INTERNATIONAL STANDARD

ISO 14739-1

First edition
2014-12-15

Document management — 3D use of Product Representation Compact (PRC) format —

Part 1: PRC 10001

Gestion de documents — Utilisation en 3D du format compact de représentation de produit (PRC) —

Partie 1: PRC 10001

Reference number
ISO 14739-1:2014(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

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

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

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

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

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/TC 171, *Document management applications*, Subcommittee SC 2, *Application issues*.

Introduction

The data representations in PRC allows 3D design data, typically created in CAD and PLM systems, to be viewed and interrogated by visualization applications and to be integrated into complex documents.

This document specifies a wide range of data forms. The wide range is necessary to:

— Achieve a high fidelity, visually equivalent representation of 3D design data produced by an advanced CAD or PLM system without requiring the original application.

— Allow applications to compute high accuracy product shape measurements.

PRC is intended to complement native or open standard CAD and PLM formats as a compact, concise binary form for visualization and documentation. PRC is not intended as a data format for CAD interoperability or use in factory automation systems, e.g. automated manufacturing and inspection systems, which is addressed by the ISO 10303 standards.

Document management — 3D use of Product Representation Compact (PRC) format —

Part 1:
PRC 10001

1 Scope

This International Standard describes PRC 10001 of a product representation compact (PRC) file format for three dimensional (3D) content data. This format is designed to be included in PDF (ISO 32000) and other similar document formats for the purpose of 3D visualization and exchange. It can be used for creating, viewing, and distributing 3D data in document exchange workflows. It is optimized to store, load, and display various kinds of 3D data, especially that coming from computer aided design (CAD) systems.

This International Standard does not apply to:

— Method of electronic distribution

— Converting CAD system generated datasets to the PRC format

— Specific technical design, user interface, implementation, or operational details of rendering

— Required computer hardware and/or operating systems

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 12651:1999, *Electronic imaging — Vocabulary*

ISO 24517-1:2008, *Document management — Engineering document format using PDF — Part 1: Use of PDF 1.6 (PDF/E-1)*

ISO 32000, *Document management — Portable document format*

IEEE 754, *Floating-Point Arithmetic*

The OpenGL Graphics System, A Specification, Version 4.1 (Core Profile), July 25, 2010[1]

[1] Available at http://www.opengl.org/registry/doc/glspec41.core.20100725.pdf

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 32000-1, ISO 24517-1 and ISO 12651 and the following apply.

3.1
PRC File Writer
software application which writes a particular PRC file

3.2
PRC File Reader
software application which reads a particular PRC file

3.3
byte
group of eight bits processed as a single unit of data

4 Document syntax conventions

4.1 Conventions

The following conventions are used within this document to describe data within a PRC File.

Terms highlighted in bold within this document signify field names in the description of entity types. Entity types are denoted in *italic*.

A table with three columns is used to describe the data within a contiguous portion of the file.

The first column indicates the name of the field. Field names are not unique and can be considered to have a scope limited to the data class.

The second column describes the type of the data. This might be

— A simple data type such as a Boolean, UnsignedInteger, or Double

— A simple class of data such as PRC_TYPE_TOPO_Body or PtrTopology where the name of the class is used to define the data stored for that class

— An Array<data class>[<size>] which indicates an array of data of the specified class. An array has <size> elements. Elements of an array are referenced beginning at 0.

The third column indicates if the field is required or optional. If the field is optional a condition is described when the field is present. The field may also be described.

4.2 Example Structure

Table 1 — PRC file structure example

Name	Type	Value
flag1	*Boolean*	(Required) describe **flag1**
data_field1	*<data class1>*	(Optional; if **flag1** is TRUE) describe data_field1
data_field2	*<data class2>*	(Optional; if **flag1** is FALSE) describe data_field2
topological_body_data	*PRC_TYPE_TOPO_Body*	(Required) describe **topological_body_data**
data_type	*<enumerated type> or Integer*	(Required) describe **data_type**
data_of_type3	*<data class3>*	(Optional; if **data_type** is 1) describe **data_type3**
data_of_type4	*<data class4>*	(Optional; if **data_type** is 2) describe **data_type4**
Size	*UnsignedInteger*	(Required) describe **size**
array_of_type5	*Array<data class5>*[**size**]	(Required) describe the array of <data class5>

5 PRC file concepts

5.1 The PRC file

A PRC File is a sequential binary file, written in a way to make the file portable across machine architectures and operating systems.

PRC is optimized to store various kinds of 3D data, especially those coming from computer-aided design (CAD) systems. Most of the main constructs of CAD systems are supported within the PRC File Format:

— Assemblies and parts

— Trees of 3D entities (coordinate systems, wireframes, surfaces, and solids)

— Exact geometry representation for curves, surfaces

— Tessellated (triangulated) representation

— Markup data

PRC is meant to be multipurpose. There are two ways to store exact geometry and tessellation depending on the usage of the file and on the original information:

— Regular compression is used to directly represent CAD data without loss or transformation from the originating CAD system.

— High compression is used to store very small files, which have a specified physical tolerance from the originating shape. The tolerance is typically 0,001 mm for exact geometric data and 0,01 mm for tessellation data.

Each PRC File corresponds to a single model file (see 8.3.3) which defines the root product occurrences within the FileStructures of the PRC File. A PRC File is a collection of FileStructures which are independent from each other and can come from various authoring PRC File Writers. A FileStructure is the representation of an independent physical file denoting an independent 3D part, assembly, etc. within a PRC file. Hence, there is one header for each FileStructure with specific information and one global header for the model file which contains information about the PRC File Writer that assembled all these individual FileStructures into the final PRC File. Header sections gather primarily information on file version, FileStructure ids and offsets for reading / skipping sections.

Each FileStructure contains one or more product occurrences (see 8.3.10.2). The product occurrences denote the assembly hierarchy of the FileStructure. A product occurrence can have child nodes, which are also product occurrences. There is exactly one root product occurrence in each FileStructure. The root product occurrence is the only entry point to the FileStructure. (refer to Figure 1 below)

In parallel to the FileStructures, the model file also contains an array of root product occurrences that comprise the starting point for the entire assembly description. A product occurrence may refer to a corresponding part definition (see 8.3.11) which in turn contains:

— Geometrical data stored in a tree of representation items (see 8.6.3)

— An optional tree of part markups, grouped into annotation entities and views (see 8.7.2)

A product occurrence can contain:

— A tree of product markups

— Filters used to redefine the loading and presentation of data defined in the child product occurrences or in the part definition (see 8.3.12)

The representation items are defined through a combination of tessellation or topology and geometry data, which may be highly compressed. The markups are defined by tessellation data.

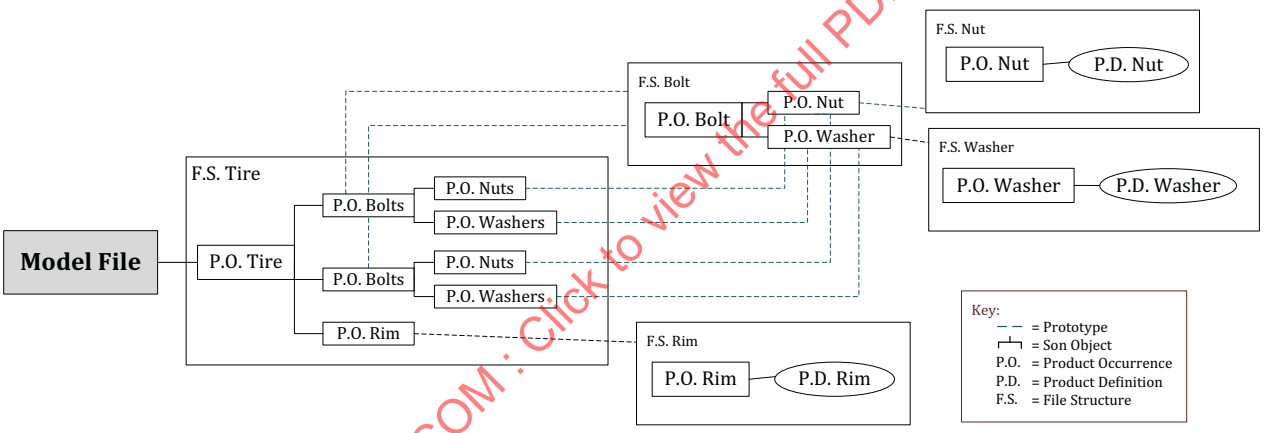


Figure 1 —Tessellation data

To optimize reading, a PRC File is arranged so that referenced entities are read before being referenced. Therefore, the FileStructures are ordered using parts, then subassemblies, and finally, the top (root) assembly.

A PRC File is composed of one header section, which starts with uncompressed data, one or more FileStructures, and one model file (*PRC_TYPE_ASM_ModelFile*) data section at the end, each individually compressed. Refer to 8.3.10 for more information about the PRC file structure.

Table 2 — PRC file structure

Section	Sub Sections	Compression	Description
FileHeader		Uncompressed	Defines originating author of the file; specifies start, and possibly end, location of other sections in file
File Structure 1	*FileStructureHeader*	Uncompressed	Identifiers of other File Structures referenced from within this FileStructure;
	Schema	Compressed	Description of changes between minimal version and authoring version of the FileStructure within the PRC File Format Specification
	Globals	Compressed	Referenced FileStructures and colors, line styles, and coordinate systems for each tree entity of the FileStructure
	Tree	Compressed	a description of the tree of items (product occurrences, part definitions, representation items, and markup)
	Tesselation	Compressed	All tessellated (triangulated) data in the leaf entities of the tree (representation items and markups).
	Geometry	Compressed	All exact geometry and topology data of the leaf entities of the tree(representation items)
	Extra Geometry	Compressed	Geometry summary data, which allow for partial loading of the FileStructure without loading the entire geometry
			Additional FileStructure sections in the PRC File
File Structure N		Compressed	Last FileStructure section in the PRC File
Schema	*ModelFile Schema*	Compressed	Description of changes between minimal version and authoring version of the ModelFile Section of the PRC File Format Specification
Model File Data	*PRC_TYPE_ASM_ModelFile*	Compressed	

5.2 Versioning

A file format version number is used to define the particular version of this international standard that a PRC File Reader or PRC File Writer conforms to.

Version numbers consist of the year modulo 2000 followed by three digits representing the day of the year. This international standard shall have the version 10001, corresponding to the 1st day of the year 2010.

The **current_version** is the maximum version number that a PRC File Reader or PRC File Writer conforms to.

Each FileStructure and the FileHeader are independent and can be copied around or written by PRC File Writers of different versions. Each of them have their own **authoring_version** and **minimal_version_for_read** version numbers. The **authoring_version** represents the version of the international standard that the PRC File Writer conformed to at the time the FileHeader or the FileStructure was written. The **minimal_version_for_read** represents the version of the international standard that a PRC File Reader shall conform to to successfully read the FileHeader or the FileStructure. If the **current_version** of the PRC File Reader is less than the **minimal_version_for_read**, then the PRC File Reader shall not continue to process the file and report an error.

If **minimal_version_for_read** is lower than **authoring_version**, the PRC File Writer shall write a schema description in the PRC File providing the differences between the two versions of this international standard. This description will enable a PRC File Reader to read and skip new information, since these new data cannot be interpreted as they are from a newer version.

A PRC File schema contains a description of new fields or new entity types added between the **minimal_version_for_read** and the **authoring_version**. See 6.3 for a description of a schema.

A PRC File Reader uses the information in the schema to read and skip new information:

— After reading each entity type, the schema information is queried and new data are skipped accordingly, following the tokens.

— Each time an entity type is read, if the type is unknown, the schema is searched and its data is skipped.

5.3 Unique identifiers

5.3.1 General

A PRC File reader/writer shall generate (writer) or interprete (reader) unique identifiers for information within the PRC File. Each FileStructure within a PRC File has an identifier (UUID) which uniquely identifies this particular FileStructure among all of the FileStructures within the PRC File. Within each FileStructure, unique identifiers for referenceable entities are generated using the order that they are first encountered in the FileStructure. Thus, the first referenceable entity in the FileStructure has the number 1 as its identifier. Subsequent identifiers of referenceable entities are incremented by 1.

5.3.2 File structure

A PRC FileStructure is identified by an identifier (see *UncompressedUniqueId*) which is unique among all of the FileStructures within a single PRC File.

The method to calculate a unique identifier for a given FileStructure is not part of this international standard.

This approach offers an advantage, for example, such as when there is some intent to repurpose FileStructures inside other PRC files without entirely rewriting them.

5.3.3 Base entities

The PRC format provides support for referencing entities. See Entity Types in 8.2.1 for a list of entity types whose entities are referenceable.

The purpose of using references on entities is to enable an interpreter to handle the same entity several times without any duplication of data, either by any other program, or by another structure in the same or another PRC file.

A referenceable entity is retrieved using the following:

— The UUID of the FileStructure.

— The entity's unique identifier (a non-zero unsigned 32-bit integer) within the FileStructure.

Just as the FileStructure UUID is unique among all of the FileStructures in the PRC File, the entity identifier is unique inside a FileStructure for all referenceable entities.

A PRC File Writer shall ensure that two referenceable entities inside the same FileStructure do not have the same identifier. The next available index within a FileStructure is the maximum value that has been assigned for identifiers to date, and is stored in the PRC File (see 8.3.4) so it is possible to safely add new entities to a FileStructure and assign them a unique identifier greater than this maximum value.

Data within the FileStructure tessellation and geometry sections are not accessible from outside the FileStructure using the approach for referenceable entities. These data are referenced only by data within the tree section (see 5.1 above) of the same FileStructure. However, it is possible (see 8.4.8) to reference topological data from outside the particular FileStructure which the topological entity lies in. This is restricted to faces in the current version of PRC but may be extended to other data in future versions of the format.

5.3.4 Other systems

In addition to the unique identifier mechanisms described above, this international standard provides for storage of identifiers from the originating system. The external identifiers and their persistence flag are stored as information in PRC strictly to support external workflows. These identifiers by themselves only exist to convey information and do not play any role in PRC.

5.4 Current data values

A PRC File reader and writer shall implement the concept of **CURRENT** for various data.

NOTE This enables smaller file sizes since duplicate data need not be written to the file. It also enables faster readers because some of the data is not being read. Default initial values are in parentheses.

— Current name (NULL)

— Current graphics

  — Index of layer (-1)

  — Index of line style (-1)

  — Behavior bit field (-1)

Current values shall be updated as they are encountered in the file. Values shall be reset when serialization is flushed at the end of every flate-compressed section (see 5.8 Compressed File Sections).

5.5 Userdata

The PRC File format provides a mechanism for a PRC File Writer to write private data within various sections of a PRC File. Such data shall consist of a bit stream of data containing the size of the bit stream followed by the specified number of bits.

NOTE Any PRC File Reader can read the bit stream and can even resave the private data, but it may not be able to interpret the data.

