NOTICE: All information contained herein is the property of UGS Corp. No part of this publication (whether in hardcopy or electronic form) may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of UGS Corp. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement. UGS, Parasolid and Transforming the process of innovation are registered trademarks or trademarks of UGS Corp. in the United States and/or other countries. All other trademarks are the property of their respective owners.

This publication and the information herein are furnished AS IS, are furnished for informational use only, are subject to change without notice, and should not be construed as a commitment by UGS Corp. UGS Corp. EXPRESSLY DISCLAIMS AND ASSUMES NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR INACCURACIES THAT MAY APPEAR IN THE INFORMATIONAL CONTENT CONTAINED IN THIS GUIDE, MAKES NO WARRANTY OF ANY KIND (EXPRESS, IMPLIED, OR STATUTORY) WITH RESPECT TO THIS PUBLICATION, AND EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSES, AND NONINFRINGEMENT OF THIRD-PARTY RIGHTS.

This document is subject to all United States government laws, regulations, orders or other restrictions regarding export from the United States of services, commodities, software, technology or derivatives thereof, as such laws, regulations, orders, or other restrictions may be enacted, amended or modified from time to time. Notwithstanding anything to the contrary in this document, You will not directly or indirectly, separately or as part of a system, export or reexport any UGS Corp. services, commodity, software, technology or derivatives thereof or permit the use by or shipment of same to: (i) a national or resident of Cuba, Iran, North Korea, Sudan, Syria, or any other country embargoed or restricted by the United States; (ii) anyone or any entity on the U.S. Treasury Department's List of Specially Designated Nationals and Blocked Persons, List of Specially Designated Terrorists or List of Specially Designated Narcotics Traffickers, or the U.S. Commerce Department's Denied Parties List or the U.S. Commerce Department's Entity List; or (iii) any country or destination for which the United States government or a United States governmental agency requires an export license or other approval for export without first having obtained such license or other approval. You acknowledge and agree that, unless a validated export license is obtained from the United States Department of Commerce or other applicable authority where required, You will not use the UGS Corp. services, commodities, software, technology or derivatives thereof in the design, development, production, stockpiling or use of nuclear weapons, missiles, or chemical or biological weapons. You agree to indemnify and hold UGS Corp. harmless from and against all claims, losses, damages and expenses arising out of or resulting from Your failure to comply with the provisions set forth in this Section.
# Table of Contents

Introduction to the Parasolid XT Format ................................................... 1  
Types of File Documented ........................................................................ 1  
Text and Binary Formats........................................................................... 2  
Standard File Names and Extensions....................................................... 2  
The Alternative Solution............................................................................ 2  
Logical Layout........................................................................................... 3  
Schema..................................................................................................... 4  
Embedded schemas ................................................................................. 5  
  Physical layout .......................................................................................... 5  
  XT format .................................................................................................. 6  
Space compression ................................................................................... 7  
Field types................................................................................................. 7  
Point.......................................................................................................... 8  
Pointer classes.......................................................................................... 9  
Variable-length nodes ............................................................................... 9  
Unresolved indices.................................................................................. 10  
Simple example....................................................................................... 10  
Physical Layout....................................................................................... 12  
Common header ..................................................................................... 12  
  Keyword Syntax .......................................................................................... 14  
Text ......................................................................................................... 15  
Binary...................................................................................................... 16  
  bare binary.................................................................................................. 16  
  neutral binary............................................................................................ 16  
Model Structure....................................................................................... 18
# Parasolid XT Format Reference

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology</td>
<td>18</td>
</tr>
<tr>
<td>General points</td>
<td>18</td>
</tr>
<tr>
<td>Entity definitions</td>
<td>18</td>
</tr>
<tr>
<td>Assembly</td>
<td>18</td>
</tr>
<tr>
<td>Instance</td>
<td>19</td>
</tr>
<tr>
<td>Body</td>
<td>19</td>
</tr>
<tr>
<td>Region</td>
<td>19</td>
</tr>
<tr>
<td>Shell</td>
<td>20</td>
</tr>
<tr>
<td>Face</td>
<td>20</td>
</tr>
<tr>
<td>Loop</td>
<td>21</td>
</tr>
<tr>
<td>Fin</td>
<td>21</td>
</tr>
<tr>
<td>Edge</td>
<td>22</td>
</tr>
<tr>
<td>Vertex</td>
<td>22</td>
</tr>
<tr>
<td>Attributes</td>
<td>22</td>
</tr>
<tr>
<td>Groups</td>
<td>23</td>
</tr>
<tr>
<td>Node-ids</td>
<td>23</td>
</tr>
<tr>
<td>Entity matrix</td>
<td>23</td>
</tr>
<tr>
<td>Representation of manifold bodies</td>
<td>24</td>
</tr>
<tr>
<td>Body types</td>
<td>24</td>
</tr>
<tr>
<td>Schema Definition</td>
<td>26</td>
</tr>
<tr>
<td>Underlying types</td>
<td>26</td>
</tr>
<tr>
<td>Geometry</td>
<td>27</td>
</tr>
<tr>
<td>Curves</td>
<td>28</td>
</tr>
<tr>
<td>LINE</td>
<td>29</td>
</tr>
<tr>
<td>CIRCLE</td>
<td>30</td>
</tr>
<tr>
<td>ELLIPSE</td>
<td>32</td>
</tr>
<tr>
<td>B_CURVE (B-spline curve)</td>
<td>33</td>
</tr>
</tbody>
</table>
INTERSECTION ........................................................................................................... 42
TRIMMED_CURVE .................................................................................................... 46
PE_CURVE (Foreign Geometry curve) ........................................................................ 48
SP_CURVE ................................................................................................................ 50
Surfaces .......................................................................................................................... 51
PLANE .......................................................................................................................... 52
CYLINDER .................................................................................................................. 53
CONE ............................................................................................................................ 55
SPHERE ....................................................................................................................... 56
TORUS .......................................................................................................................... 58
BLENDED_EDGE (Rolling Ball Blend) ......................................................................... 59
BLEND_BOUND (Blend boundary surface) ................................................................. 62
OFFSET_SURF .......................................................................................................... 63
B_SURFACE ............................................................................................................... 65
SWEPT_SURF ............................................................................................................. 71
SPUN_SURF ............................................................................................................... 72
PE_SURF (Foreign Geometry surface) ........................................................................ 74
Point .............................................................................................................................. 75
Transform ..................................................................................................................... 76
Curve and Surface Senses ........................................................................................... 78
Geometric_owner ......................................................................................................... 78
Topology ....................................................................................................................... 80
WORLD ......................................................................................................................... 80
ASSEMBLY ................................................................................................................... 81
INSTANCE .................................................................................................................... 83
BODY .......................................................................................................................... 85
REGION ....................................................................................................................... 88
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hatching</td>
<td>116</td>
</tr>
<tr>
<td>Planar Hatch</td>
<td>117</td>
</tr>
<tr>
<td>Radial Hatch</td>
<td>118</td>
</tr>
<tr>
<td>Parametric Hatch</td>
<td>118</td>
</tr>
<tr>
<td>Density Attributes</td>
<td>119</td>
</tr>
<tr>
<td>Density (of a body)</td>
<td>119</td>
</tr>
<tr>
<td>Region Density</td>
<td>119</td>
</tr>
<tr>
<td>Face Density</td>
<td>120</td>
</tr>
<tr>
<td>Edge Density</td>
<td>120</td>
</tr>
<tr>
<td>Vertex Density</td>
<td>121</td>
</tr>
<tr>
<td>Region</td>
<td>121</td>
</tr>
<tr>
<td>Colour</td>
<td>122</td>
</tr>
<tr>
<td>Reflectivity</td>
<td>122</td>
</tr>
<tr>
<td>Translucency</td>
<td>122</td>
</tr>
<tr>
<td>Name</td>
<td>123</td>
</tr>
</tbody>
</table>
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. The main audience for this manual is people who intend to write translators from or to the Parasolid transmit format.

Reading and writing transmit files are significantly different problems. Reading is simply a question of traversing the transmit file and interpreting the records stored within it. Writing is a significantly harder process; as well as getting the data format of the transmit file correct applications must also ensure that the many complex and subtle inter-relationships between the geometric nodes in the file are satisfied.

The presentation of material in this manual is structured to help the construction of applications that perform read operations. It is strongly advised that the construction of applications that write files is only attempted when a copy of Parasolid is available during the development process to check the consistency and validity of files being produced.

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 three different formats:

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

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.

**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:


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

**The Alternative Solution**
An alternative solution for reading and writing XT data is to license the Parasolid software, which is available in Designer, Editor, Communicator and Educator packages. For further details on these packages, and contact information, visit the Parasolid website at http://www.parasolid.com/.
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:

<table>
<thead>
<tr>
<th>Contents of file</th>
<th>Type of root node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body</td>
<td>BODY</td>
</tr>
<tr>
<td>Assembly</td>
<td>ASSEMBLY</td>
</tr>
<tr>
<td>Array of parts</td>
<td>POINTER_LIS_BLOCK</td>
</tr>
<tr>
<td>Partition</td>
<td>WORLD</td>
</tr>
</tbody>
</table>

**Schema**

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

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

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

e.g.

29 POINT; Point; 1 6 0

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

Field specifiers have the format:

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

e.g.

owner; p; 1 1011 1

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

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

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

**Embedded schemas**

When reading a part, partition, or delta, Parasolid converts any data that it encounters from older versions of Parasolid to the current format using a mixture of automatic table conversion (driven by the appropriate schemas), and explicit code for more complex algorithms.

However, backwards compatibility of file information – that is, reading data created by a newer version of Parasolid into an application (such as data created by a subcontractor) – can never be guaranteed to work using this method, because the older version does not contain any special-case conversion code.

From Parasolid V14 onwards, parts, partitions and deltas can be transmitted with extra information that is intended to replace the schema normally loaded to describe the data layout. This information contains the **differences** between its schema and a defined base schema (currently V13’s SCH_13006).

This enables parts, partitions, and deltas to be successfully read into older versions of Parasolid without loss of information.

Snapshots (see PK_SESSION_transmit) are not affected.

