetype does not exist in the base schema then it is output as follows:
  - number of fields (**byte**)
  - name and description (**short string**s)
  - fields one by one as

`name`	short string	
`ptr_class`	Short	
`n_elts`	Positive integer	
`type`	short string	The field type. Allowed values are described in "Field types", below. Omitted if **`ptr_class`** non-zero
`xmt_code`	logical (byte)	Omitted for fixed-length (**`n_elts`** != 1)

- If the two arrays match (equal length and all fields match in **`name`**, **`xmt_code`**, **`ptr_class`**, **`n_elts`** and **`type`**) then output the flag value 255 (**byte** 0xff).
- If the two arrays do not match, output the number of effective fields in the current schema (**byte**), and an edit sequence as follows.
  - Initialize pointers to the first base field and first current field, then while there are still unprocessed base and current fields, output a sequence of Copy, Delete and Insert instructions
    - If the base field matches the current field, output 'C' (**char**) to indicate an unchanged (Copied) field and advance to the next base and current fields;
    - If the base field does not match any unprocessed current field, output 'D' (**char**) to indicate a Deleted field and advance to the next base field;
    - Otherwise, output 'I' (**char**) to indicate an Inserted field, followed by the current field in the above format, and advance to the next current field.
  - If there are any unprocessed current fields, then output an Append sequence, each instruction being 'A' (**char**) followed by the field.
- Finally, output 'Z' (**char**) to signal the end.

Space compression

For text data in transmit formats PK_transmit_format_text_c and PK_transmit_format_xml_c, a new escape sequence is defined: the 2-character sequence **\9** denotes a sequence of nine spaces. At V14, this applies to attribute definition names, field names, and attribute strings.

Field types

The XT format is not itself a binary protocol, and so does not define data sizes; the only requirement is that a runtime implementation has sufficient room for the information. The available implementations run with 8bit ASCII characters, 8bit unsigned bytes (0..255), 16bit short integers (0..65535 or -32768..32767), 32bit integers (0..4G-1, -2G..2G-1) and IEEE reals. The implementation used in a given binary file is specified by the "PS<code>" at the start of the file. See the chapter on “Physical Layout” for more information.

The full list of field types used in transmit files is as follows:

u unsigned byte 0-255

c char

l unsigned byte 0-1 (i.e. logical)

```
typedef char logical;
```

n short int

w unicode character, output as a short int

d int

p pointer-index

Small indices (less than 32767) are treated specially in binary files to save space.
See the section below on binary format.

f double

i These correspond to a region of the real line:

typedef struct { double low, high; }interval;

v array [3] of doubles

These correspond to a 3-space position or direction:

typedef struct { double x,y,z; } vector;

b array [6] of doubles

These correspond to a 3-spce region:

typedef struct { interval x,y,z; } box;

Note that the ordering is not the same as presented at Parasolid's external PK or KI interfaces.

h array [3] of doubles

These represent points of intersection between two surfaces; only the position vector is written to a transmit file, as Parasolid will recalculate other data as required. The structure is documented further in the section on intersection curves.

Point

As an example, consider a POINT; its formal description is

```
struct POINT_s          // Point
    {
    int                             node_id;                // $d
    union  ATTRIB_GROUP_u           attributes_groups;      // $p
    union  POINT_OWNER_u            owner;                  // $p
    struct POINT_s                  *next;                  // $p
    struct POINT_s                  *previous;              // $p
    vector                          pvec;                   // $v
    };
typedef struct POINT_s          *POINT;
```

Its corresponding schema file entry is

29 POINT; Point; 1 6 0

node_id; d; 1 0 0

attributes_groups; p; 1 1019 0

owner; p; 1 1011 0

next; p; 1 29 0

previous; p; 1 29 0

pvec; v; 1 0 0

Pointer classes

In the above example, the attributes_groups field must be of class ATTRIB_GROUP_cl, the owner must be of class POINT_OWNER_cl, and the next and previous fields must refer to POINTs. A full list of node types and node classes is given at the end of the document.

Each node class corresponds to a union of pointers given in the Schema Definition section.

Variable-length nodes

Variable-length nodes differ from fixed-length nodes in that their last field is of variable length, i.e. different nodes of the same type may have different lengths. In the schema the length is notionally given as 1, e.g.

```
struct REAL_VALUES_s          // Real values
    {
    Double                        values[1];                    // $f[]
    };
```

Its schema file entry would be

83 REAL_VALUES; Real values; 1 1 1

values; f; 1 0 1

The number of entries in each such node is indicated by an integer in the transmit file between its nodetype and index, so an example might be

```
    83 3 15 1 2 3
```

Unresolved indices

In some cases a node will contain an index field which does not correspond to a node in the transmit file, in this case the index is to be interpreted as zero.

Simple example

Here is a reformatted text example of a sheet circle with a color attribute on its single edge:

```
**ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz**********
**PARASOLID !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~0123456789**********
**PART1;MC=osf65;MC_MODEL=alpha;MC_ID=sdlosf6;OS=OSF1;OS_RELEASE=V4.0;FRU=sdl_parasolid
_test_osf64;APPL=unknown;SITE=sdl-cambridge-
u.k.;USER=davidj;FORMAT=text;GUISE=transmit;DATE=29-mar-2000;
**PART2;SCH=SCH_1200000_12006;USFLD_SIZE=0;
**PART3;
```

```
**END_OF_HEADER*****************************************************
T51 : TRANSMIT FILE created by modeller version 120000017 SCH_1200000_120060
12 1 12 0 2 0 0 0 0 1e3 1e-8 0 0 0 1 0 3 1 3 4 5 0 6 7 0          body
 70 2 0 1 0 0 4 1 20 8 8 8 1 T                                    list
13 3 3 0 1 0 9 0 0 6 9                                            shell
 50 4 11 0 9 0 0 0 +0 0 0 0 0 1 1 0 0                             plane
 31 5 10 0 7 0 0 0 +0 0 0 0 0 1 1 0 0 1                           circle
 19 6 5 0 1 0 0 3 V                                               region
16 7 6 0 ?10 0 0 5 0 0 1                                          edge
 17 10 0 11 10 10 0 12 7 0 0 +                                    fin
15 11 7 0 10 9 0                                                  loop
 17 12 0 0 0 0 0 10 7 0 0 -                                       fin (dummy)
14 9 2 13 ?0 0 11 3 4 +0 0 0 0 3                                  face
 81 1 13 12 14 9 0 0 0 0 15                                       attribute (variable 1)
 80 1 14 0 16 8001 0 0 0 0 3 5 0 0 FFFFTFTFFFFFF2                 attrib_def (variable 1)
83 3 15 1 2 3                                                     real_values (variable 3)
 79 15 16 SDL/TYSA_COLOUR                                         att_def_id (variable 15)
74 20 8 1 0 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  pointer_lis_block
 1 0                                                              terminator
```

Note that the tolerance fields of the face and edge are unset, and represented as ‘?’ in the text transmit file and that the annotations in the column ‘body’ to ‘terminator’ give the node type of each line and are not part of the actual file. If the above file had no trailing spaces, it would be a valid XT file (the leading spaces on some of the lines are necessary).

Physical Layout

Parasolid transmit files have two headers:

- a textual introduction containing human-directed information about the part, written by the Frustrum and not accessible to Parasolid, and
- an internal prefix to the part data, describing to Parasolid the format of the part data and thus not seen explicitly by an application's Frustrum.

Common header

The Parasolid common header recommended to Frustrum writers consists of:

- A preamble containing all characters in the ASCII printing set. This is used by the KID Frustrum to detect obvious network corruption, but could be used to attempt to translate a text file from one character set to another.
- Part 1 data: a sequence of keyword-value pairs, separated by semicolons, of possibly interesting information. All are optional.

```
MC              =       vax, hppa, sparc, ...
                                // make of computer
        MC_MODEL    =       4090, 9000/780, sun4m, ...
                                //  model of computer
        MC_ID       =       ...
                                //  unique machine identifier
        OS          =       vms, HP-UX, SunOS, ...
                                //  name of operating system
OS_RELEASE =    V6.2, B.10.20, 5.5.1, ...
                                //  version of operating system
FRU     =   sdl_parasolid_test_vax,
              mdc_ugii_v7.0_djl_can_vrh, ...
//  frustrum supplier and implementation name
        APPL        =       kid, unigraphics, ...
//  application which is using Parasolid
        SITE        =       ...
//  site at which application is running
        USER        =       ...
                                //  login name of user
        FORMAT      =       binary, text, applio
                                //  format of file
        GUISE       =       transmit, transmit_partition
```

<value> consists of 1 or more ASCII printing characters (except space)

Escape sequences provide a way of being able to use the full (7 bit) set of ASCII printing characters and the new line character within keyword values. Certain characters must be escaped if they are to appear in a keyword value:

• Character	Escape sequence
newline	^n
space	^_
semicolon	^;
uparrow	^^

The two character escape sequences may be split by a new line character as they are written to file. They must not cause any output lines to be longer than 80 characters.

Only those characters which belong to the ASCII (7 bit) printing sequence, plus the new line character, can be included as part of a keyword value.

Text

Parasolid has no knowledge of how files are stored. On writing, Parasolid produces an internal bytestream which is then split into roughly 80-character records separated by newline characters ('\n'). The newlines are not significant.

As operating systems vary in their treatment of text data, on reading all newline and carriage return characters ('\r') are ignored, along with any trailing spaces added to the records. However, leading spaces are not ignored, and the file must not contain adjacent space characters not at the end of a record.

Text XT files written by version 12.1 and later versions use escape sequences to output the following characters, except for '\n' at the end of each line:

null "\0"

carriage return "\n"

line feed "\r"

backslash "\\"

These characters are not escaped by versions 12.0 and earlier.

The flag sequence is the character ‘T’. This is followed by the length of the modeler version, separated by a space from the characters of the modeler version, similarly the schema version, finally the userfield size. For example:

T

51 : TRANSMIT FILE created by modeller version 1200000

17 SCH_1200000_12006

0

NB: because of ignored layout, what Parasolid would read is

T51 : TRANSMIT FILE created by modeller version 120000017 SCH_1200000_120060

For partition files, the modeller version string would be given as

63 : TRANSMIT FILE (partition) created by modeller version 1200000

All numbers are followed by a single space to separate them from the next entry. Fields of type c and l are not followed by a space.

Logical values (0,1) are represented as characters F,T.

There are two special numeric values (-32764 for integral values, -3.14158e13 for floating point) which are used inside Parasolid to mark an ‘unset’ or ‘null’ value, and they are represented in a text transmit file as the question mark ‘?’. If a vector has one component null, then all three components must be null, and it will be output in a text file as a single ‘?’.

Binary

There are three types of binary file: `bare' binary, typed binary, and neutral binary. They are distinguished by a short flag sequence at the beginning of the file. In all cases, the flag sequence is followed by the length of the modeller version as a 2-byte integer, the characters of the modeller version, the length of the schema version as a 4-byte integer, the characters of the schema version, and finally the userfield size as a 4-byte integer.

As with text files, there are two special numeric values (-32764 for integral values, -3.14158e13 for floating point) which are used inside Parasolid to mark an ‘unset’ or ‘null’ value, and they are represented in a text transmit file as the question mark ‘?’.

bare binary

In bare binary, data is represented in the natural format of the machine which wrote the data. The flag sequence is the single character 'B' (for ASCII machines, '\102'). The data must be read on a machine with the same natural format with respect to character set, endianness and floating point format.

typed binary

In typed binary, data is represented in the natural format of the machine that wrote the data. The flag sequence is the 4-byte sequence “PS” followed by a zero byte and a one byte, i.e., ‘P’ ‘S’ ‘\0’ ‘\1’, followed by a 3-byte sequence of machine description.

	Byte order	**Double representation**	**Character representation**
0	Big-endian	IEEE	ASCII
1	Little-endian	VAX D-float	EBCDIC

neutral binary

In neutral binary, data is represented in big-endian format, with IEEE floating point numbers and ASCII characters. The flag sequence is the 4-byte sequence "PS" followed by two zero bytes, i.e., 'P' 'S' '\0' '\0'. At Parasolid V9, the initial letters are ASCII, thus '\120' '\123'.

