Parasolid

Parasolid XT Format Reference

May 2023

Important Note

Tel: +44 (0)1223 371555
email: parasolid.support.plm@siemens.com
Web: www.parasolid.com

SIEMENS

Trademarks

Table of Contents

About This Manual 1

1.1 Introduction to the XT format

This XT data 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 data 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 XT data 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 XT data formats these changes will not invalidate applications written based on the information herein.

1.1.1 Parasolid XT format terminology

Some of the terminology used in the XT data format differs from that used in the documentation for Parasolid. The following table lists these differences:

PK Interface	XT format
Fin	Halfedge
Group	Feature
Identifier	Node-id

1.1.2 Types of file document

There are a number of different interface routines in Parasolid for writing XT data. 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.

1.1.3 Text and binary formats

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

- Text (usually ASCII)
- Neutral binary
- Bare binary (this is not recommended)
- Typed binary

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

Neutral binary is a machine independent binary format.

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

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

1.1.4 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 as follows:

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

Extensions that have been used in the past are:

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

1.1.5 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 2

2.1 Layout of the XT data

The logical layout of the XT data is as follows:

- A human-oriented text header.
- The initial text header is read and written by applications' Frustrums and is not accessible to XT. Its detailed format is described in the section `Physical layout'.
- A short flag sequence describing the data format, followed by modeller identification information and user field size.
- The various flag sequences (mixtures of text and numbers) are documented under “Physical layout”; the content of the modeller identification information is:
- The version of the Parasolid Kernel used to write the data, as a text string of the form:
  - TRANSMIT FILE created by modeler version 3400000. 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_3400000_34000. This example denotes XT data written by the Parasolid Kernel V34.00.000 using schema number 34000: there will be a corresponding file `sch_34000` in the Parasolid schema distribution.

Note: Applications writing XT data using information from this version of the documentation should use version 3400000 and schema number 34000.

- The user field size is a simple integer.
- Objects in the XT data are called nodes. Every node in the XT data is assigned an integer index from 1 upwards (some indices may not be used). Pointer fields are output as these indices, or as zero for a null pointer.
- The first node in the XT data must be the root node. The following nodes can be in an unordered sequence, followed by a terminator.

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 XT data 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 XT data.

The node with index 1 is the root node of the XT data as follows:

Contents of XT data	Type of Root Node
Body	BODY
Assembly	ASSEMBLY
Array of Parts	PART_XMT_BLOCK (or POINTER_LIS_BLOCK for older files)
Array of Meshes	POINTER_LIS_BLOCK
Partition	WORLD

2.1.1 Schema

XT 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 the Parasolid Kernel, along with aschema file used during transmit and receive. For example, the schema file for version 31.0 is named sch_31000 and is distributed with the Parasolid Kernel.It is not necessary to have a copy of this file to understand the XT data 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 XT data.

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 XT data. 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:

Item	Description
0	A scalar, a single value
1	A variable length field (see below)
n > 1	An array of n values

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

2.1.2 Embedded schemas

When reading XT parts, partitions, or deltas, the Parasolid Kernel converts any data that it encounters from older versions of the Parasolid Kernel 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 the Parasolid Kernel 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, XT 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 XT parts, partitions, and deltas to be successfully read into older versions of Parasolid without loss of information.

The only fields that are included in this information are those which can be referenced in a cut-down version of the schema pertaining only to the XT part data that is transmitted. Specifically, a full schema definition can contain fields that are not relevant in the context of the transmitted data (fields relating to snapshots, for example), and these fields are excluded.

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

2.1.2.1 Physical layout

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

2.1.2.2 XT format

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

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

- The arrays of effective fields in the base schema node and the current schema node are assembled.
- If the nodetype does not exist in the base schema then it is output as follows:
- Number of fields (byte).
- Name and description (short strings).
- Fields one by one as follows;

Name	Data Type	Notes
`name`	Pointer	
`ptr_class`	Short	
`n_elts`	Positive Integer	
`type`	Short String	The field type. Allowed values are described in “Field types”, below. Omitted if `ptr_class` non-zero
`xmt_code`	Logical (byte)	Omitted for fixed-length (n_elts != 1)

- If the two arrays match (equal length and all fields match in `name`, `xmt_code`, `ptr_class`, `n_elts` and `type`) then output the flag value 255 (byte0xff).
- If the two arrays do not match, output the number of effective fields in the current schema (byte), and an edit sequence as follows.
- Initialize pointers to the first base field and first current field, then while there are still unprocessed base and current fields, output a sequence of Copy, Delete and Insert instructions.
- If the base field matches the current field, output 'C' (char) to indicate an unchanged (Copied) field and advance to the next base and current fields;
- If the base field does not match any unprocessed current field, output 'D' (char) to indicate a Deleted field and advance to the next base field;
- Otherwise, output 'I' (char) to indicate an Inserted field, followed by the current field in the above format, and advance to the next current field.
- If there are any unprocessed current fields, then output an Append sequence, each instruction being 'A' (char) followed by the field.
- Finally, output 'Z' (char) to signal the end.

2.1.3 Space compression

For text data in XT 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 of the Parasolid Kernel, this applies to attribute definition names, field names, and attribute strings.

2.1.4 Field types

The XT format is not itself a binary protocol, and so does not define data sizes; the only requirement is that a runtime implementation has sufficient room for the information. The available implementations run with 8bit ASCII characters, 8bit unsigned bytes (0..255), 16bit short integers (0..65535 or -32768..32767), 32bit integers (0..4G-1, -2G..2G-1) and IEEE reals.

The implementation used in a given binary XT data is specified by the "PS<code>" at the start of the file. See Chapter 3, “Physical Layout”, for more information.

The full list of field types used in XT data is as follows:

Item	Description
u	Unassigned byte 0-255
c	char
l	Unassigned byte 0-1 (i.e. logical) `typedef char logical`
n	Short int
w	Unicode character, output as short int
d	int
p	Pointer-index. Small indices (less than 32767) are treated specially in binary XT data to save space. See the section below on binary format.
f	Double
i	These correspond to a region of the real line: `typedef struct { double low, high; }interval;`
v	Array [3] of doubles. These correspond to a 3-space position or direction: `typedef struct { double x,y,z; } vector;`
b	Array [6] of doubles. These correspond to a 3-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 XT data, as the Parasolid Kernel will recalculate other data as required. The structure is documented further in the section on intersection curves.

2.1.4.1 Point

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

```
struct POINT_s          // Point
{
int             node_id;    // $d
union  ATTRIB_FEAT_u attributes_features;// $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 data entry is as follows;

```
29 POINT; Point; 1 6 0
node_id; d; 1 0 0
attributes_features; p; 1 1019 0
owner; p; 1 1011 0
next; p; 1 29 0
previous; p; 1 29 0
pvec; v; 1 0 0
```

2.1.4.2 Pointer classes

In the above example, the `attributes_features` field must be of class `ATTRIB_FEAT_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.

2.1.4.3 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 as follows;

```
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 XT data between its nodetype and index, so an example might be

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

2.1.4.4 Unresolved indices

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

2.1.4.5 Simple example

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

```
**ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz************
**PARASOLID !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~0123456789************
**PART1;MC=osf65;MC_MODEL=alpha;MC_ID=sdlosf6;OS=OSF1;OS_RELEASE=V4.0;FRU=sdl_par
asolid_test_osf64;APPL=unknown;SITE=sdl-cambridge-
u.k.;USER=davidj;FORMAT=text;GUISE=transmit;DATE=29-mar-2000;
**PART2;SCH=SCH_1200000_12006;USFLD_SIZE=0;
**PART3;
**END_OF_HEADER*****************************************************
T51 : TRANSMIT FILE created by modeller version 120000017 SCH_1200000_120060
12 1 12 0 2 0 0 0 0 1e3 1e-8 0 0 0 1 0 3 1 3 4 5 0 6 7 0 body
70 2 0 1 0 0 4 1 20 8 8 8 1 T                              list
13 3 3 0 1 0 9 0 0 6 9                                     shell
50 4 11 0 9 0 0 0 +0 0 0 0 0 1 1 0 0                       plane
31 5 10 0 7 0 0 0 +0 0 0 0 0 1 1 0 0 1                     circle
19 6 5 0 1 0 0 3 V                                         region
16 7 6 0 ?10 0 0 5 0 0 1                                   edge
17 10 0 11 10 10 0 12 7 0 0 +                              fin
15 11 7 0 10 9 0                                           loop
17 12 0 0 0 0 0 10 7 0 0 -                                 fin (dummy)
14 9 2 13 ?0 0 11 3 4 +0 0 0 0 3                           face
81 1 13 12 14 9 0 0 0 0 15                                 attribute (variable 1)
80 1 14 0 16 8001 0 0 0 0 3 5 0 0 FFFFTFTFFFFFF2           attrib_def (variable 1)
83 3 15 1 2 3                                              real_values (variable 3)
79 15 16 SDL/TYSA_COLOUR                                   att_def_id (variable 15)
74 20 8 1 0 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     pointer_lis_block
1 0 terminator
```

Note: The tolerance fields of the face and edge are un-set, and represented as '?' in the text XT data 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 valid XT data (the leading spaces on some of the lines are necessary).

Physical Layout 3

The XT data has two headers:

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

3.1 Common header

The XT common header recommended to Frustrum writers consists of:

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

```
    MC      =   vax, hppa, sparc, ...
                // make of computer
    MC_MODEL  =  4090, 9000/780, sun4m, ...
                //  model of computer
     MC_ID  =  ...
                //  unique machine identifier
      OS    =   vms, HP-UX, SunOS, ...
                //  name of operating system
OS_RELEASE =  V6.2, B.10.20, 5.5.1, ...
                //  version of operating system
FRU         =     sdl_parasolid_test_vax,
                    mdc_ugii_v7.0_djl_can_vrh, ...
//  frustrum supplier and implementation name
      APPL  =   kid, unigraphics, ...
//  application which is using Parasolid
      SITE  =   ...
//  site at which application is running
      USER  =   ...
                //  login name of user
    FORMAT  =  binary, text, applio
                //  format of file
     GUISE  =  transmit, transmit_partition
                //  guise of file
      KEY  =  ...
                //  name of key
      FILE  =   ...
                //  name of file
      DATE  =   dd-mmm-yyyy
//  e.g. 5-apr-1998
```

The ‘part 1’ data is ‘standard’ information which should be accessible to the Frustrum (e.g. by operating system calls). It is the responsibility of the Frustrum to gather the relevant information and to format it as described in this specification.

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

```
SCH     =   SCH_m_n
//  name of schema key e.g.SCH_3400000_34000
USFLD_SIZE=  m
//  length of user field (0 - 16 integer words)
```

Applications writing XT data must use a schema name of SCH_3400000_34000

- 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 given below:

```
**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.X
MT_TXT;DATE=8-sep-1997;
**PART2;SCH=SCH_701169_7007;USFLD_SIZE=0;
**PART3;
**END_OF_HEADER**********************************************
```

3.1.1 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 are to appear in a keyword value:

Character	Escape Sequence
newline	^n
space	^-
semicolon	^;
uparrow	^^

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

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

3.2 Text

XT has no knowledge of how data is stored. On writing, XT 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 XT data 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:

Item	Description
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 ‘, finally the userfield size. For example:

```
T
51 : TRANSMIT FILE created by modeller version 3000000
17 SCH_3000000_30000
0
```

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

```
T51 : TRANSMIT FILE created by modeller version 300000017
SCH_3000000_300000
```

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

```
63 : TRANSMIT FILE (partition) created by modeller version
3000000
```

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 ‘?’.

3.3 Binary

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

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

3.3.1 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.

3.3.2 Typed binary

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

	Byte Order	Double Representation	Character Representation
0	Big-endian	IEEE	ASCII
1	Little-endian	VAX D-float	EBCDIC

3.3.3 Neutral binary

In neutral binary, data is represented in big-endian format, with IEEE floating point numbers and ASCII characters. The flag sequence is the 4-byte sequence "PS" followed by two zero bytes,

i.e., 'P' 'S' '\0' '\0'. At Parasolid V9, the initial letters are ASCII, thus '\120' '\123'. The nodetype at the start of a node is a 2-byte integer, the variable length which may follow it is a 4-byte integer.

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

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

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

where op_short outputs a 2-byte integer.

The inverse is performed on reading:

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

where ip_short reads a 2-byte integer.

Model Structure 4

4.1 Topology

This section describes the XT Topology model, it gives an overview of how the nodes in the XT data are joined together. In this section the word ‘entity’ means a node which is visible to an XT application – a table of which nodes are visible at the XT 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.
- Compound bodies.
- Child bodies.
- Standard bodies.

Compound bodies are containers for child bodies that are expected to be related in some way such that they are able to share some physical aspects. Within compound bodies, a **child body** is used to define one representation of a part. **Standard** bodies are the basic “unit” of modelling used in Parasolid. A child body is identical to a standard body except that it can share geometry where appropriate with other child bodies within the compound body.

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

4.2 General points

This section provides information on some XT Format terminology used in this manual.

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.

4.2.1 Linear and angular resolution

XT data structures use fixed accuracies called **linear resolution** and **angular resolution**, which can be described as follows:

Resolution	Description
Linear resolution	The linear precision. Distances less than this value are considered to be zero and distances that differ by no more than this value are considered to be equal.
Angular resolution	The smallest angle (in radians) that is considered to be different from zero. Angles less than this value are considered to be zero and angles that differ by no more than this value are treated as equal.

By default, in XT data points are not considered coincident unless they are less than $1.0e^{-8}$ units apart (linear resolution). Directions are considered to be parallel if they differ by less than $1.0e^{-11}$ radians (angular resolution). It is important that any data passed to a Parasolid-enabled application is at least this accurate. You are recommended not to change these values when authoring XT data.

All parts of a body must be within a box called the **size box**, as shown in *Figure 4–1*, whose size is 1000 by 1000 by 1000 and is centered at the origin.

You are highly recommended to set the default unit to one meter, giving 1 kilometer as the maximum distance, in any one direction, that can be modelled.

To handle the angular resolution of arcs correctly, the radius used when representing an arc must be less than a factor of 10 times the dimension of the size box.

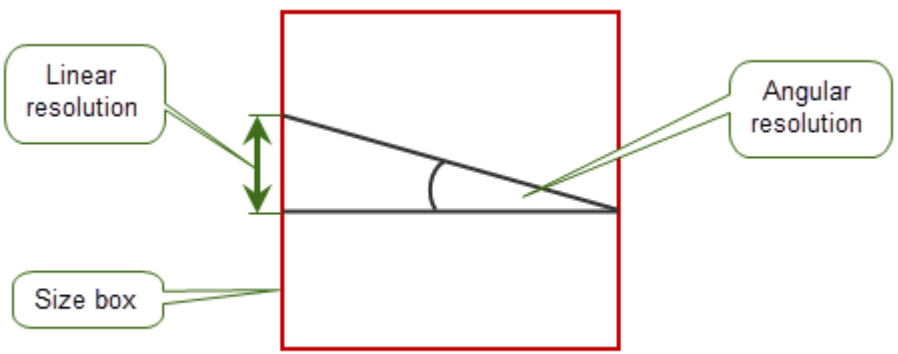


Figure 4–1 *Linear and angular resolution*

4.3 Entity definitions

4.3.1 Assembly

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

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

4.3.2 Instance

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

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

4.3.3 Body

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

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

A body has the following fields:

- A set of regions.

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

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

4.3.4 Region

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

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

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

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

A region has the following fields:

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

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

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

4.3.5 Shell

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

A shell has the following fields:

- A set of (face, logical) pairs.

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

- A set of wireframe edges.

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

- A vertex.