**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 only relevant to applications that attempt to read or write Parasolid data directly rather than via a Parasolid image. Two important elements are

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

- **positive integers**
  These are transmitted similarly to the pointer indices which link individual objects together, i.e., small values 0..32766 are transmitted as a single **short** integer, larger ones encoded into two.
**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, where an effective field is either transmittable (xmt_code == 1) or variable-length (n_elts == 1).

- If the nodetype does not exist in the base schema then it is output as follows:
  - number of fields (byte)
  - name and description (short strings)
  - fields one by one as

<table>
<thead>
<tr>
<th>name</th>
<th>short string</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr_class</td>
<td>Short</td>
</tr>
<tr>
<td>n_elts</td>
<td>Positive integer</td>
</tr>
<tr>
<td>type</td>
<td>short string</td>
</tr>
<tr>
<td>xmt_code</td>
<td>logical (byte)</td>
</tr>
</tbody>
</table>

- 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).

- Otherwise 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 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) and advance to the next base and current fields;
• If the base field does not match any unprocessed current field, output 'D' (char) and advance to the next base field;
• Output 'I' (char), 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 full list of field types used in transmit files is as follows:

u   unsigned byte 0-255

char

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

typedef char logical;

short int

w   unicode character, output as a short int

int

pointer-index

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

double

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-space 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

```c
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:

**ABCDEFHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz***********
* **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
12 112 0 2 0 0 0 1 e3 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 1 1 0 0 plane
31 5 10 0 7 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 FFFFTFTFFFTFFFTFFFTF2 attrib_def (variable 1)
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, ...
```
The ‘part 1’ data is ‘standard’ information which should be accessible to the Frustrum (e.g. by operating system calls). It is the responsibility of the Frustrum to gather the relevant information and to format it as described in this specification.

- part 2 data: a sequence of keyword-value pairs, separated by semicolons.

  SCH = SCH_m_n

  // name of schema key e.g.SCH_1200000_12006

  USFLD_SIZE = m

  // length of user field (0 - 16 integer words)

  Applications writing XT files must use a schema name of SCH_1200000_12006

- part 3 data: non-standard information, which is only comprehensible to the Frustrum which wrote it.

The ‘part 3’ data is non-standard information, which is only comprehensible to the Frustrum which wrote it. However, other Frustrum implementations must be able to parse it (in order to reach the end of the header), and it should therefore conform to
the same keyword/value syntax as for ‘part 1’ and ‘part 2’ data. However, the choice and interpretation of keywords for the ‘part 3’ data is entirely at the discretion of the Frustrum which is writing the header.

- a trailer record.

An example is:

**ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz***********
*

**PARASOLID !"#$%&'()*+,-./:;<=>?@\]^_`{|}~0123456789******************

**PART1;MC=vax;MC_MODEL=4090;MC_ID=VAX14;OS=vms;OS_RELEASE=V6.2;FRU=sdl_parasolid_test_vax;APPL=unknown;SITE=sdl-cambridge u.k.;USER=ALANS;FORMAT=text;GUISE=transmit;KEY=temp;FILE=TEMP.XMT_X;DATE=8-sep-1997;

**PART2;SCH=SCH_701169_7007;USFLD_SIZE=0;

**PART3;

**END_OF_HEADER***************************************************

**Keyword Syntax**

All keyword definitions which appear in the three parts of data are written in the form

```
<name>=<value> e.g. MC=hppa;MC_MODEL=9000/710;
```

where

- `<name>` consists of 1 to 80 uppercase, digit, or underscore characters
- `<value>` consists of 1 or more ASCII printing characters (except space)

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 appear to appear in a keyword value:

<table>
<thead>
<tr>
<th>Character</th>
<th>Escape sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>newline</td>
<td>^n</td>
</tr>
</tbody>
</table>
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
```
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 two types of binary file: `bare' 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.

**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.

**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:

```c
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:

```c
short q = 0, r;
ip_short( &r );
if (r < 0) {
    // case: big index
    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.

<table>
<thead>
<tr>
<th></th>
<th>Body</th>
<th>Region</th>
<th>Shell</th>
<th>Face</th>
<th>Loop</th>
<th>Fin</th>
<th>Edge</th>
<th>Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body</td>
<td>-</td>
<td>&gt;0</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>Region</td>
<td>1</td>
<td>-</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>Shell</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>-------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>Face</td>
<td>1</td>
<td>1-2</td>
<td>1-2</td>
<td>-</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>Loop</td>
<td>1</td>
<td>1-2</td>
<td>1-2</td>
<td>1</td>
<td>-</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>Fin</td>
<td>1</td>
<td>1-2</td>
<td>1-2</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>0-2</td>
</tr>
<tr>
<td>Edge</td>
<td>1</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>-</td>
</tr>
<tr>
<td>Vertex</td>
<td>1</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
</tr>
</tbody>
</table>

**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_s *member_of_group;
 };
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:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>Integer value unique to curve in part</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>Attributes and groups associated with curve</td>
</tr>
<tr>
<td>owner</td>
<td>pointer0</td>
<td>topological owner</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>next curve in geometry chain</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>previous curve in geometry chain</td>
</tr>
<tr>
<td>geometric_owner</td>
<td>pointer0</td>
<td>geometric owner node</td>
</tr>
<tr>
<td>sense</td>
<td>char</td>
<td>sense of curve: ‘+’ or ‘-’ (see end of Geometry section)</td>
</tr>
</tbody>
</table>
struct ANY_CURVE_s // Any Curve
{
    int node_id; // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union CURVE_OWNER_u owner; // $p
    union CURVE_u next; // $p
    union CURVE_u previous; // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char sense; // $c
};

typedef struct ANY_CURVE_s *ANY_CURVE;

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_s *geometric_owner; // $p
    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.
<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>centre</td>
<td>vector</td>
<td>Centre of circle</td>
</tr>
<tr>
<td>normal</td>
<td>vector</td>
<td>Normal to the plane containing the circle (a unit vector)</td>
</tr>
<tr>
<td>x_axis</td>
<td>vector</td>
<td>X axis in the plane of the circle (a unit vector)</td>
</tr>
<tr>
<td>radius</td>
<td>double</td>
<td>Radius of circle</td>
</tr>
</tbody>
</table>

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

```c
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
```
ELLIPSE
An ellipse has a parametric representation of the form
\[ R(t) = C + aX \cos(t) + bY \sin(t) \]
where
- \( C \) is the centre of the circle
- \( X \) is the major axis
- \( r \) is the major radius
- \( Y \) and \( b \) are the minor axis and minor radius respectively.

- \( Y \) and \( b \) are the minor axis and minor radius respectively.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>centre</td>
<td>Vector</td>
<td>Centre of ellipse</td>
</tr>
<tr>
<td>normal</td>
<td>Vector</td>
<td>Normal to the plane containing the ellipse (a unit vector)</td>
</tr>
<tr>
<td>x_axis</td>
<td>Vector</td>
<td>major axis in the plane of the ellipse (a unit vector)</td>
</tr>
<tr>
<td>major_radius</td>
<td>Double</td>
<td>major radius</td>
</tr>
<tr>
<td>minor_radius</td>
<td>Double</td>
<td>minor radius</td>
</tr>
</tbody>
</table>
The minor axis (Y) in the definition above is the vector cross product of the normal and x_axis.

struct ELLIPSE_s == ANY_CURVE_s // Ellipse
{
    int node_id; // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union CURVE_OWNER_u owner; // $p
    union CURVE_u next; // $p
    union CURVE_u previous; // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    vector centre; // $v
    vector normal; // $v
    vector x_axis; // $v
    double major_radius; // $f
    double minor_radius; // $f
};
typedef struct ELLIPSE_s *ELLIPSE;

B_CURVE (B-spline curve)
Parasolid supports B spline curves in full NURBS format. The mathematical description of these curves is:

- Non Uniform Rational B-splines as (NURBS)

\[ P(t) = \frac{\sum_{i=0}^{n-1} b_i(t)w_i V_i}{\sum_{i=0}^{n-1} b_i(t)w_i} \]

- and the more simple Non Uniform B-spline

\[ P(t) = \sum_{i=0}^{n-1} b_i(t) V_i \]
Where:

\[ n = \text{number of vertices (n_vertices in the PK standard form)} \]
\[ V_0 \ldots V_{n-1} \text{ are the B-spline vertices} \]
\[ w_0 \ldots w_{n-1} \text{ are the weights} \]
\[ b_i(t), i = 0 \ldots n-1 \text{ are the B-spline basis functions} \]

**KNOT VECTORS**

The parameter \( t \) above is global. The user supplies an ordered set of values of \( t \) at specific points. The points are called knots and the set of values of \( t \) is called the knot vector. Each successive value in the set must be greater than or equal to its predecessor. Where two or more such values are the same we say that the knots are coincident, or that the knot has multiplicity greater than 1. In this case it is best to think of the knot set as containing a null or zero length span. The principal use of coincident knots is to allow the curve to have less continuity at that point than is formally required for a spline. A curve with a knot of multiplicity equal to its \textit{degree} can have a discontinuity of first derivative and hence of tangent direction. This is the highest permitted multiplicity except at the first or last knot where it can go as high as \((\text{degree} + 1)\).

In order to avoid problems associated, for example with rounding errors in the knot set, Parasolid stores an array of distinct values and an array of integer multiplicities. This is reflected in the standard form used by the PK for input and output of B-curve data.

Most algorithms in the literature, and the following discussion refer to the expanded knot set in which a knot of multiplicity \( n \) appears explicitly \( n \) times.

**THE NUMBER OF KNOTS AND VERTICES**

The knot set determines a set of basis functions which are bell shaped, and non zero over a span of \((\text{degree} + 1)\) intervals. One basis function starts at each knot, and each one finishes \((\text{degree} + 1)\) 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 \( n_{\text{knots}} = n_{\text{vertices}} + \text{degree} + 1 \)

**THE VALID RANGE OF THE B-CURVE**

So if the knot set is numbered \( \{t_0 \ldots t_{n_{\text{knots}} - 1}\} \) it can be seen then that it is only after \( t_{\text{degree}} \) that sufficient \((\text{degree} + 1)\) basis functions are present for the curve to be fully defined, and that the B-curve ceases to be fully defined after \( t_{n_{\text{knots}} - 1} - \text{degree} \).
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 \( t_{\text{degree}} \) to \( t_{n \text{, knots} - 1 - \text{degree}} \) then the difference between these values is called the period and the curve can continue to be evaluated with the same point reoccurring every period.

The minimal smoothness requirement for periodic curves in Parasolid is tangent continuity, but we strongly recommend \( C_{\text{degree}-1} \), or continuity in the \((\text{degree}-1)^{\text{th}}\) derivative. This in turn is best achieved by repeating the first *degree* vertices at the end, and by matching knot intervals so that counting from the start of the defined range, \( t_{\text{degree}} \), the first *degree* intervals between knots match the last *degree* intervals, and similarly matching the last *degree* knot intervals before the end of the defined range to the first *degree* intervals.

**CLOSED B-CURVES**

A periodic B-curve must also be closed, but is permitted to have a closed B-curve that is not periodic.

In this case the rules for continuity are relaxed so that only \( C_0 \) or positional continuity is required between the start and end. Such closed non-periodic curves are not able to be attached to topology.

**RATIONAL B-CURVE**

In the rational form of the curve, each vertex is associated with a weight, which increases or decreases the effect of the vertex without changing the curve hull. To ensure that the convex hull property is retained, the curve equation is divided by a denominator which makes the coefficients of the vertices sum to one.

\[
P(t) = \frac{\sum_{i=0}^{n-1} w_i \times P_i}{\sum_{i=0}^{n-1} w_i}
\]

Where \( w_0 \ldots w_{n-1} \) are weights.

Each weight may take any positive value, and the larger the value, the greater the effect of the associated vertex. However, it is the relative sizes of the weights which is
important, as may be seen from the fact that in the equation given above, all the weights may be multiplied by a constant without changing the equation.

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

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

For attachment to topology a B-surface is required to have $G_1$ continuity. That is to say that the surface normal direction must be continuous.

Parasolid does not support modelling with surfaces that are self-intersecting or contain cusps. Although they can be created they are not permitted to be attached to topology.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nurbs</td>
<td>Pointer</td>
<td>Geometric definition</td>
</tr>
<tr>
<td>data</td>
<td>Pointer0</td>
<td>Auxiliary information</td>
</tr>
</tbody>
</table>