The nodetype at the start of a node is a 2-byte integer, the variable length which may follow it is a 4-byte integer.

Logical values (0,1) are represented as themselves in 1 byte.

Small pointer indices (in the range 0-32766) are implemented as a 2-byte integer, larger indices are represented as a pair, thus:

```
if (index < 32767)
        {                               // case: small index
        op_short( index + 1 );          // offset so is > 0
        }
else
        {                               // case: big index
        op_short( -(index % 32767 + 1) );   // remainder: add 1 so > 0
        op_short( index / 32767 );      // nonzero quotient
        }
```

where op_short outputs a 2-byte integer.

The inverse is performed on reading:

```
short q = 0, r;
ip_short( &r );
if (r < 0)
        {
        ip_short( &q );
        r = -r;
        }
index = q * 32767 + r - 1;
```

where ip_short reads a 2-byte integer.

Model Structure

Topology

This section describes the Parasolid Topology model, it gives an overview of how the nodes in an XT file are joined together. In this section the word ‘entity’ means a node which is visible to a PK application – a table of which nodes are visible at the PK interface appears in the section `Node Types'.

The topological representation allows for:

- Non-manifold solids
- Solids with internal partitions
- Bodies of mixed dimension (i.e. with wire, sheet, and solid `bits')
- Pure wire-frame bodies
- Disconnected bodies

Each entity is described, and its properties and links to other entities given.

General points

In this section a set is called **finite** if it can be enclosed in a ball of finite radius - not that it has a finite number of members.

A set of points in 3-dimensional space is called **open** if it does not contain its boundary.

Back-pointers, next and previous pointers in a chain, and derived pointers are not described explicitly here. For information on this see the following description of the schema-level model.

Entity definitions

Assembly

An assembly is a collection of instances of bodies or assemblies. It may also contain construction geometry. An assembly has the following fields:

- A set of instances.
- A set of geometry (surfaces, curves and points).

Instance

An instance is a reference to a body or an assembly, with an optional transform:

- Body or assembly.
- Transform. If null, the identity transform is assumed.

Body

A body is a collection of faces, edges and vertices, together with the 3-dimensional connected regions into which space is divided by these entities. Each region is either **solid** or **void** (indicating whether it represents material or not).

The point-set represented by the body is the disjoint union of the point-sets represented by its solid regions, faces, edges, and vertices. This point-set need not be connected, but it must be finite.

A body has the following fields:

- A set of regions.

  A body has one or more regions. These, together with their boundaries, make up the whole of 3-space, and do not overlap, except at their boundaries. One region in the body is distinguished as the exterior region, which must be infinite; all other regions in the body must be finite.

- A set of geometry (surfaces, curve and/or points).
- A body-type. This may be wire, sheet, solid or general.

Region

A region is an open connected subset of 3-dimensional space whose boundary is a collection of vertices, edges, and oriented faces.

Regions are either solid or void, and they may be non-manifold. A solid region contributes to the point-set of its owning body; a void region does not (although its boundary will).

Two regions may share a face, one on each side.

A region may be infinite, but a body must have exactly one infinite region. The infinite region of a body must be void.

A region has the following fields:

- A logical indicating whether the region is solid.
- A set of shells. The positive shell of a region, if it has one, is not distinguished.

The shells of a region do not overlap or share faces, edges or vertices.

A region may have no shells, in which case it represents all space (and will be the only region in its body, which will have no faces, edges or vertices).

Shell

A shell is a connected component of the boundary of a region. As such it will be defined by a collection of faces, each used by the shell on one `side', or on both sides; and some edges and vertices.

A shell has the following fields:

- A set of (face, logical) pairs.

  Each pair represents one side of a face (where true indicates the front of the face, i.e. the side towards which the face normal points), and means that the region to which the shell belongs lies on that side of the face. The same face may appear twice in the shell (once with each orientation), in which case the face is a 2-dimensional cut subtracted from the region which owns the shell.

- A set of wireframe edges.

  Edges are called **wireframe** if they do not bound any faces, and so represent 1-dimensional cuts in the shell's region. These edges are not shared by other shells.

- A vertex.

  This is only non-null if the shell is an **acorn** shell, i.e. it represents a 0-dimensional hole in its region, and has one vertex, no edges and no faces.

A shell must contain at least one vertex, edge, or face.

Face

A face is an open finite connected subset of a surface, whose boundary is a collection of edges and vertices. It is the 2-dimensional analogy of a region.

A face has the following fields:

- A set of loops. A face may have zero loops (e.g. a full spherical face), or any number.
- Surface. This may be null, and may be used by other faces.
- Sense. This logical indicates whether the normal to the face is aligned with or opposed to that of the surface.

Loop

A loop is a connected component of the boundary of a face. It is the 2-dimensional analogy of a shell. As such it will be defined by a collection of fins and a collection of vertices.

A loop has the following fields:

- An ordered ring of fins.

  Each fin represents the oriented use of an edge by a loop. The sense of the fin indicates whether the loop direction and the edge direction agree or disagree. A loop may not contain the same edge more than once in each direction.

  The ordering of the fins represents the way in which their owning edges are connected to each other via common vertices in the loop (i.e. nose to tail, taking the sense of each fin into account).

  The loop direction is such that the face is locally on the left of the loop, as seen from above the face and looking in the direction of the loop.

- A vertex.

  This is only non-null if the loop is an **isolated** loop, i.e. has no fins and represents a 0-dimensional hole in the face.

Consequently, a loop must consist either of:

- A single fin whose owning **ring** edge has no vertices, or
- At least one fin and at least one vertex, or
- A single vertex.

Fin

A fin represents the oriented use of an edge by a loop.

A fin has the following fields:

- A logical **sense** indicating whether the fin's orientation (and thus the orientation of its owning loop) is the same as that of its owning edge, or different.
- A curve. This is only non-null if the fin’s edge is tolerant, in which case every fin of that edge will reference a trimmed SP-curve. The underlying surface of the SP-curve must be the same as that of the corresponding face. The curve must not deviate by more than the edge tolerance from curves on other fins of the edge, and its ends must be within vertex tolerance of the corresponding vertices.

Note that fins are referred to as ‘halfedges’ in the Schema file.

Edge

An edge is an open finite connected subset of a curve; its boundary is a collection of zero, one or two vertices. It is the 1-dimensional analogy of a region.

An edge has the following fields:

- Start vertex.
- End vertex. If one vertex is null, then so is the other; the edge will then be called a **ring** edge.
- An ordered ring of distinct fins.

  The ordering of the fins represents the spatial ordering of their owning faces about the edge (with a right-hand screw rule, i.e. looking in the direction of the edge the fin ordering is clockwise). The edge may have zero or any number of fins; if it has none, it is called a **wireframe** edge.
- A curve. This will be null if the edge has a tolerance. Otherwise, the vertices must lie within vertex tolerance of this curve, and if it is a Trimmed Curve, they must lie within vertex tolerance of the corresponding ends of the curve. The curve must also lie in the surfaces of the faces of the edge, to within modeller resolution.
- Sense. This logical indicates whether the direction of the edge (start to end) is the same as that of the curve.
- A tolerance. If this is null-double, the edge is **accurate** and is regarded as having a tolerance of half the modeller linear resolution, otherwise the edge is called **tolerant**.

Vertex

A vertex represents a point in space. It is the 0-dimensional analogy of a region.

A vertex has the following fields:

- A geometric point.
- A tolerance. If this is null-double, the vertex is **accurate** and is regarded as having a tolerance of half the modeller linear resolution.

Attributes

An attribute is an entity which contains data, and which can be attached to any other entity except attributes, fins, lists, transforms or attribute definitions. An attribute has the following fields:

- Definition. An attribute definition is an entity which defines the number and type of the data fields in a specific type of attribute, which entities may have such an attribute attached, and what happens to the attribute when its owning entity is changed. An XT document must not contain duplicate attribute definitions. Each attribute of a given type should reference the same instance of the attribute definition for that type. It is incorrect, for example, to create a copy of an attribute definition for each instance of the attribute of that type. Only those attribute definitions referenced by attributes in the part occur in the transmit file.
- Owner.
- Fields. These are data fields consisting of one or more integers, doubles, vectors etc.

There are a number of system attribute definitions which Parasolid creates on startup. These are documented in the section `System Attribute Definitions'. Parasolid applications can create user attribute definitions during a Parasolid session. These are transmitted along with any attributes that use them.

Groups

A group is a collection of entities in the same part. Groups in assemblies may contain instances, surfaces, curves and points. Groups in bodies may contain regions, faces, edges, vertices, surfaces, curves and points. Groups have

- Owning part.
- A set of member entities.
- Type. The type of the group specifies the allowed type of its members, e.g. a ‘face’ group in a body may only contain faces, whereas a ‘mixed’ group may have any valid members.

Node-ids

All entities in a part, other than fins, have a non-zero integer node-id which is unique within a part. This is intended to enable the entity to be identified within a transmit file.

Entity matrix

Thus the relations between entities can be represented in matrix form as follows. The numbers represent the number of distinct entities connected (either directly or indirectly) to the given one.

	Body	**Region**	**Shell**	**Face**	**Loop**	**Fin**	**Edge**	**Vertex**
Body	-	>0	any	any	any	any	any	any
Region	1	-	any	any	any	any	any	any
Shell	1	1	-	any	any	any	any	any
Face	1	1-2	1-2	-	any	any	any	any
Loop	1	1-2	1-2	1	-	any	any	any
Fin	1	1-2	1-2	1	1	-	1	0-2
Edge	1	any	any	any	any	any	-	0-2
Vertex	1	any	any	any	any	any	any	-

Representation of manifold bodies

Body types

Parasolid bodies have a field body_type which takes values from an enumeration indicating whether the body is

- **solid**, representing a manifold 3-dimensional volume, possibly with internal voids. It need not be connected.
- **sheet**, representing a 2-dimensional subset of 3-space which is either manifold or manifold with boundary (certain cases are not strictly manifold – see below for details). It need not be connected.
- **wire**, representing a 1-dimensional subset of 3-space which is either manifold or manifold with boundary, and which need not be connected. An **acorn** body, which represents a single 0-dimensional point in space, also has body-type wire.
- **general** - none of the above.

A general body is not necessarily non-manifold, but at the same time it is not constrained to be manifold, connected, or of a particular dimensionality (indeed, it may be of mixed dimensionality).

Restrictions on entity relationships for manifold body types

Solid, sheet, and wire bodies are best regarded as special cases of the topological model; for convenience we call them the manifold body types (although as stated above, a general body may also be manifold).

In particular, bodies of these manifold types must obey the following constraints:

- An acorn body must consist of a single void region with a single shell consisting of a single vertex.
- A wire body must consist of a single void region, with one or more shells, consisting of one or more wireframe edges and zero or more vertices (and no faces). Every vertex in the body must be used by exactly one or two of the edges (so, in particular, there are no acorn vertices).

  So each connected component will be either: closed, where every vertex has exactly two edges; or open, where all but two vertices have exactly two edges each, and the

  A wire is called open if all its components are open, and closed if all its components are closed.
- Solid and sheet bodies must each contain at least one face; they may not contain any wireframe edges or acorn vertices.
- A solid body must consist of at least two regions; at least one of its regions must be solid. Every face in a solid body must have a solid region on its negative side and a void region on its positive side (in other words, every face forms part of the boundary of the solid, and the face normals always point away from the solid).
- Every edge in a solid body must have exactly two fins, which will have opposite senses. Every vertex in a solid body must either belong to a single isolated loop, or belong to one or more edges; in the latter case, the faces which use those edges must form a single edgewise-connected set (when considering only connections via the edges which meet at the vertex).

  These constraints ensure that the solid is manifold.
- All the regions of a sheet body must be void. It is known as an open sheet if it has one region, and a closed sheet if it has no boundary.
- Every edge in a sheet body must have exactly one or two fins; if it has two, these must have opposite senses. In a closed sheet body, all the edges will have exactly two fins. Every vertex in a sheet body must either belong to a single isolated loop, or belong to one or more edges; in the latter case, the faces which use those edges must either form a single edgewise-connected set where all the edges involved have exactly two fins, or any number of edgewise-connected sets, each of which must involve exactly two edges with one fin each (again, considering only connections via the edges which meet at the vertex).