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

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

4.3.6 Face

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

A face has the following fields:

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

4.3.7 Loop

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

A loop has the following fields:

- An ordered ring of fins.

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

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

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

- A vertex.

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

Consequently, a loop shall consist either of:

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

4.3.8 Halfedge

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

A halfedge has the following fields:

- A logical sense indicating whether the orientation of the halfedge (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 edge of the halfedge is tolerant, in which case every halfedge of that edge will reference a trimmed SP-curve. The underlying surface of the SP-curve shall be the same as that of the corresponding face. The curve shall not deviate by more than the edge tolerance from curves on other halfedges of the edge, and its ends shall be within vertex tolerance of the corresponding vertices.

Note: Halfedges are referred to as ‘fins’ in the PK Interface.

4.3.9 Edge

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

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

  The ordering of the fins represents the spatial ordering of their owning faces about the edge (with a right-hand screw rule, i.e. looking in the direction of the edge the fin ordering is clockwise). The edge may have zero or any number of fins; if it has none, it is called a **wireframe** edge.

- A curve. This will be null if the edge has a tolerance. Otherwise, the vertices shall lie within vertex tolerance of this curve, and if it is a Trimmed Curve, they shall lie within vertex

tolerance of the corresponding ends of the curve. The curve shall also lie in the surfaces of the faces of the edge, to within modeller resolution.

- Sense. This logical indicates whether the direction of the edge (start to end) is the same as that of the curve.
- A tolerance. If this is null-double, the edge is accurateand is regarded as having a tolerance of half the modeller linear resolution, otherwise the edge is called tolerant.

4.3.10 Vertex

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

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

4.3.11 Attributes

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

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

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

4.3.12 Features

A feature is a collection of entities in the same part. Features in assemblies may contain instances, surfaces, curves and points. Features in bodies may contain regions, faces, edges, vertices, surfaces, curves, points, loops and other features.

Featureshave:

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

Note: Features are referred to as ‘groups’ in the PK Interface.

4.3.13 Identifiers

All entities in a part, other than halfedges and 2D B-curves referenced by SP-curves, have a non-zero integer identifier. All non-zero integer identifiers are unique within a part. This is intended to enable the entity to be identified within the XT data.

Note: Identifiers are referred to as ‘node-ids’ in the PK Interface.

4.4 Entity matrix

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

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

4.5 Representation of manifold bodies

4.5.1 Body types

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

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

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

4.5.1.1 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 shall consist of a single void region with a single shell consisting of a single vertex.
- A wire body shall consist of a single void region, with one or more shells, consisting of one or more wireframe edges and zero or more vertices (and no faces). Every vertex in the body 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.

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

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

Note: 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).

Schema Definitions 5

5.1 Underlying types

```
union CURVE_OWNER_u
     {
     struct EDGE_s                *edge;
     struct HALFEDGE_s            *halfedge;
     struct BODY_s                *body;
     struct ASSEMBLY_s            *assembly;
     struct WORLD_s               *world;
     };
typedef union CURVE_OWNER_u CURVE_OWNER;
union SURFACE_OWNER_u
     {
     struct FACE_s                *face;
     struct BODY_s                *body;
     struct ASSEMBLY_s            *assembly;
     struct WORLD_s               *world;
     };
typedef union SURFACE_OWNER_u SURFACE_OWNER;
union ATTRIB_FEAT_u
     {
     struct ATTRIBUTE_s           *attribute;
     struct FEATURE_s             *feature;
     struct MEMBER_OF_FEATURE_s   *member_of_feature;
     };
typedef union ATTRIB_FEAT_u ATTRIB_FEAT;
```

5.2 Geometry

```
union CURVE_u
     {
     struct LINE_s                *line;
     struct CIRCLE_s              *circle;
     struct ELLIPSE_s             *ellipse;
     struct INTERSECTION_s        *intersection;
     struct TRIMMED_CURVE_s       *trimmed_curve;
     struct PE_CURVE_s            *pe_curve;
     struct B_CURVE_s             *b_curve;
     struct SP_CURVE_s            *sp_curve;
     struct POLYLINE_s            *polyline;
     };
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;
     struct MESH_s                *mesh;
     };
typedef union SURFACE_u    SURFACE;
union GEOMETRY_u
     {
     struct LATTICE_s             *lattice;
     union SURFACE_u               surface;
     union CURVE_u                 curve;
     struct POINT_s               *point;
     struct TRANSFORM_s           *transform;
     };
typedef union GEOMETRY_u GEOMETRY;
```

5.2.1 Curves

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

All curve nodes share the following common fields:

Field Name	Data Type	Description
node_id	int	Integer value unique to curve in part
attributes_features	pointer0	Attributes and featuresassociated with any curve
owner	pointer0	topological owner
next	pointer0	next curve in geometry chain

Field Name	Data Type	Description
previous	pointer0	previous curve in geometry chain
geometric_owner	pointer0	geometric owner node
sense	char	sense of curve “+” or “-” (see end of Geometry section)

```
struct ANY_CURVE_s              //Any Curve
    {
    int                           node_id;              // $d
    union  ATTRIB_FEAT_u          attributes_features;  // $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;
```

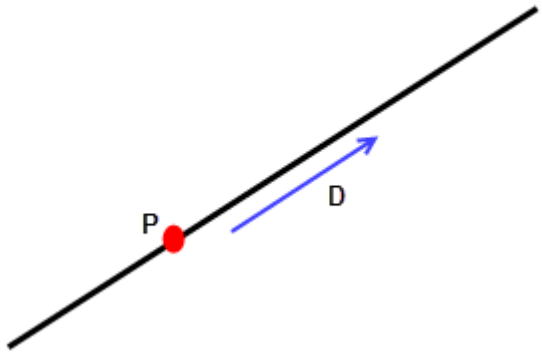
5.2.1.1 Line

A straight line has a parametric representation of the form:

`R(t) = P + t D`

Where:

- P is a point on the line
- D is its direction



Field Name	Data Types	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_FEAT_u          attributes_features;  // $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;
```

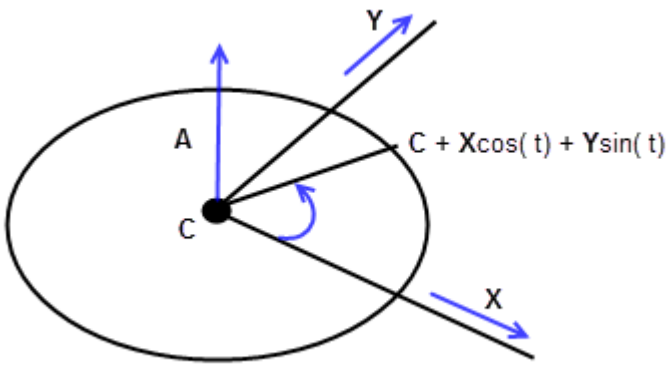
5.2.1.2 Circle

A circle has a parametric representation of the form:

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

Where:

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



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

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

```
struct CIRCLE_s == ANY_CURVE_s    //Circle
    {
    int                           node_id;              // $d
    union  ATTRIB_FEAT_u          attributes_features;  // $p
    union  CURVE_OWNER_u          owner;                // $p
    union  CURVE_u                next;                 // $p
    union  CURVE_u                previous;             // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;      // $p
    char                          sense;                // $c
    vector                        centre;               // $v
    vector                        normal;               // $v
    vector                        x_axis;               // $v
    double                        radius;               // $f
    };
typedef struct CIRCLE_s    *CIRCLE;
```

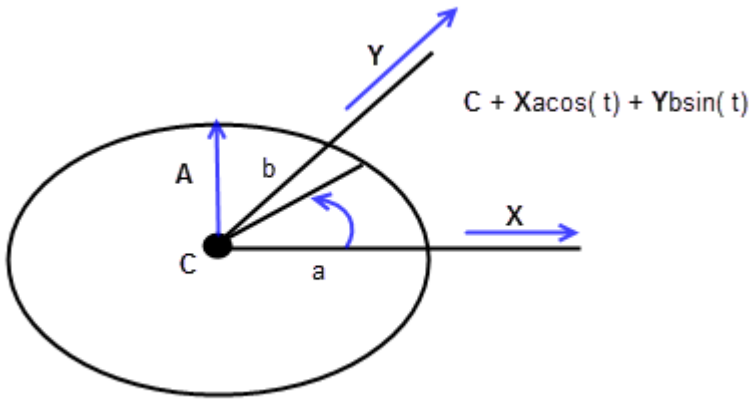
5.2.1.3 Ellipse

An ellipse has a parametric representation of the form:

`R(t) = C + a X cos(t) + b Y sin(t)`

Where:

- `C` is the centre of the ellipse
- `X` is the major axis
- a is the major radius
- `Y` and b are the minor axis and minor radius respectively



Field Name	Data Type	Description
`centre`	Vector	Centre of ellipse
`normal`	Vector	Normal to the plane containing the ellipse (a unit vector)

Field Name	Data Type	Description
x_axis	Vector	major axis in the plane of the ellipse (a unit vector)
major_radius	Double	major radius
minor_radius	Double	minor radius

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

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

5.2.1.4 B_CURVE (B-spline curve)

XT supports B-spline curves in full NURBS format. The mathematical description of these curves is:

- Non Uniform Rational B-splines as (NURBS), and

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

- the more simple Non Uniform B-spline
- Where:

  n = number of vertices (n_vertices in the PK standard form)

  $V_0 \ldots V_{n-1}$ are the B-spline vertices

  $w_0 \ldots w_{n-1}$ are the weights

$$P(t) = \sum_{i=0}^{n-1} b_i(t) V_i$$

$b_i\ (t), i = 0 \ldots n-1$ are the B-spline basis functions

KNOT VECTORS

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

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

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

THE NUMBER OF KNOTS AND VERTICES

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

THE VALID RANGE OF THE B-CURVE

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

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

PERIODIC B-CURVES

When the end of a B-curve meets its start sufficiently smoothly XT allows it to be defined to have periodic parameterisation. That is to say that if the valid range were from t_{degree} to $t_{n\_knots\ -\ 1\ -\ degree}$ then the difference between these values is called the period and the curve can continue to be evaluated with the same point reoccurring every period.

The minimal smoothness requirement for periodic curves in XT is tangent continuity to angular resolution, but we strongly recommend $C_{degree-1}$, or continuity in the $(degree\text{-}1)^{th}$ derivative. This

in turn is best achieved by repeating the first degree vertices at the end, and by matching knot intervals so that counting from the start of the defined range, t_{degree}, the first degree intervals between knots match the last degree intervals, and similarly matching the last degree knot intervals before the end of the defined range to the first degree intervals.

CLOSED B-CURVES

A periodic B-curve shall also be closed, but it 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.

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

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

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

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

```
struct B_CURVE_s == ANY_CURVE_s   //B curve
    {
    int                           node_id;              // $d
    union  ATTRIB_FEAT_u          attributes_features;  // $p
    union  CURVE_OWNER_u          owner;                // $p
    union  CURVE_u                next;                 // $p
    union  CURVE_u                previous;             // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;      // $p
    char                          sense;                // $c
    struct NURBS_CURVE_s         *nurbs;                // $p
    struct CURVE_DATA_s          *data;                 // $p
    };
typedef struct B_CURVE_s     *B_CURVE;
```

The data stored in the XT data for a NURBS_CURVE is

Field Name	Data Type	Description
degree	Short	degree of the curve
n_vertices	Int	number of control vertices (“poles”)
vertex_dim	Short	dimension of control vertices
n_knot	Int	number of distinct knots
knot_type	Byte	form of knot vector
periodic	Logical	true if curve is periodic
closed	Logical	true if curve is closed
rational	Logical	true if curve is rational
curve_form	Byte	shape of curve, if special
bspline_vertices	Pointer	control vertices node
knot_mult	Pointer	knot multiplicities node
knot	Pointer	knots node

The knot_type enum is used to describe whether or not the knot vector has a certain regular spacing or other common property:

```
typedef enum
    {
    SCH_unset = 1,             //Unknown
    SCH_non_uniform = 2,       //Known to be not special
    SCH_uniform = 3,           //Uniform knot set
    SCH_quasi_uniform = 4,     //Uniform apart from bezier ends
    SCH_piecewise_bezier = 5,  //Internal multiplicity of
                                 order-1
    SCH_bezier_ends = 6        //Bezier ends, no other property
    }
    SCH_knot_type_t;
```

A uniform knot set is one where all the knots are of multiplicity one and are equally spaced. A curve has bezier ends if the first and last knots both have multiplicity ‘order’.

The curve_form enum describes the geometric shape of the curve. The parameterisation of the curve is not relevant.

```
typedef enum
    {
    SCH_unset          = 1,     //Form is not known
    SCH_arbitrary      = 2,     //Known to be of no particular
                                  shape
    SCH_polyline       = 3,
    SCH_circular_arc   = 4,
    SCH_elliptic_arc   = 5,
    SCH_parabolic_arc  = 6,
    SCH_hyperbolic_arc = 7
    }
    SCH_curve_form_t;
struct NURBS_CURVE_s             //NURBS curve
    {
    short                          degree;             // $n
    int                            n_vertices;         // $d
    short                          vertex_dim;         // $n
    int                            n_knots;            // $d
    SCH_knot_type_t                knot_type;          // $u
    logical                        periodic;           // $l
    logical                        closed;             // $l
    logical                        rational;           // $l
    SCH_curve_form_t               curve_form;         // $u
    struct BSPLINE_VERTICES_s     *bspline_vertices;   // $p
    struct KNOT_MULT_s            *knot_mult;          // $p
    struct KNOT_SET_s             *knots;              // $p
    };
typedef struct NURBS_CURVE_s *NURBS_CURVE;
```

The bspline vertices node is simply an array of doubles; vertex_dim doubles together define one control vertex. Thus the length of the array is n_vertices * vertex_dim.