```
struct B_CURVE_s == ANY_CURVE_s                          // B curve
{
    int          node_id;                  // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union CURVE_OWNER_u owner;             // $p
    union CURVE_u next;                   // $p
    union CURVE_u previous;               // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char sense;                           // $c
    struct NURBS_CURVE_s *nurbs;         // $p
```

- 37 -
struct CURVE_DATA_s *data; // $p
}

typedef struct B_CURVE_s *B_CURVE;

The data stored in an XT file for a NURBS_CURVE is

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>Short</td>
<td>degree of the curve</td>
</tr>
<tr>
<td>n_vertices</td>
<td>Int</td>
<td>number of control vertices (‘poles’)</td>
</tr>
<tr>
<td>vertex_dim</td>
<td>Short</td>
<td>dimension of control vertices</td>
</tr>
<tr>
<td>n_knots</td>
<td>Int</td>
<td>number of distinct knots</td>
</tr>
<tr>
<td>knot_type</td>
<td>Byte</td>
<td>form of knot vector</td>
</tr>
<tr>
<td>periodic</td>
<td>Logical</td>
<td>true if curve is periodic</td>
</tr>
<tr>
<td>closed</td>
<td>Logical</td>
<td>true if curve is closed</td>
</tr>
<tr>
<td>rational</td>
<td>Logical</td>
<td>true if curve is rational</td>
</tr>
<tr>
<td>curve_form</td>
<td>Byte</td>
<td>shape of curve, if special</td>
</tr>
<tr>
<td>bspline_vertices</td>
<td>Pointer</td>
<td>control vertices node</td>
</tr>
<tr>
<td>knot_mult</td>
<td>Pointer</td>
<td>knot multiplicities node</td>
</tr>
<tr>
<td>knots</td>
<td>Pointer</td>
<td>knots node</td>
</tr>
</tbody>
</table>

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_piecewiseBezier = 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
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.

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

typedef struct KNOT_SET_s *KNOT_SET;

and

typedef struct KNOT_MULT_s *KNOT_MULT;

The data stored in an XT file for a CURVE_DATA node is:

typedef enum
Parasolid XT Format Reference

{  
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.
<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface</td>
<td>pointer array [2]</td>
<td>Surfaces of intersection curve</td>
</tr>
<tr>
<td>chart</td>
<td>Pointer</td>
<td>array of hvecs on the curve – see below</td>
</tr>
<tr>
<td>start</td>
<td>Pointer</td>
<td>start limit of the curve</td>
</tr>
<tr>
<td>end</td>
<td>Pointer</td>
<td>end limit of the curve</td>
</tr>
</tbody>
</table>

```
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  *ge ometric_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

- 43 -
double t; // curve parameter
}

where

- \textit{pvec} is a point common to both surfaces
- \textit{u[]} and \textit{v[]} are the \textit{u} and \textit{v} parameters of the \textit{pvec} on each of the surfaces.
- \textit{tangent} is the tangent to the curve at \textit{pvec}. This will be equal to the (normalised) vector cross product of the surface normals at \textit{pvec}, when this cross product is non-zero. These surface normals take account of the surface sense fields.

Note that only the \textit{pvec} part of an \textit{hvec} is actually transmitted.

The chart data structure essentially describes a piecewise-linear (chordal) approximation to the true curve. As well as containing the ordered array of \textit{hvecs} defining this approximation, it contains extra information pertaining to the accuracy of the approximation:

```c
struct CHART_s                               // Chart
{
  double Base_parameter;                      // $f
  double Base_scale;                          // $f
  int Chart_count;                            // $d
  double Chordal_error;                      // $f
  double Angular_error;                      // $f
  double Parameter_error[2];                 // $f[2]
  hvec Hvec[ 1 ];                            // $h[
};
```

where

- \textit{base\_parameter} is the parameter of the first \textit{hvec} in the chart
- \textit{base\_scale} determines the scale of the parameterisation (see below)
- \textit{chart\_count} is the length of the \textit{hvec} array
- \textit{chordal\_error} is an estimate of the maximum deviation of the curve from the piecewise-linear approximation given by the \textit{hvec} array. It may be null.
- \textit{angular\_error} is the maximum angle between the tangents of two sequential \textit{hvecs}. It may be null.
The limits of the intersection curve are stored in the following data structure:

```c
struct LIMIT_s                          // Limit
{
    char                    type;             // $c
    hvec                    hvec[ 1 ];        // $h[]
};
```

The 'type' field may take one of the following values

- const char SCH_help                = 'H';    // help hvec
- const char SCH_terminator          = 'T';    // terminator
- const char SCH_limit               = 'L';    // arbitrary limit
- const char SCH_boundary            = 'B';    // spine boundary

The length of the hvec array depends on the type of the limit.

- a SCH_help limit is an arbitrary point on a closed intersection curve. There will be one hvec in the hvec array, locating the curve.
- a SCH_terminator limit is a point where one of the surface normals is degenerate, or where their cross-product is zero. Typically, there will be more than one branch of intersection between the two surfaces at these singularities. There will be two values in the hvec array. The first will be the exact position of the singularity, and the second will be a point on the curve a small distance away from the terminator. This ‘branch point’ identifies which branch relates to the curve in question. The branch point is the one which appears in the chart, at the corresponding end – so the singularity lies just outside the parameter range of the chart.
- a SCH_limit limit is an artificial boundary of an intersection curve on an otherwise potentially infinite branch. The single hvec describes the end of the curve.
- a SCH_boundary limit is used to describe the end of a degenerate rolling-ball blend. It is not relevant to intersection curves.

The parameterization of the curve is given as follows. If the chart points are $P_i$, $i = 0$ to $n$, with parameters $t_i$, and natural tangent vectors $T_i$, then define

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

$$\cos(a_i) = T_i \cdot (P_{i+1} - P_i)$$
\[ \cos(b_i) = T_i \cdot (P_i - P_{i-1}) \]

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 \( f_0 = \text{base\_scale} \)

\[ f_i = \left( \frac{\cos(b_i)}{\cos(a_i)} \right) f_{i-1} \]

Then \( t_0 = \text{base\_parameter} \)

\[ t_i = t_{i-1} + C_{i-1} f_{i-1} \]

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.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basis_curve</td>
<td>pointer</td>
<td>Basis curve</td>
</tr>
<tr>
<td>point_1</td>
<td>vector</td>
<td>start of trimmed portion</td>
</tr>
<tr>
<td>point_2</td>
<td>vector</td>
<td>end of trimmed portion</td>
</tr>
<tr>
<td>parm_1</td>
<td>double</td>
<td>parameter on basis curve corresponding to point_1</td>
</tr>
<tr>
<td>parm_2</td>
<td>double</td>
<td>parameter on basis curve corresponding to point_2</td>
</tr>
</tbody>
</table>

```c
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.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>char</td>
<td>whether internal or external</td>
</tr>
<tr>
<td>data</td>
<td>pointer</td>
<td>internal or external data</td>
</tr>
<tr>
<td>tf</td>
<td>pointer0</td>
<td>transform applied to geometry</td>
</tr>
<tr>
<td>internal geom</td>
<td>pointer array</td>
<td>reference to other related geometry</td>
</tr>
</tbody>
</table>

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
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.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>surface</td>
<td>pointer</td>
<td>surface</td>
</tr>
<tr>
<td>b_curve</td>
<td>pointer</td>
<td>2D Bcurve</td>
</tr>
<tr>
<td>original</td>
<td>pointer0</td>
<td>not used</td>
</tr>
<tr>
<td>tolerance_to_original</td>
<td>double</td>
<td>not used</td>
</tr>
</tbody>
</table>

struct SP_CURVE_s == ANY_CURVE_s         // SP curve
{
    int                     node_id;       // $d
    union ATTRIB_GROUP_u    attributes_groups; // $p
    union CURVE_OWNER_u     owner;         // $p
}
union CURVE_u next;               // $p
union CURVE_u previous;           // $p
struct GEOMETRIC_OWNER_s *geometric_owner;   // $p
char sense;                      // $c
union SURFACE_u surface;         // $p
struct B_CURVE_s *b_curve;       // $p
union CURVE_u original;          // $p
double tolerance_to_original;    // $f
}

typedef struct SP_CURVE_s *SP_CURVE;

**Surfaces**

All surface nodes share the following common fields:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>Integer value unique to surface in part</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>Attributes and groups associated with surface</td>
</tr>
<tr>
<td>owner</td>
<td>pointer</td>
<td>topological owner</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>next surface in geometry chain</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>previous surface in geometry chain</td>
</tr>
<tr>
<td>geometric_owner</td>
<td>pointer0</td>
<td>geometric owner node</td>
</tr>
<tr>
<td>sense</td>
<td>char</td>
<td>sense of surface: ‘+’ or ‘-’ (see end of Geometry section)</td>
</tr>
</tbody>
</table>

struct ANY_SURF_s             // Any Surface
{
  int node_id;            // $d
  union ATTRIB_GROUP_u attributes_groups; // $p
  union SURFACE_OWNER_u owner; // $p
}
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 plane
- \( X \) and \( Y \) are axes in the plane.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvec</td>
<td>vector</td>
<td>point on the plane</td>
</tr>
<tr>
<td>normal</td>
<td>vector</td>
<td>normal to the plane (a unit vector)</td>
</tr>
<tr>
<td>x_axis</td>
<td>vector</td>
<td>X axis of the plane (a unit vector)</td>
</tr>
</tbody>
</table>
The Y axis in the definition above is the vector cross product of the normal and \( x_{-axis} \).