  Note that, although the constraints on edges and vertices in a sheet body are very similar to those which apply to a solid, in this case they do not guarantee that the body will be manifold; indeed, the rather complicated rules about vertices in an open sheet body specifically allow bodies which are non-manifold (such as a body consisting of two square faces which share a single corner vertex, say).

Schema Definition

Underlying types

```
union CURVE_OWNER_u
        {
        struct EDGE_s                   *edge;
        struct FIN_s                    *fin;
        struct BODY_s                   *body;
        struct ASSEMBLY_s               *assembly;
        struct WORLD_s                  *world;
        };

union SURFACE_OWNER_u
        {
        struct FACE_s                   *face;
        struct BODY_s                   *body;
        struct ASSEMBLY_s               *assembly;
        struct WORLD_s                  *world;
        };

union ATTRIB_GROUP_u
        {
        struct ATTRIBUTE_s              *attribute;
        struct GROUP_s                  *group;
        struct                          *member_of_group;
        MEMBER_OF_GROUP_s
        };
typedef union ATTRIB_GROUP_u  ATTRIB_GROUP;
```

Geometry

```
union CURVE_u
        {
        struct LINE_s                   *line;
```

```
        struct CIRCLE_s                 *circle;
        struct ELLIPSE_s                *ellipse;
        struct INTERSECTION_s           *intersection;
        struct TRIMMED_CURVE_s          *trimmed_curve;
        struct PE_CURVE_s               *pe_curve;
        struct B_CURVE_s                *b_curve;
        struct SP_CURVE_s               *sp_curve;
        };
typedef union CURVE_u      CURVE;

union SURFACE_u
        {
        struct PLANE_s                  *plane;
        struct CYLINDER_s               *cylinder;
        struct CONE_s                   *cone;
        struct SPHERE_s                 *sphere;
        struct TORUS_s                  *torus;
        struct BLENDED_EDGE_s           *blended_edge;
        struct BLEND_BOUND_s            *blend_bound;
        struct OFFSET_SURF_s            *offset_surf;
        struct SWEPT_SURF_s             *swept_surf;
        struct SPUN_SURF_s              *spun_surf;
        struct PE_SURF_s                *pe_surf;
        struct B_SURFACE_s              *b_surface;
        };
typedef union SURFACE_u    SURFACE;

union GEOMETRY_u
        {
        union  SURFACE_u                 surface;
        union  CURVE_u                   curve;
        struct POINT_s                  *point;
        struct TRANSFORM_s              *transform;
        };
typedef union GEOMETRY_u GEOMETRY;
```

Curves

In the following field tables, ‘pointer0’ means a reference to another node which may be null. ‘pointer’ means a non-null reference.

All curve nodes share the following common fields:

Field name	Data type	Description
node_id	int	Integer value unique to curve in part
attributes_groups	pointer0	Attributes and groups associated with curve
owner	pointer0	topological owner
next	pointer0	next curve in geometry chain
previous	pointer0	previous curve in geometry chain
geometric_owner	pointer0	geometric owner node
sense	char	sense of curve: ‘+’ or ‘-’ (see end of Geometry section)

```
struct ANY_CURVE_s          // Any Curve
    {
    int                         node_id;                // $d
    union  ATTRIB_GROUP_u       attributes_groups;      // $p
    union  CURVE_OWNER_u        owner;                  // $p
    union  CURVE_u              next;                   // $p
    union  CURVE_u              previous;               // $p
    struct                      *geometric_owner;       // $p
    GEOMETRIC_OWNER_s
    char                        sense;                  // $c
    };
typedef struct ANY_CURVE_s *ANY_CURVE;
```

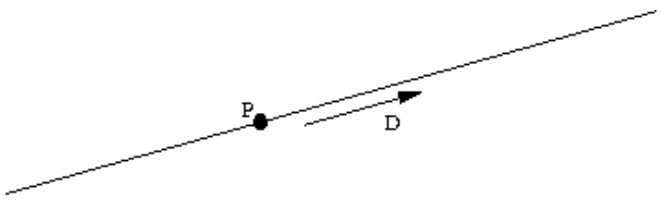
- **LINE**

A straight line has a parametric representation of the form:

$R(t) = P + t D$

where

- P is a point on the line



- D is its direction

Field name	Data type	Description
pvec	vector	point on the line
direction	vector	direction of the line (a unit vector)

```
struct LINE_s == ANY_CURVE_s     // Straight line
    {
    int                          node_id;                 // $d
    union  ATTRIB_GROUP_u        attributes_groups;       // $p
    union  CURVE_OWNER_u         owner;                   // $p
    union  CURVE_u               next;                    // $p
    union  CURVE_u               previous;                // $p
    struct                       *geometric_ owner;       // $p
    GEOMETRIC_OWNER_s
    char                         sense;                   // $c
    vector                       pvec;                    // $v
    vector                       direction;               // $v
    };
typedef struct LINE_s      *LINE;
```

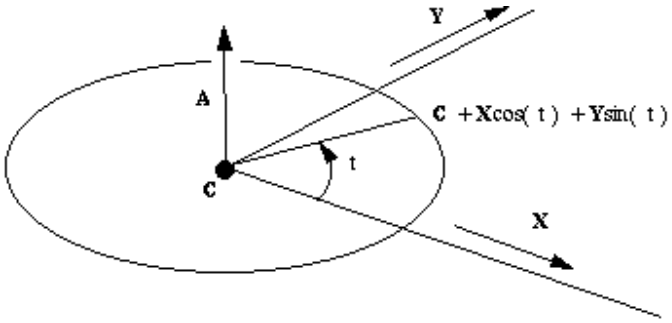
CIRCLE

A circle has a parametric representation of the form

R(t) = C+ r X cos(t) + r Y sin(t)

Where

- C is the centre of the circle
- r is the radius of the circle
- X and Y are the axes in the plane of the circle.



Field name	Data type	Description
centre	vector	Centre of circle
normal	vector	Normal to the plane containing the circle (a unit vector)
x_axis	vector	X axis in the plane of the circle (a unit vector)
radius	double	Radius of circle

The Y axis in the definition above is the vector cross product of the normal and x_axis.

```
struct CIRCLE_s == ANY_CURVE_s                  // Circle
    {
    int                             node_id;                // $d
    union  ATTRIB_GROUP_u           attributes_groups;      // $p
    union  CURVE_OWNER_u            owner;                  // $p
    union  CURVE_u                  next;                   // $p
    union  CURVE_u                  previous;               // $p
    struct GEOMETRIC_OWNER_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;
```

- **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 radiu

```
    vector                          normal;                 // $v
    vector                          x_axis;                 // $v
    double                          major_radius;           // $f
    double                          minor_radius;           // $f
};
typedef struct ELLIPSE_s  *ELLIPSE;
```

- **THE NUMBER OF KNOTS AND VERTICES**

The knot set determines a set of basis functions which are bell shaped, and non zero over a span of *(degree+1)* intervals. One basis function starts at each knot, and each one finishes *(degree +1)* knots higher. The control vectors are the coefficients applied to these basis functions in a linear sum to obtain positions on the curve. Thus it can be seen that we require the number of knots n_knots = n_vertices + degree + 1

THE VALID RANGE OF THE B-CURVE

So if the knot set is numbered $\{t_0$ to $t_{n\_knots-1}\}$ it can be seen then that it is only after t_{degree} that sufficient *(degree + 1)* basis functions are present for the curve to be fully defined, and that the B-curve ceases to be fully defined after $t_{n\_knots - 1 - degree}$.

The first *degree* knots and the last *degree* knots are known as the imaginary knots because their parameter values are outside the defined range of the B-curve.

PERIODIC B-CURVES

When the end of a B-curve meets its start sufficiently smoothly Parasolid allows it to be defined to have periodic parametrisation. That is to say that if the valid range were from $t_{\ldots}$ to $t_{\ldots}$ then the difference between

In Parasolid the weights are stored with the vertices by treating these as having an extra dimension. In the usual case of a curve in 3-d cartesian space this means that vertex_dim is 4, the x, y, z values are multiplied through by the corresponding weight and the 4th value is the weight itself.

B-SURFACE DEFINITION

$$P(u,v) = \frac{\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} b_i(u) \quad b_j(v) \quad w_{ij}V_{ij}}{\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} b_i(u) \quad b_j(v) \quad w_{ij}}$$

The B-surface definition is best thought of as an extension of the B-curve definition into two parameters, usually called u and v. Two knot sets are required and the number of control vertices is the product of the number that would be required for a curve using each knot vector. The rules for periodicity and closure given above for curves

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 parameterisation 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 an XT file 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:

The SCH_checked_ok_in_old_version enum indicates that the self-intersection check has been performed by a Parasolid version 5 or earlier but not since.

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.

INTERSECTION

An intersection curve is one of the branches of a surface / surface intersection. Parasolid represents these curves exactly; the information held in an intersection curve node is sufficient to identify the particular intersection branch involved, to identify the behavior 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.

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 t

```
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 parameterisation (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. Ther 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

$$C_i = | P_{i+1} - P_i |$$

$$\cos(a_i) = T_i \,.\, (P_{i+1} - P_i)$$

$$\cos(b_i) = T_i \,.\, (P_i - P_{i-1})$$

Then at any chart point P_i the angles a_i and b_i are the deviations between the tangent at the chart point and the next and previous chords respectively.

Let f_0 = base_scale

$$f_i = (\cos(b_i) / \cos(a_i)) \, f_{i-1}$$

Then t_0 = base_parameter

$$t_i = t_{i-1} + C_{i-1} \, f_{i-1}$$

The parameter of a point between two chart points is given by projecting the point onto the tangent line at the previous chart point. The factors f_i are chosen so that the parameterization is C_1.

TRIMMED_CURVE

A trimmed curve is a bounded region of another curve, referred to as its basis curve. It is defined by the basis curve and two points and their corresponding parameters. Trimmed curves are most commonly attached to fins (fins) of tolerant edges in order to specify which portion of the underlying basis curve corresponds to the tolerant edge. They are necessary since the tolerant vertices of the edge do not necessarily lie exactly on the basis curve; the 'point' fields of the trimmed curve lie exactly on the basis curve, and within tolerance of the relevant vertex.

The rules governing the parameter fields and points are:

- point_1 and point_2 correspond to parm_1 and parm_2 respectively.
- If the basis curve has positive sense, parm_2 > parm_1.
- If the basis curve has negative sense, parm_2 < parm_1.

In addition,

For open basis curves.

- Both parm_1 and parm_2 must be in the parameter range of the basis curve.
- point_1 and point_2 must not be equal.

For periodic basis curves

- parm_1 must 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 must not be more than a period apart.

For closed but non-periodic basis curves

- Both parm_1 and parm_2 must be in the parameter range of the basis curve.
- If the whole of the basis curve is required, parm_1 and parm_2 must lie close enough to each end of the valid parameter range in order that point_1 and point_2 are coincident to Parasolid tolerance (1.0e-8 by default).

The sense of a trimmed curve is positive.

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;
```

PE_CURVE (Foreign Geometry curve)

Foreign geometry in Parasolid is a type used for representing customers' in-house proprietary data. It is also known as PE (parametrically evaluated) geometry. It can also be used internally for representing geometry connected with this data (for example, offsets of foreign surfaces). These two types of foreign geometry usage are referred to as 'external' and 'internal' PE data respectively. Internal PE curves are not used at present.

Applications not using foreign geometry will never encounter either external or internal PE data structures at Parasolid V9 or beyond.