Each FileStructure within a PRC File may contain UserData. UserData are defined in conjunction with an application unique identifier (UUID) which allows for their interpretation. This application identifier

shall be stored in the FileStructurerHeading. By default, any user data within the FileStructure shall be assumed to be written by this one writer. However, UserData may contain streams from different writers. Therefore, different application UUIDs shall be stored accordingly with each of these other data. UserData are meant to be interpreted only by software which is aware of its meaning according to a given writers UUID. A conforming PRC File Reader should either ignore those UserData, or interpret them according to the writers UUID. PRC also allows applications to define special attributes which behave similarly to UserData,as discussed in 7.3.3.8.

Unique application identifiers are assigned through Adobe Systems, initially, and probably ISO in the long term.. Each company should have it's own unique application identifier, and if they want to share data with another company, they should share application identifiers.

5.6 Units

A conforming writer shall define the units used in a PRC File. This should be done at both the model file level (see 8.3.3) and at the product occurrence level (see 8.3.10). The unit may come from an actual CAD file and therefore be considered reliable to represent physical values in the data. If a unit is valid/reliable, the flag unit_from_CAD_file shall be set to TRUE. However, some formats do not contain units. Regardless, it is mandatory to define one unit at both the model file level and at the product occurrence level .

The unit which shall be used is the first valid unit in the ModelFile / ProductOccurrence chain. If a valid unit is defined at the ModelFile level, it will apply to all product occurrences. Once a valid unit is found, the remainder of the data shall be interpreted with respect of that unit, even for occurrences higher in the product occurrence hierarchy.

NOTE In other words, if a product occurrence having no valid unit has a child with a valid unit, it is assumed that the entire hierarchy of model file and product occurrence are to be interpreted and used according to this unit.

If no valid unit is found at either the ModelFile level or any ProductOccurrence level (i.e. unit_from_CAD_file is always set to FALSE), a conforming PRC File Reader shall clearly indicate that the unit defined is not valid for measurement purposes.

5.7 Tolerances

PRC distinguishes between several notions of tolerances:

— A first notion of tolerance represents the maximum deviation between compressed and original data, as introduced by the lossy compression of geometry or topology. When provided, this tolerance shall be a user-defined value which represents a physical length given with a unit.

— A second notion of tolerance is introduced by numerical uncertainty inherent in every 3D modelling system. This form of tolerance is non-dimensional. Its purpose is to perform consistent numerical operations. This tolerance value corresponds to coincidence (e.g. of two 3D vertices) and is generally defined in conjunction with a minimal value representing zero and a maximal value representing infinity. For instance, a system might define coincidence at 1e-3, zero as any value less than 1e-12 and infinity as any value greater than 1e6. Then, additional logic outside the modelling system should define the unit so that the numerical values can be interpreted by the computer as physical values (i.e. in a particular unit).

In PRC, the 3D modelling system corresponds to entities in tessellation section (8.8) and topology section (7.9). Hence the various tolerances stored within these sections are always numerical values with no unit. A conforming writer shall never store a tolerance which can be directly interpreted as a physical value.

NOTE For instance, brep_data_compressed_tolerance in 8.9.19 which represents the deviation introduced between original data and compressed one is stored without unit, even if it might be derived from a user interface

which takes those units into account. Then, outside this 3D modelling system, unit is defined in ProductOccurrences and ModelFile as discussed in previous chapter. The physical interpretation of those tolerances should then be done from both indications. For instance, if a tolerance in a topology section is stored as 0,001 and if the unit of the ProductOccurrence it belongs to is 1000 meters, then the physical tolerance on data is actually 1 meter.

5.8 Compressed file sections

Within a PRC File, all sections, except header sections, are individually compressed with a Flate method.

NOTE This form of compression is considered to be "lossless". It occurs systematically whatever the actual content of the PRC file, and even if it contains compressed geometry or tessellation.

5.9 Compressed geometry

Compression of geometry results in very small files. This compression is "lossy" in that the geometry is an approximation to geometry to within a specified tolerance (typically 0,001mm). See 8.9.21 and 8.9.20 for entities representing compressed geometry. Note that the methodology to determine an approximation of the original geometry (e.g. analytic recognition) is not part of PRC standard. Only the resulting entities and the method to store them are described in PRC.

5.10 Compressed tessellation

Compression of tessellation data results in very small files. This compression is "lossy" in that the tessellation data is an approximation to tessellation data to within a specified tolerance (typically 0,01mm). See 8.8.9 for an entity representing compressed tessellation. Tessellation data is not necessarily generated from or considered as an "approximation" of geometry; it is an alternative way to convey data. Note that the methodology to determine an approximation of the original tessellation (e.g. polygon decimation) is not part of PRC standard. Only the resulting entities and the method to store them are described in PRC.

6 PRC file contents

6.1 Fileheader

6.1.1 General

The Header contains the file version for the authoring version (PRC Format Specification) that the PRC File Writer is based on and the minimal version for read (PRC Format Specification) that a conforming PRC File Reader is based on that write/read the data outside of the individual FileStructures in the PRC File.

Each FileStructure has a identifier which is unique among all of the FileStructures within this PRC File.

Each application has a unique identifier which enables the interpretation of UserData. This identifier shall be set to 0000 or to a valid application identifier. 0000 indicates that the authoring application is not "registered" but (as discussed above) there can still be user data from another application in the file. Valid identifiers are distributed by Adobe Systems, Inc. upon request.

A valid PRC File shall contain at least one FileStructure.

Table 3 — Fileheader

Name	Data Type	Data Description
	3 bytes: P, R, C	(Required) Characters "PRC"
minimal_version_for_read	*UncompressedUnsignedInteger*	(Required) Minimal version for read (see 5.2)
authoring_version	*UncompressedUnsignedInteger*	(Required) Authoring version (see 5.2)
unique_id_file	*UncompressedUniqueId*	(Required) Unique ID for file structure; The first and last sections of PRC File (before and after the FileStructures themselves) are a kind of special file structure which bear an ID as well.
unique_id_application	*UncompressedUniqueId*	(Required) Unique ID for application
filestructure_count	*UncompressedUnsignedInteger*	(Required) Number of FileStructures in PRC file
file_info	Array <*FileStructureDescription*>[**filestructure_count**]	(Required) Information describing each FileStructure in PRC File; the ordering of this array reflects the ordering of the FileStructure within the file.
start_offset	*UncompressedUnsignedInteger*	(Required) Start offset of ModelFileData section from beginning of PRC File (in bytes)
end_offset	*UncompressedUnsignedInteger*	(Required) End offset of ModelFileData section from beginning of PRC File (in bytes)
file_count	*UncompressedUnsignedInteger*	(Required) Number of uncompressed files that are saved in the PRC File
files:	Array <*UncompressedFiles*>[**file_count**]	(Optional; if **file_count** is greater than 0) Array of uncompressed files stored in the PRC File

6.1.2 *FileStructureDescription*

The *FileStructureDescription* contains information defining a particular FileStructure within the PRC File:

— Each FileStructure has an identifier which is unique among the FileStructures within a PRC File.

— The starting offset (in bytes from the beginning of the PRC File) of each of the following sections within a FileStructure: header, globals, tree, tessellation, geometry, and extra geometry.

Table 4 — *FileStructureDescription*

Name	Data Type	Data Description
unique_id	*UncompressedUniqueId*	(Required) Unique id for this file structure
	UncompressedUnsignedInteger	(Required) Reserved; shall be 0
section_count	*UncompressedUnsignedInteger*	(Required) Number of sections in this FileStructure (6 for header, globals, tree, tesselation, geometry, and extra geometry)
section_offsets	Array <*UncompressedUnsignedInteger*>[**section_count**]	(Required) Start offset (in bytes) of each section from the beginning of PRC File

6.1.3 *UncompressedFiles*

Directly embeds private data inside a PRC File. This may be referenced by objects in the same PRC File using the index of the uncompressed file within this array, and interpreted accordingly. Up to this version of the PRC File Format Specification, only picture objects make reference to these files (see 8.5.5)

Table 5 — *UncompressedFiles*

Name	Data Type	Data Description
count	*UncompressedUnsignedInteger*	(Required) Number of uncompressed files
array_of_files	Array<*UncompressedBlock*>[count]	(Required) Arbitrary data to embed within a PRC File

6.2 Filestructure

6.2.1 General

Table 6 — *Filestructure*

Name	Data Type	Data Description
header	*FileStructureHeader*	(Required)
schema	*FileStructureSchema*	(Required) Define the **schema** for the entities in this FileStructure which have changed between the **minimal_version_for_read** and the **authoring_version**
globals	*PRC_TYPE_ASM_FileStructureGlobals*	(Required) Referenced FileStructures and colors, line styles, and coordinate systems for each tree entity of the file structure
tree	*PRC_TYPE_ASM_FileStructureTree*	(Required) a description of the tree of items (product occurrences, part definitions, representation items, and markup)
tessellation	*PRC_TYPE_ASM_FileStructureTessellation*	(Requireed) All tessellated (triangulated) data in the leaf entities of the tree (representation items and markups).

Table 6 *(continued)*

Name	Data Type	Data Description
Geometry	*PRC_TYPE_ASM_FileStructureGeometry*	(Required) All exact geometry and topology data of the leaf entities of the tree (representation items)
extra_geometry	*PRC_TYPE_ASM_FileStructureExtraGeometry*	(Required) Geometry summary data, which allow for partial loading of the file structure without loading the entire geometry

6.2.2 *FileStructureHeader*

A *FileStructureHeader* defines various properties of a FileStructure:

— The minimal file version of the a conforming PRC File Reader;

— The authoring version of a conforming PRC File Writer that wrote this FileStructure;

— The unique ID of this FileStructure; this ID shall be the same as the identifier in the PRC File Header section;

— The unique ID of the conforming PRC File Writer creating this file structure;

— A variable number of private uncompressed data files.

Table 7 — *FileStructureHeader*

Name	Data Type	Data Description
	3 bytes: P, R, C	(Required) Characters "PRC"
minimal_version_for_read	*UncompressedUnsignedInteger*	(Required) Minimal version for read (see 5.2)
authoring_version	*UncompressedUnsignedInteger*	(Required) Authoring version (see 5.2)
unique_id_file	*UncompressedUniqueId*	(Required) Unique ID of this FileStructure
unique_id_application	*UncompressedUniqueId*	(Required) Unique ID for application
file_count	*UncompressedUnsignedInteger*	(Required) Number of uncompressed files
files	Array <*UncompressedFile*>[file_count]	(Required) Array of uncompressed data

6.2.3 *FileStructureSchema*

Each FileStructure in a PRC File may represent a different version of the PRC Format Specification written by a different application (PRC File Writer). Each FileStructure shall define the schema for changes (new entities and new data fields to existing entities) for the entities that are stored within it. See 6.3 for a description of the facilities for describing a schema.

6.3 PRC Schema

6.3.1 General

A schema describes changes between versions of PRC File Format Specification. See 5.2 for a basic description of versioning in PRC. This mechanism allows information to be added for a given entity type (**schema_type**) and still be readable by previous versions of the software.

Only those types which have changed between the **minimal_version_for_read** and **authoring_version** require a schema description, and only if they are present in the file.

Table 8 — PRC Schema

Name	Data Type	Data Description
schema_count	*UnsignedInteger*	(Required) Number of entity types that have changed between minimal version for read and authoring version
schemas	Array <*Entity_schema_definition*>[schema_count]	(Required) Schema definition of each type that has changed

6.3.2 *Entity_schema_definition*

This provides the schema definition for an entity type in a PRC File. The entity is described by an array of schema tokens which in turn should be viewed as a list of versioned blocks which describe versions of the entity. See 9.3.20 for a description.

Table 9 — *Entity_schema_definition*

Name	Data Type	Data Description
entity_type	*UnsignedInteger*	(Required) entity_type represents the entity type, such as 8.10.1, that is being described by the array of schema tokens.
token_count	*UnsignedInteger*	(Required) Number of tokens describing this entity type
schema_tokens	Array <*UnsignedInteger*>[token_count]	(Required) Array of schema tokens describing this entity type

7 PRC basic types

7.1 General

All data within a physical PRC File exists in a section which is a contiguous stream of bytes which starts and ends on a byte boundary. A section may be either uncompressed (header sections) or compressed (all other sections).

Data within an uncompressed section is written so that individual variables occupy a specific number of bytes.

Data within a compressed section has been written in a bit-by-bit manner with no restrictions on individual variables crossing byte boundaries. At the end of the section, the last byte is padded with zero bits, the entire section is compressed using flate (see Bibliography) and the compressed section is written to the file. At this point, the current name and current graphics are reset.

Reading one section is independent from reading other sections and one can skip directly to a section since the offset (in bytes) of the section from the beginning of the file is known as it is also stored in the PRC File.

7.2 Uncompressed types

7.2.1 *UncompressedUnsignedInteger*

This type represents writing an unsigned integer. An unsigned integer is converted into a 4 byte array of unsigned characters which is independent of machine byte ordering using the algorithm MakePortable32BitsUnsigned.

Table 10 — *UncompressedUnsignedInteger*

Name	Data Type	Data Description
	4 bytes	(Required) 4 bytes representing unsigned integer

7.2.2 *UncompressedBlock*

The purpose of this function is to write the number of bytes being written followed by the specified number of bytes without the need to further interpret the content of the byte stream.

Table 11 — *UncompressedBlock*

Name	Data Type	Data Description
block_size	*UncompressedUnsignedInteger*	(Required) size (bytes) of uncompressed block
block	*Byte stream of the specified size*	(Required) Block of specified size

7.2.3 *UncompressedFiles*

This type represents writing the data for multiple uncompressed file data.

Table 12 — *UncompressedFiles*

Name	Data Type	Data Description
file_count	*UncompressedUnsignedInteger*	(Required) Number of uncompressed Files
files	Array <*UncompressedBlock*>[file_count]	(Required) Array of uncompressed file data

7.2.4 *UncompressedUniqueId*

This saves information on uniqueid in uncompressed mode.

Table 13 — *UncompressedUniqueId*

Name	Data Type	Data Description
unique_id0	*UncompressedUnsignedInteger*	(Required)
unique_id1	*UncompressedUnsignedInteger*	(Required)
unique_id2	*UncompressedUnsignedInteger*	(Required)
unique_id3	*UncompressedUnsignedInteger*	(Required)

7.3 Compressed types

7.3.1 General

The following types are used to define data contained within compressed sections of the PRC File. Each type may start or end at any bit position within a byte in the compressed section.

7.3.2 Atomic Types

7.3.2.1 *Bits(n)*

Writes n-number of bits from left to right.

7.3.2.2 *Boolean*

Boolean values shall be written as a single bit

Table 14 — *Boolean*

Name	Data Type	Data Description
	Bits(1)	(Required) Single bit in a bit stream

7.3.2.3 *Character*

Characters will be written as a single 8 bits.

Table 15 — *Character*

Name	Data Type	Data Description
	Bits(8)	(Required) Single character in a bit stream

7.3.2.4 *Integer*

This requires a special algorithm. See 10.11.