```c
struct PLANE_s == ANY_SURF_s                       // Plane
{
    int   node_id; // $d
    union ATTRIB_GROUP_u  attributes_groups; // $p
    union SURFACE_OWNER_u(owner); // $p
    union SURFACE_u(next); // $p
    union SURFACE_u(previous); // $p
    struct GEOMETRIC_OWNER_s *geometric_owner; // $p
    char   sense; // $c
    vector pvec; // $v
    vector normal; // $v
    vector x_axis; // $v
};
typedef struct PLANE_s *PLANE;

CYLINDER
A cylinder has a parametric representation of the form:
\[
R(u, v) = P + rX\cos(u) + rY\sin(u) + vA
\]
where
- P is a point on the cylinder axis
- r is the cylinder radius
- A is the cylinder axis
- X and Y are unit vectors such that A, X and Y form an orthonormal set

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvec</td>
<td>vector</td>
<td>point on the cylinder axis</td>
</tr>
<tr>
<td>axis</td>
<td>vector</td>
<td>direction of the cylinder axis (a unit vector)</td>
</tr>
<tr>
<td>radius</td>
<td>double</td>
<td>radius of cylinder</td>
</tr>
<tr>
<td>x_axis</td>
<td>vector</td>
<td>X axis of the cylinder (a unit vector)</td>
</tr>
</tbody>
</table>

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

```c
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. \( Y = A \times X \).
- \( a \) is the cone half angle.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvec</td>
<td>vector</td>
<td>point on the cone axis</td>
</tr>
<tr>
<td>axis</td>
<td>vector</td>
<td>direction of the cone axis (a unit vector)</td>
</tr>
<tr>
<td>radius</td>
<td>double</td>
<td>radius of the cone at its pvec</td>
</tr>
<tr>
<td>sin_half_angle</td>
<td>double</td>
<td>sine of the cone’s half angle</td>
</tr>
<tr>
<td>cos_half_angle</td>
<td>double</td>
<td>cosine of the cone’s half angle</td>
</tr>
<tr>
<td>x_axis</td>
<td>vector</td>
<td>X axis of the cone (a unit vector)</td>
</tr>
</tbody>
</table>
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; // $p
  GEOMETRIC_OWNER_s
    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:

\[ R( u, v ) = C + ( X\cos( u ) + Y\sin( u ) ) r\cos( v ) + r\sin( v ) \]

where

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

• $A$, $X$ and $Y$ form an orthonormal axis set.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>centre</td>
<td>vector</td>
<td>centre of the sphere</td>
</tr>
<tr>
<td>radius</td>
<td>double</td>
<td>radius of the sphere</td>
</tr>
<tr>
<td>axis</td>
<td>vector</td>
<td>$A$ axis of the sphere (a unit vector)</td>
</tr>
<tr>
<td>x_axis</td>
<td>vector</td>
<td>$X$ axis of the sphere (a unit vector)</td>
</tr>
</tbody>
</table>

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

```c
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; // $c$
```
TORUS
A torus has a parametric representation of the form
\[ R(u, v) = C + (X \cos(u) + Y \sin(u))(a + b \cos(v)) + bA \sin(v) \]
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 (\(a \leq b\), \(a > 0\))
- **Lemon** - the inner part of a self-intersecting torus (\(a < 0\), \(|a| < b\))

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

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

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>centre</td>
<td>vector</td>
<td>centre of the torus</td>
</tr>
<tr>
<td>axis</td>
<td>vector</td>
<td>axis of the torus (a unit vector)</td>
</tr>
<tr>
<td>major_radius</td>
<td>double</td>
<td>major radius</td>
</tr>
<tr>
<td>minor_radius</td>
<td>double</td>
<td>minor radius</td>
</tr>
</tbody>
</table>
The Y axis in the definition above is the vector cross product of the axis of the torus and the x_axis.

```c
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

\[ R( u, v ) = C( u ) + r X( u ) \cos( v a( u ) ) + r Y( u ) \sin( va( u ) ) \]

where

- \( C( u ) \) is the spine curve
- \( r \) is the blend radius
- 60 -

• \( \mathbf{X}(u) \) and \( \mathbf{Y}(u) \) are unit vectors such that \( \mathbf{C}'(u) \cdot \mathbf{X}(u) = \mathbf{C}'(u) \cdot \mathbf{Y}(u) = 0 \)

• \( a(u) \) is the angle subtended by points on the boundary curves at the spine

\[
\text{X, Y and } a \text{ are expressed as functions of } u, \text{ as their values change with } u.
\]

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.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>char</td>
<td>type of blend: ‘R’ or ‘E’</td>
</tr>
<tr>
<td>surface</td>
<td>pointer[2]</td>
<td>supporting surfaces (adjacent to original edge)</td>
</tr>
<tr>
<td>spine</td>
<td>pointer</td>
<td>spine of blend</td>
</tr>
<tr>
<td>range</td>
<td>double[2]</td>
<td>offsets to be applied to surfaces</td>
</tr>
<tr>
<td>thumb_weight</td>
<td>double[2]</td>
<td>always [1,1]</td>
</tr>
<tr>
<td>boundary</td>
<td>pointer[2]</td>
<td>always [0, 0]</td>
</tr>
<tr>
<td>start</td>
<td>pointer0</td>
<td>Start LIMIT in certain degenerate cases</td>
</tr>
<tr>
<td>end</td>
<td>pointer0</td>
<td>End LIMIT in certain degenerate cases</td>
</tr>
</tbody>
</table>

```c
struct BLENDED_EDGE_s == ANY_SURF_s  // Blended edge
{
    int node_id;  // $d
    union ATTRIB_GROUP_u attributes_groups;  // $p
    union SURFACE_OWNER_u owner;  // $p
    union SURFACE_u next;  // $p
```
union SURFACE_u previous; // $p
struct
  GEOMETRIC_OWNER_s
    *geometric_owner; // $p
char
  sense; // $c
char
  blend_type; // $c
union SURFACE_u
  surface[2]; // $p[2]
union CURVE_u
  spine; // $p
double
  range[2]; // $f[2]
double
  thumb_weight[2]; // $f[2]
union SURFACE_u
  boundary[2]; // $p[2]
struct LIMIT_s
  *start; // $p
struct LIMIT_s
  *end; // $p
};

typedef struct BLENDED_EDGE_s *BLENDED_EDGE;
.
The parameterisation of the blend is as follows. The u parameter is inherited from the spine, the constant u lines being circles perpendicular to the spine curve. The v parameter is zero at the blend boundary on the first surface, and one on the blend boundary on the second surface; unless the sense of the spine curve is negative, in which case it is the other way round. The v parameter is proportional to the angle around the circle.

Transmit files can contain blends of the following types:

 const char SCH_rolling_ball = 'R'; // rolling ball blend
 const char SCH_cliff_edge   = 'E'; // cliff edge blend

For rolling ball blends, the spine curve will be the intersection of the two surfaces obtained by offsetting the supporting surfaces by an amount given by the respective entry in range[]. Note that the offsets to be applied may be positive or negative, and that the sense of the surface is significant; i.e. the offset vector is the natural unit surface normal, times the range, times $-1$ if the sense is negative.

For cliff edge blends, one of the surfaces will be a blended_edge with a range of [0,0]; its spine will be the cliff edge curve, and its supporting surfaces will be the surfaces of the faces adjacent to the cliff edge. Its type will be R.
The limit fields will only be non-null if the spine curve is periodic but the edge curve being blended has terminators – for example if the spine is elliptical but the blend degenerates. In this case the two LIMIT nodes, of type ‘L’, determine the extent of the spine.