Field name	Data type	Description
type	char	whether internal or external
data	pointer	internal or external data
tf	pointer0	transform applied to geometry
internal geom	pointer array	reference to other related geometry

```
union PE_DATA_u                         // PE_data_u
    {
    struct EXT_PE_DATA_s        *external;                  // $p
    struct INT_PE_DATA_s        *internal;                  // $p
    };
typedef union PE_DATA_u PE_DATA;
```

The PE internal geometry union defined below is used by internal foreign geometry only.

```
union PE_INT_GEOM_u
    {
    union SURFACE_u             surface;                    // $p
    union CURVE_u               curve;                      // $p
    };
typedef union PE_INT_GEOM_u PE_INT_GEOM;

struct PE_CURVE_s == ANY_CURVE_s            // PE_curve
    {
    int                         node_id;                    // $d
    union  ATTRIB_GROUP_u       attributes_groups;          // $p
    union  CURVE_OWNER_u        owner;                      // $p
    union  CURVE_u              next;                       // $p
    union  CURVE_u              previous;                   // $p
    struct                      *geometric_owner;           // $p
    GEOMETRIC_OWNER_s
    char                        sense;                      // $c
    char                        type;                       // $c
    union  PE_DATA_u            data;                       // $p
    struct TRANSFORM_s          *tf;                        // $p
    union  PE_INT_GEOM_u        internal_geom[ 1 ];         // $p[]
```

```
    };
typedef struct PE_CURVE_s       *PE_CURVE;
```

The type of the foreign geometry (whether internal or external) is identified in the PE curve node by means of the char ‘type’ field, taking one of the values

```
    const char SCH_external  = 'E';          // external PE geometry
    const char SCH_interna   = 'I';          // internal PE geometry
```

The PE_data union is used in a PE curve or surface node to identify the internal or external evaluator corresponding to the geometry, and also holds an array of real and/or integer parameters to be passed to the evaluator. The data stored corresponds exactly to that passed to the PK routine PK_FSURF_create when the geometry is created.

```
struct EXT_PE_DATA_s                // ext_PE_data
    {
    struct KEY_s                *key;                   // $p
    struct REAL_VALUES_s        *real_array;            // $p
    struct INT_VALUES_s         *int_array;             // $p
    };
typedef struct EXT_PE_DATA_s *EXT_PE_DATA;

struct INT_PE_DATA_s                // int_PE_data
    {
    int                         geom_type;              // $d
    struct REAL_VALUES_s        *real_array;            // $p
    struct INT_VALUES_s         *int_array;             // $p
    };
typedef struct INT_PE_DATA_s *INT_PE_DATA;
```

The only internal pe type in use at the moment is the offset PE surface, for which the geom_type is 2.

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 must be a 2D BCURVE; that is it must either be a rational B curve with a vertex dimensionality of 3, or a non-rational B curve with a vertex dimensionality of 2.

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;           // $p
      GEOMETRIC_OWNER_s
      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;
```

Surfaces

All surface nodes share the following common 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;
```

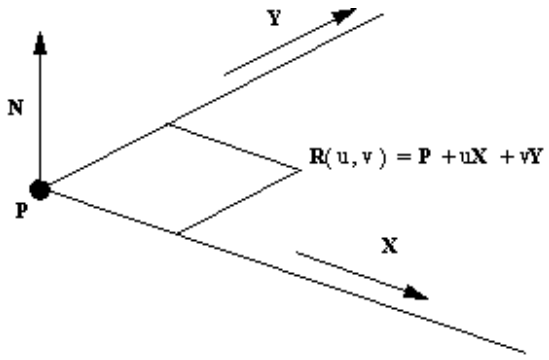
PLANE

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.

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
    {
```

J

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;
```

CONE

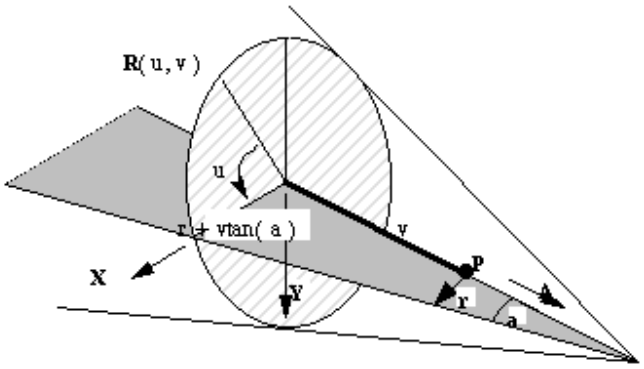
A cone in Parasolid is only half of a mathematical cone. By convention, the cone axis points away from the half of the cone in use. A cone has a parametric representation of the form:

$R(u, v) = P - vA + (X\cos(u) + Y\sin(u))(r + v\tan(a))$

where

- P is a point on the cone axis
- r is the cone radius at the point P
- A is the cone axis
- X and Y are unit vectors such that A, X and Y form an orthonormal set, i.e. Y = A x X.

- a is the cone half angle.



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;
```

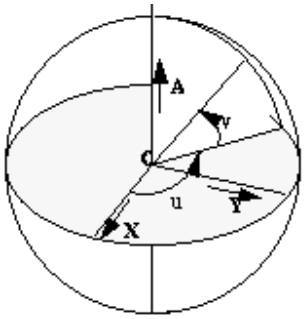
SPHERE

A sphere has a parametric representation of the form:

R(u, v) = C + (Xcos(u) + Ysin(u)) rcos(v) + rAsin(v)

where

- C is centre of the sphere
- r is the sphere radius



- A, X and Y form an orthonormal axis set.

Field name	Data type	Description
centre	vector	centre of the sphere
radius	double	radius of the sphere
axis	vector	A axis of the sphere (a unit vector)
x_axis	vector	X axis of the sphere (a unit vector)

The Y axis of the sphere is the vector cross product of its A and X axes.

```
struct SPHERE_s == ANY_SURF_s                // Sphere
    {
    int                         node_id;                 // $d
    union  ATTRIB_GROUP_u       attributes_groups;       // $p
    union  SURFACE_OWNER_u      owner;                   // $p
    union  SURFACE_u            next;                    // $p
    union  SURFACE_u            previous;                // $p
```

```
struct GEOMETRIC_OWNER_s    *geometric_owner;    // $p
char                        sense;               // $c
vector                      centre;              // $v
double                      radius;              // $f
vector                      axis;                // $v
vector                      x_axis;              // $v
};
typedef struct SPHERE_s    *SPHERE;
```

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 center 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 Parasolid, there are three types of torus:

Doughnut - the torus is not self-intersecting (a > b)

Apple - the outer part of a self-intersecting torus (a <= b, a > 0)

Lemon - the inner part of a self-intersecting torus (a < 0, |a| < b)

The limiting case a = b is allowed; it is called an ‘osculating apple’, but there is no ‘lemon’ surface corresponding to this case.

The limiting case a = 0 cannot be represented as a torus; this is a sphere.

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;
```

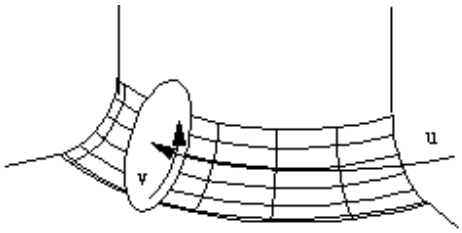
BLENDED_EDGE (Rolling Ball Blend)

Parasolid supports exact rolling ball blends. They have a parametric representation of the form

R(u, v) = C(u) + rX(u)cos(v a(u)) + rY(u)sin(va(u))

where

- C(u) is the spine curve
- r is the blend radius
- X(u) and Y(u) are unit vectors such that C'(u) . X(u) = C'(u) . Y(u) = 0
- a(u) is the angle subtended by points on the boundary curves at the spine



X, Y and a are expressed as functions of u, as their values change with u.

The spine of the rolling ball blend is the center line of the blend; i.e. the path along which the center of the ball moves.

Field name	Data type	Description
type	char	type of blend: ‘R’ or ‘E’
surface	pointer[2]	supporting surfaces (adjacent to original edge)
spine	pointer	[redacted]

```
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.

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. Since the actual shape of the surface is not significant for the blend geometry, it is not described here.

Blend boundary surfaces are most commonly referenced by the intersection curve representing the boundary curve of the blend.

The data stored in an XT file for a blend_bound is only that necessary to identify the relevant blend and supporting surface:

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;           // $p
    GEOMETRIC_OWNER_s
    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].

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.

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 must not be within modeller linear resolution of zero
- The sense of the offset surface must 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
```

B_SURFACE

Parasolid supports B spline curves in full NURBS format.

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 an XT file for a NURBS surface is

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 BCURVE.

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. It is recommended that it only be set by Parasolid.

```
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 *SU
```

t SURFAC–

The ‘original_’ and ‘extended_’ parameter intervals and corresponding character fields original_u_start etc. are all connected with Parasolid’s 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

9 Tc 8.39375 0 Td ((Z)' 0.00000197 Tc 5.24375 0 Td ( )Tj -0.0501084 Tc 5.24883 0 Td SO)Tj 0.00183443 Tc 8.3 Td (H[WQ

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.

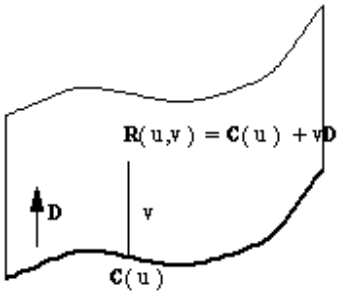
SWEPT_SURF

A swept surface has a parametric representation of the form:

R(u, v) = C(u) + vD

where

- C(u) is the section curve.
- D is the sweep direction (unit vector).



- C must not be an intersection curve or a trimmed curve.

Field name	Data type	Description
section	pointer	section curve
sweep	vector	sweep direction (a unit vector)
scale	double	for internal use only – may be set to null

struct SWEPT_SURF_s == ANY_SURF_s // Swept surface

```
{
int                             node_id;                        // $d
union  ATTRIB_GROUP_u           attributes_groups;
```

Field name	Data type	Description
profile	pointer	profile curve
base	vector	point on spin axis
axis	vector	spin axis direction (a unit vector)
start	vector	position of degeneracy at low u (may be null)
end	vector	position of degeneracy at low v (may be null)
start_param	double	curve parameter at low u degeneracy (may be null)
end_param	double	curve parameter at high u degeneracy (may be null)
x_axis	vector	unit vector in profile plane if common with spin axis
scale	double	for internal use only – may be set to null

```
struct SPUN_SURF_s == ANY_SURF_s              // Spun surface
    {
    int                             node_id;                // $d
    union  ATTRIB_GROUP_u           attributes_groups;      // $p
    union  SURFACE_OWNER_u          owner;                  // $p
    union  SURFACE_u                next;                   // $p
    union  SURFACE_u                previous;               // $p
    struct                         *geometric_owner;        // $p
    GEOMETRIC_OWNER_s
    char                            sense;                  // $c
    union  CURVE_u                  profile;                // $p
    vector                          base;                   // $v
    vector                          axis;                   // $v
    vector                          start;                  // $v
    vector                          end;                    // $v
    double                          start_param;            // $f
    double                          end_param;              // $f
    vector                          x_axis;                 // $v
    double                          scale;                  // $f
    };
typedef struct SPUN_SURF_s *SPUN_SURF;
```

The ‘start’ and ‘end’ vectors correspond to physical degeneracies on the spun surface caused by the profile curve crossing the spin axis at that point. The values start_param and end_param are the corresponding parameters on the curve. These parameter values define the valid range for the u parameter of the surface. If either value is null, then the valid range for u is infinite in that direction. For example, for a straight line profile curve intersecting the

spin axis at the parameter t=1, values of null for start_param and 1 for end_param would define a cone with u parameterisation (-infinity, 1].

If the profile curve lies in a plane containing the spin axis, then x_axis must be set to a vector perpendicular to the spin axis and in the plane of the profile, pointing from the spin axis to a point on the profile curve in the valid range. If the profile curve is not planar, or its plane does not contain the spin axis, then x_axis should be set to null.

PE_SURF (Foreign Geometry surface)

Foreign (or 'PE') geometry in Parasolid is a type used for representing customers' in-house proprietary data. It can also be used internally for representing geometry connected with this data (for example, offset foreign surfaces). These two types of foreign geometry usage are referred to as 'external' and 'internal' respectively. The only internal PE surface is the offset PE surface.