```
struct BSPLINE_VERTICES_s        // B-spline vertices
    {
    double                 vertices[ 1 ];              // $f[]
    };
typedef struct BSPLINE_VERTICES_s *BSPLINE_VERTICES;
```

The knot vector of the NURBS_CURVE is stored as an array of distinct knots and an array describing the multiplicity of each distinct knot. Hence the two nodes:

```
struct KNOT_SET_s                //Knot set
    {
    double                 knots[ 1 ];                 // $f[]
    };
typedef struct KNOT_SET_s *KNOT_SET;
```

and

```
struct KNOT_MULT_s              //Knot multiplicities
    {
    short                  mult[ 1 ];                  // $n[]
    };
typedef struct KNOT_MULT_s *KNOT_MULT;
```

The data stored in the XT data for a CURVE_DATA node is:

```
typedef enum
    {
    SCH_unset                      = 1,  //check has not been
                                          performed
    SCH_no_self_intersections      = 2,  //passed checks
    SCH_self_intersects            = 3,  //fails checks
    SCH_checked_ok_in_old_version = 4  //see below
    }
    SCH_self_int_t;
struct CURVE_DATA_s  //curve_data
    {
    SCH_self_int_t               self_int;             // $u
    Struct HELIX_CU_FORM_s      *analytic_form         // $p
    };
typedef struct CURVE_DATA_s *CURVE_DATA;
```

The self-intersection enum describes whether or not the geometry has been checked for self-intersections, and whether such self-intersections were found to exist:

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 B-curve fits this specification.

5.2.1.5 Intersection

An intersection curve is one of the branches of a surface / surface intersection. XT represents these curves exactly; the information held in an intersection curve node is sufficient to identify the particular intersection branch involved, to identify the behaviour of the curve at its ends, and to evaluate precisely at any point in the curve. Specifically, the data is:

- The two surfaces involved in the intersection.
- The two ends of the intersection curve. These are referred to as the 'limits' of the curve. They identify the particular branch involved.
- An ordered array of points along the curve. This array is referred to as the 'chart' of the curve. It defines the parameterisation of the curve, which increases as the array index increases.

The natural tangent to the curve at any point (i.e. in the increasing parameter direction) is given by the vector cross-product of the surface normals at that point, taking into account the senses of the surfaces.

Singular points where the cross-product of the surface normals is zero, or where one of the surfaces is degenerate, are called terminators. Intersection curves do not contain terminators in their interior. At terminators, the tangent to the curve is defined by the limit of the curve tangent as the curve parameter approaches the terminating value.

Field Name	Data Type	Description
surface	pointer array [2]	surfaces of intersection curve
chart	Pointer	array of hvecs on the curve – see below
start	Pointer	start limit of the curve
end	Pointer	end limit of the curve
intersection_data	Pointer	optional structure for storing additional information associated with an intersection curve

```
struct INTERSECTION_s == ANY_CURVE_s // Intersection
    {
    int                              node_id;             // $d
    union  ATTRIB_FEAT_u             attributes_features; // $p
    union  CURVE_OWNER_u             owner;               // $p
    union  CURVE_u                   next;                // $p
    union  CURVE_u                   previous;            // $p
    struct GEOMETRIC_OWNER_s        *geometric_owner;     // $p
    char                             sense;               // $c
    union  SURFACE_u                surface[ 2 ];         // $p[2]
    struct CHART_s                  *chart;               // $p
    struct LIMIT_s                  *start;               // $p
    struct LIMIT_s                  *end;                 // $p
nolog struct INTERSECTION_DATA *intersection_data         // $p
    };
typedef struct INTERSECTION_s *INTERSECTION;
```

A point on an intersection curve is stored in a data structure called an hvec (hepta-vec, or 7-vector):

```
typedef struct hvec_s         // hepta_vec
    {
    vector          Pvec;               //position
    double          u[2];               //surface parameters
    double          v[2];
    vector          Tangent;            //curve tangent
    double          t;                  //curve parameter
    } hvec;
```

Where:

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

Note: Only the pvec part of an hvec is actually transmitted.

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

```
struct CHART_s          //Chart
    {
    double              base_parameter;             // $f
    double              base_scale;                 // $f
    int                 chart_count;                // $d
    double              chordal_error;              // $f
    double              angular_error;              // $f
    double              parameter_error[2];         // $f[2]
    hvec                hvec[ 1 ];                  // $h[]
    };
typedef struct CHART_s *CHART;
```

Where:

- base_parameter is the parameter of the first hvec in the chart.
- base_scale determines the scale of the parameterization (see below).
- chart_count is the length of the hvec array.
- chordal_error is an estimate of the maximum deviation of the curve from the piecewise-linear approximation given by the hvec array. It may be null.
- angular_error is the maximum angle between the tangents of two sequential hvecs. It may be null.
- parameter_error[] is always [null, null].
- hvec[] is the ordered array of hvec's.

The limits of the intersection curve are stored in the following data structure:

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

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 ‘term_use’ field takes one of the following values

```
const char SCH_unset_char = '?' //generic uninvestigated value
const char SCH_first ='F'       //first item
const char SCH_second ='S'      //second item
```

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

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

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

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

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

Let f_0 = base_scale

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

Then t_0 = base_parameter

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

The factors f_i are chosen so that the parameterisation is C1. The parameter of a point between two chart points is given by projecting the point onto the chord between the previous and next chart points.

The point on the intersection curve corresponding to a given parameter is defined as follows:

- For a parameter equal to that of a chart point, it is the position of the chart point.
- For a parameter interior to the chart, it is the local point of intersection of three surfaces: the two surfaces of the intersection, and a plane defined by the chart. If the parameter t lies between chart parameters t_i, t_{i+1}, then the chord point corresponding to t lies at

$(t_{i+1} - t) / (t_{i+1} - t_i) P_i + (t - t_i) / (t_{i+1} - t_i) P_{i+1}$

The plane lies through this point and is orthogonal to the chord (P_{i+1}, P_i).

- For a parameter between a branch chart point and a terminator, it is the local point of intersection of three surfaces: one of the intersection surfaces and two planes. Surface[0] is used unless
  - it is singular at the terminator and surface[1] is not
  - or it has the node type BLEND_BOUND_nt
  - or 'term_use' specifies to use surface[1]

The first plane contains the chord between the branch and the terminator, and the normal of the chosen intersection surface at the terminator or the curve tangent at the branch chart point if the surface normal cannot be defined. The second plane is the plane orthogonal to the chord between the branch and terminator points through the chord point as calculated above.

The intersection_data node is an optional structure for storing surface uv parameters from hvecs that are associated with an intersection curve.

Note: The intersection_data must match the hvecs

```
struct INTERSECTION_DATA_s          //Intersection data
    {
    SCH_intersection_uv_type_t uv_type; // $u
    double values [1]; // $f[]
    };
typedef struct INTERSECTION_DATA_s*INTERSECTION_DATA;
              inline double      *SCH_INTERSECTION_DATA_values(INTERSECTION_DATA
self)
{
   return self -> values;
}
SCH_define_init_fn_m(INTERSECTION_DATA_s, self,
                     self -> uv_type = SCH_intersection_uv_none;
                     double*values   = SCH_INTERSECTION_DATA_values(self);
                         for (int i = 0; i< n_variable; ++i)
                         values [i] = null;
                    )
```

The intersection_data node contains an enum value and a variable length double array. The enum value specifies the uv values stored in the values array and is set based on the following:

```
typedef enum
    {
    SCH_intersection_uv_none      = 1,
    SCH_intersection_uv_first     = 2,
    SCH_intersection_uv_second    = 3,
    SCH_intersection_uv_both      = 4,
    }
    SCH_intersection_uv_type_t;
char *SCH_intersection_uv_type_sprintf
```

The uv values are converted to the number of parameters which are stored for each chart hvec as follows:

- If SCH_intersection_uv_none, the number of parameters is 0
- If SCH_intersection_uv_first or SCH_intersection_uv_second, the number of parameters is 2
- If SCH_intersection_uv_both, the number of parameters is 4

The variable length double array contains these parameters, and the start and end limits. The values for the start and end limits can be found in the variable length arrays in the LIMIT start, and LIMIT end fields of the INTERSECTION node.

The number of values in the double array is calculated as:

(The number of chart points + The number of terminator limits) * (The number of parameters per hvec)

For each terminator present in the array the number of values will increase by 0, 2, or 4 depending on the intersection_uv_type field. For example, if both the start and the end limits are terminators and the intersection_uv_type is set to SCH_intersection_uv_both the value will increase by 8.

The order of values in the array is as follows:

If the start limit is a terminator:

If the intersection_uv_type is...	The order of values in array is...
SCH_intersection_uv_first or SCH_intersection_uv_both	intersection node ->start->hvec[0].u[0] intersection node ->start->hvec[0].v[0]
SCH_intersection_uv_second or SCH_intersection_uv_both	intersection node ->start ->hvec[0].u[1] intersection node ->start ->hvec[0].v[1]

For each hvec in the chart:

If the intersection_uv_type is...	The order of values in array is...
SCH_intersection_uv_first or SCH_intersection_uv_both	intersection node->chart ->hvec[i].u[0] intersection node->chart ->hvec[i].v[0]
SCH_intersection_uv_second or SCH_intersection_uv_both	intersection node->chart ->hvec[i].u[1] intersection node->chart ->hvec[i].v[1]

chart hvecs are wrapped in a loop where i = 0 to the (number of chart hvecs -1).

If end limit is a terminator:

If the intersection_uv_type is...	The order of values in array is...
SCH_intersection_uv_first or SCH_intersection_uv_both	intersection node->end ->hvec[0].u[0] intersection node->end ->hvec[0].v[0]
SCH_intersection_uv_second or SCH_intersection_uv_both	intersection node->end ->hvec[0].u[1] intersection node->end ->hvec[0].v[1]

5.2.1.6 Trimmed_curve

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

The rules governing the parameter fields and points are:

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

In addition,

For open basis curves.

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

For periodic basis curves

- parm_1 shall lie in the base range of the basis curve.
- If the whole basis curve is required then parm_1 and parm_2 should be a period apart and point_1 = point_2. Equality of parm_1 and parm_2 is not permitted.
- parm_1 and parm_2 shall not be more than a period apart.

For closed but non-periodic basis curves

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

The sense of a trimmed curve is positive.

Field name	Data type	Description
basis curve	pointer	basis curve
point_1	vector	start of trimmed portion
point_2	vector	end of trimmed portion
parm_1	double	parameter on basis curve corresponding to point_1
parm_2	double	parameter on basis curve corresponding to point_2

```
struct TRIMMED_CURVE_s == ANY_CURVE_s  //Trimmed Curve
    {
    int                            node_id;             // $d
    union  ATTRIB_FEAT_u           attributes_features; // $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;
```

5.2.1.7 PE_CURVE (Foreign geometry curve)

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

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

Field name	Data type	Description
type	char	whether internal or external
data	pointer	internal or external data
tf	pointer(0)	transform applied to geometry
internal geom	pointer array	reference to other related geometry

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

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

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

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

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

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

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

5.2.1.8 SP_CURVE

An SP-curve is the 3D curve resulting from embedding a 2D curve in the parameter space of a surface.

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

Field name	Data type	Description
surface	pointer	surface
b_curve	pointer	2D B-curve
original	pointer(0)	not used
tolerance_to_original	double	not used

```
struct SP_CURVE_s == ANY_CURVE_s //SP curve
    {
    int                            node_id;                 // $d
    union  ATTRIB_FEAT_u           attributes_features;     // $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;
```

5.2.1.9 Polyline

A polyline describes a connected chain of linear segments. It takes the following additional field:

Field name	Data type	Description
data	pointer	contains the data information of the polyline.

```
struct POLYLINE_s == ANY_CURVE_s  //Polyline
{
  int                          node_id;             //$d
  union ATTRIB_FEAT_u          attributes_features; //$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;               //$c
  struct POLYLINE_DATA_s      *data;                //$p
};
typedef struct POLYLINE_s *POLYLINE;
```

The data stored in the XT data for a POLYLINE is as follows:

```
struct POLYLINE_DATA_s  //Polyline data
{
  int       n_pvecs;              //$d
  logical   closed;               //$I
  double    base_parm;            //$f
  struct    POINT_VALUES_s*pvec;  //$p
};
typedef struct POLYLINE_DATA_s*POLYLINE_DATA;
```

Where:

Field name	Data type	Description
n_pvecs	integer	number of point vectors
closed	logical	true if polyline is closed
base_parm	double	the parameter of the first pvec in the polyline
pvec	pointer	point vectors that describe the shape of the polyline

5.2.2 Surfaces

All surface nodes share the following common fields:

Field name	Data type	Description
node_id	int	integer value unique to surface in part
attributes_features	pointer0	attributes and features associated with surface
owner	pointer	topological owner
next	pointer0	next surface in geometry chain
previous	pointer0	previous surface in geometry chain
geometric_owner	pointer0	geometric owner node
sense	char	sense of surface: '+' or '-'(see end of Geometry section)

```
struct ANY_SURF_s     //Any Surface
    {
    int                             node_id;             // $d
    union  ATTRIB_FEAT_u            attributes_features; // $p
    union  SURFACE_OWNER_u          owner;               // $p
    union  SURFACE_u                next;                // $p
    union  SURFACE_u                previous;            // $p
    struct GEOMETRIC_OWNER_s       *geometric_owner;     // $p
    char                            sense;               // $c
    };
typedef struct ANY_SURF_s *ANY_SURF;
```

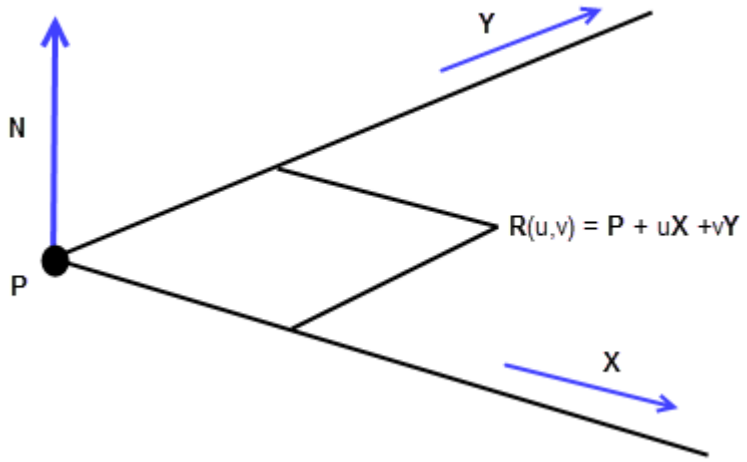
5.2.2.1 Plane

A plane has a parametric representation of the form

$R(u, v) = P + uX + vY$

Where:

- P is a point on the plane



- X and Y are axes in the plane

Field name	Data type	Description
pvec	vector	point on the plane
normal	vector	normal to the plane (a unit vector)
x_axis	vector	X axis of the plane (a unit vector)

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

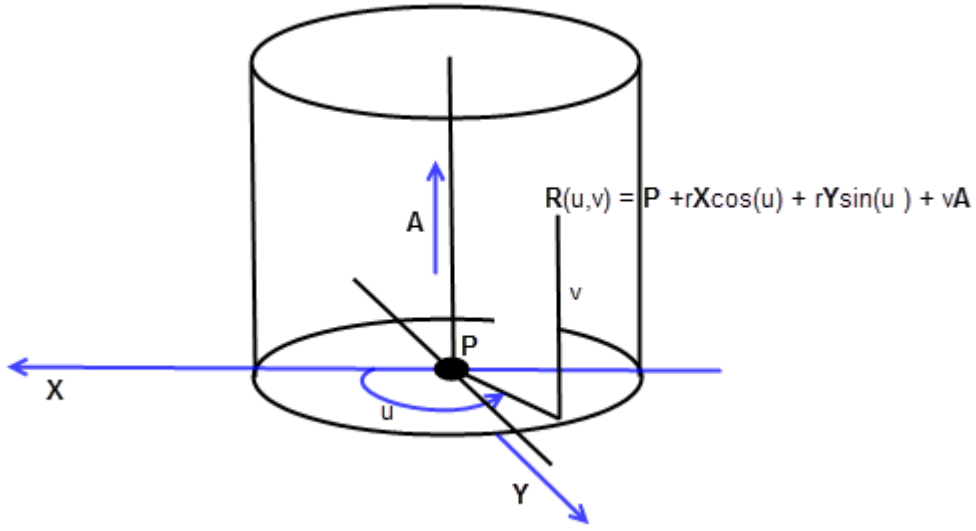
```
struct PLANE_s == ANY_SURF_s      //Plane
    {
    int                           node_id;             // $d
    union  ATTRIB_FEAT_u          attributes_features; // $p
    union  SURFACE_OWNER_u        owner;               // $p
    union  SURFACE_u              next;                // $p
    union  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;
```

5.2.2.2 Cylinder

A cylinder has a parametric representation of the form:

R(u,v) = P + rXcos(u) + rYsin(u) + vA

Where:



- P is a point on the cylinder axis.
- r is the cylinder radius.
- A is the cylinder axis.
- X and Y are unit vectors such that A, X and Y form an orthonormal set.

Field name	Data type	Description
pvec	vector	point on the cylinder axis
axis	vector	direction of the cylinder axis (a unit vector)
radius	double	radius of cylinder
x_axis	vector	X axis of the cylinder (a unit vector)

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