**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:
<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundary</td>
<td>short</td>
<td>index into supporting surface array</td>
</tr>
<tr>
<td>blend</td>
<td>pointer</td>
<td>corresponding blend surface</td>
</tr>
</tbody>
</table>

```c
struct BLEND_BOUND_s == ANY_SURF_s       // Blend boundary
{
    int         node_id;  // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union SURFACE_OWNER_u owner;     // $p
    union SURFACE_u next;            // $p
    union SURFACE_u previous;        // $p
    struct GEOMETRIC_OWNER_s         *geometric_owner; // $p
    char            sense;           // $c
    short           boundary;        // $n
    union SURFACE_u   blend;         // $p
};
```

typedef struct BLEND_BOUND_s *BLEND_BOUND;

The supporting surface corresponding to the blend_bound is

```
    blend_bound->blend.blended_edge->surface[1 - blend_bound->boundary].
```

**OFFSET_SURF**

An offset surface is the result of offsetting a surface a certain distance along its normal, taking into account the surface sense. It inherits the parameterization of this underlying surface.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>char</td>
<td>check status</td>
</tr>
<tr>
<td>true_offset</td>
<td>logical</td>
<td>not used</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nurbs</td>
<td>pointer</td>
<td>Geometric definition</td>
</tr>
<tr>
<td>data</td>
<td>pointer0</td>
<td>Auxiliary information</td>
</tr>
</tbody>
</table>

```c
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
<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_periodic</td>
<td>logical</td>
<td>true if surface is periodic in u parameter</td>
</tr>
<tr>
<td>v_periodic</td>
<td>logical</td>
<td>true if surface is periodic in v parameter</td>
</tr>
<tr>
<td>u_degree</td>
<td>short</td>
<td>u degree of the surface</td>
</tr>
<tr>
<td>v_degree</td>
<td>short</td>
<td>v degree of the surface</td>
</tr>
<tr>
<td>n_u_vertices</td>
<td>int</td>
<td>number of control vertices (‘poles’) in u direction</td>
</tr>
<tr>
<td>n_v_vertices</td>
<td>int</td>
<td>number of control vertices (‘poles’) in v direction</td>
</tr>
<tr>
<td>u_knot_type</td>
<td>byte</td>
<td>form of u knot vector – see “B curve”</td>
</tr>
<tr>
<td>v_knot_type</td>
<td>byte</td>
<td>form of v knot vector</td>
</tr>
<tr>
<td>n_u_knots</td>
<td>int</td>
<td>number of distinct u knots</td>
</tr>
<tr>
<td>n_v_knots</td>
<td>int</td>
<td>number of distinct v knots</td>
</tr>
<tr>
<td>rational</td>
<td>logical</td>
<td>true if surface is rational</td>
</tr>
<tr>
<td>u_closed</td>
<td>logical</td>
<td>true if surface is closed in u</td>
</tr>
<tr>
<td>v_closed</td>
<td>logical</td>
<td>true if surface is closed in v</td>
</tr>
<tr>
<td>surface_form</td>
<td>byte</td>
<td>shape of surface, if special</td>
</tr>
<tr>
<td>vertex_dim</td>
<td>short</td>
<td>dimension of control vertices</td>
</tr>
<tr>
<td>bspline_vertices</td>
<td>pointer</td>
<td>control vertices (poles) node</td>
</tr>
<tr>
<td>u_knot_mult</td>
<td>pointer</td>
<td>multiplicities of u knot vector</td>
</tr>
<tr>
<td>v_knot_mult</td>
<td>pointer</td>
<td>multiplicities of v knot vector</td>
</tr>
<tr>
<td>u_knots</td>
<td>pointer</td>
<td>u knot vector</td>
</tr>
<tr>
<td>v_knots</td>
<td>pointer</td>
<td>v knot vector</td>
</tr>
</tbody>
</table>

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.

static 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
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
- “extended_u_int” and “extended_v_int” are the valid parameter ranges for a B surface once it has been extended.

The character fields ‘original_u_start’ etc. all refer to the status of the corresponding parameter boundary of the surface before or after an extension has taken place. For B surfaces, the character can have one of the following values:

```c
const char SCH_degenerate = 'D';  // Degenerate edge
const char SCH_periodic   = 'P';   // Periodic parameterisation
const char SCH_bounded    = 'B';   // Parameterisation bounded
const char SCH_closed     = 'C';    // Closed, but not periodic
```
The separate fields original_u_start and extended_u_start etc. are necessary because an extension may cause the corresponding parameter boundary to become degenerate.

If the surface_data node is present, then the original_u_int, original_v_int, original_u_start, original_u_end, original_v_start and original_v_end fields should be set to their appropriate values. If the surface has not been extended, the extended_u_int and extended_v_int fields should contain null, and the extended_u_start etc. fields should contain

\[
\text{const char SCH_unset_char} = '?'; // generic uninvestigated value
\]

As soon as any parameter boundary of the surface is extended, all the fields should be set, regardless of whether the corresponding boundary has been affected by the extension.

The SCH_self_int_t enum is documented in the corresponding curve_data structure under B curve.

The ‘swept_form_type’, ‘spun_form_type’ and ‘blend_form_type’ characters and the corresponding pointers swept_form, spun_form and blend_form, are for future use and are not implemented in Parasolid V12.0. The character fields should be set to SCH_unset_char (‘?’) and the pointers should be set to null pointer.

If the analytic_form field is not null, it will point to a HELIX_SU_FORM node, which indicates that the surface has a helical shape. In this case the analytic_form_type field will be set to ‘H’.

```c
struct HELIX_SU_FORM_s {
    vector axis_pt // $v
    vector axis_dir // $v
    char hand // $c
    interval turns // $i
    double pitch // $f
    double gap // $f
    double tol // $f
};
```

typedef struct HELIX_SU_FORM_s *HELIX_SU_FORM;

The axis_pt and axis_dir fields define the axis of the helix. The hand field is ‘+’ for a right-handed and ‘-’ for a left-handed helix. The turns field gives the extent of the helix relative to the profile curve which was used to generate the surface. For instance, an interval [0 10] indicates a start position at the profile curve and an end 10 turns along the
axis. Pitch is the distance travelled along the axis in one turn. Tol is the accuracy to which the owning bsurface fits this specification. Gap is for future expansion and will currently be zero. The v parameter increases in the direction of the axis.

**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.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>section</td>
<td>pointer</td>
<td>section curve</td>
</tr>
<tr>
<td>sweep</td>
<td>vector</td>
<td>sweep direction (a unit vector)</td>
</tr>
<tr>
<td>scale</td>
<td>double</td>
<td>for internal use only – may be set to null</td>
</tr>
</tbody>
</table>

```
struct SWEPT_SURF_s == ANY_SURF_s // Swept surface
{
    int node_id; // $d
    union ATTRIB_GROUP_u attributes_groups; // $p
    union SURFACE_OWNER_u owner; // $p
```

- 71 -
union SURFACE_u next; // $p
union SURFACE_u previous; // $p
struct *geometric_owner; // $p
GEOMETRIC_OWNER_s
char sense; // $c
union CURVE_u section; // $p
vector sweep; // $v
double scale; // $f
};
typedef struct SWEPT_SURF_s *SWEPT_SURF;

SPUN_SURF
A spun surface has a parametric representation of the form:
\[ R( u, v ) = Z( u ) + ( C( u ) - Z( u )) \cos( v ) + A \times ( C( u ) - Z( u )) \sin( v) \]
where

- \( C(u) \) is the profile curve
- \( Z(u) \) is the projection of \( C(u) \) onto the spin axis
- \( A \) is the spin axis direction (unit vector)
- \( C \) must not be an intersection curve or a trimmed curve

NOTE: \( Z(u) = P + ( ( C( u ) - P ) \cdot A )A \) where \( P \) is a reference point on the axis.
<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>profile</td>
<td>pointer</td>
<td>profile curve</td>
</tr>
<tr>
<td>base</td>
<td>vector</td>
<td>point on spin axis</td>
</tr>
<tr>
<td>axis</td>
<td>vector</td>
<td>spin axis direction (a unit vector)</td>
</tr>
<tr>
<td>start</td>
<td>vector</td>
<td>position of degeneracy at low u (may be null)</td>
</tr>
<tr>
<td>end</td>
<td>vector</td>
<td>position of degeneracy at low v (may be null)</td>
</tr>
<tr>
<td>start_param</td>
<td>double</td>
<td>curve parameter at low u degeneracy (may be null)</td>
</tr>
<tr>
<td>end_param</td>
<td>double</td>
<td>curve parameter at high u degeneracy (may be null)</td>
</tr>
<tr>
<td>x_axis</td>
<td>vector</td>
<td>unit vector in profile plane if common with spin axis</td>
</tr>
<tr>
<td>scale</td>
<td>double</td>
<td>for internal use only – may be set to null</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>char</td>
<td>whether internal or external</td>
</tr>
<tr>
<td>data</td>
<td>pointer</td>
<td>internal or external data</td>
</tr>
<tr>
<td>tf</td>
<td>pointer0</td>
<td>transform applied to geometry</td>
</tr>
<tr>
<td>internal geom</td>
<td>pointer array</td>
<td>reference to other related geometry</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>integer unique within part</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>attributes and groups associated with point</td>
</tr>
<tr>
<td>owner</td>
<td>pointer</td>
<td>Owner</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>next point in chain</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>previous point in chain</td>
</tr>
<tr>
<td>pvec</td>
<td>vector</td>
<td>position of point</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>integer unique within part</td>
</tr>
<tr>
<td>owner</td>
<td>pointer</td>
<td>owning instance or world</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>next transform in chain</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>previous pointer in chain</td>
</tr>
<tr>
<td>rotation_matrix</td>
<td>double[3][3]</td>
<td>rotation component</td>
</tr>
<tr>
<td>translation_vector</td>
<td>vector</td>
<td>translation component</td>
</tr>
<tr>
<td>scale</td>
<td>double</td>
<td>scaling factor</td>
</tr>
<tr>
<td>flag</td>
<td>byte</td>
<td>binary flags indicating non-trivial components</td>
</tr>
<tr>
<td>perspective_vector</td>
<td>vector</td>
<td>perspective vector (always null vector)</td>
</tr>
</tbody>
</table>
The transform acts as

\[ x' = (\text{rotation\_matrix} \cdot x + \text{translation\_vector}) \cdot \text{scale} \]

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

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Binary Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>translation</td>
<td>00001</td>
<td>set if translation vector non-zero</td>
</tr>
<tr>
<td>rotation</td>
<td>00010</td>
<td>set if rotation matrix is not the identity</td>
</tr>
<tr>
<td>scaling</td>
<td>00100</td>
<td>set if scaling component is not 1.0</td>
</tr>
<tr>
<td>reflection</td>
<td>01000</td>
<td>set if determinant of rotation matrix is negative</td>
</tr>
<tr>
<td>general affine</td>
<td>10000</td>
<td>set if the rotation_matrix is not a rigid rotation</td>
</tr>
</tbody>
</table>

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
}
Curve and Surface Senses

The ‘natural’ tangent to a curve is that in the increasing parameter direction, and the ‘natural’ normal to a surface is in the direction of the cross-product of $\frac{\mathbf{dP}}{\mathbf{du}}$ and $\frac{\mathbf{dP}}{\mathbf{dv}}$. For some purposes these are modified by the curve and surfaces senses, respectively – for example in the definition of blend surfaces, offset surfaces and intersection curves.

At the PK interface, the edge/curve and face/surface sense orientations are regarded as properties of the topology/geometry combination. In the XT format, this orientation information resides in the curves, surfaces and faces as follows:

The edge/curve orientation is stored in the curve->sense field. The face/surface orientation is a combination of sense flags stored in the face->sense and surface->sense fields, so the face/surface orientation is true (i.e. the face normal is parallel to the natural surface normal) if neither, or both, of the face and surface senses are positive.

Geometric_owner

Where geometry has dependants, the dependants point back to the referencing geometry by means of Geometric Owner nodes. Each geometric node points to a doubly-linked ring of Geometric Owner nodes which identify its referencing geometry. Referenced geometry is as follows:

- Intersection: 2 surfaces
- SP-curve: Surface
- Trimmed curve: basis curve
- Blended edge: 2 supporting surfaces, 2 blend_bound surfaces, 1 spine curve
- Blend bound: blend surface
- Offset surface: underlying surface
- Swept surface: section curve
- Spun surface: profile curve
Note that the 2D B-curve referenced by an SP-curve is not a dependent in this sense, and does not need a geometric owner node.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>pointer</td>
<td>referencing geometry</td>
</tr>
<tr>
<td>next</td>
<td>pointer</td>
<td>next in ring of geometric owners referring to the same geometry</td>
</tr>
<tr>
<td>previous</td>
<td>pointer</td>
<td>previous in above ring</td>
</tr>
<tr>
<td>shared_geometry</td>
<td>pointer</td>
<td>referenced (dependent) geometry</td>
</tr>
</tbody>
</table>