7.3.2.5 *IntegerWithVariableBitNumber*

This requires a special algorithm. See 10.12.

7.3.2.6 *UnsignedInteger*

This requires a special algorithm. See 10.10.

7.3.2.7 *UnsignedIntegerWithVariableBitNumber*

This requires a special algorithm. See 10.13.

7.3.2.8 *NumberOfBitsThenUnsignedInteger*

This requies a special algorithm. See 10.15.

7.3.2.9 *Float*

This requires a special algorithm. See 10.5.

7.3.2.10 *Double*

This requires a special algorithm. See 10.17.

7.3.2.11 *DoubleWithVariableBitNumber*

This requires a special algorithm. See 10.14.

7.3.3 Compound Types

7.3.3.1 *CompressedEntityType*

This requires a special algorithm. See 10.16.

7.3.3.2 *CompressedUniqueId*

This saves information on uniqueid in compressed mode. See 5.3

Table 16 — *CompressedUniqueId*

Name	Data Type	Data Description
unique_id0	*UnsignedInteger*	(Required)
unique_id1	*UnsignedInteger*	(Required)
unique_id2	*UnsignedInteger*	(Required)
unique_id3	*UnsignedInteger*	(Required)

7.3.3.3 *String*

This is a UTF8-encoded string without a terminating NULL character. The number of characters does not include a terminating null character.

Table 17 — *String*

Name	Data Type	Data Description
null_flag	*Boolean*	(Required) TRUE if the string is not NULL; else FALSE
size	*UnsignedInteger*	(Optional; if **null_flag** is TRUE) Size of character array
string	Array <*Character*>[**size**]	(Optional; if **null_flag** is TRUE) Array of characters in String

7.3.3.4 *CharacterArray*

This requires a special algorithm. See 10.6.

7.3.3.5 *ShortArray*

This requires a special algorithm. See 10.7.

7.3.3.6 *CompressedIntegerArray*

This requires a special algorithm. See 10.8.

7.3.3.7 *CompressedIndiceArray*

This requires a special algorithm. See 10.9.

7.3.3.8 *UserData*

Applications that write a PRC File may store an arbitrary bit stream of private data for the following data types :

— Subtypes of PRC_TYPE_ASM

— Subtypes of PRC_TYPE_RI

— Subtypes of PRC_TYPE_MKP

— Subtypes of PRC_TYPE_MISC

All those entities have UserData at the end of their data definition.

Table 18 — *UserData*

Name	Data Type	Data Description
stream_size	*UnsignedInteger*	(Required) Number of bits in the *UserDataStream*
stream	*UserDataStream*	(Optional; if **stream_size** > 0) Arbitrary bit stream of **stream_size** length

7.3.3.8.1 *UserDataStream*

Table 19 — *UserDataStream*

Name	Data Type	Data Description
section_count	*UnsignedInteger*	(Required) Number of UserData sub sections in the bit stream
sub_sections	Array <*UserDataSubSection*>[section_count]	(Required) Array of sub-sections

7.3.3.8.2 *UserDataSubSection*

Name	Data Type	Data Description
same_flag	*Boolean*	(Required) Same Application UUID as FileStructure
unique_id_application	*CompressedUniqueId*	(Optional; if **same_flag** is FALSE) Application UUID
stream_size	*UnsignedInteger*	(Required) Number of bits in the bit stream for this user data subsection
stream	*Bits(n)*	(Optional; if stream_size is > 0) Arbitrary bit stream of the stream_size length

7.3.4 Parameter Range Types

7.3.4.1 *Infinite_param*

The constant **infinite_param** shall have the double precision floating point value 12345. It is used to represent infinity, and as **–infinite_param** also minus infinity, for parameter values in **interval**.

7.3.4.2 *Interval*

An Interval is a subset of R^1. It is represented by two double precision numbers defining the minimum and maximum values of the interval. An interval is defined to be all values between the minimum and maximum

minimum ≤ t ≤ maximum

The minimum shall not be smaller than **–infinite_param** and the maximum shall not be larger than **infinite_param**. Infinite or semi-infinite intervals may be specified by using infinite_param to represent infinity. The following are examples of intervals:

— [0.0, 1.0]

— [-infinite_param, infinite_param]

— [0.0, infinite_param]

The primary use of an interval is to define the domain of a curve. For a non-periodic curve, whether an open or closed curve, the interval defines the domain of legal parameter values for the curve. For a periodic closed curve, such as a circle or periodic closed NURBS curve, the interval defines the period (i.e. the length of the interval) as well as the primary domain of the curve. Any value is a legal parameter when evaluated modulo the period. For a periodic open curve, such as a proper subset of a circle, the interval defines the valid subset of R^1 that is the domain of legal parameter values.

Table 20 — *Interval*

Name	Data Type	Data Description
min_value	*Double*	(Required) Minimum value
max_value	*Double*	(Required) MaximumValue

7.3.4.3 *Parameterization*

Parameterization data provide a way to reparameterize a curve and to define the domain of the curve. The domain of the curve define legal parameter values.

Two doubles, **coeff_a** and **coeff_b**, are used to reparameterize a curve towards its implicit parameterization.

The evaluation formula to calculate the implicit parameter from a parameter is:

implicit_parameter ← . **coeff_a** * parameter + **coeff_b**

If there is no reparameterization of the curve, **coeff_a** shall be set to 1.0 and **coeff_b** shall be set to 0.0. In this case implicit_parameter equals the specified parameter.

Parameterization data also contain an interval used to define the domain of the curve. This interval restricts the curve before applying the reparameterization formula. The reparameterization formula can be used to calculate the implicit_interval from the given interval in the same way that it is used to calculate the implicit_parameter from a given parameter.

All curves shall have an interval to define the legal parameter values of the curve. In the case of base curves (curves not defined by reference to other curves), this interval defines the domain of definition for the curve. For curves which are defined in terms of other curves, the interval represents a subset of the curve which may be the entire curve or a portion of it. The interpretation of the interval depends upon the curve being periodic or non-periodic (see the definition of Interval).

For instance, a circle has the implicit domain of [0.0, 2π] and a line has the implicit domain of [-infinite_param, infinite_param]. A portion of the circle might be limited to [0.0, π].

Table 21 — *Parameterization*

Name	Data Type	Data Description
trim_interval	*Interval*	(Required) trim interval
coeff_a	*Double*	(Required) Coeff_a
coeff_b	*Double*	(Required) Coeff_b

As an example, consider a circle. It has an implicit parameterization on the interval [0.0, 2π]. For this case:

— Coeff_a = 1.0

— Coeff_b = 0.0

— Interval = [0.0, 2π]

To reparameterize the circle so the parameter values are in the interval [0.0, 1.0] would give

— Coeff_a = 2π

— Coeff_b = 0.0

— Interval = [0.0, 1.0]

To define a semi circle

— Coeff_a = 1.0

— Coeff_b = 0.0

— Interval = [-π/2.0, π/2.0]

7.3.4.4 *Domain*

A Domain is a subset of R^2 that is used to define the domain of definition of a surface. It is represented by a 2D vector defining the minimum UV parameter values and a 2D vector defining the maximum UV parameters.

The domain of the surface is

minimum U ≤ u ≤ maximum U

minimum V ≤ v ≤ maximum V

The interval in U (or V) when used with a specific surface definition will define a surface that is open, closed, or periodic in that parameter. For a non-periodic surface, whether an open or closed surface, the interval defines the domain of legal parameter values for the surface. For a periodic closed surface, such as a cone or periodic closed NURBS surface, the interval defines the period (i.e. the length of the interval) as well as the primary domain of the surface. Any value is a legal parameter when evaluated modulo the period. For a periodic open surface, such as a proper subset of a cone, the interval defines the valid subset of R^2 that is the domain of legal parameter values.

Table 22 — *Domain*

Name	Data Type	Data Description
min_uv	*Vector2d*	(Required) Minimum U and V
max_uv	*Vector2d*	(Required) Maximum U and V

7.3.4.5 *UVParameterization*

This describes a reparameterization of a surface towards implicit parameterization. Coeff_a shall be set to 1.0 and Coeff_b shall be set to 0.0 if there is no reparameterization.

The evaluation formula is:

If (swap_uv is FALSE) {

implicit_param.u ← **u_param_coeff_a** * param.u + **u_param_coeff_b**

implicit_param.v ← **v_param_coeff_a** * param.v + **v_param_coeff_b**

} else {

implicit_param.u ← **v_param_coeff_a** * param.v + **v_param_coeff_b**

implicit_param.v ← **u_param_coeff_a** * param.u + **u_param_coeff_b**

}

Table 23 — *UVParameterization*

Name	Data Type	Data Description
swap_uv	*Boolean*	(Required) **swap_uv** TRUE implies swap the uv param; FALSE implies do not swap
suface_domain	*Domain*	(Required) The domain of the surface is specified in numbers before the previous reparameterization formula is aplied.
u_param_coeff_a	*Double*	(Required)
v_param_coeff_a	*Double*	(Required)
u_param_coeff_b	*Double*	(Required)
v_param_coeff_b	*Double*	(Required)

7.3.5 Basic Geometry Types

7.3.5.1 *Vector2d*

Representation of a 2D vector. This type may be used to represent either 2D positions or vectors. The context shall be used to determine if a position or vector is meant.

Table 24 — *Vector2d*

Name	Data Type	Data Description
x_value	*Double*	(Required)
y_value	*Double*	(Required)

7.3.5.2 *Vector3d*

Representation of a 3D vector. This type may be used to represent either 3D positions or vectors. The context shall be used to determine if a position or vector is meant.

Table 25 — *Vector3d*

Name	Data Type	Data Description
x_value	*Double*	(Required)
y_value	*Double*	(Required)
z_value	*Double*	(Required)

7.3.5.3 *BoundingBox*

Define a bounding box with sides parallel to the XYZ coordinate planes using two diagonal corners of the box. The minimum and maximum shall satisfy

— Xmin < Xmax

— Ymin < Ymax

— Zmin < Zmax

If the above is not satisfied the bounding box is considered invalid.

Table 26 — *BoundingBox*

Name	Data Type	Data Description
minimum_corner	*Vector3d*	(Required)
maximum_corner	*Vector3d*	(Required)

8 Base entities

8.1 General

BASE ENTITIES represent high level concepts such as curves, surfaces, topology, parts, assemblies, markups, or tessellation data which are stored in a PRC File. Entities are defined by a type name used for descriptive purposes, a type value which is stored in the PRC File to indicate that the data defining the entity follows, and an indication if the entity is referenceable. An entity is referenceable if it may be referenced using a unique identifier.

8.2 Abstract root types

8.2.1 Entity types

Table 27 — Abstract root types entity types

Type Name	Type Value
PRC_TYPE_ROOT	0
PRC_TYPE_ROOTBase	*PRC_TYPE_ROOT*+ 1
PRC_TYPE_ROOT_PRCBaseWithGraphics	*PRC_TYPE_ROOT*+ 2
PRC_TYPE_ROOT_PRCBaseNoReference	*PRC_TYPE_ROOT* + 3

8.2.2 *PRC_TYPE_ROOT*

An entity of this type in a PRC File is to be interpreted as a NULL pointer or entity depending on the specific circumstances.

8.2.3 *PRC_TYPE_ROOTBase*

8.2.3.1 General

This is the abstract root type for any PRC entity.

8.2.3.2 *ContentPRCBase*

This represents common data for all PRC base entities that are not eligble to be referenced.. All PRC base entities have attribute and name information.

Table 28 — *ContentPRCBase*

Name	Data Type	Data Description
attribute_data	*AttributeData*	(Required) Attribute data associated with the entity
entity_name	*Name*	(Required) Entity Name

8.2.3.3 *ContentPRCRefBase*

This represents common data for all PRC base entities that are eligible to be referenced. All PRC base entities have attribute, name information, a PRC FileStructure unique identifier and a persistent and non-persistent identifier from the originating CAD file.

Table 29 — *ContentPRCRefBase*

Name	Data Type	Data Description
attribute_data	*AttributeData*	(Required) Attribute data associated with the entity
entity_name	*Name*	(Required) Entity Name
unique_id_cad	*UnsignedInteger*	(Required) persistent CAD identifier from originating CAD file
unique_id	*UnsignedInteger*	(Required) PRC FileStructure unique identifier

8.2.3.3.1 *AttributeData*

A base entity may have zero or more associated attributes.

Table 30 — *AttributeData*

Name	Data Type	Data Description
attribute_count	*UnsignedInteger*	(Required) Number of attributes
attributes	Array <*PRC_TYPE_MISC_Attribute*>[attribute_count]	(Required) An array of attributes

8.2.3.3.2 *Name*

PRC employs the concept of a current_name which retains the name of the last entity being read or written. If the name of the subsequent entity being read or written is the same as the current name, no name will be in the file. Otherwise, a name for the entity is read from the file and the current name is updated to this new name. This is done to optimize on space within the file.

Table 31 — *Name*

Name	Data Type	Data Description
same_name	*Boolean*	(Required) TRUE implies the name of this entity is the same as the current name; FALSE implies that a new name is in the file
name	*String*	(Optional; if **same_name** is FALSE) Name of the entity and the current name is set to this name

8.2.4 *PRC_TYPE_ROOT_PRCBaseWithGraphics*

8.2.4.1 General

Information for any base PRC entity which can be referenced and which contains graphics.

PRC employs the concept of current graphics content which retains the graphics content of the last entity being read or written. If the graphics content of the subsequent entity being read or written is the same as that of the current graphics content, no graphics content will be in the file. Otherwise, a graphics content for the entity is read from the file and the current graphics content is updated to this new graphics content. This is done to optimize on space within the file.

Table 32 — *PRC_TYPE_ROOT_PRCBaseWithGraphics*

Name	Data Type	Data Description
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
same_graphics	*Boolean*	(Required) SameGraphicsAsCurrent
graphic_content	*GraphicsContent*	(Optional; if **same_graphics** is TRUE) Graphical data associated with the entity and the current graphics content is set to this graphics content

8.2.4.2 *GraphicsContent*

index_of_line_style represents the index into the array of styles, which is stored in FileStructureInternalGlobalData (See section 8.3.5.2below). The array of styles is a set of PRC_TYPE_GRAPH_Style entities (See 8.5.3) and should have a value less than 65535.

behaviour_bit_field is an unsigned short integer (2 bytes). This bit field controls visibility and removal of an entity as well as how values of entities (visibility, color, transparency, layer, line pattern, and line width) are inherited in the tree of entities.

Each value that is subject to inheritance has two bit flags in the **behavior_bit_field**: one flag is called Childherit. If ChildHerit is set, the actual value of ParentHerit is ignored and assumed to be FALSE.

The actual value that the entity uses depends on the path of entites from the root of the tree of entites to this entity and the **behavior_bit_field** settings along this path.

If there are ChildHerit flags in the tree, it is the lowest node in the tree which has this flag which defines the value. Else if there are ParentHerit flags in the tree, it is the highest node in the tree which has this flag which defines the value. Else if there is no flag, the current value is set, if any.