Applications not using foreign geometry will never encounter either external or internal PE data structures at Parasolid V9 or beyond.

Field name	Data type	Description
type	char	whether internal or external
data	pointer	internal or external data
tf	pointer0	transform applied to geometry
internal geom	pointer array	reference to other related geometry

```
struct PE_SURF_s == ANY_SURF_s              // PE_surface
    {
    int                                 node_id;                    // $d
    union  ATTRIB_GROUP_u               attributes_groups;          // $p
    union  SURFACE_OWNER_u              owner;                      // $p
    union  SURFACE_u                    next;                       // $p
    union  SURFACE_u                    previous;                   // $p
    struct GEOMETRIC_OWNER_s           *geometric_owner;            // $p
    char                                sense;                      // $c
    char                                type;                       // $c
    union  PE_DATA_u                    data;                       // $p
    struct TRANSFORM_s                 *tf;                         // $p
    union  PE_INT_GEOM_u                internal_geom[ 1 ];         // $p[]
    };
typedef struct PE_SURF_s  *PE_SURF;
```

The PE_DATA and PE_INT_GEOM unions are defined under 'PE curve'.

Point

Field name	Data type	Description
node_id	int	integer unique within part
attributes_groups	pointer0	attributes and groups associated with point
owner	pointer	Owner
next	pointer0	next point in chain
previous	pointer0	previous point in chain
pvec	vector	position of point

```
union POINT_OWNER_u
    {
    struct VERTEX_s             *vertex;
    struct BODY_s               *body;
    struct ASSEMBLY_s           *assembly;
    struct WORLD_s              *world;
    };

struct POINT_s                  // Point
    {
    int                          node_id;                   // $d
    union  ATTRIB_GROUP_u        attributes_groups;         // $p
    union  POINT_OWNER_u         owner;                     // $p
    struct POINT_s              *next;                      // $p
    struct POINT_s              *previous;                  // $p
    vector                       pvec;                      // $v
    };
typedef struct POINT_s    *POINT;
```

Transform

Field name	Data type	Description
node_id	int	integer unique within part
owner	pointer	owning instance or world
next	pointer0	next transform in chain

previous	pointer0	previous pointer in chain
rotation_matrix	double[3][3]	rotation component
translation_vector	vector	translation component
scale	double	scaling factor
flag	byte	binary flags indicating non-trivial components
perspective_vector	vector	perspective vector (always null vector)

The transform acts as

x’ = (rotation_matrix . x + translation_vector) * scale

The ‘flag’ field contains various bit flags which identify the components of the transformation:

Flag Name	**Binary Value**	**Description**
translation	00001	set if translation vector non-zero
rotation	00010	set if rotation matrix is not the identity
scaling	00100	set if scaling component is not 1.0
reflection	01000	set if determinant of rotation matrix is negative
general affine	10000	set if the rotation_matrix is not a rigid rotation

```
union TRANSFORM_OWNER_u
    {
    struct INSTANCE_s          *instance;
    struct WORLD_s             *world;
    };

struct TRANSFORM_s             // Transformation
    {
    int                            node_id;                    // $d
    union TRANSFORM_OWNER_u        owner;                      // $p
    struct TRANSFORM_s             *next;                      // $p
    struct TRANSFORM_s             *previous;                  // $p
    double                         rotation_matrix[3][3];      // $f[9]
```

```
vector          translation_vector;     // $v
double          scale;                  // $f
unsigned        flag;                   // $d
vector          perspective_vector;     // $v
};
typedef struct TRANSFORM_s *TRANSFORM;
```

Curve and Surface Senses

The 'natural' tangent to a curve is that in the increasing parameter direction, and the '
is in the direction of the cross-product of dP/du and dP/dv. For some purposes thes
and surfaces senses, respectively – for example in the definition of blend surface
curves.

At the PK interface, the edge/curve and face/surface sense orientations are
topology/geometry combination. In the XT format, this orientation infor
faces as follows:

The edge/curve orientation is stored in the curve->sense field. The
sense flags stored in the face->sense and surface->sense fields, s
normal is parallel to the natural surface normal) if neither, or b

Geometric_owner

Where geometry has dependants, the dependants point b
Owner nodes. Each geometric node points to a doubly
referencing geometry. Referenced geometry is as fol

• **Field name**	• **Data type**	• **Description**
• owner	• pointer	• referencing geometry
• next	• pointer	• next in ring of geometric owners referring to the same geometry
• previous	• pointer	• previous in above ring
• shared_geometry	• pointer	• referenced (dependent) geometry

•

```
struct GEOMETRIC_OWNER_s        //  geometric owner of geometry
    {
    union  GEOMETRY_u               owner;                  //  $p
    struct GEOMETRIC_OWNER_s    *next;                      //  $p
    struct GEOMETRIC_OWNER_s    *previous;                  // $p
    union  GEOMETRY_u               shared_geometry;        //  $p
    };
typedef struct GEOMETRIC_OWNER_s *GEOMETRIC_OWNER;
```

Topology

In the following tables, ‘ignore’ means this may be set to null (zero) if an XT file is created outside Parasolid. For an XT file created by Parasolid, this may take any value, but should be ignored.

Unless otherwise stated, all chains of nodes are doubly-linked and null-terminated.

- **WORLD**

• **Field name**	• **Type**	• **Description**
• assembly	• pointer0	• Head of chain of assemblies
• attribute	• pointer0	• Ignore
• body	• pointer0	• Head of chain of bodies
• transform	• pointer0	• Head of chain of transforms
• surface	• pointer0	• Head of chain of surfaces
• curve	• pointer0	• Head of chain of curves
• point	• pointer0	• Head of chain of points
• alive	• logical	• True unless partition is at initial pmark
• attrib_def	• pointer0	• Head of chain of attribute definitions
• highest_id	• int	• Highest pmark id in partition
• current_id	• int	• Id of current pmark
• index_map_offset	• int	• Must be set to 0
• index_map	• pointer0	• Must be set to null
• schema_embedding_map	• pointer0	• Must be set to null

•

The World node is only used when a partition is transmitted. Because some of the attribute definitions may be referenced by nodes which have been deleted, but which may reappear on rollback, the attribute definitions are chained off the World node rather than simply being referenced by attributes.

The fields index_map_offset, index_map, and schema_embedding_map are used for Indexed Transmit; applications writing XT data must set them to 0 and null.

```
struct WORLD_s                          // World
    {
    struct ASSEMBLY_s       *assembly;                  // $p
    struct ATTRIBUTE_s      *attribute;                 // $p
```

```
    struct BODY_s               *body;                  // $p
    struct TRANSFORM_s          *transform;             // $p
    union  SURFACE_u             surface;               // $p
    union  CURVE_u               curve;                 // $p
    struct POINT_s              *point;                 // $p
    logical                      alive;                 // $l
    struct ATTRIB_DEF_s         *attrib_def;            // $p
    int                          highest_id;            // $d
    int                          current_id;            // $d
    };
typedef struct WORLD_s  *WORLD;
```

ASSEMBLY

highest_node_id	int	Highest node-id in assembly
attributes_groups	pointer0	Head of chain of attributes of, and groups in, assembly
attribute_chains	pointer0	List of attributes, one for each attribute definition used in the assembly
list	pointer0	Null
surface	pointer0	Head of construction surface chain
curve	pointer0	Head of construction curve chain
point	pointer0	Head of construction point chain
key	pointer0	Ignore
res_size	double	Value of ‘size box’ when transmitted (normally 1000)
res_linear	double	Value of modeller linear precision when transmitted (normally 1.0e-8).
ref_instance	pointer0	Head of chain of instances referencing this assembly
next	pointer0	Ignore
previous	pointer0	Ignore
state	byte	Set to 1.
owner	pointer0	Ignore
type	byte	Always 1.
sub_instance	pointer0	Head of chain of instances in assembly

The value of the ‘state’ field should be ignored, as should any nodes of type ‘KEY’ referenced by the assembly. If an XT file is constructed outside Parasolid, the state field should be set to 1, and the key to null.

```
    struct ASSEMBLY_s       *next;                  //  $p
    struct ASSEMBLY_s       *previous;              //  $p
    SCH_part_state           state;                 //  $u
    struct WORLD_s          *owner;                 //  $p
    SCH_assembly_type        type;                  //  $u
    struct INSTANCE_s       *sub_instance;          //  $p
    };
typedef struct ASSEMBLY_s *ASSEMBLY;
struct KEY_s                                        // Key
    {
    string[1];              char                    // $c[]
    };
typedef struct KEY_s *KEY;
```

INSTANCE

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of instance and member_of_groups of instance
type	byte	Always 1
part	pointer	Part referenced by instance
transform	pointer0	Transform of instance
assembly	pointer	Assembly in which instance lies
next_in_part	pointer0	Next instance in assembly
prev_in_part	pointer0	Previous instance in assembly
next_of_part	pointer0	Next instance of instance->part
prev_of_part	pointer0	Previous instance of instance->part

```
typedef enum
    {
    SCH_positive_instance = 1,
    SCH_negative_instance = 2
    }
    SCH_instance_type;
```

```
union  PART_u
	{
	struct BODY_s			*body;
	struct ASSEMBLY_s		*assembly;
	};
typedef union PART_u	PART;

struct INSTANCE_s			// Instance
	{
	int				node_id;		// $d
	union  ATTRIB_GROUP_u		attributes_groups;	// $p
	SCH_instance_type		type;			// $u
	union  PART_u			part;			// $p
	struct TRANSFORM_s		*transform;		// $p
	struct ASSEMBLY_s		*assembly;		// $p
	struct INSTANCE_s		*next_in_part;		// $p
	struct INSTANCE_s		*prev_in_part;		// $p
	struct INSTANCE_s		*next_of_part;		// $p
	struct INSTANCE_s		*prev_of_part;		// $p
	};
typedef struct INSTANCE_s *INSTANCE;
```

BODY

Field name	**Type**	**Description**
highest_node_id	int	Highest node-id in body
attributes_groups	pointer0	Head of chain of attributes of, and groups in, body
attribute_chains	pointer0	List of attributes, one for each attribute definition used in the body
surface	pointer0	Head of construction surface chain
curve	pointer0	Head of construction curve chain
point	pointer0	Head of construction point chain
key	pointer0	Ignore

res_size	double	Value of ‘size box’ when transmitted (normally 1000)
res_linear	double	Value of modeller linear precision when transmitted (normally 1.0e-8)
ref_instance	pointer0	Head of chain of instances referencing this part
next	pointer0	Ignore
previous	pointer0	Ignore
state	byte	Set to 1 (see below)
owner	pointer0	Ignore
body_type	byte	Body type
nom_geom_state	byte	Set to 1 (for future use)
shell	pointer0	For general bodies: null For solid bodies: the first shell in one of the solid regions For other bodies: the first shell in one of the regions This field is **obsolete**, and should be ignored by applications reading XT files. When writing XT files, it must be set as above.
boundary_surface	pointer0	Head of chain of surfaces attached directly or indirectly to faces or edges or fins
boundary_curve	pointer0	Head of chain of curves attached directly or indirectly to edges or faces or fins
boundary_point	pointer0	Head of chain of points attached to vertices
region	pointer	Head of chain of regions in body; this is the infinite region
edge	pointer0	Head of chain of all non-wireframe edges in body
vertex	pointer0	Head of chain of all vertices in body
index_map_offset	int	Must be set to 0
index_map	pointer0	Must be set to null
node_id_index_map	pointer0	Must be set to null
schema_embedding_map	pointer0	Must be set to null

The value of the ‘state’ field should be ignored, as should any nodes of type ‘KEY’ referenced by the body. If an XT file is constructed outside Parasolid, the state field should be set to 1, and the key to null.

The highest_node_id gives the highest node of any node in this body. Most nodes in a body which are visible at the PK interface have node-ids, which are non-zero integers unique to that node within the body. Applications writing XT files must ensure that node-ids are present and distinct. The details of which nodes have node ids are given in an appendix.