```c
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

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assembly</td>
<td>pointer0</td>
<td>Head of chain of assemblies</td>
</tr>
<tr>
<td>attribute</td>
<td>pointer0</td>
<td>Ignore</td>
</tr>
<tr>
<td>body</td>
<td>pointer0</td>
<td>Head of chain of bodies</td>
</tr>
<tr>
<td>transform</td>
<td>pointer0</td>
<td>Head of chain of transforms</td>
</tr>
<tr>
<td>surface</td>
<td>pointer0</td>
<td>Head of chain of surfaces</td>
</tr>
<tr>
<td>curve</td>
<td>pointer0</td>
<td>Head of chain of curves</td>
</tr>
<tr>
<td>point</td>
<td>pointer0</td>
<td>Head of chain of points</td>
</tr>
<tr>
<td>alive</td>
<td>logical</td>
<td>True unless partition is at initial pmark</td>
</tr>
<tr>
<td>attrib_def</td>
<td>pointer0</td>
<td>Head of chain of attribute definitions</td>
</tr>
<tr>
<td>highest_id</td>
<td>int</td>
<td>Highest pmark id in partition</td>
</tr>
<tr>
<td>current_id</td>
<td>int</td>
<td>Id of current pmark</td>
</tr>
</tbody>
</table>

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.

```
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

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

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.
The highest_node_id gives the highest node-id of any node in the assembly. Certain nodes within the assembly (namely instances, transforms, geometry, attributes and groups) have unique node-ids which are non-zero integers.

typedef enum
{
    SCH_collective_assembly = 1,
    SCH_conjunctive_assembly = 2,
    SCH_disjunctive_assembly = 3
} SCH_assembly_type;

typedef enum
{
    SCH_new_part       = 1,
    SCH_stored_part    = 2,
    SCH_modified_part  = 3,
    SCH_anonymous_part = 4,
    SCH_unloaded_part  = 5
} SCH_part_state;

struct ASSEMBLY_s                             // Assembly
{
    int                          highest_node_id;  // $d
    union ATTRIB_GROUP_u        attributes_groups; // $p
    struct LIST_s               *attribute_chains; // $p
    struct LIST_s               *list;            // $p
    union SURFACE_u             surface;         // $p
union CURVE_u   curve;  // $p
struct POINT_s   *point;  // $p
struct KEY_s    *key;  // $p
double   res_size;  // $f
double   res_linear;  // $f
struct INSTANCE_s  *ref_instance;  // $p
struct ASSEMBLY_s   *next;  // $p
struct ASSEMBLY_s   *previous;  // $p
SCH_part_state    state;  // $u
struct WORLD_s    *owner;  // $p
SCH_assembly_type    type;  // $u
struct INSTANCE_s  *sub_instance;  // $p
}

typedef struct ASSEMBLY_s  *ASSEMBLY;

struct KEY_s                     // Key
{
    string[1];  char  // $c[]
};

typedef struct KEY_s  *KEY;

### INSTANCE

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>Node-id</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>Head of chain of attributes of instance and member_of_groups of instance</td>
</tr>
<tr>
<td>type</td>
<td>byte</td>
<td>Always 1</td>
</tr>
<tr>
<td>part</td>
<td>pointer</td>
<td>Part referenced by instance</td>
</tr>
<tr>
<td>transform</td>
<td>pointer0</td>
<td>Transform of instance</td>
</tr>
<tr>
<td>assembly</td>
<td>pointer</td>
<td>Assembly in which instance lies</td>
</tr>
<tr>
<td>next_in_part</td>
<td>pointer0</td>
<td>Next instance in assembly</td>
</tr>
<tr>
<td>prev_in_part</td>
<td>pointer0</td>
<td>Previous instance in assembly</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>highest_node_id</td>
<td>int</td>
<td>Highest node-id in body</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>Head of chain of attributes of, and groups in, body</td>
</tr>
<tr>
<td>attribute_chains</td>
<td>pointer0</td>
<td>List of attributes, one for each attribute definition used in the body</td>
</tr>
<tr>
<td>surface</td>
<td>pointer0</td>
<td>Head of construction surface chain</td>
</tr>
<tr>
<td>curve</td>
<td>pointer0</td>
<td>Head of construction curve chain</td>
</tr>
<tr>
<td>point</td>
<td>pointer0</td>
<td>Head of construction point chain</td>
</tr>
<tr>
<td>key</td>
<td>pointer0</td>
<td>Ignore</td>
</tr>
<tr>
<td>res_size</td>
<td>double</td>
<td>Value of ‘size box’ when transmitted (normally 1000)</td>
</tr>
<tr>
<td>res_linear</td>
<td>double</td>
<td>Value of modeller linear precision when transmitted (normally 1.0e-8)</td>
</tr>
<tr>
<td>ref_instance</td>
<td>pointer0</td>
<td>Head of chain of instances referencing this part</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>Ignore</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>Ignore</td>
</tr>
<tr>
<td>state</td>
<td>byte</td>
<td>Set to 1 (see below)</td>
</tr>
<tr>
<td>owner</td>
<td>pointer0</td>
<td>Ignore</td>
</tr>
<tr>
<td>body_type</td>
<td>byte</td>
<td>Body type</td>
</tr>
<tr>
<td>nom_geom_state</td>
<td>byte</td>
<td>Set to 1 (for future use)</td>
</tr>
<tr>
<td>shell</td>
<td>pointer0</td>
<td>For general bodies: null</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For solid bodies: the first shell in one of the solid regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For other bodies: the first shell in one of the regions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is <strong>obsolete</strong>, and should be ignored by applications reading XT files. When writing XT files, it must be set as above.</td>
</tr>
<tr>
<td>boundary_surface</td>
<td>pointer0</td>
<td>Head of chain of surfaces attached directly or indirectly to faces or edges or fins</td>
</tr>
<tr>
<td>boundary_curve</td>
<td>pointer0</td>
<td>Head of chain of curves attached directly or indirectly to edges or faces or fins</td>
</tr>
<tr>
<td>boundary_point</td>
<td>pointer0</td>
<td>Head of chain of points attached to vertices</td>
</tr>
<tr>
<td>region</td>
<td>pointer</td>
<td>Head of chain of regions in body; this is the infinite region</td>
</tr>
<tr>
<td>edge</td>
<td>pointer0</td>
<td>Head of chain of all non-wireframe edges in body</td>
</tr>
<tr>
<td>vertex</td>
<td>pointer0</td>
<td>Head of chain of all vertices in body</td>
</tr>
</tbody>
</table>
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.

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

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
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
};

typedef struct BODY_s     *BODY;
**Attaching Geometry to Topology**

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

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

<table>
<thead>
<tr>
<th>Geometry</th>
<th>Owner</th>
<th>Whether chained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attached to face</td>
<td>face</td>
<td>In boundary_surface chain</td>
</tr>
<tr>
<td>Attached to edge or fin</td>
<td>edge or fin</td>
<td>In boundary_curve chain</td>
</tr>
<tr>
<td>Attached to vertex</td>
<td>vertex</td>
<td>In boundary_point chain</td>
</tr>
<tr>
<td>Indirectly attached to face or edge or fin</td>
<td>body</td>
<td>In boundary_surface chain or boundary_curve chain</td>
</tr>
<tr>
<td>Construction geometry</td>
<td>body or assembly</td>
<td>In surface, curve or point chain</td>
</tr>
<tr>
<td>2D B-curve in SP-curve</td>
<td>null</td>
<td>Not chained</td>
</tr>
</tbody>
</table>

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**

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

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>Node-id</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>Head of chain of attributes of shell</td>
</tr>
<tr>
<td>body</td>
<td>pointer0</td>
<td>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.</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>Next shell in region</td>
</tr>
<tr>
<td>face</td>
<td>pointer0</td>
<td>Head of chain of back-faces of shell (i.e. faces with face normal pointing out of region of shell).</td>
</tr>
<tr>
<td>edge</td>
<td>pointer0</td>
<td>Head of chain of wire-frame edges of shell</td>
</tr>
<tr>
<td>vertex</td>
<td>pointer0</td>
<td>If shell consists of a single vertex, this is it; else null</td>
</tr>
<tr>
<td>region</td>
<td>pointer</td>
<td>Region of shell</td>
</tr>
<tr>
<td>front_face</td>
<td>pointer0</td>
<td>Head of chain of front-faces of shell (i.e. faces with face normal pointing into region of shell)</td>
</tr>
</tbody>
</table>

struct SHELL_s                                    //  Shell


```c
{
  int node_id; // $d
  union ATTRIB_GROUP_u attributes_groups; // $p
  struct BODY_s *body; // $p
  struct SHELL_s *next; // $p
  struct FACE_s *face; // $p
  struct EDGE_s *edge; // $p
  struct VERTEX_s *vertex; // $p
  struct REGION_s *region; // $p
  struct FACE_s *front_face; // $p
};

typedef struct SHELL_s *SHELL;