```
struct CYLINDER_s == ANY_SURF_s   //Cylinder
    {
    int                           node_id;             // $d
    union  ATTRIB_FEAT_u          attributes_features; // $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;
```

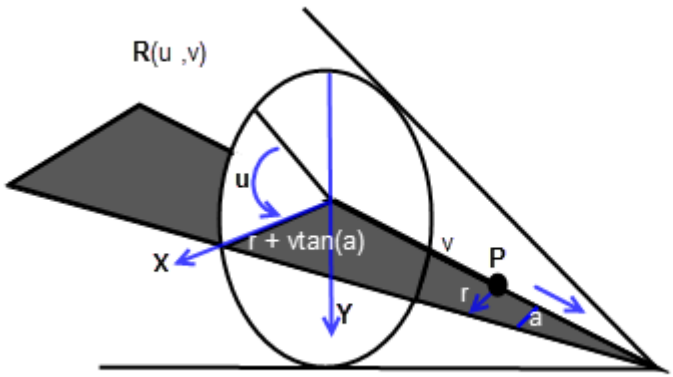
5.2.2.3 Cone

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

`R(u, v) = P - vA + (Xcos(u) + Ysin(u))(r + vtan(a))`

Where:

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



Note: At the PK interface, the cone axis is in the opposite direction to that stored in the XT data.

Field name	Data type	Description
pvec	vector	point on the cone axis
axis	vector	direction of the cone axis (a unit vector)
radius	double	radius of the cone at its pvec
sin_half_angle	double	sine of the cone's half angle
cos_half_angle	double	cosine of the cone's half angle
x_axis	vector	X axis of the cone (a unit vector)

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

```
struct CONE_s == ANY_SURF_s       //Cone
    {
    int                           node_id;             // $d
    union  ATTRIB_FEAT_u          attributes_features; // $p
    union  SURFACE_OWNER_u        owner;               // $p
    union  SURFACE_u              next;                // $p
    union  SURFACE_u              previous;            // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;     // $p
    char                          sense;               // $c
    vector                        pvec;                // $v
    vector                        axis;                // $v
    double                        radius;              // $f
    double                        sin_half_angle;      // $f
    double                        cos_half_angle;      // $f
    vector                        x_axis;              // $v
    };
typedef struct CONE_s *CONE;
```

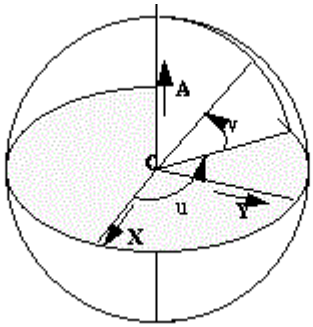
5.2.2.4 Sphere

A sphere has a parametric representation of the form:

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

Where:

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



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

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

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

```
struct SPHERE_s == ANY_SURF_s     //Sphere
    {
    int                           node_id;             // $d
    union  ATTRIB_FEAT_u          attributes_features; // $p
    union  SURFACE_OWNER_u        owner;               // $p
    union  SURFACE_u              next;                // $p
    union  SURFACE_u              previous;            // $p
    struct GEOMETRIC_OWNER_s     *geometric_owner;     // $p
    char                          sense;               // $c
    vector                        centre;              // $v
    double                        radius;              // $f
    vector                        axis;                // $v
    vector                        x_axis;              // $v
    };
typedef struct SPHERE_s *SPHERE;
```

5.2.2.5 Torus

A torus has a parametric representation of the form

$$R(u, v) = C + (X \cos(u) + Y \sin(u))(a + b \cos(v)) + b A \sin(v)$$

Where:

- C is centre of the torus.
- A is the torus axis.
- a is the major radius.
- b is the minor radius.
- X and Y are unit vectors such that A, X and Y form an orthonormal set.

In XT, there are three types of torus:

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

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

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

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

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

Field name	Data type	Description
`centre`	vector	centre of the torus
`axis`	vector	axis of the torus (a unit vector)
`major_radius`	double	major radius
`minor_radius`	double	minor radius
`x_axis`	vector	X axis of the torus (a unit vector)

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

```
struct TORUS_s == ANY_SURF_s      //Torus
    {
    int                           node_id;             // $d
    union  ATTRIB_FEAT_u          attributes_features; // $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;
```

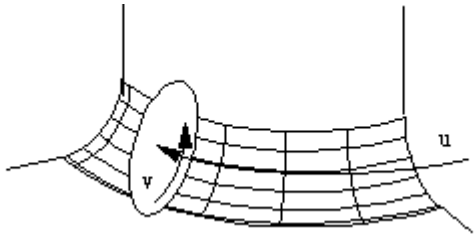
5.2.2.6 Blended_edge (Rolling ball blend)

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

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

Where:

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



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

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

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

```
struct BLENDED_EDGE_s == ANY_SURF_s  //Blended edge
    {
    int                             node_id;             // $d
    union  ATTRIB_FEAT_u            attributes_features; // $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.

XT data can contain blends of the following types:

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

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

For cliff edge blends, one of the surfaces will be a `blended_edge` with a range of [0,0]; its spine will be the cliff edge curve, and its supporting surfaces will be the surfaces of the faces adjacent to the cliff edge. Its type will be `R`.

The limit fields will only be non-null if the spine curve is periodic but the edge curve being blended has terminators – for example if the spine is elliptical but the blend degenerates. In this case the two LIMIT nodes, of type 'L', determine the extent of the spine.

5.2.2.7 Blend_bound (Blend boundary surface)

A `blend_bound` surface is a construction surface, used to define the boundary curve where a blend becomes tangential to its supporting surface. It is an implicit surface defined internally so that it intersects one of the supporting surfaces along the boundary curve. It is orthogonal to the blend and the supporting surface along this boundary curve. The supporting surface corresponding to the `blend_bound` is

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

Blend boundary surfaces have no parameterisation, but are defined by the distance function

```
f(X) = f0 (X + r1 * grad_f1 (X)) – r0
```

Where:

- `f0` is the surface distance function of the supporting surface corresponding to the `blend_bound`
- `r0` is the blend radius corresponding to that supporting surface
- `f1` is the surface distance function of the other supporting surface of the blend
- `r1` is the blend radius corresponding to the other supporting surface

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

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

Field name	Data type	Description
`boundary`	short	index into supporting surface array
`blend`	pointer	corresponding blend surface

```
struct BLEND_BOUND_s == ANY_SURF_s  //Blend boundary
    {
    int                             node_id;             // $d
    union  ATTRIB_FEAT_u            attributes_features; // $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]`

5.2.2.8 Offset_surf

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

Field name	Data type	Description
`check`	char	check status
`true_offset`	logical	not used
`surface`	pointer	underlying surface
`offset`	double	signed offset distance
`scale`	double	for internal use only – may be set to null

```
struct OFFSET_SURF_s == ANY_SURF_s  //Offset surface
    {
    int                             node_id;             // $d
    union  ATTRIB_FEAT_u            attributes_features; // $p
    union  SURFACE_OWNER_u          owner;               // $p
    union  SURFACE_u                next;                // $p
    union  SURFACE_u                previous;            // $p
    struct GEOMETRIC_OWNER_s       *geometric_owner;     // $p
    char                            sense;               // $c
    char                            check;               // $c
    logical                         true_offset;         // $l
    union  SURFACE_u                surface;             // $p
    double                          offset;              // $f
    double                          scale;               // $f
    };
typedef struct OFFSET_SURF_s *OFFSET_SURF;
```

The offset surface is subject to the following restrictions:

- The offset distance shall not be within modeller linear resolution of zero
- The sense of the offset surface shall be the same as that of the underlying surface
- Offset surfaces may not share a common underlying surface

The ‘check’ field may take one of the following values:

```
const char SCH_valid      = 'V';    // valid
const char SCH_invalid    = 'I';    // invalid
const char SCH_unchecked  = 'U';    // has not been checked
```

5.2.2.9 B_surface

XT supports B-spline surfaces in full NURBS format.

B-SURFACE DEFINITION

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

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

For attachment to topology a B-surface is required to have G1 continuity. That is to say that the surface normal direction shall be continuous.

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

Field name	Data type	Description
nurbs	pointer	Geometric definition
data	pointer0	Auxiliary information

```
struct B_SURFACE_s == ANY_SURF_s  //B surface
    {
    int                            node_id;             // $d
    union  ATTRIB_FEAT_u           attributes_features; // $p
    union  SURFACE_OWNER_u         owner;               // $p
    union  SURFACE_u               next;                // $p
    union  SURFACE_u               previous;            // $p
    struct GEOMETRIC_OWNER_s      *geometric_owner;     // $p
    char                           sense;               // $c
    struct NURBS_SURF_s           *nurbs;               // $p
    struct SURFACE_DATA_s         *data;                // $p
    };
typedef struct B_SURFACE_s *B_SURFACE;
```

The data stored in the XT data for a NURBS surface is:

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

The surface form enum is defined below.

```
typedef enum
    {
    SCH_unset              = 1,          //Unknown
    SCH_arbitrary          = 2,          //No particular shape
    SCH_planar             = 3,
    SCH_cylindrical        = 4,
    SCH_conical            = 5,
    SCH_spherical          = 6,
    SCH_toroidal           = 7,
    SCH_surf_of_revolution = 8,
    SCH_ruled              = 9,
    SCH_quadric            = 10,
    SCH_swept              = 11
    }
    SCH_surface_form_t;
```

```
struct NURBS_SURF_s                //NURBS surface
    {
    logical                        u_periodic;          // $l
    logical                        v_periodic;          // $l
    short                          u_degree;            // $n
    short                          v_degree;            // $n
    int                            n_u_vertices;        // $d
    int                            n_v_vertices;        // $d
    SCH_knot_type_t                u_knot_type;         // $u
    SCH_knot_type_t                v_knot_type;         // $u
    int                            n_u_knots;           // $d
    int                            n_v_knots;           // $d
    logical                        rational;            // $l
    logical                        u_closed;            // $l
    logical                        v_closed;            // $l
    SCH_surface_form_t             surface_form;        // $u
    short                          vertex_dim;          // $n
    struct BSPLINE_VERTICES_s     *bspline_vertices;    // $p
    struct KNOT_MULT_s            *u_knot_mult;         // $p
    struct KNOT_MULT_s            *v_knot_mult;         // $p
    struct KNOT_SET_s             *u_knots;             // $p
    struct KNOT_SET_s             *v_knots;             // $p
    };
typedef struct NURBS_SURF_s *NURBS_SURF;
```

The `bspline_vertices`, `knot_set` and `knot_mult` nodes and the `knot_type` enum are described in the documentation for BCURVE.

The ‘surface data’ field in a B-surface node is a structure designed to hold auxiliary or ‘derived’ data about the surface: it is not a necessary part of the definition of the B-surface. It may be null, or the majority of its individual fields may be null. It is recommended that it only be set by Parasolid.

```
struct SURFACE_DATA_s          //auxiliary surface data
    {
    interval                   original_uint;           // $i
    interval                   original_vint;           // $i
    interval                   extended_uint;           // $i
    interval                   extended_vint;           // $i
    SCH_self_int_t             self_int;                // $u
    char                       original_u_start;        // $c
    char                       original_u_end;          // $c
    char                       original_v_start;        // $c
    char                       original_v_end;          // $c
    char                       extended_u_start;        // $c
    char                       extended_u_end;          // $c
    char                       extended_v_start;        // $c
    char                       extended_v_end;          // $c
    char                       analytic_form_type;      // $c
    char                       swept_form_type;         // $c
    char                       spun_form_type;          // $c
    char                       blend_form_type;         // $c
    void                      *analytic_form;           // $p
    void                      *swept_form;              // $p
    void                      *spun_form;               // $p
    void                      *blend_form;              // $p
    };
typedef struct SURFACE_DATA_s *SURFACE_DATA;
```

The ‘original_’ and ‘extended_’ parameter intervals and corresponding character fields `original_u_start` etc. are all connected with XT’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:

```
const char SCH_degenerate = 'D';    //Degenerate edge
const char SCH_periodic   = 'P';    //Periodic parameterization
const char SCH_bounded    = 'B';    //Parameterization bounded
const char SCH_closed     = 'C';    //Closed, but not periodic
```

The separate fields original_u_start and extended_u_start etc. are necessary because an extension may cause the corresponding parameter boundary to become degenerate.

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

```
const char SCH_unset_char = '?'; //generic uninvestigated value
```

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

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

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

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

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

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

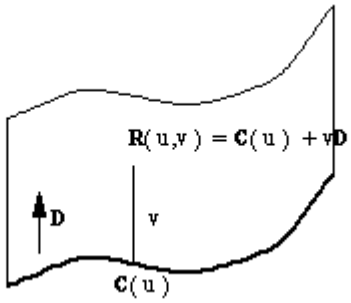
5.2.2.10 Swept_surf

A swept surface has a parametric representation of the form:

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

Where:

- `C(u)` is the section curve.
- `D` is the sweep direction (unit vector).
- `C` shall not be an intersection curve, a trimmed curve, an SP-curve or a PE-curve/foreign curve. It must be analytic or a B-curve.
- The swept surface inherits its u parameterisation from the section curve.



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

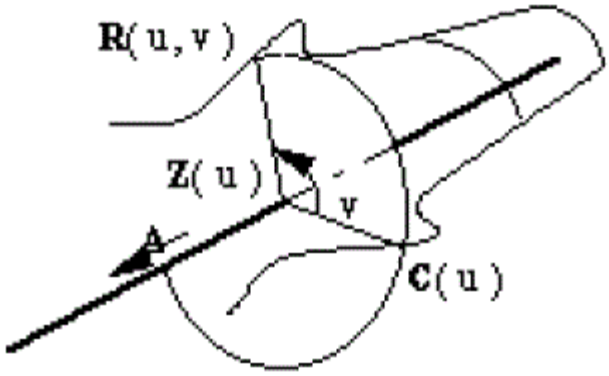
```
struct SWEPT_SURF_s == ANY_SURF_s  //Swept surface
    {
    int                             node_id;             // $d
    union  ATTRIB_FEAT_u            attributes_features; // $p
    union  SURFACE_OWNER_u          owner;               // $p
    union  SURFACE_u                next;                // $p
    union  SURFACE_u                previous;            // $p
    struct GEOMETRIC_OWNER_s       *geometric_owner;     // $p
    char                            sense;               // $c
    union  CURVE_u                  section;             // $p
    vector                          sweep;               // $v
    double                          scale;               // $f
    };
typedef struct SWEPT_SURF_s *SWEPT_SURF;
```

5.2.2.11 Spun_surf

A spun surface has a parametric representation of the form:

`R(u, v) = Z(u) + (C(u) - Z(u))cos(v) + A X (C(u) - Z(u)) sin(v)`

Where:



- C(u) is the profile curve.
- Z(u) is the projection of C(u) onto the spin axis.
- A is the spin axis direction (unit vector).
- C shall not be an intersection curve,a trimmed curve, an SP-curve or a PE-curve/foreign curve. It must be analytic or a B-curve.

Note: Z(u) = P + ((C(u) - P). A)A where P is a reference point on the axis.

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

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

The ‘start’ and ‘end’ vectors correspond to physical degeneracies on the spun surface caused by the profile curve crossing the spin axis at that point. The values `start_param` and `end_param` are the corresponding parameters on the curve. These parameter values define the valid range for the u parameter of the surface. If either value is null, then the valid range for u is infinite in that direction. For example, for a straight line profile curve intersecting the spin axis at the parameter `t=1`, values of null for `start_param` and 1 for `end_param` would define a cone with u parameterisation(-infinity, 1].

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

5.2.2.12 PE_surf (Foreign geometry surface)

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

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

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

```
struct PE_SURF_s == ANY_SURF_s  //PE_surface
{
int  node_id;  //$d
union ATTRIB_FEAT_u               attributes_features;   //$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’.