The fields index_map_offset, index_map, node_id_index_map, and schema_embedding_map are used for Indexed Transmit; applications writing XT files must ensure that these fields are set to 0 and null.

```
typedef enum
    {
    SCH_solid_body     = 1,
    SCH_wire_body      = 2,
    SCH_sheet_body     = 3,
    SCH_general_body   = 6
    }
    SCH_body_type;


typedef short short enum
     {
     SCH_nom_geom_off = 1,          --- Entirely off
     SCH_nom_geom_on  = 2           --- Entirely on
     }
     SCH_nom_geom_state_t;

struct BODY_s                       //  Body
    {
    int                             highest_node_id;            //  $d
    union ATTRIB_GROUP_u            attributes_groups;          //  $p
    struct LIST_s                  *attribute_chains;           //  $p
    union SURFACE_u                 surface;                    //  $p
    union CURVE_u                   curve;                      //  $p
    struct POINT_s                 *point;                      //  $p
    struct KEY_s                   *key;                        //  $p
    double                          res_size;                   //  $f
    double                          res_linear;                 //  $f
    struct INSTANCE_s              *ref_instance;               //  $p
    struct BODY_s                  *next;                       //  $p
    struct BODY_s                  *previous;                   //  $p
```

```
    SCH_part_state          state;                  // $u
    struct WORLD_s          *owner;                 // $p
    SCH_body_type           body_type;              // $u
    SCH_nom_geom_state_t    nom_geom_state;         // $u
    struct SHELL_s          *shell;                 // $p
    union SURFACE_u         boundary_surface;       // $p
    union CURVE_u           boundary_curve;         // $p
    struct POINT_s          *boundary_point;        // $p
    struct REGION_s         *region;                // $p
    struct EDGE_s           *edge;                  // $p
    struct VERTEX_s         *vertex;                // $p
    int                     index_map_offset;       // $d
    struct INT_VALUES_s     *index_map;             // $p
    struct INT_VALUES_s     *node_id_index_map;     // $p
    struct INT_VALUES_s     *schema_embedding_map;  // $p
    };
typedef struct BODY_s   *BODY;
```

Attaching Geometry to Topology

The faces which reference a surface are chained together, surface->owner is the head of this chain. Similarly the edges which reference the same curve are chained together. Fins do not share curves.

Geometry in parts may be chained into one of the three boundary geometry chains, or one of the three construction geometry chains. A geometric node will fall into one of the following cases:

Geometry	**Owner**	**Whether chained**
Attached to face	face	In boundary_surface chain
Attached to edge or fin	edge or fin	In boundary_curve chain
Attached to vertex	vertex	In boundary_point chain
Indirectly attached to face or edge or fin	body	In boundary_surface chain or boundary_curve chain
Construction geometry	body or assembly	In surface, curve or point chain
2D B-curve in SP-curve	null	Not chained

Here ‘indirectly attached’ means geometry which is a dependent of a dependent of (... etc) of geometry attached to an edge, face or fin.

Geometry in a construction chain may reference geometry in a boundary chain, but not vice-versa.

REGION

Field name	**Type**	**Description**
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of region and member_of_groups of region
body	pointer	Body of region
next	pointer0	Next region in body
prev	pointer0	Previous region in body
shell	pointer0	Head of singly-linked chain of shells in region
type	char	Region type – solid (‘S’) or void (‘V’)

```
struct REGION_s                    //  Region
    {
    int                            node_id;                    //  $d
```

```
    union  ATTRIB_GROUP_u       attributes_groups;          //  $p
    struct BODY_s               *body;                      //  $p
    struct REGION_s             *next;                      //  $p
    struct REGION_s             *previous;                  //  $p
    struct SHELL_s              *shell;                     //  $p
    char                        type;                       //  $c
    };
typedef struct REGION_s   *REGION;
```

SHELL

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of shell
body	pointer0	For shells in wire and sheet bodies, and for shells bounding a solid region of a solid body, this is set to the body of the shell. For shells in general bodies, or void shells in solid bodies, it is null. This field is **obsolete**, and should be ignored by applications reading XT files. When writing XT files, it must be set as above.
next	pointer0	Next shell in region
face	pointer0	Head of chain of back-faces of shell (i.e. faces with face normal pointing out of region of shell).
edge	pointer0	Head of chain of wire-frame edges of shell
vertex	pointer0	If shell consists of a single vertex, this is it; else null
region	pointer	Region of shell
front_face	pointer0	Head of chain of front-faces of shell (i.e. faces with face normal pointing into region of shell)

```
struct SHELL_s                  //  Shell
```

```
    struct FACE_s                   *face;                  //  $p
    struct EDGE_s                   *edge;                  //  $p
    struct VERTEX_s                 *vertex;                //  $p
    struct REGION_s                 *region;                //  $p
    struct FACE_s                   *front_face;            //  $p
    };
typedef struct SHELL_s    *SHELL;
```

FACE

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of face and member_of_groups of face
tolerance	double	Not used (null double)
next	pointer0	Next back-face in shell
previous	pointer0	Previous back-face in shell
loop	pointer0	Head of singly-linked chain of loops
shell	pointer	Shell of which this is a back-face
surface	pointer0	Surface of face
sense	char	Face sense – positive (‘+’) or negative (‘-’)
next_on_surface	pointer0	Next in chain of faces sharing the surface of this face
previous_on_surface	pointer0	Previous in chain of faces sharing the surface of this face
next_front	pointer0	Next front-face in shell
previous_front	pointer0	Previous front-face in shell
front_shell	pointer	Shell of which this is a front-face

```
struct FACE_s                       //  Face
    {
    int                             node_id;                //  $d
    union  ATTRIB_GROUP_u           attributes_groups;      //  $p
    double                          tolerance;              //  $f
    struct FACE_s                   *next;                  //  $p
    struct FACE_s                   *previous;              //  $p
```

```
    struct LOOP_s                   *loop;                  //  $p
    struct SHELL_s                  *shell;                 //  $p
    union  SURFACE_u                 surface;               //  $p
    char                             sense;                 //  $c
    struct FACE_s                   *next_on_surface;       //  $p
    struct FACE_s                   *previous_on_surface;   //  $p
    struct FACE_s                   *next_front;            //  $p
    struct FACE_s                   *previous_front;        //  $p
    struct SHELL_s                  *front_shell;           //  $p
    };
typedef struct FACE_s     *FACE;
```

LOOP

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of loop
fin	pointer	One of ring of fins of loop
face	pointer	Face of loop
next	pointer0	Next loop in face

- **Isolated Loops**

An isolated loop (one consisting of a single vertex) does not refer directly to a vertex, but points to a fin which refers to that vertex. This isolated fin has fin->forward = fin->backward = fin, and fin->other = fin->curve = fin->edge = null. Its sense is not significant. The fin is chained into the chain of fins referencing the isolated vertex.

```
struct LOOP_s                       //  Loop
    {
    int                              node_id;               //  $d
    union  ATTRIB_GROUP_u            attributes_groups;     //  $p
    struct FIN_s                    *fin;                   //  $p
    struct FACE_s                   *face;                  //  $p
    struct LOOP_s                   *next;                  //  $p
    };
typedef struct LOOP_s     *LOOP;
```

FIN

Field name	Type	Description
attributes_groups	pointer0	Head of chain of attributes of fin
loop	pointer0	Loop of fin
forward	pointer0	Next fin around loop
backward	pointer0	Previous fin around loop
vertex	pointer0	Forward vertex of fin
other	pointer0	Next fin around edge, clockwise looking along edge
edge	pointer0	Edge of fin
curve	pointer0	For a non-dummy fin of a tolerant edge, this will be a trimmed SP-curve, otherwise null.
next_at_vx	pointer0	Next fin referencing the vertex of this fin
sense	char	Positive (‘+’) if the fin direction is parallel to that of its edge, else negative (‘-’)

Dummy Fins

An application will see edges as having any number of fins, including zero. However internally, they have at least two. This is so that the forward and backward vertices of an edge can always be found as edge->fin->vertex and edge->fin->other->vertex respectively - the first one being a positive fin, the second a negative fin. If an edge does not have both a positive and a negative externally-visible fin, **dummy** fins will exist for this purpose. Dummy fins have fin->loop = fin->forward = fin->backward = fin->curve = fin->next_at_vx = null. For example the boundaries of a sheet always have one dummy fin.

```
struct FIN_s                    //  Fin
    {
```

```
    };
typedef struct FIN_s *FIN;
```

VERTEX

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of vertex and member_of_groups of vertex
fin	pointer0	Head of singly-linked chain of fins referencing this vertex
previous	pointer0	Previous vertex in body
next	pointer0	Next vertex in body
point	pointer	Point of vertex
tolerance	double	Tolerance of vertex (null-double for accurate vertex)
owner	pointer	Owning body (for non-acorn vertices) or shell (for acorn vertices)

```
union SHELL_OR_BODY_u
    (
    struct BODY_s               *body;
    struct SHELL_s              *shell;
    };
typedef union SHELL_OR_BODY_u SHELL_OR_BODY;

struct VERTEX_s                 // Vertex
    {
    int                         node_id;            // $d
    union  ATTRIB_GROUP_u       attributes_groups;  // $p
    struct FIN_s                *fin;               // $p
    struct VERTEX_s             *previous;          // $p
    struct VERTEX_s             *next;              // $p
    struct POINT_s              *point;             // $p
    double                      tolerance;          // $f
    union  SHELL_OR_BODY_u      owner;              // $p
    };
```

typedef struct VERTEX_s *VERTEX;

EDGE

Field name	Type	Description
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of edge and member_of_groups of edge
tolerance	double	Tolerance of edge (null-double for accurate edges)
fin	pointer	One of singly-linked ring of fins around edge
previous	pointer0	Previous edge in body or shell
next	pointer0	Next edge in body or shell
curve	pointer0	Curve of edge, zero for tolerant edge. If edge is accurate, but any of its vertices are tolerant, this will be a trimmed curve
next_on_curve	pointer0	Next in chain of edges sharing the curve of this edge
previous_on_curve	pointer0	Previous in chain of edges sharing the curve of this edge
owner	pointer	Owning body (for non-wireframe edges) or shell (for wireframe edges)

```
struct EDGE_s                    //  Edge
    {
    int                          node_id;                  //  $d
    union  ATTRIB_GROUP_u        attributes_groups;        //  $p
    double                       tolerance;                //  $f
    struct FIN_s                *fin;                      //  $p
    struct EDGE_s               *previous;                 //  $p
    struct EDGE_s               *next;                     //  $p
    union  CURVE_u               curve;                    //  $p
    struct EDGE_s;              *next_on_curve             //  $p
    struct EDGE_s               *previous_on_curve;        //  $p
    union SHELL_OR_BODY_u        owner;                    //  $p
    };
typedef struct EDGE_s    *EDGE;
```

Associated Data

LIST

Field name	Type	Description
node_id	int	Zero
list_type	byte	Always 4
notransmit	logical	Ignore
owner	pointer	Owning part
next	pointer0	Ignore
previous	pointer0	Ignore
list_length	int	Length of list (>= 0)
block_length	int	Length of each block of list. Always 20
size_of_entry	int	Ignore
finger_index	int	Any integer between 1 and list->list_length (set to 1 if length is zero). Ignore
finger_block	pointer	Any block e.g. the first one. Ignore
list_block	pointer	Head of singly-linked chain of pointer list blocks

Lists only occur in part files as the list of attributes referenced by a part.

```
typedef enum
    {
    LIS_pointer    = 4
    }
LIS_type_t;

union LIS_BLOCK_u
    {
    struct POINTER_LIS_BLOCK_s      *pointer_block;
    };
typedef union LIS_BLOCK_u      LIS_BLOCK;

union LIST_OWNER_u
    {
    struct BODY_s                   *body;
```

```
    struct ASSEMBLY_s               *assembly;
    struct WORLD_s                  *world;
    };
typedef union LIST_OWNER_u LIST_OWNER;

struct LIST_s                       //  List Header
    {
    int                             node_id;            //  $d
    LIS_type_t                      list_type;          //  $u
    logical                         notransmit;         //  $l
    union LIST_OWNER_u              owner;              //  $p
    struct LIST_s                  *next;               //  $p
    struct LIST_s                  *previous;           //  $p
    int                             list_length;        //  $d
    int                             block_length;       //  $d
    int                             size_of_entry;      //  $d
    int                             finger_index;       //  $d
    union LIS_BLOCK_u               finger_block;       //  $p
    union LIS_BLOCK_u               list_block;         //  $p
    };
typedef struct LIST_s *LIST;
```

POINTER_LIS_BLOCK:

Field name	Type	Description
n_entries	int	Number of entries in this block (0 <= n_entries <= 20). Only the first block may have n_entries = 0.
index_map_offset	int	Must be set to 0
next_block	pointer0	Next pointer list block in chain
Entries[20]	pointer0	Pointers in block, those beyond n_entries must be zero

When the pointer_lis_block is used as the root node in a transmit file containing more than one part, the restriction n_entries <= 20 does not apply.