FACE

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

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>Node-id</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>Head of chain of attributes of loop</td>
</tr>
<tr>
<td>fin</td>
<td>pointer</td>
<td>One of ring of fins of loop</td>
</tr>
<tr>
<td>face</td>
<td>pointer</td>
<td>Face of loop</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>Next loop in face</td>
</tr>
</tbody>
</table>

**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

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

**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.

```c
struct FIN_s                                        // Fin
{
    union ATTRIB_GROUP_u   attributes_groups;   // $p
    struct LOOP_s           *loop;                // $p
    struct FIN_s            *forward;             // $p
    struct FIN_s            *backward;            // $p
    struct VERTEX_s         *vertex;               // $p
    struct FIN_s            *other;                // $p
    struct EDGE_s           *edge;                 // $p
    union CURVE_u           curve;                 // $p
    struct FIN_s            *next_at_vx;           // $p
    char                    sense;                 // $c
};
typedef struct FIN_s *FIN;
```

**VERTEX**

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

EDGEEEE

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>Node-id</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>Head of chain of attributes of edge and member_of_groups of edge</td>
</tr>
<tr>
<td>tolerance</td>
<td>double</td>
<td>Tolerance of edge (null-double for accurate edges)</td>
</tr>
<tr>
<td>fin</td>
<td>pointer</td>
<td>One of singly-linked ring of fins around edge</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>Previous edge in body or shell</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>Next edge in body or shell</td>
</tr>
</tbody>
</table>
### Parasolid XT Format Reference

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>curve</td>
<td>pointer0</td>
<td>Curve of edge, zero for tolerant edge. If edge is accurate, but any of its vertices are tolerant, this will be a trimmed curve</td>
</tr>
<tr>
<td>next_on_curve</td>
<td>pointer0</td>
<td>Next in chain of edges sharing the curve of this edge</td>
</tr>
<tr>
<td>previous_on_curve</td>
<td>pointer0</td>
<td>Previous in chain of edges sharing the curve of this edge</td>
</tr>
<tr>
<td>owner</td>
<td>pointer</td>
<td>Owning body (for non-wireframe edges) or shell (for wireframe edges)</td>
</tr>
</tbody>
</table>

```c
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**

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

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

typedef enum

```c
{
    LIS_pointer = 4
}
```

LIS_type_t;

union LIS_BLOCK_u

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

POINTER_LIS_BLOCK:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_entries</td>
<td>int</td>
<td>Number of entries in this block (0 &lt;= n_entries &lt;= 20). Only the first block may have n_entries = 0.</td>
</tr>
</tbody>
</table>
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.

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

ATT_DEF_ID

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string[]</td>
<td>char</td>
<td>String name e.g. &quot;SDL/TYSA_COLOUR&quot;</td>
</tr>
</tbody>
</table>

struct ATT_DEF_ID_s                   //  name field type for attrib def.
{
    char                          String[1];       //  $c[]
};
typedef struct ATT_DEF_ID_s *ATT_DEF_ID;

FIELD_NAMES

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>names[]</td>
<td>pointer</td>
<td>Array of field names – unicode or char</td>
</tr>
</tbody>
</table>
typedef union FIELD_NAME_u
{
    struct CHAR_VALUES_s    *name
    struct UNICODE_VALUES_s *uname
};
FIELD_NAME_t;

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

ATTRIB_DEF

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>pointer0</td>
<td>Next attribute definition. This can be ignored, except in a partition transmit file.</td>
</tr>
<tr>
<td>identifier</td>
<td>pointer</td>
<td>Pointer to string name</td>
</tr>
<tr>
<td>type_id</td>
<td>int</td>
<td>Numeric id, e.g. 8001 for color. 9000 for user-defined attribute definitions</td>
</tr>
<tr>
<td>actions[8]</td>
<td>byte</td>
<td>Required actions on various events</td>
</tr>
<tr>
<td>field_names</td>
<td>pointer0</td>
<td>Names of fields (unicode or char)</td>
</tr>
<tr>
<td>legal_owners[14]</td>
<td>logical</td>
<td>Allowed owner types</td>
</tr>
<tr>
<td>fields[]</td>
<td>byte</td>
<td>Array of field types. Note that the number of fields is given by the length of the variable length part of this node, i.e. the integer following the node type in the transmit file.</td>
</tr>
</tbody>
</table>

The legal_owners array is an array of logicals determining which node types may own this type of attribute, e.g. if faces are allowed attrib_def -> legal_owners [SCH_fa_owner] = true.
Note that if the file contains user fields, the ‘fields’ field of an attribute definition may contain extra values, set to zero. These are to be ignored.

The ‘actions’ field in an attribute definition defines the behaviour of the attribute when an event (rotate, scale, translate, reflect, split, merge, transfer, change) occurs. The actions are:

| 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_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:

<table>
<thead>
<tr>
<th></th>
<th>split</th>
<th>merge</th>
<th>transfer</th>
<th>change</th>
<th>Rotate</th>
<th>scale</th>
<th>translate</th>
<th>reflect</th>
</tr>
</thead>
<tbody>
<tr>
<td>class 1</td>
<td>propagate</td>
<td>keep_equal</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
</tr>
<tr>
<td>class 2</td>
<td>delete</td>
<td>delete</td>
<td>delete</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
</tr>
<tr>
<td>class 3</td>
<td>delete</td>
<td>delete</td>
<td>delete</td>
<td>delete</td>
<td>Delete</td>
<td>delete</td>
<td>delete</td>
<td>delete</td>
</tr>
<tr>
<td>class 4</td>
<td>propagate</td>
<td>keep_equal</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>Transform</td>
<td>transform</td>
<td>transform</td>
<td>transform</td>
</tr>
<tr>
<td>class 5</td>
<td>delete</td>
<td>delete</td>
<td>delete</td>
<td>do_nothing</td>
<td>Transform</td>
<td>transform</td>
<td>transform</td>
<td>transform</td>
</tr>
<tr>
<td>class 6</td>
<td>propagate</td>
<td>combine</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>do_nothing</td>
</tr>
<tr>
<td>class 7</td>
<td>propagate</td>
<td>combine</td>
<td>do_nothing</td>
<td>do_nothing</td>
<td>Transform</td>
<td>transform</td>
<td>transform</td>
<td>transform</td>
</tr>
</tbody>
</table>

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

typedef enum

```c
{
  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 = 3,
    SCH_keep_if_equal = 5,
    SCH_combine = 6
}
SCH_action_on_fields_t;

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

} SCH_attrib_owners_t;

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

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

typedef struct ATTRIB_DEF_s *ATTRIB_DEF;

ATTRIBUTE

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>Node-id</td>
</tr>
<tr>
<td>definition</td>
<td>pointer</td>
<td>Attribute definition</td>
</tr>
<tr>
<td>owner</td>
<td>pointer</td>
<td>Attribute owner</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>Next attribute, group, or member_of_group</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>Previous ditto</td>
</tr>
<tr>
<td>next_of_type</td>
<td>pointer0</td>
<td>Next attribute of this type in this part</td>
</tr>
<tr>
<td>previous_of_type</td>
<td>pointer0</td>
<td>Previous attribute of this type in this part</td>
</tr>
<tr>
<td>fields[]</td>
<td>pointer</td>
<td>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.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>values[]</th>
<th>int count</th>
<th>Integer values</th>
</tr>
</thead>
<tbody>
<tr>
<td>values[]</td>
<td>int</td>
<td>$d</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>values[]</th>
<th>vector</th>
<th>Point values</th>
</tr>
</thead>
</table>

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

VECTOR_VALUES

<table>
<thead>
<tr>
<th>values[]</th>
<th>vector</th>
<th>Vector values</th>
</tr>
</thead>
</table>

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

DIRECTION_VALUES

<table>
<thead>
<tr>
<th>values[]</th>
<th>vector</th>
<th>Direction values</th>
</tr>
</thead>
</table>

```c
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.

```c
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.

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

GROUP

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_id</td>
<td>int</td>
<td>Node-id</td>
</tr>
<tr>
<td>attributes_groups</td>
<td>pointer0</td>
<td>Head of chain of attributes of this group</td>
</tr>
<tr>
<td>owner</td>
<td>pointer</td>
<td>Owning part</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>Next group or attribute</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>Previous group or attribute</td>
</tr>
<tr>
<td>type</td>
<td>byte</td>
<td>Type of node allowed in group</td>
</tr>
<tr>
<td>first_member</td>
<td>pointer0</td>
<td>Head of chain of member_of_group nodes in group</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dummy_node_id</td>
<td>int</td>
<td>Zero</td>
</tr>
<tr>
<td>owning_group</td>
<td>pointer</td>
<td>Owning group</td>
</tr>
<tr>
<td>owner</td>
<td>pointer</td>
<td>Referenced member of group</td>
</tr>
<tr>
<td>next</td>
<td>pointer0</td>
<td>Next attribute, group or member_of_group</td>
</tr>
<tr>
<td>previous</td>
<td>pointer0</td>
<td>Previous ditto</td>
</tr>
<tr>
<td>next_member</td>
<td>pointer0</td>
<td>Next member_of_group in this group</td>
</tr>
<tr>
<td>previous_member</td>
<td>pointer0</td>
<td>Previous ditto</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Node name</th>
<th>Node type</th>
<th>Visible at PK</th>
<th>Has node-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSEMBLY</td>
<td>10</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>INSTANCE</td>
<td>11</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BODY</td>
<td>12</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SHELL</td>
<td>13</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FACE</td>
<td>14</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LOOP</td>
<td>15</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EDGE</td>
<td>16</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FIN</td>
<td>17</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VERTEX</td>
<td>18</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>REGION</td>
<td>19</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>POINT</td>
<td>29</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LINE</td>
<td>30</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CIRCLE</td>
<td>31</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ELLIPSE</td>
<td>32</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>INTERSECTION</td>
<td>38</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CHART</td>
<td>40</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LIMIT</td>
<td>41</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>BSPLINE_VERTICES</td>
<td>45</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PLANE</td>
<td>50</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CYLINDER</td>
<td>51</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>CONE</td>
<td>52</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SPHERE</td>
<td>53</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TORUS</td>
<td>54</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BLENDED_EDGE</td>
<td>56</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BLEND_BOUND</td>
<td>59</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>OFFSET_SURF</td>
<td>60</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SWEPT_SURF</td>
<td>67</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SPUN_SURF</td>
<td>68</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LIST</td>
<td>70</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>POINTER_LIS_BLOCK</td>
<td>74</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ATT_DEF_ID</td>
<td>79</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ATTRIB_DEF</td>
<td>80</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ATTRIBUTE</td>
<td>81</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>INT_VALUES</td>
<td>82</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>REAL_VALUES</td>
<td>83</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>CHAR_VALUES</td>
<td>84</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>POINT_VALUES</td>
<td>85</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>VECTOR_VALUES</td>
<td>86</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>AXIS_VALUES</td>
<td>87</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>TAG_VALUES</td>
<td>88</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>DIRECTION_VALUES</td>
<td>89</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>GROUP</td>
<td>90</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MEMBER_OF_GROUP</td>
<td>91</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>UNICODE_VALUES</td>
<td>98</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>FIELD_NAMES</td>
<td>99</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>100</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>WORLD</td>
<td>101</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>KEY</td>
<td>102</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE_SURF</td>
<td>120</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>INT_PE_DATA</td>
<td>121</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>EXT_PE_DATA</td>
<td>122</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>B_SURFACE</td>
<td>124</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SURFACE_DATA</td>
<td>125</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>NURBS_SURF</td>
<td>126</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>KNOT_MULT</td>
<td>127</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>KNOT_SET</td>
<td>128</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE_CURVE</td>
<td>130</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TRIMMED_CURVE</td>
<td>133</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>B_CURVE</td>
<td>134</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CURVE_DATA</td>
<td>135</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>NURBS_CURVE</td>
<td>136</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SP_CURVE</td>
<td>137</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GEOMETRIC_OWNER</td>
<td>141</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>HELIX_CU_FORM</td>
<td>163</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>HELIX_SU_FORM</td>
<td>184</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>
# Node Classes