5.2.3 Mesh Surfaces

Each MESH surface node references a PSM_MESH node containing facet data. Meshes cannot be shared by more than one face of a body.

Field name	Data type	Description
mesh box	box	may contain an axis-aligned box bounding the mesh
transform	pointer0	transform applied
rcv_key	pointer0	key of XMM file containing PSM mesh data. This field is not used.
rcv_index	int	unique integer value corresponding to the index of the PSM_MESH in the POINTER_LIS_BLOCK used as the root node in the corresponding XT mesh data file, if used
psm_imesh	pointer0	a pointer to the PSM_MESH node when mesh data is embedded in the XT part or partition file
pff_imesh	pointer0	a pointer to mesh data in the internal debug format. This is for internal use only.

```
struct MESH_s == ANY_SURF_s  //Mesh
{
int                          node_id;              //$d
union  ATTRIB_FEAT_u         attributes_features;  //$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;                //$c
box                          mesh_box;             //$b
struct TRANSFORM_s          *transform;            //$p
union  MESH_KEY_u            rcv_key;              //$p
int                          rcv_index;            //$d
struct PSM_MESH_s           *psm_imesh;            //$p
struct PFF_MESH_s           *pff_imesh;            //$p
};
typedef struct MESH_s *MESH;
```

The `rcv_key` field is not used and must always be set to null pointer.

The PSM mesh data may be embedded in the XT part or partition file, or stored in an associated XT mesh data file. The definition of the PSM mesh data is identical in both cases. When PSM mesh data is stored in an associated XT mesh data file the `psm_imesh` field must be set to null pointer.

5.2.3.1 PSM mesh

Mesh data is stored in a PSM_MESH node, which is referenced by the MESH node.

The mesh data is described using the following fields:

Field Name	Data Type	Description
`precision`	byte	number format used to store the mesh data
`owner`	pointer0	mesh data owner
`position_pool`	pointer	array of positions. See Section 5.2.3.2
`normal_pool`	pointer0	array of directions stored in polar coordinates. See Section 5.2.3.4
`position_indices`	pointer	array of 3N indices into the position pool, where N is the number of facets. See Section 5.2.3.3
`normal_type`	byte	form of mesh normals
`normal_indices`	pointer0	array of 3N indices into the normal pool, where N is the number of facets. See Section 5.2.3.5

```
typedef enum
{
SCH_mesh_normal_none       = 1,
SCH_mesh_normal_per_vertex = 2,
SCH_mesh_normal_per_facet  = 3
}
SCH_mesh_normal_type_t;

typedef enum
{
SCH_mesh_precision_double  = 1,
SCH_mesh_precision_single  = 2
}
SCH_mesh_precision_t;

struct PSM_MESH_s  //PSM Mesh
{
SCH_mesh_precision_t        precision;  //$u
struct MESH_s              *owner;  //$p
struct VECTOR_COMB_s       *position_pool;  //p
struct VECTOR_COMB_s       *normal_pool;  //$p
struct INTEGER_COMB_s      *position_indices;  //$p
SCH_mesh_normal_type_t      normal_type;  //$u
struct INTEGER_COMB_s      *normal_indices;  //$p
};
```

The `owner` field is reserved for future use and must be set to null pointer.

The `precision` field is reserved for future use and must be set to 1, i.e. double precision.

The `normal_type` defines whether normals are stored, and if so, whether storage is on a per-facet or per-vertex basis. See Section 5.2.3.5, “Normal indices”, for more information.

5.2.3.2 Position pool

The position pool is an indexed point cloud. The vector array is stored using “comb” nodes, each of which consists of a “spine” array containing pointers to “tooth” arrays. Each tooth array contains the vector information. All the teeth have the same length, which is a power of 2. See Section 5.2.7, “Comb nodes” for more information on the VECTOR_COMB node.

5.2.3.3 Position indices

The mesh is defined by each facet specifying 3 positions from the position pool. The indices for these positions are stored in an integer comb, similar to the vector comb. See Section 5.2.7, “Comb nodes” for more information on INTEGER_COMB nodes.

5.2.3.4 Normal pool

The normal pool is an indexed cloud of normals, i.e. unit vectors. These are stored in double-precision spherical polar coordinates.

The normal pool is optional, so need not exist at all.

5.2.3.5 Normal indices

If normals are stored with the mesh they must be stored for each vertex of each facet. The type of normal storage used by the mesh data is specified by the `normal_type` field.

- If no normals are stored, normal indices are not needed and must be set to null pointer.
- If the normals are per-facet, then the number of normal indices is the same as the number of position indices, and the normal indices and position indices are parallel arrays.
- Normal storage per-vertex is not implemented.

5.2.4 Lattices

Each LATTICE node references a LATTICE_DATA node containing the lattice geometry data.

Field name	Data type	Description
`node_id`	int	integer unique within part
`attribute_feature`	pointer0	attributes and features associated with lattice
`owner`	pointer0	owner
`next`	pointer0	next lattice in geometry chain
`previous`	pointer0	previous lattice in geometry chain
`geometric_owner`	pointer0	geometric owner node
`sense`	char	sense of lattice: '+' or '-' (see Section 5.2.8, "Curve, surface and lattice senses")
`data`	pointer0	lattice geometry data

```
struct LATTICE_s                    //Lattice
    {
    int                            node_id;             // $d
    union  ATTRIB_FEAT_u           attributes_feature;  // $p
    union  LATTICE_OWNER_u         owner;               // $p
    struct LATTICE_s              *next;                // $p
    struct LATTICE_s              *previous;            // $p
    struct GEOMETRIC_OWNER_S      *geometric_owner;     // $p
    char                           sense;               // $c
    union  LATTICE_DATA_u          data;                // $p
    };
typedef struct LATTICE_s *LATTICE
```

```
union LATTICE_OWNER_u
    {
    struct BODY_s                 *body;
    struct ASSEMBLY_s             *assembly;
    struct WORLD_s                *world;
    };

typedef union LATTICE_OWNER_u LATTICE_OWNER;
```

```
union LATTICE_DATA_u
    {
    DS_block_p_t                    any;
    struct LATTICE_DATA_IRREGULAR_s *irregular;
    };

typedef union LATTICE_DATA_u LATTICE_DATA;
```

5.2.4.1 Irregular lattice node

Lattice data is stored in a LATTICE_DATA_IRREGULAR node, which is referenced by the LATTICE node. The lattice data is described using the following fields:

Field name	Data type	Description
connectivity	pointer	graph of lattice. See the GRAPH_COMPACT node below for more information.
positions	pointer	array of positions
ball_type	byte	form of ball radii
ball_radius	double	constant radius for balls
ball_radii	pointer	array of variable radii for balls
ball_rod_radii	pointer	array of variable radii for rods at balls
ball_blend_type	byte	type of blend for the balls
ball_blend_size	double	constant blend size for balls
ball_blend_sizes	pointer	array of variable blend sizes for balls if ball_blend_type is anything other than SCH_ball_blend_none and ball_blend_size is zero.
rod_term_type	byte	form of rod terminal radii
rod_term_radius	double	constant terminal radius for rods
rod_start_radii	pointer	array of variable radii for start of rod
rod_end_radii	pointer	array of variable radii for ends of rod
rod_mid_type	byte	form of mid radius of rods
rod_mid_radius	double	constant mid radius for rods
rod_mid_radii	pointer	array of mid radii for rods

```
struct LATTICE_DATA_IRREGULAR_s            //General lattice data
    {
    struct GRAPH_COMPACT_s        *connectivity;       // $p
    struct VECTOR_COMB_s_         *positions;          // $p
    SCH_lattice_ball_type_t        ball_type;          // $u
    double                         ball_radius;        // $f
    struct REAL_COMB_s            *ball_radii;         // $p
    struct REAL_COMB_s            *ball_rod_radii;     // $p
    SCH_lattice_ball_blend_type_t  ball_blend_type;    // $u
    double                         ball_blend_size;    // $f
    struct REAL_COMB_s            *ball_blend_sizes;   // $p
    SCH_lattice_rod_term_type_t    rod_term_type;      // $u
    double                         rod_term_radius;    // $f
    struct REAL_COMB_s            *rod_start_radii;    // $p
    struct REAL_COMB_s            *rod_end_radii;      // $p
    SCH_lattice_rod_mid_type_t     rod_mid_type;       // $u
    double                         rod_mid_radius;     // $f
    struct REAL_COMB_s            *rod_mid_radii;      // $p
  };
typedef struct LATTICE_DATA_IRREGULAR_s *LATTICE_DATA_IRREGULAR;
```

The GRAPH_COMPACT node stores graph data for a lattice and is defined as follows:

```
typedef struct GRAPH_COMPACT_s  //Graph data
{
    struct INTEGER_COMB_s       *adjacency_indices;        // $p
    struct INTEGER_COMB_s_      *adjacencies;              // $p
};

typedef struct GRAPH_COMPACT_s *GRAPH_COMPACT;


SCH_define_init_fn_m( GRAPH_COMPACT_S, SELF,
                      self -> adjacency_indices = null;
                      self -> adjacencies       = null;
                    )
```

The `ball_type` enum defines the type of ball radii

```
typedef short short enum
    {
    SCH_ball_unset    = 0,
    SCH_ball_const    = 1,
    SCH_ball_variable = 2
    }
    SCH_lattice_ball_type_t;
```

The `ball_blend_type` enum defines the type of ball blend

```
typedef short short enum
    {
    SCH_ball_blend_none = 0,
    SCH_ball_blend_absolute = 1,
    SCH_ball_blend_relative = 2,
    }
    SCH_lattice_ball_blend_type_t;
```

The rod_term_type enum defines the type of rod terminal radii

```
typedef short short enum
    {
    SCH_rod_term_unset      = 0,
    SCH_rod_term_const      = 1,
    SCH_rod_term_derived    = 2,
    SCH_rod_term_variable_1 = 3,
    SCH_rod_term_variable_2 = 4
    }
    SCH_lattice_rod_term_type_t;
```

The rod_mid_type enum defines the type of the rod mid radii

```
typedef short short enum
    {
    SCH_rod_mid_unset      = 0,
    SCH_rod_mid_none       = 1,
    SCH_rod_mid_const      = 2,
    SCH_rod_mid_variable   = 3
    }
    SCH_lattice_rod_mid_type_t;
```

The arrays in the LATTICE_DATA_IRREGULAR node are stored using “comb” nodes, each of which consists of a “spine” array containing pointers to “tooth” arrays. Each tooth array contains the vector or real information. See Section 5.2.7, “Comb nodes” for more information.

5.2.5 Point

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

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

```
struct POINT_s                    //Point
    {
    int                           node_id;            // $d
    union  ATTRIB_FEAT_u
    attributes_features;                              // $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;
```

5.2.6 Transform

Field name	Data type	Description
node_id	int	integer unique within part
owner	pointer	owning instance or world
next	pointer0	next transform in chain
previous	pointer0	previous pointer in chain
rotation_matrix	double[3][3]	rotation component
translation_vector	vector	translation component
scale	double	scaling factor
flag	byte	binary flags indicating non-trivial components
perspective_vector	vector	perspective vector (always null vector)
precision	pointer0	additional precision data for the transform

The transform acts as:

```
x' = (rotation_matrix . x + translation_vector) * scale
```

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

Field name	Data type	Description
translation	00001	set if translation vector non-zero
rotation	00010	set if rotation matrix is not the identity
scaling	00100	set if scaling component is not 1.0
reflection	01000	set if determinant of rotation matrix is negative
general affine	10000	set if the rotation_matrix is not a rigid rotation

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

```
struct TRANSFORM_s               //Transformation
    {
    int                           node_id;                 // $d
    union TRANSFORM_OWNER_u       owner;                   // $p
    struct TRANSFORM_s           *next;                    // $p
    struct TRANSFORM_s           *previous;                // $p
    double                        rotation_matrix[3][3];   // $f[9]
    vector                        translation_vector;      // $v
    double                        scale;                   // $f
    unsigned                      flag;                    // $d
    vector                        perspective_vector;      // $v
struct TRANSFORM_PRECISION_s     *precision;               // $p
    };
typedef struct TRANSFORM_s *TRANSFORM;
```

The TRANSFORM_PRECISION node is an optional structure for storing additional precision information of the transform and is defined as follows:

```
struct TRANSFORM_PRECISION_s     //Transformation precision
{
double  rotation_matrix[3][3];    //$f[3*3]
vector  translation_vector;       //$v
};
typedef struct TRANSFORM_PRECISION_s *TRANSFORM_PRECISION;
```

Note: When consuming Parasolid-native XT, the values in the TRANSFORM_PRECISION structure (when not NULL) should be added to those in the `rotation_matrix` and `translation_vector` values in the parent TRANSFORM structure.

When using the format reference to write XT data, the TRANSFORM_PRECISION structure should not typically be used.

5.2.7 Comb nodes

5.2.7.1 VECTOR_COMB nodes

The VECTOR_COMB node stores an array of double precision vectors.

The vector array is stored using “comb” nodes, each of which consists of a “spine” array containing pointers to “tooth” arrays. Each tooth array contains the vector information. All the teeth have the same length, which is a power of 2.

Field Name	Data Type	Description
encoding	byte	Form of the vector coordinates
n_vectors	int	Number of vectors stored in the comb
n_max_vectors	int	Maximum number of vectors for which space has been allocated in the comb
shift	int	Capacity of the tooth, defined as 2^{shift}
teeth	pointer array	Array of pointers to vector teeth

```
struct VECTOR_TOOTH_s                          //Vector Tooth
{
double                       values[1];        //$f[]
};

typedef enum
{
SCH_vector_simple    = 1,
SCH_vector_spherical = 2
}
SCH_vector_encoding_t;

struct VECTOR_COMB_s  //Vector Comb
{
SCH_vector_encoding_t       encoding;        //$u
int                         n_vectors;       //$d
int                         n_max_vectors;   //$d
int                         shift;           //$d
struct VECTOR_TOOTH_s      *teeth[1]     ;  //$p[]
};
```

The encoding enum determines the number of values used to store the vectors:

- simple vectors are stored as triplets of doubles corresponding to Cartesian coordinates;
- spherical vectors are stored as pairs of doubles corresponding to (theta, phi) spherical coordinates. This encoding is only used for unit vectors.

5.2.7.2 INTEGER_COMB nodes

Similar to vector comb nodes and is defined as follows:

```
struct INTEGER_TOOTH_s                          //Integer Tooth
{
int                         values[1];          //$d[]
};

typedef enum
{
SCH_no_encoding       = 1
}
SCH_comb_encoding_t;

struct INTEGER_COMB_s                           //Integer Comb
{
SCH_comb_encoding_t         encoding;           //$u
int                         n_integers;         //$d
int                         n_max_integers;     //$d
int                         n_bits_per_integer; //$d
int                         shift;              //$d
struct INTEGER_TOOTH_s     *teeth[1];           //$p[]
};
```

The `encoding` enum is reserved for future use. The `n_bits_per_integer` integer is not used and must be set to 32.

5.2.7.3 REAL_COMB nodes

These are similar to vector nodes and are described as follows:

Field Name	Data Type	Description
`encoding`	byte	Reserved for future use
`length`	int	Number of reals stored in the comb
`max_length`	int	Maximum length
`shift`	int	Capacity of the tooth, defined as 2^{shift}
`teeth`	pointer array	Array of pointers to real teeth

```
struct REAL_TOOTH_s                              //Real Tooth
{
int                      values[1];              //$d[]
};

typedef enum
{
SCH_no_encoding       = 1
}
SCH_comb_encoding_t;

struct REAL_COMB_s                                //Real Comb
    {
    SCH_comb_encoding_t         encoding;         //$u
    int                         length;           //$d
    int                         max_length;       //$d
    int                         shift;            //$d
    struct REAL_TOOTH_s        *teeth[1];         //$p[]
    };
typedef struct REAL_COMB_s *REAL_COMB;
```

5.2.8 Curve, surface and lattice senses

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

At the XT 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.