The index_map_offset field is used for Indexed Transmit; applications writing XT files must ensure this field is set to 0.

```
struct POINTER_LIS_BLOCK_s                  //  Pointer List
    {
    int                                      n_entries;                  //  $d
    int                                      index_map_offset            // $d
    struct POINTER_LIS_BLOCK_s               *next_block;                //  $p
    void                                     *entries[ 1 ];              //  $p[]
    };
typedef struct POINTER_LIS_BLOCK_s *POINTER_LIS_BLOCK;
```

```
    {
    union FIELD_NAME_u          names[1];                   //  $p[]
    };
typedef struct FIELD_NAME_s *FIELD_NAME;
```

ATTRIB_DEF

Field name	Type	Description

do_nothing	Leave attribute as it is
delete	Delete the attribute
transform	Transform the transformable fields (point, vector, direction, axis) by appropriate part of transformation
propagate	Copy attribute onto split-off node
keep_sub_dominant	Move attribute(s) from deleted node onto surviving node in a merge, but any such attributes already on the surviving node are deleted.
keep_if_equal	Keep attribute if present on both nodes being merged, with the same field values.
combine	Move attribute(s) from deleted node onto surviving node, in a merge

The PK attribute classes 1-7 correspond as follows:

	split	merge	transfer	change	Rotate	scale	translate	reflect
class 1	propagate	keep_equal	do_nothing	do_nothing	do_nothing	do_nothing	do_nothing	do_nothing
class 2	delete	delete	delete	delete	do_nothing	delete	do_nothing	do_nothing
class 3	delete	delete	delete	delete	Delete	delete	delete	delete
class 4	propagate	keep_equal	do_nothing	do_nothing	Transform	transform	transform	transform
class 5	delete	delete	delete	delete	Transform	transform	transform	transform
class 6	propagate	combine	do_nothing	do_nothing	do_nothing	do_nothing	do_nothing	do_nothing
class 7	propagate	combine	do_nothing	do_nothing	Transform	transform	transform	transform

Certain attribute definitions are created by Parasolid on startup, these are documented in an appendix.

typedef enum

{

SCH_rotate = 0,

SCH_scale = 1,

SCH_translate = 2,

SCH_reflect = 3,

SCH_split = 4,

SCH_merge = 5,

SCH_transfer = 6,

SCH_change = 7,

SCH_max_logged_event // last entry; value in $d[] code for actions

```
    }
    SCH_logged_event_t;

typedef enum
    {
    SCH_do_nothing          = 0,
    SCH_delete              = 1,
    SCH_transform           = 2,
    SCH_propagate           = 3,
    SCH_keep_sub_dominant = 4,
    SCH_keep_if_equal       = 5,
    SCH_combine             = 6
    }
    SCH_action_on_fields_t;

typedef enum
    {
    SCH_as_owner  = 0,
    SCH_in_owner  = 1,
    SCH_by_owner  = 2,
    SCH_sh_owner  = 3,
    SCH_fa_owner  = 4,
    SCH_lo_owner  = 5,
    SCH_ed_owner  = 6,
    SCH_vx_owner  = 7,
    SCH_fe_owner  = 8,
    SCH_sf_owner  = 9,
    SCH_cu_owner  = 10,
    SCH_pt_owner  = 11,
    SCH_rg_owner  = 12,
    SCH_fn_owner  = 13,
    SCH_max_owner               //  last entry; value in $l[] for
                                .legal_owners
    } SCH_attrib_owners_t;
```

```
typedef enum
    {
    SCH_int_field          = 1,
    SCH_real_field         = 2,
    SCH_char_field         = 3,
    SCH_point_field        = 4,
    SCH_vector_field       = 5,
    SCH_direction_field    = 6,
    SCH_axis_field         = 7,
    SCH_tag_field          = 8,
    SCH_pointer_field      = 9,
    SCH_unicode_field      = 10
    } SCH_field_type_t;


struct ATTRIB_DEF_s             //  attribute definition
    {
    struct ATTRIB_DEF_s         *next;                          //  $p
    struct ATT_DEF_ID_s         *identifier;                    //  $p
    int                          type_id;                       //  $d
    SCH_action_on_fields_t       actions
                                 [(int)SCH_max_logged_event];   //  $u[8]
    struct FIELD_NAMES_s        *field_names                    //  $p
    logical                      legal_owners
                                 [(int)SCH_max_owner];          //  $l[14]
    SCH_field_type_t             fields[1];                     //  $u[]
    };
typedef struct ATTRIB_DEF_s    *ATTRIB_DEF;
```

ATTRIBUTE

Field name	Type	Description
node_id	int	Node-id
definition	pointer	Attribute definition
owner	pointer	Attribute owner
next	pointer0	Next attribute, group, or member_of_group
previous	pointer0	Previous ditto

next_of_type	pointer0	Next attribute of this type in this part
previous_of_type	pointer0	Previous attribute of this type in this part
fields[]	pointer	Fields, of type int_values etc. The number of fields is given by the length of the variable part of the node. There may be no fields.

The attributes of a node are chained using the next and previous pointers in the attribute. The attribute_groups pointer in the node points to the head of this chain. This chain also contains the member_of_groups of the node.

Attributes within the same part, with the same attribute definition, are chained together by the next_of_type and previous_of_type pointers. The part points to the head of this chain as follows. The attribute_chains pointer in the part points to a list which contains the heads of these attribute chains, one for each attribute definition which has attributes in the part. The list may be null.

Note that the attributes_groups chains in parts, groups and nodes contain the following types of node:

Part: attributes and groups

Group: attributes

Node: attributes and member_of_groups

Fields of type ‘pointer’ can be used in Parasolid V12.0, but they are always transmitted as empty.

```
union ATTRIBUTE_OWNER_u
      {
     struct ASSEMBLY_s          *assembly;
     struct INSTANCE_s          *instance;
     struct BODY_s              *body;
     struct SHELL_s             *shell;
     struct REGION_s            *region;
     struct FACE_s              *face;
     struct LOOP_s              *loop;
     struct EDGE_s              *edge;
     struct FIN_s               *fin;
     struct VERTEX_s            *vertex;
     union  SURFACE_u            Surface;
     union  CURVE_u              Curve;
     struct POINT_s             *point;
     struct GROUP_s             *group;
     };
```

```
typedef union ATTRIBUTE_OWNER_u ATTRIBUTE_OWNER;

union FIELD_VALUES_u
    {
    struct INT_VALUES_s             *int_values;
    struct REAL_VALUES_s            *real_values;
    struct CHAR_VALUES_s            *char_values;
    struct POINT_VALUES_s           *point_values;
    struct VECTOR_VALUES_s          *vector_values;
    struct DIRECTION_VALUES_s       *direction_values;
    struct AXIS_VALUES_s            *axis_values;
    struct TAG_VALUES_s             *tag_values;
    struct UNICODE_VALUES_s         *unicode_values;
    };
typedef union FIELD_VALUES_u FIELD_VALUES;

struct ATTRIBUTE_s                  // Attribute
    {
    int                             node_id;            // $d
    struct ATTRIB_DEF_s             *definition;        // $p
    union  ATTRIBUTE_OWNER_u        owner;              // $p
    union  ATTRIB_GROUP_u           next;               // $p
    union  ATTRIB_GROUP_u           previous;           // $p
    struct ATTRIBUTE_s              *next_of_type;      // $p
    struct ATTRIBUTE_s              *previous_of_type;  // $p
    union  FIELD_VALUES_u           fields[1];          // $p[]
    };
typedef struct ATTRIBUTE_s *ATTRIBUTE;
```

INT_VALUES

values[]	int	Integer values

```
struct INT_VALUES_s                 // Int values
    {
```

```
    int                         values[1];                  // $d[]
    };
typedef struct INT_VALUES_s *INT_VALUES;
```

REAL_VALUES

values[]	double	Real values

```
struct REAL_VALUES_s            // Real values
    {
    double                      values[1];                  // $f[]
    };
typedef struct REAL_VALUES_s *REAL_VALUES;
```

CHAR_VALUES

values[]	char	Character values

```
struct CHAR_VALUES_s            // Character values
    {
    char                        values[1];                  // $c[]
    };
typedef struct CHAR_VALUES_s *CHAR_VALUES;
```

UNICODE_VALUES

values[]	short	Unicode character values

```
struct UNICODE_VALUES_s         // Unicode character values
    {
    short                       values[1];                  // $w[]
    };
typedef struct UNICODE_VALUES_s *UNICODE_VALUES;
```

POINT_VALUES

values[]	vector	Point values

```
struct POINT_VALUES_s                // Point values
    {
    vector                  values[1];                  // $v[]
    };
typedef struct POINT_VALUES_s *POINT_VALUES;
```

VECTOR_VALUES

values[]	vector	Vector values

```
struct VECTOR_VALUES_s               // Vector values
    {
    vector                  values[1];                  // $v[]
    };
typedef struct VECTOR_VALUES_s *VECTOR_VALUES;
```

DIRECTION_VALUES

values[]	vector	Direction values

```
struct DIRECTION_VALUES_s            // Direction values
    {
    vector                  values[1];                  // $v[]
    };
typedef struct DIRECTION_VALUES_s *DIRECTION_VALUES;
```

AXIS_VALUES

values[]	vector	Axis values

Note that an axis takes up two vectors.

```
struct AXIS_VALUES_s                 // Axis values
    {
    vector                  values[1];                  // $v[]
```

```
};
typedef struct AXIS_VALUES_s *AXIS_VALUES;
```

TAG_VALUES

values[]	int	Integer tag values

The tag field type and the tag_values node are not available for use in user-defined attributes, they occur only in certain system attributes.

```
struct TAG_VALUES_s                 //  Tag values
    {
    int                             values[1];                          //  $t[]
    };
typedef struct TAG_VALUES_s *TAG_VALUES;
```

GROUP

Field name	**Type**	**Description**
node_id	int	Node-id
attributes_groups	pointer0	Head of chain of attributes of this group
owner	pointer	Owning part
next	pointer0	Next group or attribute
previous	pointer0	Previous group or attribute
type	byte	Type of node allowed in group
first_member	pointer0	Head of chain of member_of_group nodes in group

The groups in a part are chained by the next and previous pointers in a group. The attributes_groups pointer in the part points to the head of the chain. This chain also contains the attributes attached directly to the part - groups and attributes are intermingled in this chain, the order is not significant.

Each group has a chain of member_of_groups. These are chained together using the next_member and previous_member pointers. The first_member pointer in the group points to the head of the chain. Each member_of_group has an owning_group pointer which points back to the group.

Each member_of_group has an owner pointer which points to a node. Thus the group references its member nodes via the member_of_groups.