<table>
<thead>
<tr>
<th>Node class name</th>
<th>Node class</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEOMETRY</td>
<td>1003</td>
</tr>
<tr>
<td>PART</td>
<td>1005</td>
</tr>
<tr>
<td>SURFACE</td>
<td>1006</td>
</tr>
<tr>
<td>SURFACE_OWNER</td>
<td>1007</td>
</tr>
<tr>
<td>CURVE</td>
<td>1008</td>
</tr>
<tr>
<td>CURVE_OWNER</td>
<td>1010</td>
</tr>
<tr>
<td>POINT_OWNER</td>
<td>1011</td>
</tr>
<tr>
<td>LIS_BLOCK</td>
<td>1012</td>
</tr>
<tr>
<td>LIST_OWNER</td>
<td>1013</td>
</tr>
<tr>
<td>ATTRIBUTE_OWNER</td>
<td>1015</td>
</tr>
<tr>
<td>GROUP_OWNER</td>
<td>1016</td>
</tr>
<tr>
<td>GROUP_MEMBER</td>
<td>1017</td>
</tr>
<tr>
<td>FIELD_VALUES</td>
<td>1018</td>
</tr>
<tr>
<td>ATTRIB_GROUP</td>
<td>1019</td>
</tr>
<tr>
<td>TRANSFORM_OWNER</td>
<td>1023</td>
</tr>
<tr>
<td>PE_DATA</td>
<td>1027</td>
</tr>
<tr>
<td>PE_INT_GEOM</td>
<td>1028</td>
</tr>
<tr>
<td>SHELL_OR_BODY</td>
<td>1029</td>
</tr>
<tr>
<td>FIELD_NAME</td>
<td>1037</td>
</tr>
</tbody>
</table>
System Attribute Definitions

All system attribute definitions are of class 1.

Hatching

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_HATCHING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type_id</td>
<td>8003</td>
</tr>
<tr>
<td>Entity types</td>
<td>face</td>
</tr>
<tr>
<td>Fields</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>real 1</td>
</tr>
<tr>
<td></td>
<td>real 2</td>
</tr>
<tr>
<td></td>
<td>real 3</td>
</tr>
<tr>
<td></td>
<td>real 4</td>
</tr>
<tr>
<td></td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>Hatching type</td>
</tr>
<tr>
<td>Set by</td>
<td>Application</td>
</tr>
<tr>
<td>Used by</td>
<td>Parasolid hidden line and wireframe images</td>
</tr>
</tbody>
</table>

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**

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

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**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_RADIAL_HATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type_id</strong></td>
<td>8027</td>
</tr>
<tr>
<td><strong>Entity types</strong></td>
<td>face</td>
</tr>
<tr>
<td><strong>Fields</strong></td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>radial around</td>
</tr>
<tr>
<td></td>
<td>radial along</td>
</tr>
<tr>
<td></td>
<td>radial about</td>
</tr>
<tr>
<td></td>
<td>radial around start</td>
</tr>
<tr>
<td></td>
<td>radial along start</td>
</tr>
<tr>
<td></td>
<td>radial about start</td>
</tr>
<tr>
<td><strong>Set by</strong></td>
<td>Application</td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>Parasolid hidden line and wireframe images</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_PARAM_HATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type_id</strong></td>
<td>8028</td>
</tr>
<tr>
<td><strong>Entity types</strong></td>
<td>face</td>
</tr>
<tr>
<td><strong>Fields</strong></td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>u spacing</td>
</tr>
<tr>
<td></td>
<td>v spacing</td>
</tr>
<tr>
<td></td>
<td>u start</td>
</tr>
<tr>
<td></td>
<td>v start</td>
</tr>
<tr>
<td><strong>Set by</strong></td>
<td>Application</td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>Parasolid hidden line and wireframe images</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_DENSITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type_id</td>
<td>8004</td>
</tr>
<tr>
<td>Entity types</td>
<td>body</td>
</tr>
<tr>
<td>Fields</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>Density</td>
</tr>
<tr>
<td></td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Units</td>
</tr>
<tr>
<td>Set by</td>
<td>Application</td>
</tr>
<tr>
<td>Used by</td>
<td>Parasolid Mass Properties - calculation of mass</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_REGION_DENSITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type_id</td>
<td>8023</td>
</tr>
<tr>
<td>Entity types</td>
<td>region</td>
</tr>
<tr>
<td>Fields</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>Density of region</td>
</tr>
<tr>
<td></td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Units</td>
</tr>
<tr>
<td>Set by</td>
<td>Application</td>
</tr>
<tr>
<td>Used by</td>
<td>Parasolid Mass Properties - calculation of mass</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_FACE_DENSITY</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type id</strong></td>
<td>8024</td>
</tr>
<tr>
<td><strong>Entity types</strong></td>
<td>face</td>
</tr>
<tr>
<td><strong>Fields</strong></td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>Density of face</td>
</tr>
<tr>
<td></td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Units</td>
</tr>
<tr>
<td><strong>Set by</strong></td>
<td>Application</td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>Parasolid Mass Properties - calculation of mass</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_EDGE_DENSITY</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type id</strong></td>
<td>8025</td>
</tr>
<tr>
<td><strong>Entity types</strong></td>
<td>edge</td>
</tr>
<tr>
<td><strong>Fields</strong></td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>Density of edge</td>
</tr>
<tr>
<td></td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Units</td>
</tr>
<tr>
<td><strong>Set by</strong></td>
<td>Application</td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>Parasolid Mass Properties - calculation of mass</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TTYA_VERTEX_DENSITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type_id</td>
<td>8026</td>
</tr>
<tr>
<td>Entity types</td>
<td>vertex</td>
</tr>
<tr>
<td>Fields</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>string</td>
</tr>
<tr>
<td>Mass of vertex</td>
<td></td>
</tr>
<tr>
<td>Units</td>
<td></td>
</tr>
<tr>
<td>Set by</td>
<td>Application</td>
</tr>
<tr>
<td>Used by</td>
<td>Parasolid Mass Properties - calculation of mass</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TTYA_REGION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type_id</td>
<td>8013</td>
</tr>
<tr>
<td>Entity types</td>
<td>face</td>
</tr>
<tr>
<td>Fields</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Unused</td>
</tr>
<tr>
<td>Set by</td>
<td>Application</td>
</tr>
<tr>
<td>Used by</td>
<td>Parasolid hidden line images</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_COLOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>8001</td>
</tr>
<tr>
<td>Entity types</td>
<td>face, edge</td>
</tr>
<tr>
<td>Fields</td>
<td>real</td>
</tr>
<tr>
<td>Red value</td>
<td>These three values should be in the range 0.0 to 1.0</td>
</tr>
<tr>
<td>Green value</td>
<td></td>
</tr>
<tr>
<td>Blue value</td>
<td></td>
</tr>
<tr>
<td>Set by</td>
<td>Application</td>
</tr>
<tr>
<td>Used by</td>
<td>Application</td>
</tr>
</tbody>
</table>

### Reflectivity

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_REFLECTIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>8014</td>
</tr>
<tr>
<td>Entity types</td>
<td>face</td>
</tr>
<tr>
<td>Fields</td>
<td>real</td>
</tr>
<tr>
<td>Coefficient of specular reflection</td>
<td></td>
</tr>
<tr>
<td>Proportion of colored light in highlights</td>
<td></td>
</tr>
<tr>
<td>Coefficient of diffuse reflection</td>
<td></td>
</tr>
<tr>
<td>Coefficient of ambient reflection</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>Reflection power</td>
</tr>
<tr>
<td>Set by</td>
<td>Application</td>
</tr>
<tr>
<td>Used by</td>
<td>Application</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_TRANSLUCENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>8015</td>
</tr>
</tbody>
</table>
### Entity types

<table>
<thead>
<tr>
<th>Entity types</th>
<th>face</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Fields</th>
<th>real</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transparency coefficient</td>
<td>range 0.0 to 1.0, where 0 is opaque and 1 is transparent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set by</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used by</td>
<td>Application</td>
</tr>
</tbody>
</table>

### Name

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDL/TYSA_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>8017</td>
</tr>
<tr>
<td>Entity types</td>
<td>assembly, body, instance, shell, face, loop, edge, vertex, group, surface, curve, point</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fields</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set by</td>
<td>Application</td>
</tr>
<tr>
<td>Used by</td>
<td>Application</td>
</tr>
</tbody>
</table>

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.