In the XT Format, the orientation information of lattices is stored in the `sense` field. If the sense is positive, the interior of the lattice is considered solid, and void if the sense is negative.

5.2.9 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: The 2D B-curve referenced by an SP-curve is not a dependent in this sense, and does not need a geometric owner node.

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

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

5.3 Topology

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

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

5.3.1 World

Field name	Data type	Description
`assembly`	pointer0	Head of chain of assemblies
`attribute`	pointer0	Ignore

Field name	Data type	Description
body	pointer0	Head of chain of bodies. This chain contains standard and compound bodies but does not contain any child bodies.
transform	pointer0	Head of chain of transforms
lattice	pointer0	Head of chain of lattices
surface	pointer0	Head of chain of surfaces
curve	pointer0	Head of chain of curves
point	pointer0	Head of chain of points
mesh	pointer0	Head of chain of meshes
polyline	pointer0	Head of chain of polylines
alive	logical	True unless partition is at initial pmark
attrib_def	pointer0	Head of chain of attribute definitions
attdef_list	pointer0	Shallbe set to null
highest_id	int	Highest pmark id in partition
current_id	int	Id of current pmark
index_map_offset	int	Shallbe set to 0
index_map	pointer0	Shall be set to null
schema_embedding_map	pointer0	Shallbe set to null
mesh_offset_data	pointer0	Data for embedded meshes. If the XT file does not contain embedded meshes, this field must be set to null.

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

The `attdef_list` field is not used and shall be set to null pointer.

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

The `mesh_offset_data` field is used for embedded mesh data.

```
struct WORLD_s                      //World
    {
    struct ASSEMBLY_s             *assembly;              // $p
    struct ATTRIBUTE_s            *attribute;             // $p
    struct BODY_s                 *body;                  // $p
    struct TRANSFORM_s            *transform;             // $p
    struct LATTICE_s              *lattice;               // $p
    union  SURFACE_u               surface;               // $p
    union  CURVE_u                 curve;                 // $p
    struct POINT_s                *point;                 // $p
    union SURFACE_u                mesh;                  // $p
    union CURVE_u                  polyline;              // $p
    logical                        alive;                 // $l
    struct  ATTRIB_DEF_s          *attrib_def;            // $p
    struct  POINTER_LIS_BLOCK_s   *attdef_list;           // $p
    int                            highest_id;            // $d
    int                            current_id;            // $d
    int                            index_map_offset:      // $d
    struct INT_VALUES_s            *index_map;            // $p
    struct INT_VALUES_s            *schema_embedding_map; // $p
    struct MESH_OFFSET_DATA_s      *mesh_offset_data;     // $p
    };
typedef struct WORLD_s *WORLD;
```

5.3.2 Assembly

Field name	Data type	Description
highest_node_id	int	Highest identifier in assembly
attributes_features	pointer0	Head of chain of attributes of/and features in assembly
attribute_chains	pointer0	List of attributes, one for each attribute definition used in the assembly
list	pointer0	Null
lattice	pointer0	Head of construction lattice chain
surface	pointer0	Head of construction surface chain
curve	pointer0	Head of construction curve chain
point	pointer0	Head of construction point chain
mesh	pointer0	Head of construction mesh chain
polyline	pointer0	Head of construction polyline chain
key	pointer0	Ignore
res_size	double	Value of ‘size box’ when transmitted (normally 1000)
res_linear	double	Value of linear resolution when transmitted (normally 1.0e-8). For more information on linear resolution, see Section 4.2, “General points”.
ref_instance	pointer0	Head of chain of instances referencing this assembly
next	pointer0	Ignore

Field name	Data type	Description
previous	pointer0	Ignore
state	byte	Set to 1
owner	pointer0	Ignore
type	byte	Always 1
sub_instance	pointer0	Head of chain of instances in assembly
mesh_offset_data	pointer0	Data for embedded meshes. If the XT file does not contain embedded meshes, this field must be set to null.

The value of the ‘state’ field should be ignored, as should any nodes of type ‘KEY’ referenced by the assembly. If XT data is constructed without the use of the Parasolid Kernel, the state field should be set to 1, and the key to null.

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

The mesh_offset_data field is used for embedded mesh data.

```
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_FEAT_u             attributes_features;  // $p
    struct LIST_s                  *attribute_chains;     // $p
    struct LIST_s                  *list;                 // $p
    union LATTICE_u                 lattice;              // $p
    union SURFACE_u                 surface;              // $p
    union CURVE_u                   curve;                // $p
    struct POINT_s                 *point;                // $p
    union SURFACE_u                 mesh;                 // $p
    union CURVE_u                   polyline;             // $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;         // $p
struct MESH_OFFSET_DATA_s  *mesh_offset_data;  //$p
    };
typedef struct ASSEMBLY_s *ASSEMBLY;
```

```
struct KEY_s                       //Key
    {
    string[1];                      char                  // $c[]
    };
typedef struct KEY_s  *KEY;
```

5.3.3 Instance

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

Note: Only standard bodies can be instanced. Compound or child bodies cannot be instanced.

```
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_FEAT_u           attributes_features;   // $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;
```

5.3.4 Body

Field name	Data type	Description
highest_node_id	int	Highest identifier in a standard body. For compound bodies, this is the highest identifier, including entities in child bodies. For child bodies, this is the identifier of the child body itself. It is also the highest unique identifier of the entities which were in the child body when it was added to its parent compound body.
attributes_features	pointer0	Head of chain of attributes of and features in body. All features in a compound body and its children are chained off this regardless of their contents.

Field name	Data type	Description
attribute_chains	pointer0	List of attributes, one for each attribute definition used in the body. For compound bodies, all attributes within the compound body and its children appear in the attribute_chains list for the compound body, except those attributes that are directly attached to the child bodies. These attributes appear in the attribute_chains list of their relevant child body.
lattice	pointer0	Head of construction lattice chain.
surface	pointer0	Head of construction surface chain. For a child body, these fields are always null.
curve	pointer0	Head of construction curve chain. For a child body, these fields are always null.
point	pointer0	Head of construction point chain. For a child body, these fields are always null.
mesh	pointer0	Head of construction mesh chain. For a child body, these fields are always null
polyline	pointer0	Head of construction polyline chain. For a child body, these fields are always null
key	pointer0	Ignore
res_size	double	Value of ‘size box’ when transmitted (normally 1000)
res_linear	double	Value of linear resolution when transmitted (normally 1.0e-8). For more information on linear resolution, see Section 4.2, “General points”.
ref_instance	pointer0	Head of chain of instances referencing this assembly
next	pointer0	For a child body, this field chains children of the same parent together.
previous	pointer0	For a child body, this field chains children of the same parent together.
state	byte	Set to 1 (see below)
owner	pointer0	For a child, this field references its parent compound body.
body_type	byte	Body type. If children of a compound body have the same body_type, then this will also be the body_type of the compound body. Otherwise, the body_type of the compound will be general.
nom_geom_state	byte	Set to 1 (for future use)

Field name	Data type	Description
shell	pointer0	For general bodies: null. For solid bodies: the first shell in one of the solid regions. For other bodies: the first shell in one of the regions.This field is **obsolete**, and should be ignored by applications reading XT data. When writing XT data, it shall be set as above.
boundary_surface	pointer0	Head of chain of surfaces attached directly or indirectly to faces or edges or fins. For a child body, these fields are always null
boundary_curve	pointer0	Head of chain of curves attached directly or indirectly to edges or faces or fins. For a child body, these fields are always null
boundary_point	pointer0	Head of chain of points attached to vertices. For a child body, these fields are always null
boundary_mesh	pointer0	Head of chain of meshes attached directly or indirectly to faces or edges or fins. For a child body, these fields are always null
boundary_polyline	pointer0	Head of chain of polylines attached directly or indirectly to edges or faces or fins. For a child body, these fields are always null
region	pointer0	Head of chain of regions in body; this is the infinite region. For a compound body, this is the head of chain of regions comprising of all the regions of all the child bodies. The regions of each child body are contiguous in this chain and the order of the regions corresponds to the order of the children. For a child body, this field points to the first region of the child. For an empty compound body (i.e. one without any children), this field is null.
edge	pointer0	Head of chain of all non-wireframe edges in body. For a compound body, this is the head of a chain of all non-wireframe edges in all the children.For a child body, this field is null.
vertex	pointer0	Head of chain of all vertices in body. For a compound body, this is the head of a chain of all vertices in all the children.For a child body, this field is null.
index_map_offset	int	Shallbe set to 0
index_map	pointer0	Shall be set to null
node_id_index_map	pointer0	Shallbe set to null
schema_embedding_map	pointer0	Shallbe set to null

Field name	Data type	Description
child	pointer0	For a compound body, this is the first child body. the children are chained using the next and previous pointers. For a standard or child body, this is null.
lowest_node_id	int	For a standard or compound body, this field is zero. For a child body, this is the lowest node id of the entities in the child body.
mesh_offset_data	pointer0	Data for embedded meshes. If the XT file does not contain embedded meshes, this field must be set to null.

The value of the ‘state’ field should be ignored, as should any nodes of type ‘KEY’ referenced by the body. If XT data is constructed without using the Parasolid Kernel, the state field should be set to 1, and the key to null.

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

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

The mesh_offset_data field is used for embedded mesh data.

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

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

```
struct BODY_s                       //Body
    {
    int                              highest_node_id;       // $d
    union  ATTRIB_FEAT_u             attributes_features;   // $p
    struct LIST_s                   *attribute_chains;      // $p
    union  LATTICE_u                *lattice;               // $p
    union  SURFACE_u                 surface;               // $p
    union  CURVE_u                   curve;                 // $p
    struct POINT_s                  *point;                 // $p
    union SURFACE_u                  mesh;                  // $p
    union CURVE_u                    polyline;              // $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
    union BODY_OWNER_u               owner;                 // $p
    SCH_body_type                    body_type;             // $u
    SCH_nom_geom_state_t             nom_geom_state;        // $u
    struct SHELL_s                  *shell;                 // $p
    union  SURFACE_u                 boundary_surface;      // $p
    union  CURVE_u                   boundary_curve;        // $p
    struct POINT_s                  *boundary_point;        // $p
    union SURFACE_u                  boundary_mesh;         // $p
    union CURVE_u                    boundary_polyline;     // $p
    struct REGION_s                 *region;                // $p
    struct EDGE_s                   *edge;                  // $p
    struct VERTEX_s                 *vertex;                // $p
    int                              index_map_offset;      // $d
    struct INT_VALUES_s             *index_map;             // $p
    struct INT_VALUES_s             *node_id_index_map;     // $p
    struct INT_VALUES_s             *schema_embedding_map;  // $p
struct BODY_s   *child;  //$p
int   lowest_node_id;  //$d
struct MESH_OFFSET_data_s  *mesh_offset_data;  //$p
    };
typedef struct BODY_s *BODY;
```

```
union BODY_OWNER_u
{
struct WORLD_s  *world;
struct BODY_s  *body;
};
```

5.3.4.1 Attaching geometry to topology

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

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

Geometry	Owner	Whether chained
Attached to face	face	In boundary_surface chain
Attached to edge or fin	pointer0	In boundary_curve chain
Attached to vertex	pointer	In boundary_point chain
Indirectly attached to face or edge or fin	pointer0	In boundary_surface chain or boundary_curve chain
Construction geometry	pointer0	In lattice, surface, curve or point chain
2D B-curve in SP-curve	pointer0	Not chained
type	char	Region type – solid (‘S’) or void (‘V’)

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.

5.3.5 Region

Field name	Type	Description
node_id	int	Identifier
attributes_features	pointer0	Head of chain of attributes of region and member of features of region
body	pointer	Body of region. For a region in a child body, this field references the parent compound body.
next	pointer0	Next region in body
prev	pointer0	Previous region in body
shell	pointer0	Head of singly-linked chain of shells in region
type	char	Region type – solid (‘S’) or void (‘V’)
owner	pointer0	For a region in a child body, this field references the child body. Otherwise, the field is null.

```
struct REGION_s                    //Region
    {
    int                             node_id;               // $d
    union  ATTRIB_FEAT_u            attributes_features;   // $p
    struct BODY_s                  *body;                  // $p
    struct REGION_s                *next;                  // $p
    struct REGION_s                *previous;              // $p
    struct SHELL_s                 *shell;                 // $p
    char                            type;                  // $c
struct BODY_s  *owner:  //$p
    };
typedef struct REGION_s *REGION;
```

5.3.6 Shell

Field name	Type	Description
node_id	int	Identifier
attributes_features	pointer0	Head of chain of attributes of shell
body	pointer0	For standard bodies: 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. For shells in child bodies, this field is set to the parent compound body if the child has type wire or sheet, or if it has type solid and the shell belongs to a solid region. Otherwise, this field is null. This field is **obsolete**, and should be ignored by applications reading XT data. When writing XT data, it shall be set as above.
next	pointer0	Next shell in region
face	pointer0	Head of chain of back-faces of shell (i.e. faces with face normal pointing out of region of shell).
edge	pointer0	Head of chain of wire-frame edges of shell.
vertex	pointer0	If shell consists of a single vertex, this is it; else null
region	pointer	Region of shell
front_face	pointer0	Head of chain of front-faces of shell (i.e. faces with face normal pointing into region of shell).

```
struct SHELL_s                      //Shell
    {
    int                             node_id;                // $d
    union  ATTRIB_FEAT_u            attributes_features;    // $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;
```

5.3.7 Face

Field name	Type	Description
node_id	int	Identifier
attributes_features	pointer0	Head of chain of attributes of face and member_of_feature of face.

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

```
struct FACE_s                       //Face
    {
    int                             node_id;                // $d
    union ATTRIB_FEAT_u            attributes_features;    // $
    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;
```

5.3.8 Loop

Field name	Type	Description
node_id	int	Identifier
attributes_features	pointer0	Head of chain of attributes of loop
halfedge	pointer	One of ring of fins of loop
face	pointer	Face of loop
next	pointer0	Next loop in face

5.3.8.1 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 `halfedge->forward = halfedge->backward = halfedge`, and `halfedge->other = halfedge->curve = halfedge->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_FEAT_u            attributes_features;  // $p
    struct HALFEDGE_s              *halfedge;             // $p
    struct FACE_s                  *face;                 // $p
    struct LOOP_s                  *next;                 // $p
    };
typedef struct LOOP_s *LOOP;
```

5.3.9 Fin

Note: Fins are referred to as halfedges in the PK Interface.

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

5.3.9.1 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->halfedge->vertex` and `edge->halfedge->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 `halfedge->loop = halfedge->forward = halfedge->backward = halfedge->curve = halfedge->next_at_vx = null`. For example the boundaries of a

sheet always have one dummy fin. See Section 5.3.9.2, “Fin chain at a vertex” for more information on fin chains at a vertex.