Potential values and meanings of this bit field are:

Table 33 — behavior_bit_field

Name	Value	Meaning
PRC_GRAPHICS_Show	0x0001	The entity is shown.
PRC_GRAPHICS_ChildHeritShow	0x0002	Shown entity child inheritance.
PRC_GRAPHICS_FatherHeritShow	0x0004	Shown entity parent inheritance.
PRC_GRAPHICS_ChildHeritColor	0x0008	Color/material child inheritance.
PRC_GRAPHICS_ParentHeritColor	0x0010	Color/material parent inheritance
PRC_GRAPHICS_ChildHeritLayer	0x0020	Layer child inheritance.
PRC_GRAPHICS_ParentHeritLayer	0x0040	Layer parent inheritance.
PRC_GRAPHICS_ChildHeritTransparency	0x0080	Transparency child inheritance.
PRC_GRAPHICS_ParentHeritTransparency	0x0100	Transparency parent inheritance
PRC_GRAPHICS_ChildHeritLinePattern	0x0200	Line pattern child inheritance.
PRC_GRAPHICS_ParentHeritLinePattern	0x0400	Line pattern parent inheritance.
PRC_GRAPHICS_ChildHeritLineWidth	0x0800	Line width child inheritance
PRC_GRAPHICS_ParentHeritLineWidth	0x1000	Line width parent inheritance
PRC_GRAPHICS_Removed	0x2000	The entity has been removed and no longer appears in the tree

Table 34 — *GraphicsContent*

Name	Data Type	Data Description
biased_layer_index	*UnsignedInteger*	**(Required) layer_index** + 1 where **layer_index** represents the layer the entity lies on. It should have a value less than 65535.
biased_index_of_line_style	*UnsignedInteger*	**(Required) index_of_line_style** + 1 where **index_of_line_style** represents the index into the array of styles, which is stored in *FileStructureInternalGlobalData* (See section 8.3.5.2 below). The array of styles is a set of *PRC_TYPE_GRAPH_Style* entities (See 8.5.3) and should have a value less than 65535.
behavior_bit_field1	*UnsignedCharacter*	**(Required) behavior_bit_field1 is behavior_bit_field & 0xff**
behavior_bit_field2	*UnsignedCharacter*	**(Required) behavior_bit_field2 is behavior_bit_field >> 8**

8.2.5 *PRC_TYPE_ROOT_PRCBaseNoReference*

This is the abstract root type for any PRC entity that can not be referenced.

This type is useful for schema descriptions. For example, *EPRCSchema_Parent_Type* can be used to define a new type which has one of three different ancestor types. It can be a child of either *PRC_TYPE_ROOTBase*, *PRC_TYPE_ROOT_PRCBaseWithGraphics*, or *PRC_TYPE_ROOT_PRCBaseNoReference*.

PRC_TYPE_ROOTBase: ancestor type would indicate that the new type is a referencable type

PRC_TYPE_ROOT_PRCBaseNoReference: ancestor type would indicate that the new type is not referencable.

PRC_TYPE_ROOT_PRCBaseWithGraphics.: ancestor type would indicate that the new type is a referencable type which bear graphics

Examples :

PRC_TYPE_GRAPH_LinePattern has PRC_TYPE_ROOTBase in its ancestor chain

PRC_TYPE_ASM_FileStructure has *PRC_TYPE_ROOT_PRCBaseNoReference* in its ancestor chain

PRC_TYPE_RI_Curve has *PRC_TYPE_ROOT_PRCBaseWithGraphics* in its ancestors chain

8.3 Structure and assembly

8.3.1 Entity types

Table 35 — Structure and assembly entity types

Type Name	Type Value	Referenceable
PRC_TYPE_ASM	*PRC_TYPE_ROOT* + 300	no
PRC_TYPE_ASM_ModelFile	*PRC_TYPE_ASM*+ 1	no
PRC_TYPE_ASM_FileStructure	*PRC_TYPE_ASM*+ 2	no
PRC_TYPE_ASM_FileStructureGlobals	*PRC_TYPE_ASM*+ 3	no
PRC_TYPE_ASM_FileStructureTree	*PRC_TYPE_ASM*+ 4	no
PRC_TYPE_ASM_FileStructureTessellation	*PRC_TYPE_ASM*+ 5	no
PRC_TYPE_ASM_FileStructureGeometry	*PRC_TYPE_ASM*+ 6	no
PRC_TYPE_ASM_FileStructureExtraGeometry	*PRC_TYPE_ASM*+ 7	no
PRC_TYPE_ASM_ProductOccurrence	*PRC_TYPE_ASM*+ 8	yes
PRC_TYPE_ASM_PartDefinition	*PRC_TYPE_ASM*+ 9	yes
PRC_TYPE_ASM_Filter	*PRC_TYPE_ASM*+ 10	yes

8.3.2 *PRC_TYPE_ASM*

This is the abstract type for the top level PRC structure.

8.3.3 *PRC_TYPE_ASM_ModelFile*

8.3.3.1 General

A model file (*PRC_TYPE_ASM_ModelFile*) is typically created by importing data from a CAD file. There is only a single *PRC_TYPE_ASM_ModelFile* entity in a PRC File. The model file contains product occurrences, which are split into different FileStructures.

units_from_CAD_file is to be interpreted as discussed in section *Units* (See 5.6)

number_of_root_product_occurrences is the number of root product occurrences in the model file.

product_occurrences represents the root product occurrences in the model file.

file_structure_index_in_model_file indicates the index at which the FileStructure should be stored in memory within a ModelFile (*PRC_TYPE_ASM_ModelFile*). A conforming PRC File Reader should reconstitute / maintain FileStructures in memory in the order that they appear in the ModelFile, regardless of the order in the physical PRC file. In the physical PRC file, the order of file structures shall be in accordance with their dependencies with other file structures as follows: if a given file structure A depends on another file structure B (in the sense that A references elements of B), B shall appear first in the PRC physical file. This restriction does not exist for the memory storage of the file structures.

number_of_file_structures is obtained directly from the header of the PRC File.

Table 36 — *PRC_TYPE_ASM_ModelFile*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_ASM_ModelFile*
Base	*ContentPRCBase*	(Required) Base information associated with the entity
units_from_CAD_file	*Boolean*	(Required) Units_from_CAD_file is TRUE if the units come from a CAD system and are thus suitable for measurement; else FALSE
	Double	(Optional; if **units_from_CAD_file** is TRUE) Units in multiple of mm
number_of_root_product_occurrences	*UnsignedInteger*	(Required) **number_of_root_product_occurrences**
product_occurences	Array <*ProductOccurrenceReference*> [**number_of_root_product_occurrences**]	(Required) References to the root **product occurrences** in the model file
file_structure_index_in_model_file	Array <*UnsignedInteger*> [**number_of_root_product_occurrences**]	(Required) **file_structure_index_in_model_file**: Indicies to to FileStructure within the model file; the size of the array is obtained directly from the header of the PRC file
user_data	*UserData*	(Required) User defined data

8.3.3.2 *ProductOccurrenceReference*

This defines the unique identifier of the *PRC_TYPE_ASM_FileStructure* and the index of the root product occurence in the array of product occurrences within the *PRC_TYPE_ASM_FileStructureTree*, as contained within the *PRC_TYPE_ASM_FileStructure*.

product_occurrence_is_active is reserved for future use, and is currently unused. Its default value should be TRUE. Its use will be to control storing different versions/configurations of a product in the same PRC File. Currently, only one can be active at a time but you can switch from one to the other.

Table 37 — *ProductOccurrenceReference*

Name	Data Type	Data Description
unique_id	*CompressedUniqueId*	(Required) Unique identifier of the FileStructure that contains this root product occurrence.
root_index	*UnsignedInteger*	(Required) Index of the root product occurrence within the FileStructure
product_occurence_is_active	*Boolean*	(Required) TRUE if the product occurrence is active; else FALSE

8.3.4 *PRC_TYPE_ASM_FileStructure*

This type gathers internal data of a file structure as described in *PRC_TYPE_ASM_FileStructureTree*

Table 38 — *PRC_TYPE_ASM_FileStructure*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_ASM_FileStructure*
Base	*ContentPRCBase*	(Required) Base information associated with the entity
next_available_index	*UnsignedInteger*	(Required) **next_available_index** is used when re-opening the FileStructure, to be able to safely add entities with a unique id without overlap to pre-existing entities.
index_product_occurrence	*UnsignedInteger*	(Required) **index_product_occurrence** is used to denote which product occurrence inside the FileStructure is the unique root.

8.3.5 *PRC_TYPE_ASM_FileStructureGlobals*

8.3.5.1 General

This type gathers global data of a file structure as described in Section 6.2.

Table 39 — *PRC_TYPE_ASM_FileStructureGlobals*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_ASM_FileStructureGlobals*
Base	*ContentPRCBase*	(Required) Base information associated with the entity
file_count	*UnsignedInteger*	(Required) Number of referenced FileStructures
unique_ids	Array <*CompressedUniqueId*> [**file_count**]	(Required) Unique ids for FileStructures within the PRC File which are referenced by entities in this FileStructure
global_data	*FileStructureInternalGlobalData*	(Required)
user_data	*UserData*	(Required) User defined data

8.3.5.2 *FileStructureInternalGlobalData*

8.3.5.2.1 General

This internal structure is used in ***PRC_TYPE_Asm_FileStructureGlobals***.

Table 40 — *FileStructureInternalGlobalData*

Name	Data Type	Data Description
tess_chord	*Double*	(Required) Tessellation chord height
tess_angle	*Double*	(Required) Tessellation angle (degrees)
serialize_help	*MarkupSerializationHelper*	(Required) See below
color_count	*UnsignedInteger*	(Required) Number of colors
Colors	Array <*RgbColor*>[**color_count**]	(Required) Array of color definitions
picture_count	*UnsignedInteger*	(Required) Number of pictures
Pictures	Array <*PRC_TYPE_GRAPH_Picture*>[**picture_count**]	(Required) Array of pictures
texture_count	*UnsignedInteger*	(Required) Number of textures definitions
Textures	Array <*PRC_TYPE_GRAPH_TextureDefinition*> [**texture_count**]	(Required) Array of texture definitions
material_count	*UnsignedInteger*	(Required) Number of materials
Materials	Array <*PRC_TYPE_GRAPH_Material*> [**material_count**]	(Required) Array of materials
line_pattern_count	*UnsignedInteger*	(Required) Number of line patterns
line_patterns	Array <*PRC_TYPE_GRAPH_LinePattern*> [**line_pattern_count**]	(Required) Array of line patterns
style_count	*UnsignedInteger*	(Required) Number of styles
Styles	Array <*PRC_TYPE_GRAPH_Style*>[**style_count**]	(Required) Array of category 1 line styles
fill_count	*UnsignedInteger*	(Required) Number of fill patterns
Fills	Array <*PRC_TYPE_GRAPH_FillPattern*>[**fill_count**]	(Required) Array of fill patterns
ref_coord_count	*UnsignedInteger*	(Required) Number of reference coordinate systems
ref_coords	Array <*PRC_TYPE_RI_CoordinateSystem*> [**ref_coord_count**]	(Required) Array of reference coordinate systems

8.3.5.2.2 *MarkupSerializationHelper*

8.3.5.2.2.1 General

This Global data is composed of font information for markup. The following example shows how the global data is serialized for markup font information.

Table 41 — *MarkupSerializationHelper*

Name	Data Type	Data Description
default_font_family_name	*String*	(Required) **default_font_family_name** defines the case-sensitive default font family name used for text if a given font family is not available on the computer. If this default font family itself is not available on the computer, the font will be MyriadPro from Adobe Systems, Inc.
font_keys_count	*UnsignedInteger*	(Required) Number of fonts
font_keys_of_font	Array <*FontKeysSameFont*> [**font_keys_count**]	(Required) **font_keys_of_font** represents several font keys sharing the same base font name. Indices for font keys used in *ContentMarkupTess* are calculated from this array. For example, if there are two font names representing 4 and 5 font keys respectively, index #7 would be represented by font_keys[1][3].

8.3.5.2.3 *FontKeysSameFont*

This type describes a list of usages of the same font (referred to by its name) with different metrics and attributes.

— **attributes** represents the font attributes, and is a combination of the values in the table below.

Table 42 — *FontKeySameFont*

Name	Data Type	Data Description
font_name	*String*	(Required) Font Name
character_set	*UnsignedInteger*	(Required) Character Set
key_count	*UnsignedInteger*	(Required) Number-of-font-keys
font_key_list	Array<*FontKey*> [**key_count**]	(Required)

Table 43 — *FontKey*

Name	Data Type	Data Description
font_size	*UnsignedInteger*	(Required) font size + 1
font_attributes	*Character*	(Required) Font Attributes

The following table contains the possible values for the **Character Set**.

Table 44 — Character Set

Value	Description
0	Roman
1	Japanese
2	Traditional Chinese
3	Korean
4	Arabic
5	Hebrew
6	Greek
7	Cyrillic
8	RightLeft
9	Devanagari
10	Gurmukhi
11	Gujarati
12	Oriya
13	Bengali
14	Tamil
15	Telugu
16	Kannada
17	Malayalam
18	Sinhalese
19	Burmese
20	Khmer
21	Thai
22	Laotian
23	Georgian
24	Armenian
25	Simplified Chinese
26	Tibetan
27	Mongolian
28	Geez
29	EastEuropeanRoman
30	Vietnamese
31	ExtendedArabic

font_attributes represents the font attributes, and is a combination of the following values.

Table 45 — Font attributes

Font attribute value	Description
2	Bold
4	Italic
8	Underlined
16	Strike-Out
32	Overlined
64	Stretch. In case the font to be used is not the original font, this attribute value indicates that the text shall be stretched to fit within its bounding box
128	Wire. This attribute value indicates that the original font is a wireframe font.

8.3.5.2.4 RgbColor

Color definition with 3 components.

Table 46 — *RgbColor*

Name	Data Type	Data Description
Red	*Double*	(Required) Red
Green	*Double*	(Required) Green
Blue	*Double*	(Required) Blue

8.3.6 *PRC_TYPE_ASM_FileStructureTree*

Each FileStructure within a PRC file has a FileStructureTree which defines

— the number of parts and part data

— the number of product occurrences and product occurence data

within this FileStructure.

By convention, the data within PRC Files are ordered so that data is defined before it is referenced. In the case of a FileStructureTree, part data is defined before product occurrence data which may refer to it, but within the array of part data, the order of the parts is immaterial. This is not the case for product occurrence data.

Product occurrence data represent an assembly, with a single product occurrence being the root. Each product occurrence (except the root) may be referenced by only one product occurrence. However, each product occurrence may refer to multiple product occurrences. The array of product occurence data should be ordered so that any product occurrence is defined before the product occurrence referring to it.

The FileStructureInternalData defines the index of the root product occurrence and the next available index available to use when assigning unique index (identifiers) to referenceable entities within the FileStructure.