The member_of_groups which refer to a particular node are chained using the next and previous pointers in the member_of_group. The attributes_groups pointer in the node points to the head of this chain. This chain also contains the attributes attached to the node.

```
typedef enum
    {
    SCH_instance_fe   = 1,
    SCH_face_fe       = 2,
    SCH_loop_fe       = 3,
    SCH_edge_fe       = 4,
    SCH_vertex_fe     = 5,
    SCH_surface_fe    = 6,
    SCH_curve_fe      = 7,
    SCH_point_fe      = 8,
    SCH_mixed_fe      = 9,
    SCH_region_fe     = 10
    }  SCH_group_type_t;

struct GROUP_s                  //  Group
    {
    int                                 node_id;                    //  $d
    union   ATTRIB_GROUP_u              attributes_groups;          //  $p
    union   PART_u                      owner;                      //  $p
    union   ATTRIB_GROUP_u              next;                       //  $p
    union   ATTRIB_GROUP_u              previous;                   //  $p
    SCH_group_type_t                    type;                       //  $u
    struct  MEMBER_OF_GROUP_s    *first_member;                     //  $p
    };
typedef struct GROUP_s *GROUP;
```

MEMBER_OF_GROUP

Field name	**Type**	**Description**
dummy_node_id	int	Entity label
owning_group	pointer	Owning group
owner	pointer	Referenced member of group
next	pointer0	Next attribute, group or member_of_group
previous	pointer0	Previous ditto

next_member	pointer0	Next member_of_group in this group
previous_member	pointer0	Previous ditto

```
union GROUP_MEMBER_u
        {
        struct INSTANCE_s               *instance;
        struct FACE_s                   *face;
        struct REGION_s                 *region;
        struct LOOP_s                   *loop;
        struct EDGE_s                   *edge;
        struct VERTEX_s                 *vertex;
        union  SURFACE_u                 surface;
        union  CURVE_u                   curve;
        struct POINT_s                  *point;
        };
typedef union GROUP_MEMBER_u GROUP_MEMBER;

struct MEMBER_OF_GROUP_s          //  Member of group
        {
        int                              dummy_node_id;          //  $d
        struct GROUP_s                  *owning_group;           //  $p
        union  GROUP_MEMBER_u            owner;                  //  $p
        union  ATTRIB_GROUP_u            next;                   //  $p
        union  ATTRIB_GROUP_u            previous;               //  $p
        struct MEMBER_OF_GROUP_s        *next_member;            //  $p
        struct MEMBER_OF_GROUP_s        *previous_member;        //  $p
        };
typedef struct MEMBER_OF_GROUP_s *MEMBER_OF_GROUP;
```

Node Types

Node name	Node type	Visible at PK	Has node-id
ASSEMBLY	10	Yes	No
INSTANCE	11	Yes	Yes
BODY	12	Yes	No
SHELL	13	Yes	Yes
FACE	14	Yes	Yes
LOOP	15	Yes	Yes
EDGE	16	Yes	Yes
FIN	17	Yes	No
VERTEX	18	Yes	Yes
REGION	19	Yes	Yes
POINT	29	Yes	Yes
LINE	30	Yes	Yes
CIRCLE	31	Yes	Yes
ELLIPSE	32	Yes	Yes
INTERSECTION	38	Yes	Yes
CHART	40	No	
LIMIT	41	No	
BSPLINE_VERTICES	45	No	
PLANE	50	Yes	Yes
CYLINDER	51	Yes	Yes
CONE	52	Yes	Yes
SPHERE	53	Yes	Yes
TORUS	54	Yes	Yes
BLENDED_EDGE	56	Yes	Yes
BLEND_BOUND	59	No	
OFFSET_SURF	60	Yes	Yes

SWEPT_SURF	67	Yes	Yes
SPUN_SURF	68	Yes	Yes
LIST	70	Yes	Yes
POINTER_LIS_BLOCK	74	No	
ATT_DEF_ID	79	No	
ATTRIB_DEF	80	Yes	No
ATTRIBUTE	81	Yes	Yes
INT_VALUES	82	No	
REAL_VALUES	83	No	
CHAR_VALUES	84	No	
POINT_VALUES	85	No	
VECTOR_VALUES	86	No	
AXIS_VALUES	87	No	
TAG_VALUES	88	No	
DIRECTION_VALUES	89	No	
GROUP	90	Yes	Yes
MEMBER_OF_GROUP	91	No	
UNICODE_VALUES	98	No	
FIELD_NAMES	99	No	
TRANSFORM	100	Yes	Yes
WORLD	101	No	
KEY	102	No	
PE_SURF	120	Yes	Yes
INT_PE_DATA	121	No	
EXT_PE_DATA	122	No	
B_SURFACE	124	Yes	Yes
SURFACE_DATA	125	No	
NURBS_SURF	126	No	

KNOT_MULT	127	No	
KNOT_SET	128	No	
PE_CURVE	130	Yes	Yes
TRIMMED_CURVE	133	Yes	Yes
B_CURVE	134	Yes	Yes
CURVE_DATA	135	No	
NURBS_CURVE	136	No	
SP_CURVE	137	Yes	Yes
GEOMETRIC_OWNER	141	No	
HELIX_CU_FORM	163	No	
HELIX_SU_FORM	184	No	

Node Classes

Node class name	**Node class**
GEOMETRY	1003
PART	1005
SURFACE	1006
SURFACE_OWNER	1007
CURVE	1008
CURVE_OWNER	1010
POINT_OWNER	1011
LIS_BLOCK	1012
LIST_OWNER	1013
ATTRIBUTE_OWNER	1015
GROUP_OWNER	1016
GROUP_MEMBER	1017
FIELD_VALUES	1018
ATTRIB_GROUP	1019
TRANSFORM_OWNER	1023
PE_DATA	1027
PE_INT_GEOM	1028
SHELL_OR_BODY	1029
FIELD_NAME	1037

System Attribute Definitions

All system attribute definitions are of class 1.

Hatching

Identifier	SDL/TYSA_HATCHING	
Type_id	8003	
Entity types	face	
Fields	real	real 1
		real 2
		real 3
		real 4
	integer	Hatching type
Set by	Application	
Used by	Parasolid hidden line and wireframe images	

For **planar hatching** - the four real values define the hatch orientation as a vector and a spacing between consecutive planes.

For **radial hatching** - the first three real values define the spacing of the hatch lines. The fourth value is not used.

For **parametric hatching** - the first two real values define the spacing in *u* and *v* respectively. The last two values are not used.

Planar Hatch

Identifier	SDL/TYSA_PLANAR_HATCH		
Type_id	8021		
Entity types	face		
Fields	real	x component	‘direction’ or plane normal
		y component	
		z component	
		‘pitch’ or separation	
		x component	position vector
		y component	
		z component	
Set by	Application		
Used by	Parasolid hidden line and wireframe images		

For planar hatching, an attribute with this definition takes precedence over an attribute with the SDL/TYSA_HATCHING definition, if a face has both types of attribute attached.

Radial Hatch

Identifier	SDL/TYSA_RADIAL_HATCH	
Type_id	8027	
Entity types	face	
Fields	real	radial around
		radial along
		radial about
		radial around start
		radial along start
		radial about start
Set by	Application	
Used by	Parasolid hidden line and wireframe images	

For radial hatching, an attribute with this definition takes precedence over an attribute with the SDL/TYSA_HATCHING definition, if a face has both types of attribute attached.

Parametric Hatch

Identifier	SDL/TYSA_PARAM_HATCH	
Type_id	8028	
Entity types	face	
Fields	real	u spacing
		v spacing
		u start
		v start
Set by	Application	
Used by	Parasolid hidden line and wireframe images	

For parametric hatching, an attribute with this definition takes precedence over an attribute with the SDL/TYSA_HATCHING definition, if a face has both types of attribute attached.

Density Attributes

There are density attributes for each of regions, faces, edges and vertices in addition to the system attribute for density of a body.

The region/face/edge/vertex attributes will be taken into account when finding the mass, centre of gravity and moment of inertia of a body or of the entity to which the attribute is attached:

- The mass of a region will not include that of any of its faces or edges, and the same applies to faces and edges and their boundaries.
- A void region will always have zero mass whatever its density and a solid region will inherit its density from the body if it does not have a density of its own.
- The default density for faces, edges and vertices is always zero.

Density (of a body)

Identifier	SDL/TYSA_DENSITY	
Type_id	8004	
Entity types	body	
Fields	real	Density
	string	Units
Set by	Application	
Used by	Parasolid Mass Properties - calculation of mass	

A body without a density attribute is taken to have, by default, a density of 1.0.

The character field units is not used by Parasolid but it can be set and read by the application.

- ***Region Density***

Identifier	SDL/TYSA_REGION_DENSITY

Type_id	8023	
Entity types	region	
Fields	real	Density of region
	string	Units
Set by	Application	
Used by	Parasolid Mass Properties - calculation of mass	

This attribute only makes sense for solid regions; void regions always have a mass of zero.

A solid region without a density attribute is taken to have, by default, the same density as its owning body.

The character field units is not used by Parasolid but it can be set and read by the user.

Face Density

Identifier	SDL/TYSA_FACE_DENSITY	
Type_id	8024	
Entity types	face	
Fields	real	Density of face
	string	Units
Set by	Application	
Used by	Parasolid Mass Properties - calculation of mass	

The value of this attribute is treated as a mass per unit area.

A mass will be calculated for a face only when a face possesses this attribute. In all other cases the mass of a face is not defined.

The character field units is not used by Parasolid but it can be set and read by the user.

- ### *Edge Density*

Identifier	SDL/TYSA_EDGE_DENSITY	
Type_id	8025	
Entity types	edge	
Fields	real	Density of edge
	string	Units
Set by	Application	
Used by	Parasolid Mass Properties - calculation of mass	

The value of this attribute is treated as a mass per unit length.

A mass will be calculated for an edge only when an edge possesses this attribute. In all other cases the mass of an edge is not defined.

The character field units is not used by Parasolid but it can be set and read by the user.

Vertex Density

Identifier	SDL/TYSA_VERTEX_DENSITY

Type_id	8026	
Entity types	vertex	
Fields	real	Mass of vertex
	string	Units
Set by	Application	
Used by	Parasolid Mass Properties - calculation of mass	

The value of this attribute is treated as a point mass.

A mass will be calculated for a vertex only when a vertex possesses this attribute. In all other cases the mass of a vertex is not defined.

The character field units is not used by Parasolid but it can be set and read by the user.

Region

Identifier	SDL/TYSA_REGION	
Type_id	8013	
Entity types	face	
Fields	string	Unused
Set by	Application	
Used by	Parasolid hidden line images	

Regional data will allow the application to analyze a hidden-line picture for distinct regions in the 2D view.

Colour

Identifier	SDL/TYSA_COLOUR		
Token	8001		
Entity types	face edge		
Fields	real	Red value	These three values should be in the range 0.0 to 1.0
		Green value	
		Blue value	
Set by	Application		
Used by	Application		

Reflectivity

Identifier	SDL/TYSA_REFLECTIVITY	
Token	8014	
Entity types	face	
Fields	real	Coefficient of specular reflection
		Proportion of colored light in highlights
		Coefficient of diffuse reflection
		Coefficient of ambient reflection
	integer	Reflection power
Set by	Application	
Used by	Application	

The attribute types for Reflectivity and Translucency are also used by the Parasolid routine RRPIXL, but the use of this routine is not recommended.

- ## Translucency

Identifier	SDL/TYSA_TRANSLUCENCY		
Token	8015		
Entity types	face		
Fields	real	Transparency coefficient	range 0.0 to 1.0, where 0 is opaque and 1 is transparent
Set by	Application		
Used by	Application		

Name

Identifier	SDL/TYSA_NAME	
Token	8017	
Entity types	assembly, body, instance, shell, face, loop, edge, vertex, group, surface, curve, point	
Fields	string	Name of entity
Set by	Application	
Used by	Application	

Entities read into Parasolid from a Romulus 6.0 transmit file have their names held in name attributes. Only entities to which the user has given names will be treated in this way.

Incremental faceting

Identifier	SDL/TYSA_INCREMENTAL_FACETTING 8934

Fields	string	Unused
Set by	Application	
Used by	Parasolid modeling operations	

If an edge has an attribute of this definition attached, it indicates that the edge should not be merged in any modelling operations.

Group merge behavior

Identifier	SDL/TYSA_GROUP_MERGE	
Token	TYSAGM	
Entity types	group	
Fields	string	Unused
Set by	Application	
Used by	Parasolid modeling operations	

If a group has an attribute of this definition attached, it indicates that alternative behavior should be used if an entity in the group is merged with an entity not in that group.