```
struct HALFEDGE_s  //Halfedge
    {
    union  ATTRIB_FEAT_u         attributes_features;   // $p
    struct HALFEDGE_s           *forward;               // $p
    struct HALFEDGE_s           *backward;              // $p
    struct VERTEX_s             *vertex;                // $p
    struct HALFEDGE_s           *other;                 // $p
    struct EDGE_s               *edge;                  // $p
    union  CURVE_u               curve;                 // $p
    struct HALFEDGE_s           *next_at_vx;            // $p
    char                         sense;                 // $c
    };
typedef struct HALFEDGE_s *HALFEDGE;
```

5.3.9.2 Fin chain at a vertex

The fin chain at a vertex only includes fins which are primary or secondary fins i.e `edge->halfedge` or `edge->halfedge->other` for some edge. All fins going into a vertex have that vertex as their `halfedge->vertex` but only those fins which are primary or secondary fins appear in the fin chains at the vertex. For any edge with a vertex or vertices, regardless of how many fins it has, the primary fin will be in the fin chain of the vertex towards which the edge points. The secondary fin will be in the fin chain of the vertex away from which the edge points. Any other fin will not appear in either fin chain at a vertex.

5.3.10 Vertex

Field name	Type	Description
`node_id`	int	Identifier
`attributes_features`	pointer0	Head of chain of attributes of vertex and `member_of_feature` of vertex
`halfedge`	pointer0	Head of singly-linked chain of fins referencing this vertex
`previous`	pointer0	Previous vertex in body
`next`	pointer0	Next vertex in body
`point`	pointer	Point of vertex
`tolerance`	double	Tolerance of vertex (null-double for accurate vertex)
`owner`	pointer	Owning body (for non-acorn vertices) or shell (for acorn vertices). If the vertex is in a child body and it’s not an acorn vertex then the field references the parent compound body.

```
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_FEAT_u            attributes_features;   // $p
    struct HALFEDGE_s              *halfedge;              // $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;
```

5.3.11 Edge

Field name	Type	Description
node_id	int	Identifier
attributes_features	pointer0	Head of chain of attributes of edge and member_of_feature of edge
tolerance	double	Tolerance of edge (null-double for accurate edges)
halfedge	pointer	Head of singly-linked chain of fins around edge
previous	pointer0	Previous edge in body or shell
next	pointer0	Next edge in body or shell
curve	pointer0	Curve of edge, zero for tolerant edge. If edge is accurate, but any of its vertices are tolerant, this will be a trimmed curve.
next_on_curve	pointer0	Next in chain of edges sharing the curve of this edge
previous_on_curve	pointer0	Previous in chain of edges sharing the curve of this edge
owner	pointer	Owning body (for non-wireframe edges) or shell (for wireframe edges). If the edge is in a child body and it is not a wireframe edge, then the field references the parent compound body.

```
struct EDGE_s                       //Edge
    {
    int                              node_id;               // $d
    union  ATTRIB_FEAT_u             attributes_features;   // $p
    double                           tolerance;             // $f
    struct HALFEDGE_s               *halfedge;              // $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;
```

5.4 Associated Data

5.4.1 List

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

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

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

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

```
union LIS_OWNER_u
    {
    struct BODY_s                 *body;
    struct ASSEMBLY_s             *assembly;
    struct WORLD_s                *world;
    };
typedef union LIST_OWNER_u    LIST_OWNER;
```

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

5.4.2 Pointer_lis_block

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

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

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

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

5.4.3 Att_def_id

Field name	Type	Description
string[]	char	String name e.g. "SDL/TYSA_COLOUR"

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

5.4.4 Field_names

Field name	Type	Description
names[]	pointer	Array of field names – unicode or char

```
typedef union FIELD_NAME_u
    {
    struct CHAR_VALUES_s         *name
    struct UNICODE_VALUES_s      *uname
    };
    FIELD_NAME_t;
```

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

5.4.5 Attrib_def

The XT Format provides a set of pre-defined attribute definitions known as system attribute definitions. See Appendix A, “System Attribute Definitions”, for more information on these.

Attribute definitions have the following form in the XT format:

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

The legal_owners array is an array of logicals determining which node types may own this type of attribute.

e.g. if faces are allowed attrib_def -> legal_owners [SCH_fa_owner] = true.

Note that if the XT data contains user fields, the 'fields' field of an attribute definition may contain extra values, set to zero. These are to be ignored.

The 'actions' field in an attribute definition defines the behaviour of the attribute when an event (rotate, scale, translate, reflect, split, merge, transfer, change) occurs. The actions are in table F.49:

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

The XT attribute classes 1-7 correspond as follows:

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

```
typedef enum
    {
    SCH_rotate      = 0,
    SCH_scale       = 1,
    SCH_translate   = 2,
    SCH_reflect     = 3,
    SCH_split       = 4,
    SCH_merge       = 5,
    SCH_transfer    = 6,
    SCH_change      = 7,
    SCH_max_logged_event   //last entry; value in $d[] code for
actions
    }
    SCH_logged_event_t;
```

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

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

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

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

5.4.6 Attribute

Field name	Type	Description
node_id	int	Identifier
definition	pointer	Attribute definition
owner	pointer	Attribute ~~definition~~owner
next	pointer0	Next attribute, feature, or member_of_feature
previous	pointer0	Previous ditto
next_of_type	pointer0	Next attribute of this type in this part
previous_of_type	pointer0	Previous attribute of this type in this part
fields[]	pointer	Fields, of type int_values etc. The number of fields is given by the length of the variable part of the node. There may be no fields.

The attributes of a node are chained using the next and previous pointers in the attribute. The attributes_features pointer in the node points to the head of this chain. This chain also contains the member_of_feature 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_features chains in parts, features and nodes contain the following types of node:

- Part: attributes and features.
- Feature: attributes.
- Node: attributes and member_of_feature.

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 HALFEDGE_s            *halfedge;
    struct VERTEX_s              *vertex;
    union  SURFACE_u              Surface;
    union  CURVE_u                Curve;
    struct POINT_s               *point;
    struct FEATURE_s             *feature;
    };
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_FEAT_u          next;                 // $p
    union  ATTRIB_FEAT_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;
```

5.4.7 Int_values

Field name	Type	Description
values[]	int	Integer values

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

5.4.8 Real_values

Field name	Type	Description
values[]	double	Real values

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

5.4.9 Char_values

Field name	Type	Description
values[]	char	Character values

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

5.4.10 Unicode_values

Field name	Type	Description
values[]	short	Unicode character values

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

5.4.11 Point_values

Field name	Type	Description
values[]	vector	Point values

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

5.4.12 Vector_values

Field name	Type	Description
values[]	vector	Vector values

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

5.4.13 Direction_values

Field name	Type	Description
values[]	vector	Direction values

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

5.4.14 Axis_values

Field name	Type	Description
values[]	vector	Axis values

```
struct AXIS_VALUES_s              //Axis values
    {
    vector                         values[1];          // $v[]
    };
typedef struct AXIS_VALUES_s *AXIS_VALUES;
```

5.4.15 Tag_values

Field name	Type	Description
values[]	int	Integer tag values

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

5.4.16 Feature

Note: Features are referred to as ‘groups’ in the PK Interface.

Field name	Type	Description
node_id	int	Identifier
attributes_features	pointer	Head of chain of attributes of this feature
owner	pointer	Owning part
next	pointer0	Next feature or attribute
previous	pointer0	Previous feature or attribute
type	byte	Type of node allowed in feature
first_member	pointer0	Head of chain of member_of_feature nodes in feature

The groups in a part are chained by the next and previous pointers in a group. The attributes_features 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_feature. These are chained together using the next_member and previous_member pointers. The first_member pointer in the feature points to the head of the chain. Each member_of_feature has an owning_feature pointer which points back to the feature.

Each member_of_feature has an owner pointer which points to a node. Thus the feature references its member nodes via the member_of_feature.

The member_of_feature which refer to a particular node are chained using the next and previous pointers in the member_of_feature. The attributes_features 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_pf_pline_fe   = 11,
    SCH_feature_fe    = 12
    } SCH_feature_type_t;
```

```
struct FEATURE_s  //Feature
    {
    int                          node_id;              // $d
    union  ATTRIB_FEAT_u         attributes_features;  // $p
    union  PART_u                owner;                // $p
    union  ATTRIB_FEAT_u         next;                 // $p
    union  ATTRIB_FEAT_u         previous;             // $p
    SCH_feature_type_t           type;                 // $u
    struct MEMBER_OF_FEATURE_s  *first_member;         // $p
    };
typedef struct FEATURE_s *FEATURE;
```

5.4.17 Member_of_feature

Field name	Type	Description
dummy_node_id	int	Entity label
owning_feature	pointer	Owning feature
owner	pointer	Referenced member of feature
next	pointer0	Next attribute, feature or member_of_feature
previous	pointer0	Previous ditto
next_member	pointer0	Next member_of_feature in this feature
previous_member	pointer0	Previous ditto

```
union FEATURE_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;
struct FEATURE_s  *feature;
    };
typedef union FEATURE_MEMBER_u FEATURE_MEMBER;
```

```
struct MEMBER_OF_FEATURE_s //Member of feature
   {
   int                            dummy_node_id;       // $d
   struct FEATURE_s              *owning_feature;      // $
   union  FEATURE_MEMBER_u        owner;               // $p
   union  ATTRIB_FEAT_u           next;                // $p
   union  ATTRIB_FEAT_u           previous;            // $p
   struct MEMBER_OF_FEATURE_s    *next_member;         // $p
   struct MEMBER_OF_FEATURE_s    *previous_member;     // $p
   };
typedef struct MEMBER_OF_FEATURE_s *MEMBER_OF_FEATURE;
```

5.4.18 Part_XMT_block

Field name	Type	Description
n_entries	int	Number of entries in this block. (n_entries >1)
index_map_offset	int	Must be set to 0
index_map	pointer0	Must be set to null
schema_embedding_map	pointer0	Must be set to null
mesh_offset_data	pointer0	Data for embedded meshes. If the XT file does not contain embedded meshes, this field must be set to null.
entries[]	pointer	Pointers in block

The PART_XMT_BLOCK must be the root node in XT data containing more than one part,

The fields index_map_offset, index_map, and schema_embedding_map are used for Indexed Transmit.

The mesh_offset_data field is used for embedded mesh data.

```
struct PART_XMT_BLOCK_s //Part list
{
int                              n_entries;              //$d
unsigned                         index_map_offset;       //$d
struct INT_VALUES_s             *index_map;              //$p
struct INT_VALUES_s             *schema_embedding_map;   //$p
struct MESH_OFFSET_DATA_s       *mesh_offset_data;       //$p
union PART_u                     entries[1];             //$p[]
};

typedef struct PART_XMT_BLOCK_s *PART_XMT_BLOCK;
```

5.4.19 Mesh_offset_data

The MESH_OFFSET_DATA node is used for embedded mesh data. It is the second node in an XT file with embedded meshes. At most, there can only be one MESH_OFFSET_DATA node.

Field Name	Data Type	Description
mesh_index_map	pointer0	Must be set to an offset_values node
schema_data	pointer0	See Section 5.4.19.2, “Schema_data” for information
schema_data_offset_high	int	Must be present
schema_data_offset_low	int	Must be present

The schema_data field must be present if the XT data contains embedded schema information, otherwise it must be set to null.

The schema_data_offset_high and schema_data_offset_low fields must be present and indicate either the offset of the schema_data node (if present), or the offset of the terminator.

```
struct MESH_OFFSET_DATA_s //Mesh offset data
{
struct OFFSET_VALUE_s    *mesh_index_map;           //$p
struct SCHEMA_DATA_s     *schema_data;              //$p
unsigned                 *schema_data_offset_high;  //$d
unsigned                 *schema_data_offset_low ;  //$d
};
typedef struct MESH_OFFSET_DATA_s *MESH_OFFSET_DATA;
```

5.4.19.1 Offset_values

The OFFSET_VALUES node contains the offset values of each PSM_MESH node. These offset values must be in the same order as the PSM_MESH nodes in the XT data. If this node is present, it must be the third node in the XT data.

Field Name	Data Type	Description
values[]	int	Offset values in high and low pairs

```
struct OFFSET_VALUES_s     //Offset values
{
unsigned    values[1];     //$d[]
};
typedef struct OFFSET_VALUES_s *OFFSET_VALUES;
```

5.4.19.2 Schema_data

The SCHEMA_DATA node must be present if the XT data contains embedded schema information. The SCHEMA_DATA node chains one NODE_MAP node per node type used in the mesh data section of the XT data (i.e between the first PSM_MESH node and the last node before the SCHEMA_DATA node).

There is a NEW_NODE_MAP node defining the following node types:

- PSM_MESH
- INTEGER_TOOTH
- INTEGER_COMB
- VECTOR_TOOTH
- VECTOR_COMB

These node types are relative to the defined base schema, currently SCH_13006. See Section 2.1.2, “Embedded schemas” for more information.

```
struct SCHEMA_DATA_s //Schema data
{
union NODE_MAP_u   node_map;   //$p
};
typedef struct SCHEMA_DATA_s *SCHEMA_DATA;
```

The SCHEMA_DATA node is followed by the NODE_MAP, FIELD_MAP, and SCHEMA_CHAR_VALUES nodes. These nodes will always be at the end of the embedded mesh file and indicate whether an embedded mesh file was saved using an embedded schema.

5.4.19.3 Node_map union

```
union NODE_MAP_u
{
struct ANY_NODE_MAP_s *any;
struct NEW_NODE_MAP_s *new;
struct MOD_NODE_MAP_s *mod;
struct OLD_NODE_MAP_s *old;
};
typedef union NODE_MAP_u NODE_MAP;
```

5.4.19.4 Node map nodes

All node map nodes share the following common fields:

Field Name	Data Type	Description
next	pointer0	Next node map in chain
exemplar_offset_high	int	Offset values of the first node of node_type in the XT data
exemplar_offset_low	int	Offset values of the first node of node_type in the XT data
node_type	short	Node type

5.4.19.5 Any_node_map

```
struct ANY_NODE_MAP_s //Any node map
{
union NODE_MAP_u             next;                  //$p
unsigned                     exemplar_offset_high;  //$d
unsigned                     exemplar_offset_low;   //$d
short                        node_type;             //$n
};
typedef struct ANY_NODE_MAP_s  *ANY_NODE_MAP;
```

5.4.19.6 Old_node_map

```
struct OLD_NODE_MAP_s == ANY_NODE_MAP_s //Unchanged node map
{
union NODE_MAP_u             next;                  //$p
unsigned                     exemplar_offset_high;  //$d
unsigned                     exemplar_offset_low;   //$d
short                        node_type;             //$n
};
typedef struct OLD_NODE_MAP_s *OLD_NODE_MAP;
```

5.4.19.7 New_node_map

Field Name	Data Type	Description
vla_field_xmt_code	logical	Whether the variable length array field (if present) should be saved to XT. If this field is not present, the value is set to false.
name	pointer	Points to the schema character values node that contain the node name.
description	pointer	Points to the schema character values node that contain the description.
field_maps[1]	pointer	Array of new field maps

```
struct NEW_NODE_MAP_s == ANY_NODE_MAP_s //New node map
{
union NODE_MAP_u              next; $p
unsigned                      exemplar_offset_high;  //$d
unsigned                      exemplar_offset_low;   //$d
short                         node_type;             //$n
logical                       vla_field_xmt_code;    //$l
struct SCHEMA_CHAR_VALUES_s *name;                   //$p
struct SCHEMA_CHAR_VALUES_s *description;            //$p
struct NEW_FIELD_MAP_s      *field_maps[1];          //$p[]
};
typedef struct NEW_NODE_MAP_s *NEW_NODE_MAP;
```

5.4.19.8 Modified_node_map

Field Name	Data Type	Description
vla_field_xmt_code	logical	Whether the variable length array field (if present) is transmitted. If this field is not present, the value is set to false.
field_maps[1]	pointer	Array of field maps

```
struct MOD_NODE_MAP_s == ANY_NODE_MAP_s          //Modified node map
{
union NODE_MAP_u             next;                  //$p
unsigned                     exemplar_offset_high;  //$d
unsigned                     exemplar_offset_low;   //$d
short                        node_type;             //$n
logical                      vla_field_xmt_code;    //$l
union FIELD_MAP_u            field_maps[1];         //$p[]
};
typedef struct MOD_NODE_MAP_s *MODE_NODE_MAP;
```

5.4.19.9 Field_map union

```
union FIELD_MAP_u
{
struct NEW_FIELD_MAP_s *new;
struct OLD_FIELD_MAP_s *old;
};
typedef union FIELD_MAP_u FIELD_MAP;
```

5.4.19.10 Old_field_map

Field Name	Data Type	Description
base_index	byte	The field number this field used to be in the baseline schema

```
struct OLD_FIELD_MAP_s // Old field map
{
SCH_byte_t               base_index;    //$u
};
typedef struct OLD_FIELD_MAP_s *OLD_FIELD_MAP;
```

5.4.19.11 New_field_map

Field Name	Data Type	Description
name	pointer	Points to the schema character values node that contains the field name.
ptr_class	short	Node type or class of the field if it is a pointer field
n_elts	int	Number of elements for that field
type	pointer	Points to a schema character values node that contains the field type.