Table 47 — *PRC_TYPE_ASM_FileStructureTree*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_ASM_FileStructureTree*
Base	*ContentPRCBase*	(Required) Base information associated with the entity
part_count	*UnsignedInteger*	(Required) Number of part definitions
Parts	Array <*PRC_TYPE_ASM_PartDefinition*>[**part_count**]	(Required) An array of part definitions
product_count	*UnsignedInteger*	(Required) Number of product occurrences
Products	Array <*PRC_TYPE_ASM_ProductOccurrence*>[**product_count**]	(Required) An array of product occurrences
internal_data	*PRC_TYPE_ASM_FileStructure*	(Required) FileStructureInternalData
user_data	*UserData*	(Required) User defined data

8.3.7 *PRC_TYPE_ASM_FileStructureTessellation*

This type gathers tessellation data of a file structure (See 6.2).

Table 48 — *PRC_TYPE_ASM_FileStructureTessellation*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_ASM_FileStructureTesssellation
base	*ContentPRCBase*	(Required) Base information associated with the entity
tess_count	*UnsignedInteger*	(Required) Number_of_tessellations
tess	Array <*PRC_TYPE_TESS*>[**tess_count**]	(Required) Content of all tessellations
user_data	*UserData*	(Required) User defined data

8.3.8 *PRC_TYPE_ASM_FileStructureGeometry*

8.3.8.1 General

This type gathers geometry data of a file structure (See 6.2).

Table 49 — *PRC_TYPE_ASM_FileStructureGeometry*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_ASM_FileStructureGeometry*
Base	*ContentPRCBase*	(Required) Base information associated with the entity
exact_geometry	*FileStructureExactGeometry*	(Required) Topological context entities and their associated brep bodies
user_data	*UserData*	(Required) User defined data

8.3.8.2 *FileStructureExactGeometry*

The *FileStructureExactGeometry* section consists of an array of "topological contexts". Each "topological context" contains an array of brep bodies contained within that topological context. Every geometrical and topological entity within a FileStructure may only belong to a single topological context.

A pair of indices (topological context, brep body) uniquely identifies a brep body within the topolological entities of a FileStructure.

Table 50 — *FileStructureExactGeometry*

Name	Data Type	Data Description
topo_context_count	*UnsignedInteger*	(Required) Number of topological contexts
topo_contexts	Array <*TopologicalContext*>[topo_context_count]	(Required) Array of topological contexts together with its associated brep bodies

8.3.9 *PRC_TYPE_ASM_FileStructureExtraGeometry*

8.3.9.1 General

The extra geometry data is summary data pertaining to the geometry which can be used to enable partial loading of the file structure without loading the entire geometry. This type gathers summary information of the exact geometry section, by topological context.

Table 51 — *PRC_TYPE_ASM_FileStructureExtraGeometry*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_ASM_FileStructureExtraGeometry*
Base	*ContentPRCBase*	(Required) Base information associated with the entity
extra_geom_count	*UnsignedInteger*	(Required) Number of extra geometry contexts
extra_geom	Array <*ExtraGeometry*>[**extra_geom_count**]	(Required)
user_data	*UserData*	(Required) User defined data

8.3.9.2 *ExactGeometry*

This is the summary of a topological context.

Table 52 — *ExactGeometry*

Name	Data Type	Data Description
Summary	*GeometrySummary*	(Required) Summary Data
context_data	*ContextGraphics*	(Required) Graphics Context Data

8.3.9.2.1 *GeometrySummary*

8.3.9.2.1.1 General

This describes the summary list of the bodies of the topological context.

The **number_of_bodies** shall be the same as the number of bodies in the context.

Table 53 — *GeometrySummary*

Name	Data Type	Data Description
number_of_bodies	*UnsignedInteger*	(Required) The **number_of_bodies** shall be the same as the number of bodies in the context.
Bodies	Array <*BodyInformation*>[**number_of_bodies**]	(Required) Graphics information specific to each body

8.3.9.2.1.2 *BodyInformation*

This describes the summary information of a body.

The **body_serial_type** shall be the type of the body that is in the topological context or it shall be set to *PRC_ROOT_TYPE* if the body has no geometry. See the section 8.9.15 for details.

Table 54 — *BodyInformation*

Name	Data Type	Data Description
body_serial_type	*UnsignedInteger*	**(Required)**
tolerance	*Double*	(Optional; if **body_serial_type** is *PRC_TYPE_TOPO_BrepDataCompress* or **body_serial_type** is *PRC_TYPE_TOPO_SingleWireBodyCompress* or **body_serial_type** is *PRC_TYPE_TESS_3D_Compress*) The **tolerance** which corresponds to the compression tolerance for the corresponding entity

8.3.9.2.2 *ContextGraphics*

8.3.9.2.2.1 General

This describes the summary list of the graphical attributes of entities within the topological context.

The following loop shows how to traverse a topological context to gather *GraphicInformation*. A *GraphicInformation* is gathered as soon as there is a graphic content provided for a particular entity :

```
For (i=0; i<number_of_body; i++) {
    If (body[i] is PRC_TYPE_TOPO_BrepData) {
        For (j=0; j<body[i].number_of_connex; j++) {
            For (k=0; k<body[i].connex[j].number_of_shell; k++) {
                For (l=0; l<body[i].connex[j].shell[k].number_of_face; l++) {
                    Add_to_output(body[i].connex[j].shell[k].face[l])
                }
            }
        }
    }
}
```

number_of_treat_type corresponds to the number of entity types for which *GraphicsInformation* is stored (currently, only *PRC_TYPE_TOPO_Face* is supported, so if there are graphics on some faces, **number_of_treat_type** is 1, else it is 0).

The current graphics as explained in 8.5 is reset prior to writing/reading the context graphics.

Table 55 — *ContextGraphics*

Name	Data Type	Data Description
number_of_treat_type	*UnsignedInteger*	(Required) number_of_treat_type
treat_types	Array <*GraphicsInformation*>[**number_of_treat_type**]	(Required) Graphic information for each type

8.3.9.2.2.2 GraphicsInformation

This describes the particular graphics for one particular type of the topological context.

Table 56 — *GraphicsInformation*

Name	Data Type	Data Description
element_type	*UnsignedInteger*	(Required)
number_of_element	*UnsignedInteger*	(Required) The **number_of_element** represents the number of elements collected during a recursive search on the Context data (including duplicated elements) as shown in 8.5.
element_information	Array <*ElementInformation*>[**number_of_element**]	(Required) Graphics Information for each Element

8.3.9.2.2.3 *ElementInformation*

This describes the particular graphics for one particular element of the topological context.

Table 57 — *ElementInformation*

Name	Data Type	Data Description
has_graphics	*Boolean*	(Required) **has_graphics** is TRUE if element[i] has graphics
graphic_behavior	*ElementGraphicsBehavior*	(Optional; if **has_graphics** is TRUE) If element[i] has Graphics

8.3.9.2.2.4 *ElementGraphicsBehavior*

This describes graphics for one particular element of the topological context.

Table 58 — *ElementGraphicsBehavior*

Name	Data Type	Data Description
use_context	*Boolean*	(Required) **use_context** is TRUE if use current graphic context
biased_layer_index	*UnsignedInteger*	(Optional; if use_context is FALSE) **layer_index** + 1 where **layer_index** represents the index in the array of layers stored in *FileStructureInternalGlobalData*.
biased_index_of_line_style	*UnsignedInteger*	(Optional; if use_context is FALSE) **index_of_line_style** + 1 where **index_of_line_style** represents the index in the array of styles stored in *FileStructureInternalGlobalData*.
behavior_bit_field	Array <*UnsignedCharacter*>[2]	(Optional; if use_context is FALSE) **behavior_bit_field** **behaviour_bit_field** is an unsigned short integer (2 bytes) . See Section *ContextGraphics* for details.

8.3.10 *PRC_TYPE_ASM_ProductOccurrence*

8.3.10.1 General

A product occurrence defines an assembly tree. In the case of a single part, the product occurrence points directly to a part definition (*PRC_TYPE_ASM_PartDefinition*). In the case of a more complex assembly, a product occurrence is comprised of a list of product occurences.

A product occurrence is comprised of the following data:

— Part definition: A pointer to the corresponding part definition. It can be null.

— Product prototype: A pointer to the corresponding product occurrence prototype. It can be null.

— External data: A pointer to the corresponding external product occurrence. It can be null.

— Children: An array of pointers to the child product occurrences.

The product prototype is the product occurrence of a subassembly or part to be used in the parent assembly. This prototype acts as a template for a given product occurrence, and lets you link to information inside the subpart or assembly, such as geometry. When building assemblies from

subassemblies, the tree of children of the product occurrence is duplicated from the prototype description. For external data, the tree is only described inside the external data product occurrence. (refer to Figure 1 in Section 5.1)

When the assembly is heterogeneous (originating from different CAD systems), the link is specified through the external data rather than the prototype.

A product occurrence has at most one parent.

Table 59 — *PRC_TYPE_ASM_ProductOccurrence*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_ASM_ProductOccurrence*
Base	*PRCBaseWithGraphics*	(Required)
references_product_occurence	*ReferencesOfProductOccurrence*	(Required)
product_behavior	*Character*	(Required) **product_behavior** represents the various flags for the product. In this version of PRC, only *PRC_PRODUCT_BEHAVIOUR_SUPPRESSED* == 0x01 is used. The other flags should be set to 0.
product_information	*ProductInformation*	(Required)
has_transform	*Boolean*	(Required) TRUE if there is a transformation; else FALSE
Location	*Transformation*	(Optional; if has_transform is TRUE) **location** transforms entities in the product occurrence to its parent space.
entity_ref_count	*UnsignedInteger*	(Required) Number of references
entity_reference	Array <*PRC_TYPE_MISC_EntityReference*>[**entity_ref_count**]	(Required) entity_reference is the referenced entities with possible modifiers towards their nominal definition, which may include location, color, and visibility.

Table 59 *(continued)*

Name	Data Type	Data Description
Markups	*MarkupData*	(Required) **markups** represents the markups of this product occurrence (as opposed to part definition markups); these are grouped into annotations.
number_of_views	*UnsignedInteger*	(Required) Number of **views**
Views	Array <PRC_TYPE_MKP_View>[number_of_views]	(Required) **views** represents the views which can contain annotations or specific display parameters.
has_filter	*Boolean*	(Required) TRUE if product occurrence has **entity_filter**; else FALSE
entity_filter	*PRC_TYPE_ASM_Filter*	(Optional; if **has_filter** is TRUE) **entity_filter** is a specific filter applied when loading data from product prototypes denoting sub-assemblies.
number_of_display_filters	*UnsignedInteger*	(Required) Number of **display_filters**
display_filters	Array <*PRC_TYPE_ASM_Filter*>[**number_of_display_filters**]	(Required) **display_filters** represents the filters to use for display. Several filters can be specified, but only one can be active (see 8.3.12).
number_of_scene_parameters	*UnsignedInteger*	(Required) Number of **scene_display_parameters**
scene_display_parameters	Array <*PRC_TYPE_GRAPH_SceneDisplayParameter*[**number_of_scene_param**]	(Required) **scene_display_parameters** is reserved for future use.
user_data	*UserData*	(Required) User defined data

8.3.10.2 *ReferencesOfProductOccurrence*

8.3.10.2.1 General

For a given file structure, the product occurrences should be ordered based on the following criteria:

— A product prototype in the same file structure should be stored before any occurrences that use it.

— External data in the same file structure should be stored before any occurrences that use it.

— A child occurrence should be stored before its parent.

— A part definition can be referenced several times in the same file structure.

(See to Figure 1 in Section 5.1.)

Table 60 — *ReferencesOfProductOccurrence*

Name	Data Type	Data Description
biased_index_part	*UnsignedInteger*	(Required) **index_part** + 1 where **index_part** represents the index of the part definition in the array of part definitions of the same FileStructure.
biased_index_prototype	*UnsignedInteger*	(Required) **index_prototype** + 1 where **index_prototype** represents the index of the product prototype in the FileStructure product occurrences array.
prototype_in_same_file_structure	*FileIdentifier*	(Optional; if **index_prototype** is NOT -1) **prototype_in_same_file_structure** indicates whether the prototype is in the same *FileStructure*.
biased_index_external_data	*UnsignedInteger*	(Required) **index_external_data** + 1 where **index_external_data** represents the index of the external data.
external_data_in_same_file_structure	*FileIdentifier*	(Optional; if **index_external_data** is NOT -1) **external_data_in_same_file_structure** indicates whether the external data is in the same *FileStructure*.
number_of_child_product_occurrences	*UnsignedInteger*	(Required) Number of child product occurrences
index_child_occurrence	Array <*UnsignedInteger*>[**number_of_child_product_occurrences**]	(Required) **index_child_occurrence**, which is mandatory in the same file structure, represents the index of the child product occurrence.

8.3.10.2.2 *FileIdentifier*

This is the identifier of the FileStructure that the prototype or external data if it is different from the FileStructure that the product occurrence lies in.

Table 61 —*FileIdentifier*

Name	Data Type	Data Description
Flag	*Boolean*	(Required) **flag** is TRUE if the entity exists in the same *FileStructure*;
unique_id	*CompressedUniqueId*	(Optional; if flag is FALSE) the identifier of the FileStructure that the entity lies in

8.3.10.3 *ProductInformation*

8.3.10.3.1 General

This is used to save general information associated with a product occurrence.

Table 62 — *ProductInformation*

Name	Data Type	Data Description
unit_from_CAD_file	*Boolean*	(Required) **unit_from_CAD_file** indicates whether the unit is read from the native CAD file.
Unit	*Double*	(Required) **unit** represents the units in mm.
product_information_flags	*Character*	(Required) **product_information_flags** is described in *PRCProductFlag*.
product_load_status	*Integer*	(Required) **product_load_status** is described in *EPRCProductLoadStatus*.

8.3.10.3.2 *PRCProductFlag*

These flags represent characteristics of product occurrences.

A product occurrence can be:

— Default: The product occurrence is the default container, configuration, or view. This means that it is loaded by default in the originating CAD system.

— Internal: when used as a prototype of another product occurrence, this product occurrence does not come from a different physical file. Hence it should belong to the same file structure.

— Container: The product occurrence acts as a repository of child occurrences that do not necessarily have relationships between them. This is useful for situations where a single CAD file can correspond to a whole database of parts and assemblies.

— Configuration: This is a specific arrangement of a product with respect to its whole hierarchy. Some parts may differ or are in a different position. For example, consider the case of an automobile where the steering wheel may be either on the left or right side.

— View: A product occurrence which is a view refers to another product occurrence (its prototype) to denote a particular setting of visibilities and position within the same hierarchy.

If none of these flags is specified, a product occurrence is referred to as regular. If the product occurrence has no parent, it is similar to a configuration. A product occurrence with no parent leads to a different FileStructure, unless it is internal, meaning that it represents a part, subassembly, or assembly

hierarchy inside the same FileStructure. If the flag is PRC_PRODUCT_FLAG_REGULAR or PRC_PRODUCT_FLAG_CONTAINER all other flags are ignored. Otherwise, the flags are legal in any combination.