```
struct NEW_FIELD_MAP_s // New field map
{
struct SCHEMA_CHAR_VALUES_s *name;      //$p
short                        ptr_class; //$n
int                          n_elts;    //$d
struct SCHEMA_CHAR_VALUES_s *type;      //$p
};
typedef struct NEW_FIELD_MAP_s *NEW_FIELD_MAP;
```

5.4.19.12 Schema_char_values

Field Name	Data Type	Description
values[]	char	Character values

```
struct SCHEMA_CHAR_VALUES_s     //Schema character values
{
char values[1];     //$c[]
};
typedef struct SCHEMA_CHAR_VALUES_s *SCHEMA_CHAR_VALUES;
```

Nodes and Classes 6

6.1 Node types

The following table details the node types:

Node Name	Node Type	Visible at PK	Has Node-ID
ASSEMBLY	10	Yes	No
INSTANCE	11	Yes	Yes
BODY	12	Yes	No
SHELL	13	Yes	Yes
FACE	14	Yes	Yes
LOOP	15	Yes	Yes
EDGE	16	Yes	Yes
HALFEDGE	17	Yes	No
VERTEX	18	Yes	Yes
REGION	19	Yes	Yes
POINT	29	Yes	Yes
LINE	30	Yes	Yes
CIRCLE	31	Yes	Yes
ELLIPSE	32	Yes	Yes
INTERSECTION	38	Yes	Yes
CHART	40	No	
LIMIT	41	No	
BSPLINE_VERTICES	45	No	
PLANE	50	Yes	Yes
CYLINDER	51	Yes	Yes
CONE	52	Yes	Yes
SPHERE	53	Yes	Yes
TORUS	54	Yes	Yes
BLENDED_EDGE	56	Yes	Yes
BLEND_BOUND	59	No	
OFFSET_SURF	60	Yes	Yes
SWEPT_SURF	67	Yes	Yes
SPUN_SURF	68	Yes	Yes
LIST	70	Yes	
POINTER_LIS_BLOCK	74	No	
ATT_DEF_ID	79	No	

Node Name	Node Type	Visible at PK	Has Node-ID
ATTRIB_DEF	80	Yes	No
ATTRIBUTE	81	Yes	Yes
INT_VALUES	82	No	
REAL_VALUES	83	No	
CHAR_VALUES	84	No	
POINT_VALUES	85	No	
VECTOR_VALUES	86	No	
AXIS_VALUES	87	No	
TAG_VALUES	88	No	
DIRECTION_VALUES	89	No	
FEATURE	90	Yes	Yes
MEMBER_OF_FEATURE	91	No	
UNICODE_VALUES	98	No	
FIELD_NAMES	99	No	
TRANSFORM	100	Yes	No
WORLD	101	No	
KEY	102	No	
PE_SURF	120	Yes	Yes
INT_PE_DATA	121	No	
EXT_PE_DATA	122	No	
B_SURFACE	124	Yes	Yes
SURFACE_DATA	125	No	
NURBS_SURF	126	No	
KNOT_MULT	127	No	
KNOT_SET	128	No	
PE_CURVE	130	Yes	Yes
TRIMMED_CURVE	133	Yes	Yes
B_CURVE	134	Yes	Yes
CURVE_DATA	135	No	
NURBS_CURVE	136	No	
SP_CURVE	137	Yes	Yes
GEOMETRIC_OWNER	141	No	
HELIX_SU_FORM	163	No	
PART_XMT_BLOCK	176	No	
HELIX_CU_FORM	184	No	
POLYLINE_DATA	185	No	
PSM_MESH	189	No	
INTEGER_TOOTH	190	No	
INTEGER_COMB	191	No	
VECTOR_TOOTH	192	No	

Node Name	Node Type	Visible at PK	Has Node-ID
VECTOR_COMB	193	No	
POLYLINE	200	Yes	Yes
MESH	201	Yes	Yes
INTERSECTION_DATA	204	No	
OFFSET_VALUES	205	No	
MESH_OFFSET_DATA	206	No	
SCHEMA_CHAR_VALUES	207	No	
NEW_NODE_MAP	208	No	
MOD_NODE_MAP	209	No	
NEW_FIELD_MAP	210	No	
SCHEMA_DATA	211	No	
OLD_NODE_MAP	212	No	
OLD_FIELD_MAP	213	No	
REAL_TOOTH	220	No	
REAL_COMB	221	No	
LATTICE	222	Yes	
LATTICE_DATA_IRREGULAR	223	No	
GRAPH_COMPACT	224	No	
TRANSFORM_PRECISION	229	No	

6.2 Node classes

The following table details the node classes:

Node Class Name	Node Class
GEOMETRY	1003
PART	1005
SURFACE	1006
SURFACE_OWNER	1007
CURVE	1008
CURVE_OWNER	1010
POINT_OWNER	1011
LIS_BLOCK	1012
LIST_OWNER	1013
ATTRIBUTE_OWNER	1015
FEATURE_OWNER	1016
FEATURE_MEMBER	1017
FIELD_VALUES	1018
ATTRIB_FEATURE	1019
TRANSFORM_OWNER	1023

Node Class Name	Node Class
PE_DATA	1027
PE_INT_GEOM	1028
SHELL_OR_BODY	1029
FIELD_NAME	1037
BODY_OWNER	1040
COMB	1042
NODE_MAP	1043
FIELD_MAP	1044
LATTICE_OWNER	1045
LATTICE_DATA	1046

System Attribute Definitions

A

Note: Some user-defined attribute definitions could have a similar naming convention to a system attribute definition. If not documented in this chapter, an attribute definition should be treated as user-defined.

A.1 System attribute definitions whose field values define a property

A.1.1 Colour

Identifier	SDL/TYSA_COLOUR		
Type id	8001		
Legal Owner (entity types)	face edge		
Fields	real	Red value	These three values should be in the range 0.0 to 1.0
		Green value	
		Blue value	

A.1.2 Colour 2

Identifier	SDL/TYSA_COLOUR_2		
Type id	8040		
Legal Owners	body, instance, assembly		
Fields	real	Red value	These three values should be in the range of 0.0 to 1.0
		Green value	
		Blue value	

A.1.3 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.

A.1.3.1 Density (of a body)

Identifier	SDL/TYSA_DENSITY	
Type_id	8004	
Legal Owner (entity types)	body	
Fields	real	Density
	string	Units

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

The character field “units” is optional.

A.1.3.2 Region density

Identifier	SDL/TYSA_REGION_DENSITY	
Type_id	8023	
Legal Owner (entity types)	region	
Fields	real	Density of region
	string	Units

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 optional.

A.1.3.3 Face density

Identifier	SDL/TYSA_FACE_DENSITY	
Type_id	8024	
Legal Owner (entity types)	face	
Fields	real	Density of face
	string	Units

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 optional.

A.1.3.4 Edge density

Identifier	SDL/TYSA_EDGE_DENSITY	
Type_id	8025	
Legal Owner (entity types)	edge	
Fields	real	Density of edge
	string	Units

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 optional.

A.1.3.5 Vertex density

Identifier	SDL/TYSA_VERTEX_DENSITY	
Type_id	8026	
Legal Owner (entity types)	vertex	
Fields	real	Mass of vertex
	string	Units

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 optional.

A.1.4 Group control

Identifier	SDL/TYSA_GROUP_CONTROL	
Type id	8039	
Legal Owner (entity types)	group	
Fields	integer	Two integers, containing numerical tokens from PK_GROUP_split_empty_t and PK_GROUP_merge_empty_t, in that order.

PK_GROUP_split_empty_t can take one of the following numerical tokens:

Token	Description
24780	The group remains in the original part. (Default)
24781	The group remains in the original part, and a copy is made in the split-off part.

PK_GROUP_merge_empty_t can take one of the following numerical tokens:

Token	Description
24790	The empty groups in both parts appear in the merged part. (Default)
24791	Pairs of identical empty groups, one in each part, are merged into one group in the merged part. "Identical" here means having the same attributes (including SDL/TYSA_GROUP_CONTROL attributes). Identical empty groups in the same part are not merged.

A.1.5 Hatching

Identifier	SDL/TYSA_HATCHING	
Type_id	8003	
Legal Owner (entity types)	face	
Fields	real	real 1
		real 2
		real 3
		real 4
	integer	Hatching type

For planar hatching (Hatching type is 3) - the four real values define the hatch orientation as a vector and a spacing between consecutive planes.

For radial hatching (Hatching type is 2) - the first three real values define the spacing of the hatch lines. The fourth value is not used.

For parametric hatching (Hatching type is 22) - the first two real values define the spacing in u and v respectively. The last two values are not used.

A.1.5.1 Planar hatch

Identifier	SDL/TYSA_PLANAR_HATCH		
Type_id	8021		
Legal Owner (entity types)	face		
Fields	real	x component	‘direction’ or plane normal
		y component	
		z component	
		‘pitch’ or separation	
		x component	position vector
		y component	
		z component	

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.

A.1.5.2 Radial hatch

Identifier	SDL/TYSA_RADIAL_HATCH	
Type_id	8027	
Legal Owner (entity types)	face	
Fields	real	radial around
		radial along
		radial about
		radial around start
		radial along start
		radial about start

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.

A.1.5.3 Parametric hatch

Identifier	SDL/TYSA_PARAM_HATCH	
Type_id	8028	
Legal Owner (entity types)	face	
Fields	real	*u* spacing
		v spacing
		u start
		v start

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.

A.1.5.4 Parametric hatch (numeric control)

Identifier	SDL/TYSA_PARAM_NUM_HATCH	
Type_id	8049	
Legal Owner (entity types)	face	
Fields	integer	number of u hatch lines
		number of v hatch lines

For **parametric hatching** (numeric control), an attribute with this definition takes precedence over an attribute with the SDL/TYSA_HATCHING definition or the SDL/TYSA_PARAM_HATCH definition, if a face has both types of attribute attached.

A.1.6 Layer

Identifier	SDL/TYSA_LAYER	
Type id	8042	
Legal Owner (entity types)	body, instance, assembly	
Fields	ustring	Layer number
	ustring	Layer name/label
	ustring	Extra user data
	integer	Visibility information. 0=visible, 1=invisible. Other values not allowed.

A.1.7 Name

Identifier	SDL/TYSA_NAME	
Type_id	8017	
Legal Owner (entity types)	assembly, body, instance, shell, face, loop, edge, vertex, group, surface, curve, point	
Fields	string	Name of entity

A.1.8 Reflectivity

Identifier	SDL/TYSA_REFLECTIVITY	
Type_id	8014	
Legal Owner (entity types)	face	
Fields	real	Coefficient of specular reflection
		Proportion of coloured light in highlights
		Coefficient of diffuse reflection
		Coefficient of ambient reflection
	integer	Reflection power

A.1.9 Translucency

Identifier	SDL/TYSA_TRANSLUCENCY		
Type_id	8015		
Legal Owner (entity types)	face		
Fields	real	Transparency coefficient	range 0.0 to 1.0, where 0 is opaque and 1 is transparent

A.1.10 Translucency 2

Identifier	SDL/TYSA_TRANSLUCENCY_2		
Type id	8041		
Legal Owner (entity types)	body, instance, assembly		
Fields	real	Transparency coefficient	range 0.0 to 1.0, where 0 is opaque and 1 is transparent

A.1.11 Transparency

Identifier	SDL/TYSA_TRANSPARENCY	
Type_id	8029	
Legal Owner (entity types)	body, face	
Fields	integer	Non-zero transparency coefficient value is transparent

A body may be rendered transparent if it has an attached transparency attribute with a non-zero transparency coefficient.

A.1.12 Unicode name

Identifier	SDL/TYSA_UNAME	
Type_id	8038	
Legal Owner (entity types)	assembly, body, instance, shell, face, loop, edge, vertex, group, surf, curve, point, region	
Fields	ustring	Name of entity

A.1.13 Scale factor

Identifier	SDL/TYSA_SCALE_FACTOR	
Type_id	8051	
Legal Owner (entity types)	Assembly, body	
Fields	integer	An integer representing the scale factor of the part. It must be in the range [-6, 3].

This attribute definition represents the scale factor of a part.

Note: For data interoperability purposes, we recommend applications interpret 0 as indicating a mapping 1 XT unit = 1 metre.

A.2 System attribute definitions whose presence alone defines a property

A.2.1 Group merge behaviour

Identifier	SDL/TYSA_GROUP_MERGE	
Type_id	8037	
Legal Owner (entity types)	group	
Fields	string	Unused

If a group has an attribute of this definition attached, and an entity in the group is merged with an entity not in that group, one occurrence of the resulting merged entity is added to each group that contained all of the original entities.

A.2.2 Invisibility

Identifier	SDL/TYSA_INVISIBLE
Type id	8043
Legal Owners	body, assembly, instance
Fields	none

A.2.3 Non-mergeable edges

Identifier	SDL/TYSA_EDGE_NO_MERGE	
Type_id	8032	
Legal Owner (entity types)	edge	
Fields	string	Unused

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

A.2.4 Region

Identifier	SDL/TYSA_REGION	
Type_id	8013	
Legal Owner (entity types)	face	
Fields	string	Unused

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

Document History B

Version	Date	Change
24.0	06/04/11	Updated for Parasolid v24.0 - No significant changes. Changes to document date only.
24.1	31/07/11	Updated for Parasolid v24.1 - Improvements and additions to the documentation of intersection curves and blend surfaces.
25.0	25/04/12	Updated for Parasolid v25.0 - Manual reformatted and delivered in HTML format, new system attribute definitions, and information on compound bodies.
25.1	26/10/12	Updated for Parasolid V25.1 - Manual has been updated and known errors have been removed.
27.0	19/02/14	Updated for Parasolid V27.0 - Clarified description of attribute definitions.
28.0	02/03/15	Updated for Parasolid V28.0 - Improvements to the handling of intersection curve terminators and underlying surfaces.
28.1	10/12/15	Updated for Parasolid V28.1 - Added mesh information.
30.0	07/04/17	Updated for Parasolid V30.0 - Added information on storing surface parameters of intersection curves.
30.1	01/12/17	Updated for Parasolid V30.1 - Added information on PSM Mesh data.
31.0	08/02/18	Updated for Parasolid V31.0 - Added more information on mesh data
34.0	06/04/21	Updated for Parasolid V33.1 - Added information on lattice geometry
34.0	28/06/21	Updated for Parasolid V34.0 - Added information on the precision of a transformation.
35.0	21/07/22	Updates for Parasolid V35.0 - Added more information on lattice geometry and a new system attribute definition for scaling