Table 63 — *PRCProductFlag*

Value	Type Name
0x00	PRC_PRODUCT_FLAG_REGULAR
0x01	PRC_PRODUCT_FLAG_DEFAULT
0x02	PRC_PRODUCT_FLAG_INTERNAL
0x04	PRC_PRODUCT_FLAG_CONTAINER
0x08	PRC_PRODUCT_FLAG_CONFIG
0x10	PRC_PRODUCT_FLAG_VIEW

8.3.10.3.3 *EPRCProductLoadStatus*

This represents the status of a loading of a product occurence that may exist in a different file. This status represents the availability of that external resource at the time the PRC file was created.

Table 64 — *EPRCProductLoadStatus*

Value	Type Name	Description
0	KEPRCProductLoadStatus_Error	Unknown status
1	KEPRCProductLoadStatus_NotLoaded	Loading error. For example, there is a missing file
2	KEPRCProductLoadStatus_NotLoadable	Not loadable. For example, something prevents the file from being loaded.
3	KEPRCProductLoadStatus_Loaded	The product was successfully loaded

8.3.10.4 *MarkupData*

This is all the data that are related to Markup for a part or a product.

Table 65 — *MarkupData*

Name	Data Type	Data Description
number_of_linked_items	*UnsignedInteger*	(Required)
linked_items	Array <*PRC_TYPE_MISC_MarkupLinkedItem*>[**number_of_lnked_items**]	(Required) array of linked items
number_of_leaders	*UnsignedInteger*	(Required)
Leaders	Array <*PRC_TYPE_MKP_Leader*>[**number_of_leaders**]	(Required) array of leaders
number_of_markups	*UnsignedInteger*	(Required)

Table 65 *(continued)*

Markups	ArrayOf[*PRC_TYPE_MKP_Markup*]	(Required) array of markups
number_of_annotation_entities	*UnsignedInteger*	(Required)
annotation_entities	Array <*AnnotationEntities*>[**number_of_annotation_entities**]	(Required) array of annotation entities

8.3.10.5 *AnnotationEntities*

An annotation entity can be a:

PRC_TYPE_MKP_AnnotationItem

PRC_TYPE_MKP_AnnotationSet

PRC_TYPE_MKP_AnnotationReference

8.3.11 *PRC_TYPE_ASM_PartDefinition*

This represents a part definition.

A part consists of:

— A Bounding_box

— **views** represents the views which can contain annotations or specific display parameters.

Table 66 — *PRC_TYPE_ASM_PartDefinition*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_ASM_PartDefinition*
Base	*PRC_TYPE_ROOT_PRCBaseWithGraphics*	(Required)
bounding_box	*BoundingBox*	(Required)
number_of_representation_items	*UnsignedInteger*	(Required)
representation_items	Array <*PRC_TYPE_RI*>[**number_of_representation_items**]	(Required) A collection of visible **representation_items** containing geometrical data;

Table 66 *(continued)*

Name	Data Type	Data Description
Markups	*MarkupData*	(Required) **markups** representing the markups of this part definition (as opposed to product occurrence markups); these are grouped into annotations.
number_of_views	*UnsignedInteger*	(Required) Number of views
Views	Array <*PRC_TYPE_MKP_View*>[**number_of_views**]	(Required) Array of **views** data
user_data	*UserData*	(Required) User defined data

8.3.12 *PRC_TYPE_ASM_Filter*

8.3.12.1 General

This entity specifies the filtering between parts and assemblies. A filter denotes the particular usage of a subpart or product occurrence within a more complex one. It has the following purposes:

— To represent only those items that are of interest in the complex assembly.

— To configure the display accordingly.

Table 67 — *PRC_TYPE_ASM_Filter*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_ASM_Filter
Base	*ContentPRCBase*	(Required) Base information associated with the entity
is_active	*Boolean*	(Required) **is_active**: indicates whether this filter corresponds to the active layout when loading the file.
layer_filter	*ContentLayerFilterItems*	(Required) Layer filter
entity_filter	*ContentEntityFilterItems*	(Required) Entity filter
user_data	*UserData*	(Required) User defined data

8.3.12.2 *ContentLayerFilterItems*

This saves information for filtering of entities by layer: only entities having certain layer specifications will keep the show status they have without the filter. All other entities will be set to no show.

Table 68 — *ContentLayerFilterItems*

Name	Data Type	Data Description
b_is_inclusive	*Boolean*	(Required) **b_is_inclusive** indicates whether the elements inside the filter shall be retained.
number_of_layers	*UnsignedInteger*	(Required) Number of layers
Layers	Array <*UnsignedInteger*>[**number_of_layers**]	(Required) Layer index

8.3.12.3 *ContentEntityFilterItems*

This saves information for a filtering directly by entities: only entities referred to in the array will keep the show status they have without the filter. All other entities will be set to no show.

Table 69 — *ContentEntityFilterItems*

Name	Data Type	Data Description
b_is_inclusive	*Boolean*	(Required) **b_is_inclusive** indicates whether the elements inside the filter shall be retained.
number_of_entities	*UnsignedInteger*	(Required) Number of entities
Entities	Array <PRC_TYPE_MISC_EntityReference> [number_of_entities]	(Required) Basic entity information

8.4 Miscellaneous Data

8.4.1 Entity Types

This section gathers types allowing for entities' referencing and positioning.

Table 70 — Miscellaneous data entity types

Type Name	Type Value	Referenceable
PRC_TYPE_MISC	*PRC_TYPE_ROOT* + 200	no
PRC_TYPE_MISC_Attribute	*PRC_TYPE_ MISC + 1*	No
PRC_TYPE_MISC_CartesianTransformation	*PRC_TYPE_ MISC + 2*	No
PRC_TYPE_MISC_EntityReference	*PRC_TYPE_ MISC + 3*	No
PRC_TYPE_MISC_MarkupLinkedItem	*PRC_TYPE_ MISC + 4*	No
PRC_TYPE_MISC_ReferenceOnPRCBase	*PRC_TYPE_ MISC + 5*	No
PRC_TYPE_MISC_ReferenceOnTopology	*PRC_TYPE_ MISC + 6*	No
PRC_TYPE_MISC_GeneralTransformation	*PRC_TYPE_ MISC + 7*	No

8.4.2 *PRC_TYPE_MISC*

This is the base type for *PRC_TYPE_MISC* entity types.

8.4.3 *PRC_TYPE_MISC_Attribute*

8.4.3.1 General

This represents the storage of an attribute which has a single title and a variable number of key/value pairs.

For example, an attribute might contain the coordinates of the center of gravity, which would be represented by an attribute with title "Center of Gravity" and three key/value pairs (X =, 5), (Y =, 10) and (Z =, 20).

Table 71 — *PRC_TYPE_MISC_Attribute*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_MISC_Attribute
attribute_title	*AttributeEntry*	(Required) Attribute title
number_of_attributes	*UnsignedInteger*	(Required) Number of attribute Key/Value pairs
attributes	Array <*Attribute Key/Value*>[**number_of_attributes**]	(Required) Array of Key/Value pairs

8.4.3.2 *AttributeEntry*

This represents the storage of an attribute title represented by either a string or an integer containing a predefined string.

The following are valid integer titles and their predefined character strings:

Table 72 — *PRC_TYPE_MISC_Attribute* AttributeEntry

Integer Value	Title
2	Title
3	Subject
4	Author
5	Keywords
6	Comments
7	Template
8	Last Saved By
9	Revision Number
10	Total Editing Time
11	Last Printed
12	Create Time/Date
13	Last saved Time/Date
14	Number of Pages
15	Number of Words

Table 72 *(continued)*

Integer Value	Title
16	Number of Characters
17	Thumbnail
18	Name of Creating Application
19	Security

Attributes with a title beginning with either "__PRC_RESERVED_ATTRIBUTE" or "__PRC_EXTERNAL_ATTRIBUTE" can be used in the same way as UserData as discussed in section 5.5. They are considered as conveying proprietary information which should not be interpreted by a conforming PRC File Reader. This proprietary information is to be interpreted together with the application UUID of the FileStructure the attribute belongs to, unless the first 4 key/value pairs of the attribute are:

— "__PRC_APPLICATION_UUID_1" integer,

— "__PRC_APPLICATION_UUID_2" integer,

— "__PRC_APPLICATION_UUID_3" integer,

— "__PRC_APPLICATION_UUID_4" integer

which indicates an alternate application UUID for the data to be interpreted.

Table 73 — *AttributeEntry*

Name	Data Type	Data Description
flag	*Boolean*	(Required) **flag** is TRUE if tiltle of the attribute is an integer else **flag** is FALSE
integer_title	*UnsignedInteger*	(Optional; if **flag** is TRUE) Title is an integer
string_title	*String*	(Optional; if **flag** is FALSE) Title is a string

8.4.3.3 *AttributeKey/Value*

PRC allows for six different kinds of attribute data, represented by the following key:

— 0 represents an invalid type

— 1 represents a 32 bit integer

— 2 represents a floating point

— 3 represents a 32 bit integer interpreted as seconds since midnight UTC, January 1st 1970, not counting leap seconds.

— 4 represents a UTF-8 character string

— 5 represents a 64 bit integer interpreted as seconds since midnight UTC, January 1st 1970, not counting leap seconds.

Table 74 — *AttributeKey/Value*

Name	Data Type	Data Description
title	*AttributeEntry*	(Required) Title of Key/Value Pair
type	*UnsignedInteger*	(Required) Key determines what of 4 legal types of attributes is to follow
value	*Integer*	(Optional; if **type** is 1)
value	*Double*	(Optional; if **type** is 2)
value	*Integer*	(Optional; if **type** is 3)
value	*String*	(Optional; if **type** is 4)
value_msp	*Integer*	(Optional; if **type** is 5) most significant part of the 64 bit value
value_lsp	*UnsignedInteger*	(Optional; if **type** is 5) least significant part of the 64 bit value. The 64 bit value is **value_msp** <<32 + **value_lsp**

8.4.4 *PRC_TYPE_MISC_CartesianTransformation*

This represents a 3D transformation. Only the following flags are acceptable in defining a *PRC_TYPE_MISC_CartesianTransformation*:

Table 75 — *PRC_TYPE_MISC_CartesianTransformation_name*

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x04	*PRC_TRANSFORMATION_Mirror*	Mirror
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale
0x10	*PRC_TRANSFORMATION_NonUniformScale*	Non uniform scale

Table 76 — *PRC_TYPE_MISC_CartesianTransformation*

Name	Data Type	Data Description
name	*UnsignedInteger*	(Required) PRC_TYPE_MISC_CartesianTransformation_name
transform	*Transformation*	(Required) Data defining the cartesian transformation which is limited to the above table

8.4.5 *PRC_TYPE_MISC_EntityReference*

This general type can be used to reference any referenceable entity. The data stored in the reference may include a line style, visibility, position or other property, and can be used to overwrite properties of the referenced entity.

Table 77 — *PRC_TYPE_MISC_EntityReference*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_MISC_EntityReference
content_entity_reference	*ContentEntityReference*	(Required)
user_data	*UserData*	(Required) User defined data

8.4.6 *PRC_TYPE_MISC_MarkupLinkedItem*

8.4.6.1 General

This is used to establish a cross reference between markup and geometry. It contains a reference to the geometry in a PRC File as well as a reference to the product occurrence to which the given instance of the geometry belongs.

For example, consider the case of a distance dimension between a part contained in two product occurrences (assemblies). The dimension will have two MarkupLinkedItems, the first pointing to the first product occurrence and referencing the part, the second pointing to the second product occurrence and referencing the part as well.

Table 78 — *PRC_TYPE_MISC_MarkupLinkedItem*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_MISC_MarkupLinkedItem
content_entity_reference	*ContentExtendedEntityReference*	(Required) Reference to a remote product occurrence
show_markup	*Boolean*	(Required) If TRUE, show/hide markup when showing/hiding the referenced entity
delete_markup	*Boolean*	(Required) If TRUE, delete markup when deleting the referenced entity
show_leader	*Boolean*	(Required) If TRUE, show the leader when showing/hiding the referenced entity
delete_leader	*Boolean*	(Required) If TRUE, delete the leader when deleting the referenced entity
user_data	*UserData*	(Required) User defined data

8.4.6.2 *ContentExtendedEntityReference*

Stores data to reference entities in remote product occurrences.

Table 79 — *ContentExtendedEntityReference*

Name	Data Type	Data Description
content_entity_reference	*ContentEntityReference*	(Required) Reference to a remote product occurrence
reference_data	*ReferenceData*	(Required) Reference data for the entity

8.4.7 *PRC_TYPE_MISC_ReferenceOnPRCBase*

This describes a reference to a referenceable entity. Referenceable entity types are a subset of the PRC base entities (see section 8.2.1 for the specific subset). Referenceable topological entities are handled separately (see 8.4.8).

A reference to an entity consists of the UUID of the FileStructure the referenced entity lies in (if different from the FileStructure this entity is in) and the index of the referenced entity within this FileStructure.

Table 80 — *PRC_TYPE_MISC_ReferenceOnPRCBase*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_MISC_ReferenceOnPRCBase
type_of_entity	*UnsignedInteger*	(Required) This is the type of the target entity
flag	*Boolean*	(Required) TRUE if this reference is to an entity in same File Structure as this entity exists in else FALSE
different_unique_id	*CompressedUniqueID*	(Optional if **flag** is FALSE) Unique identifier of target FileStructure if different from this FileStructure; See Section **Error! Reference source not found.** for details
unique_id	*UnsignedInteger*	(Required) Unique identifer within target File Structure

8.4.8 *PRC_TYPE_MISC_ReferenceOnTopology*

8.4.8.1 General

This describes a reference to a topological entity.

The following describe the data needed to locate the target entity:

— The **type** of topological entity being referenced shall be one of those in ReferenceOnTopology Entities **(**See 8.4.8.3**)**

— If the target entity has a body in the Exact Geometry Section of the target FileStructure, additional information is required to locate the target entity.

Table 81 — *PRC_TYPE_MISC_ReferenceOnTopology*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MISC_ReferenceOnTopology*
type	*UnsignedInteger*	(Required) **type** of the topolocial entity being referenced
flag	*Boolean*	(Required) **flag** is TRUE if the target entity has a body in the Exact Geometry Section of the target FileStructure
data	*AdditionalTargetData*	(Optional; if **flag** is TRUE) Data defining the reference to the target entity

8.4.8.2 *AdditionalTargetData*

This is used only within *PRC_TYPE_MISC_ReferenceOnTopology*.

The unique identifier of the target FileStructure is stored here if it is different from the FileStructure of the entity currently being read or written.

To locate the target entity within the target FileStructure requires the following:

— The index of the topological context of the target entity.

— The index of the body within the topological context.

— In addition, most topological entities need other indices within the body to identify themselves. The number of additional indices and an array of index values indicate other index values that are needed to uniquely identify the target entity. At present, the only topological entity which may be referenced is a *PRC_TYPE_TOPO_Face*. For this, the requisite data would be:

  — Number of additional indices is 1

  — The array would consist of one Unsigned Integer which would be the index of the face within the body.

Table 82 — *AdditionalTargetData*

Name	Data Type	Data Description
flag	*Boolean*	(Required) TRUE if the target entity is in the same FileStructure as this entity
unique_id	*CompressedUniqueId*	(Optional; if **flag** is FALSE) Unique identifier of the FileStructure the target entity lies in
index_of_topological_index	*UnsignedInteger*	(Required) Index of the topological context of the target entity within the FileStructure
index_of_body	*UnsignedInteger*	(Required) Index of the body within the topological context of the target entity
number_of_indices	*UnsignedInteger*	(Required) Number of additional indices needed to locate the target entity
indices	Array <*UnsignedInteger*>[**number_of_indices**]	(Required) Array of additional indices

8.4.8.3 ReferenceOnTopology Entities

The only topological entity which may be referenced is a *PRC_TYPE_TOPO_Face*.

The follow topological entities may be referenced in future versions:

— *PRC_TYPE_TOPO_MultipleVertex*

— *PRC_TYPE_TOPO_UniqueVertex*

— *PRC_TYPE_TOPO_WireEdge*

— *PRC_TYPE_TOPO_Edge*

— *PRC_TYPE_TOPO_Loop*

— *PRC_TYPE_TOPO_Shell*

— *PRC_TYPE_TOPO_Connex*

8.4.9 *PRC_TYPE_MISC_GeneralTransformation*

This is a general 3D transformation consisting of the sixteen coordinates of a 4x4 matrix.

To use a 4x4 matrix to convert a 3D position of vector, one pre multiplies by the matrix, that is,

New_3D_PointOrVector ← matrix * Old_3D_PointOrVector

The coefficients are stored in the following order:

Matrix (First number is row, second number is column). For example, translation is represented by Tx=M[0][3], Ty = M[1][3], Tz=M[2][3]

M[0][0] M[0][1] M[0][2] M[0][3]

M[1][0] M[1][1] M[1][2] M[1][3]

M[2][0] M[2][1] M[2][2] M[2][3]

M[3][0] M[3][1] M[3][2] M[3][3]

Storage order:

M[0][0]

M[1][0]

M[2][0]

M[3][0]

M[0][1]

M[1][1]

....

M[1][3]

M[2][3]

M[3][3]

Table 83 — *PRC_TYPE_MISC_GeneralTransformation*

Name	Data Type	Data Description
	UnsignedInteger	*(Required) PRC_TYPE_MISC_GeneralTransformation*
general_transform	Array <*Double*>[16]	(Required) 16 Coefficients of transformation

8.4.10 *ContentEntityReference*

8.4.10.1 General

This represents the data defining a reference to any referenceable entity.

The **index_of_local_coordinate_system** may be -1 indicating no local coordinate system is present. Otherwise, the value given is the index into the array of reference coordinate systems defined in 8.3.5.2.

If the referenced entity does not exist, no futher information should be stored. If the reference does exist, data describing the unique identifier of the referenced entity will be present in the PRC File.

Table 84 — *ContentEntityReference*

Name	Data Type	Data Description
base	*PRCBaseWithGraphics*	(Required)
index_of_local_coordinate	*UnsignedInteger*	(Required) Index_of_local_coordinate_system or -1 if none present
flag	*Boolean*	(Required) TRUE if the referenced entity exists (i.e. is not NULL)
reference_data	*ReferenceData*	(Optional; if **flag** is TRUE) Define the unique identifier of the reference entity.

8.4.10.2 *ReferenceData*

PRC_TYPE_MISC_ReferenceOnTopology should be used to reference an entity, whenever referencing a referencable topological entity. Any other value is an error.

Table 85 — *ReferenceData*

Name	Data Type	Data Description
topo_reference	*PRC_TYPE_MISC_ReferenceOnTopology*	(Optional if reference is to a referenceable topological entity)
non_topo_reference	*PRC_TYPE_MISC_ReferenceOnPRCBase*	(Optional; if reference is to a non-topological entity)

8.4.11 *Transformation*

The Transformation associated with an entity is defined as either 2D or 3D depending upon the dimension of the entity containing the transformation.

NOTE

— A cartesian transformation that is used to represent a 3D cartesian transformation shall be orthogonal and shall not have homogeneous values. Non-orthogonal or homogeneous transformations are used only for the transformations used in textures.

— *PRC_TYPE_MISC_GeneralTransformation* is a transformation but only matrix coefficients are stored, as described in section 8.4.9.

The Transformation is defined by its behavior which can be any combination (except as noted above) of

Table 86 — *Transformation type names*

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x04	*PRC_TRANSFORMATION_Mirror*	Mirror
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale
0x10	*PRC_TRANSFORMATION_NonUniformScale*	Non uniform scale
0x20	*PRC_TRANSFORMATION_NonOrtho*	Non orthogonal
0x40	*PRC_TRANSFORMATION_Homogeneous*	Homogeneous

A 3D transformation is defined by a 4x4 matrix. By default, this matrix is the identity. The following table is used to determine the 3D transformation.

Table 87 — *3D Transformation*

Name	Data Type	Data Description
behavior	*Character*	(Required) **behavior** determines the type of data used to define the transformation; each bit of the behavior determines if the transformation has that data
translation	*Vector3D*	(Optional; **behavior** & PRC_TRANSFORMATION_Translate is TRUE) Define the transformation translation translate -> mat[0][3] mat[1][3] mat[2][3]
non_ortho_matrix	*Array <Vector3D>[3]*	(Optional; **behavior** & PRC_TRANSFORMATION_NonOrtho is TRUE) Define the non orthogonal matrix nonortho.xaxis -> mat[0][0] mat[1][0] mat[2][0] nonortho.yaxis -> mat[0][1] mat[1][1] mat[2][1] nonortho.zaxis -> mat[0][2] mat[1][2] mat[2][2] The vectors are unit vectors

Table 87 *(continued)*

rotation	*Array <Vector3D>[2]*	(Optional; **behavior** & PRC_TRANSFORMATION_Rotate is TRUE and **behavior** & PRC_TRANSFORMATION_NonOrtho is FALSE) Define an orthogonal matrix with 2 unit vectors, X and Y. Z is the cross product of X and Y if no mirror. Else, it is the cross product of Y and X. rotate.xaxis -> mat[0][0] mat[1][0] mat[2][0] rotate.yaxis -> mat[0][1] mat[1][1] mat[2][1] If no mirror, rotate.xaxis X rotate.yaxis -> mat[0][2] mat[1][2] mat[2][2] where X is the cross product If mirror, rotate.yaxis X rotate.xaxis -> mat[0][2] mat[1][2] mat[2][2] where X is the cross product The vectors are unit vectors
non_uniform_scale	*Vector3D*	(Optional; **behavior** & PRC_TRANSFORMATION_NonUniformScale is TRUE) Scale factor for x, y, and z component of the matrix. non_uniform_scale.x to column mat[0][0] mat[1][0] mat[2][0] non_uniform_scale.y to column mat[0][1] mat[1][1] mat[2][1] non_uniform_scale.z to column mat[0][2] mat[1][2] mat[2][2]
scale	*Double*	(Optional; **behavior** & PRC_TRANSFORMATION_Scale is TRUE and **behavior** & PRC_TRANSFORMATION_NonUniformScale is FALSE) Define the scale apply scale to the 3x3 submatrix mat[0][0] ... mat[3][3]
homogeneous	*Array <Double>[4]*	(Optional; **behavior** & PRC_TRANSFORMATION_Homogeneous is TRUE) Define the homogeneous coordinate: x, y, z, w. Homogeneous.x -> mat[3][0] Homogeneous.y -> mat[3][1] Homogeneous.z -> mat[3][2] Homogeneous.w -> mat[3][3]

A 2D transformation is defined by a 3x3 matrix. By default, this matrix is the identity. Then the following table is used to determine the 2D transformation.

Table 88 — *2D Transformation*

Name	Data Type	Data Description
behavior	*Character*	(Required) **behavior** determines the type of data used to define the transformation; each bit of the behavior determines if the transformation has that data
translation	*Vector3D*	(Optional; **behavior** & PRC_TRANSFORMATION_Translate is TRUE) Define the transformation translation translate -> mat[0][2] mat[1][2]
non_ortho_matrix	*Array <Vector3D>[3]*	(Optional; **behavior** & PRC_TRANSFORMATION_NonOrtho is TRUE) Define the non orthogonal matrix nonortho.xaxis -> mat[0][0] mat[1][0] nonortho.yaxis -> mat[0][1] mat[1][1] The vectors are unit vectors.

Table 88 *(continued)*

rotation	*Array <Vector3D>[2]*	(Optional; **behavior** & PRC_TRANSFORMATION_Rotate is TRUE and **behavior** & PRC_TRANSFORMATION_NonOrtho is FALSE) xaxis -> mat[0][0] mat[1][0] yaxis -> mat[0][1] mat[1][1] The vectors are unit vectors.
non_uniform_scale	*Vector3D*	(Optional; **behavior** & PRC_TRANSFORMATION_NonUniformScale is TRUE) Scale factor for x, y. non_uniform_scale.x to column mat[0][0] mat[1][0] non_uniform_scale.y to column mat[0][1] mat[1][1]
scale	*Double*	(Optional; **behavior** & PRC_TRANSFORMATION_Scale is TRUE and **behavior** & PRC_TRANSFORMATION_NonUniformScale is FALSE) Define the scale apply scale to the 2x2 submatrix mat[0][0] ... mat[1][1].
homogeneous	*Array <Double>[4]*	(Optional; **behavior** & PRC_TRANSFORMATION_Homogeneous is TRUE) Define the homogeneous coordinate: x, y, w. Homogeneous.x -> mat[2][0] Homogeneous.y -> mat[2][1]

8.5 Graphics

8.5.1 Entity types

Table 89 — Graphics entity types

Type Name	Type Value	Referenceable
PRC_TYPE_GRAPH	*PRC_TYPE_ROOT* + 700	
PRC_TYPE_GRAPH_Style	*PRC_TYPE_GRAPH* + 1	yes
PRC_TYPE_GRAPH_Material	*PRC_TYPE_GRAPH* + 2	yes
PRC_TYPE_GRAPH_Picture	*PRC_TYPE_GRAPH* + 3	no
PRC_TYPE_GRAPH_TextureApplication	*PRC_TYPE_GRAPH* + 11	yes
PRC_TYPE_GRAPH_TextureDefinition	*PRC_TYPE_GRAPH* + 12	yes
PRC_TYPE_GRAPH_TextureTransformation	*PRC_TYPE_GRAPH* + 13	no
PRC_TYPE_GRAPH_LinePattern	*PRC_TYPE_GRAPH* + 21	yes
PRC_TYPE_GRAPH_FillPattern	*PRC_TYPE_GRAPH* + 22	no
PRC_TYPE_GRAPH_DottingPattern	*PRC_TYPE_GRAPH* + 23	yes
PRC_TYPE_GRAPH_HatchingPattern	*PRC_TYPE_GRAPH* + 24	yes
PRC_TYPE_GRAPH_SolidPattern	*PRC_TYPE_GRAPH* + 25	yes
PRC_TYPE_GRAPH_VpicturePattern	*PRC_TYPE_GRAPH* +26	yes
PRC_TYPE_GRAPH_AmbientLight	*PRC_TYPE_GRAPH* + 31	yes
PRC_TYPE_GRAPH_PointLight	*PRC_TYPE_GRAPH* + 32	yes
PRC_TYPE_GRAPH_DirectionalLight	*PRC_TYPE_GRAPH* + 33	yes

Table 89 *(continued)*

PRC_TYPE_GRAPH_SpotLight	*PRC_TYPE_GRAPH* + 34	yes
PRC_TYPE_GRAPH_SceneDisplayParameters	*PRC_TYPE_GRAPH* + 41	yes
PRC_TYPE_GRAPH_Camera	*PRC_TYPE_GRAPH* + 42	yes

8.5.2 *PRC_TYPE_GRAPH*

The abstract type for miscellaneous graphic elements not included in part geometry, topology, tesselation, or markups. Includes line and fill styles and patterns, colors, textures, pictures, lighting scenes, and camera angles. Graphic elements may be applied to other elements, such as part surfaces or markups.

8.5.3 *PRC_TYPE_GRAPH_Style*

This type contains all information used to describe the style of a line.

— **line_width** represents the line width in millimeters.

— **is_vpicture** indicates that the drawing style is a VPicture pattern instead of a line pattern. This style is to be found in the pattern array instead of the line pattern array (see *FileStructureInternalGlobalData* Section 8.3.5.2).

— **is_material** indicates that the color style is a material instead of a plain color. This style is to be found in the material array instead of the color array (see *FileStructureInternalGlobalData* Section 8.3.5.2).

— **material_index** is the index into the material array. (see *FileStructureInternalGlobalData* Section 8.3.5.2).

— **color_index** is the index into the color array. (see *FileStructureInternalGlobalData* Section 8.3.5.2).

— **transparency** values can range from 0 (transparent) to 255 (opaque).

— **rendering_parameters**

Table 90 — *PRC_TYPE_GRAPH_Style*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_GRAPH_Style
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
line_width	*Double*	(Required) **line width**
is_vpicture	*Boolean*	(Required) **is_vpicture**
biased_patern_index	*UnsignedInteger*	(Required) value is either **line_pattern_index** + 1 or **vpicture_index** + 1
is_material	*Boolean*	(Required) **is_material**
biased_color_index	*UnsignedInteger*	(Required) value is either **color_index** + 1 or **material_index** + 1

Table 90 *(continued)*

is_transparency	*Boolean*	(Required) If TRUE, **transparency** is defined
transparency	*Character*	(Optional; if **is_transparency** is TRUE) transparency
is_rendering_parameters	*Boolean*	(Required) If TRUE, **rendering_parameters** are defined
rendering_parameters	*Character*	(Optional; if **is_rendering_parameters** is TRUE) rendering_parameters
flag1	*Boolean*	(Required) Not currently used (set FALSE)
flag2	*Boolean*	(Required) Not currently used (set FALSE)

Table 91 — *PRC_TYPE_GRAPH_Style* rendering parameters

Rendering parameter	Value
special-culling strategy applies	0x01
front culling applies (ignored if no special-culling strategy)	0x02
back culling applies (ignored if no special-culling strategy)	0x04
no light applied to the corresponding object	0x08

8.5.4 *PRC_TYPE_GRAPH_Material*

This type defines basic material appearance with colors and alphas.

— **ambient_index**: index into the RGB array (see Section 8.3.5.2)

— **diffuse_index**: index into the RGB array (see Section 8.3.5.2)

— **emissive_index**: index into the RGB array (see Section 8.3.5.2)

— **specular_index**: index into the RGB array (see Section 8.3.5.2)

Table 92 — *PRC_TYPE_GRAPH_Material*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_GRAPH_Material
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_ambient_index	*UnsignedInteger*	(Required) **ambient_index** + 1
biased_difuse_index	*UnsignedInteger*	(Required) **diffuse_index** + 1
biased_emissive_index	*UnsignedInteger*	(Required) **emissive_index** + 1
biased_specular_index	*UnsignedInteger*	(Required) **specular_index** + 1
shininess	*Double*	(Required) **shininess**
ambient_alpha	*Double*	(Required) **ambient_alpha** (0.0 –> 1.0)
difuse_alpha	*Double*	(Required) **diffuse_alpha** (0.0 –> 1.0)

Table 92 *(continued)*

emissive_alpha	*Double*	(Required) **emissive_alpha** (0.0 –> 1.0)
specular_alpha	*Double*	(Required) **specular_alpha** (0.0 –> 1.0)

The definitions for **shininess**, **anbient_alpha**, **diffuse_alpha**, **emissive_alpha** , and **specular_alpha** are identical to the definitions in OpenGL.

8.5.5 *PRC_TYPE_GRAPH_Picture*

8.5.5.1 General

This type is used to define pictures embedded in the file.

Table 93 — *PRC_TYPE_GRAPH_Picture*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_GRAPH_Picture
base	*ContentPRCBase*	(Required) Base information associated with the entity
format	*EPRCPictureDataFormat*	(Required) **format**
biased_uncompressed_file_index	*UnsignedInteger*	(Required) **uncompressed_file_index** + 1
pixel_width	*UnsignedInteger*	(Required) **pixel_width**
pixel_height	*UnsignedInteger*	(Required) **pixel_height**

pixel_width and **pixel_height** are the size of the picture expressed in pixels. When **format** is 0 or 1, pixel width and pixel height fields are ignored. When **format** is one of {2,3,4,5} the size of the picture buffer when uncompressed shall be at least pixel width * pixel height * number of components per pixel .

8.5.5.2 *EPRCPictureDataFormat*

This object is used for the format of the Picture.

Table 94 — *EPRCPictureDataFormat*

Value	Type Name	Type Description
0	*KEPRCPicture_PNG*	PNG format buffer
1	*KEPRCPicture_JPG*	JPEG format buffer
2	*KEPRCPicture_BITMAP_RGB_BYTE*	flate-formatted pixel data. Each element is an RGB triplet (3 components).
3	*KEPRCPicture_BITMAP_RGBA_BYTE*	flate-formatted pixel data. Each element is an RGBA triplet (4 components).
4	*KEPRCPicture_BITMAP_GREY_BYTE*	flate-formatted pixel data. Each element is a single luminance value (1 component).
5	*KEPRCPicture_BITMAP_GREYA_BYTE*	flate-formatted pixel data. Each element is a luminance/alpha pair (2 components).

8.5.6 *PRC_TYPE_GRAPH_TextureApplication*

This type contains a definition of the complete texture pipe (multiple texturing) to be applied.

— **material_generic_index** represents an index in the material array (see *FileStructureInternalGlobalData* Section 8.3.5.2). This index should correspond to a *PRC_TYPE_GRAPH_Material*, which defines the basic material parameters of the texture.

— **texture_definition_index** represents an index in the texture definition array (see Section 8.3.5.2).

— **next_texture_index** represents an index in the material array (see Section 8.3.5.2). This index should correspond to a *PRC_TYPE_GRAPH_TextureApplication*, which is used as the next level of texture in multiple texturing. This index is set to -1 if it is the last level of texture.

— **uv_coordinates_index** represents the texture mapping coordinates index (see *PRC_TYPE_TESS_FACE* section 8.8.6 and below).

Table 95 — *PRC_TYPE_GRAPH_TextureApplication*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_TextureApplication*
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_material_generic_index	*UnsignedInteger*	(Required) **material_generic_index** + 1
biased_texture_definition_index	*UnsignedInteger*	(Required) **texture_definition_index** + 1
biased_next-texture_index	*UnsignedInteger*	(Required) **next_texture_index** + 1
biased_uv_coordinates_index	*UnsignedInteger*	(Required) **uv_coordinates_index** + 1

uv_coordinates_index denotes the set of UV coordinates to consider in the *PRC_TYPE_TESS_Face* for textured entities, as there might be several UV coordinates for each point.

See: **number_of_texture_coordinate_indexes** in *PRC_TYPE_TESS_Face*.

For example, a simple triangle with TWO texture coordinates index is described by (normal,{texture1,texture2},point,

normal, {texture1,texture2},point,

normal, {texture1,texture2},point).

UV_coordinate_index indicates which of texture1 or texture2 should be used.

8.5.7 *PRC_TYPE_GRAPH_TextureDefinition*

This type contains a single set of texture parameters to be used in a TextureApplication.

A definition for the unique variables follows:

— **picture_index** represents the index in the picture array (see *FileStructureInternalGlobalData* Section 8.3.5.2).

— **texture_dimension** represents the dimension of the image. It's possible values are 1, 2, and 3 (1 and 3 are reserved for future use).

— **texture_mapping_attributes** is a bit field that represents the procedure used to apply the texture (see texture mapping attributes table below). This information can be combined with additional information, such as intensity, and involves color or alpha components.

— **size_texture_mapping_attributes_intensities** can be set either to 0 or to the number of procedures deduced from **texture_mapping_attributes**. If it is set to 0, the intensity is set to 1. Otherwise, its values should be in the range [0.0,1.0] and should correspond to each nonzero bit of **texture_mapping_attributes**, respectively. The same is true for **size_texture_mapping_attributes_components**, for which the default value is PRC_TEXTURE_MAPPING_COMPONENTS_RGBA (see texture mapping attributes table below). Multiple procedures for texture application are reserved for future use. Therefore **size_texture_mapping_attributes_intensities** and **size_texture_mapping_attributes_components** contain at most one element. If **texture_mapping_attributes** = *PRC_TEXTURE_MAPPING_DIFFUSE*, then **size_texture_mapping_attributes_intensities** = 0. For each bit of **texture_mapping_attributes** with a value of 1, intensity will be 1.0 by default. If **size_texture_mapping_attributes_components** = 0, then for each bit of **texture_mapping_attributes** with a value of 1, components will be *PRC_TEXTURE_MAPPING_COMPONENTS_RGBA* by default. Or:

  texture_mapping_attributes = PRC_TEXTURE-MAPPING_DIFFUSE

  size_texture_mapping_attributes_intensities = 1

  texture_mapping_attributes_intensities[0] = 1.0

  size_texture_mapping_attributes_components = 1

  texture_mapping_attributes_components[0] = PRC_TEXTURE_MAPPING_COMPONENTS_RGBA

— **texture_function** : see texture function table below.

— **blend_src_rgb, blend_dst_rgb**, **blend_src_alpha**, **blend_dst_alpha**; blending modes are reserved for future use.

— **texture_applying_mode** : see texture application mode table below.

— **alpha_test** : reserved for future use.

— **alpha_test_reference** : threshold value for alpha test; used in conjunction with alpha_test.

— **texture_wrapping_mode_s** : Repeating mode; U direction; see wrapping mode table below.

— **texture_wrapping_mode_t** : Repeating mode; V direction; see wrapping mode table below.

— **texture_wrapping_mode_r** : Repeating mode; W direction (for multi dimension textures) ; see wrapping mode table below.

— **texture_transformation** : optional transformation on texture coordinates.

Table 96 — *PRC_TYPE_GRAPH_TextureDefinition*

Name	Data Type	Data Description
	UnsignedInteger	(Required)

Name	Data Type	Data Description
		PRC_TYPE_GRAPH_TextureDefinition
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_picture_index	*UnsignedInteger*	(Required) **picture_index** + 1
texture_dimension	*Character*	(Required) **texture dimension** = 2 (1 and 3 are reserved for future use)
texture_mapping_type	*Integer*	(Required) texture mapping type
texture_mapping_operator	*Integer*	(Optional; if texture_mapping_type is 4) texture mapping operator
has_transformation	*Bolean*	(Required) has_transformation
transformation	*Cartesian Transformation*	(Optional; if has_transformation is TRUE) If(has_transformation != 0)
texture_mapping-attributes	*UnsignedInteger*	(Required) **texture mapping attributes**
number_of_texture_mapping_attributes_intensities	*UnsignedInteger*	(Required) number of **texture_mapping_attributes_intensities** (shall be 0 or 1)
texture_mapping_attributes_intensities	Array <*Double*> [**number_of_texture_mapping_attributs_intensities**]	(Optional; if **number_of_texture_mapping_attributes_intensities** is not 0) texture_mapping_attributes_intensities[0] = 1.0
number_of_texture_mapping _attributes_ components	*UnsignedInteger*	(Required) number of texture_mapping_attributes_ components (shall be 0 or 1)
texture_mapping _attributes_ components	Array <*Character*> [**number_of_texture_mapping _attributes_ components**]	(Optional; if **number_of_texture_mapping _attributes_ components** is not 0) texture_mapping_attributes_components[0] = 0x000F
texture_function	*Integer*	(Required) **texture_function** (reserved for future use)
blend_src	Array <*Double*>[4]	(Optional; if **texture_function** is KEPRCTextureFunctionBlend) [red, green,blue,alpha] blend color components In the range (0.0, 1.0)
blend_src_rgb	*Integer*	(Required) **blend_src_rgb** (reserved for future use)
blend_src_alpha	*Integer*	(Required) **blend_src_alpha** (reserved for future use)
texture_application_mode	*Character*	(Required) **texture_application_mode**
alpha_test	*Integer*	(Optional; if **texture_application_mode** &

Name	Data Type	Data Description
		PRC_TEXTURE_APPLYING_MODE_ALPHATEST is TRUE) **alpha_test**
Option:alpha_test_reference	*Double*	(Optional; if **texture_application_mode** & PRC_TEXTURE_APPLYING_MODE_ALPHATEST is TRUE) **alpha_test_reference**
texture_wrapping_mode	*Character*	(Required) **texture_wrapping_mode**
texture_wrapping_mode_s	*integer*	(Required) **texture_wrapping_mode_S**
texture_wrapping_mode_t	*integer*	(Optional; if texture_dimension > 1) **texture_wrapping_mode_t**
texture_wrapping_mode_r	*Integer*	(Optional; if texture_dimension>2) **texture_wrapping_mode_r**
texture_transformation	*Boolean*	(Required) **texture_transfomation**
transformation	*PRC_TYPE_GRAPH_TextureTransformation*	(Optional; if(**texture_transfomation** is TRUE) (see section PRC_TYPE_GRAPH_TextureTransformation 8.5.8)

Table 97 — Texture mapping type

Texture mapping type	Value
Let the application choose.	1
Use the mapping coordinates that are stored on a 3D tessellation object.	2
Retrieve the UV coordinates on the surface as mapping coordinates (reserved for future use).	3
Use the defined Texture mapping operator to calculate mapping coordinates (reserved for future use)	4

Table 98 — Texture mapping operator

Texture mapping operator	Integer value
Unknown (default value)	1
Planar	2
Cylindrical	3
Spherical	4
Cubic	5

Table 99 — Texture mapping attributes

Texture mapping attributes	Integer value
Red component	0x0001
Green component	0x0002
Blue component	0x0004
RGB component	0x0007
Alpha component	0x0008
RGBA component	0x000F

Table 100 — Texture function

Texture function	Integer value
Unknown - Let the application choose.	1
Modulate - Combine lighting with texturing (default value).	2
Replace the object color with texture color data.	3
Blend	4
Decal	5

Table 101 — Texture application mode

Texture application mode	Character value
Let the application choose. (All states disabled.)	0x0000
Use lighting mode.	0x0001
Use alpha test.	0x0002
Combine a texture with one-color-per-vertex mode.	0x0004

Table 102 — Texture wrapping mode

Texture wrapping mode	Integer value
Unknown - Let the application choose.	1
Repeat - Display the repeated texture on the surface.	2
ClampToBorder - Clamp the texture to the border. Display the surface color along the texture limits.	3
Clamp	4
Clamp to edge	5
Mirrored repeat	6

8.5.8 *PRC_TYPE_GRAPH_TextureTransformation*

This type contains the transformation data used in a texture definition. In the current release, texture transformations are limited to two dimensions.

Table 103 — *PRC_TYPE_GRAPH_TextureTransformation*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_TextureTransformation*
invert_s	*Boolean*	(Required) If(TRUE) the S coordinate parameter is inverted.
invert_t	*Boolean*	(Required) If(TRUE) the T coordinate parameter is inverted.
transform_2d	*Boolean*	(Required) If(TRUE) the matrix transformation contains only 2-dimensional terms. (Always TRUE in this version.)
transform	*Transformation*	(Required) 2d transformation (see section 8.4.11 Transformations)

8.5.9 *PRC_TYPE_GRAPH_LinePattern*

This type contains the information used to display the dashes and gaps that comprise a line pattern.

Table 104 — *PRC_TYPE_GRAPH_LinePattern*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_LinePattern*
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
number_of_elements	*UnsignedInteger*	(Required) number of unique dash-array elements
lengths	Array <*Double*>[**number_of_elements**]	lengths of each type of alternating dashes and gaps, length
start_offset	*Double*	(Required) the offset within the dash pattern at which to start the dash, phase
scale	*Boolean*	(Required) If scale is TRUE the pattern aspect that scales with the view.

If a pattern scales with the view, the unit of length is the same as the product occurrence it is associated with: otherwise, lengths are to be interpreted as a ratio.

8.5.10 *PRC_TYPE_GRAPH_FillPattern*

Abstract class for a two-dimensional display style. This type contains information related to a fill pattern, which can be one of the following types of patterns:

— Dotting pattern (PRC_TYPE_GRAPH_DottingPattern)

— Hatching pattern (PRC_TYPE_GRAPH_HatchingPattern)

— Solid pattern (PRC_TYPE_GRAPH_SolidPattern)

— Vectorized picture pattern (PRC_TYPE_GRAPH_VPicturePattern)

8.5.11 *PRC_TYPE_GRAPH_DottingPattern*

This type describes a two-dimensional filling pattern with points. By default, this pattern describes a regular grid of points spaced with pitch (zizag==FALSE). If zizag is true, the points are offset in X by pitch/2.0 for the odd row.

— **next_pattern_index** represents the index of the next pattern (superimposed) in the pattern array (see *FileStructureInternalGlobalData* Section 8.3.5.2).

— **color_index** represents the index into the color array (see *FileStructureInternalGlobalData* Section 8.3.5.2).

Table 105 — *PRC_TYPE_GRAPH_DottingPattern*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_DottingPattern*
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_next_pattern_index	*UnsignedInteger*	(Required) **next_pattern_index** + 1
pitch	*Double*	(Required) pitch of point spacing
is_offset	*Boolean*	(Required) If **is_offset** is **TRUE**, the points are offset in X by (pitch/2.0) for the odd row.
biased_color_index	*Integer*	(Required) **color_index** + 1

8.5.12 *PRC_TYPE_GRAPH_HatchingPattern*

This type describes a two-dimensional filling pattern with hatches. This pattern is defined by a group of infinite lines, each having its own dash pattern and color.

— **next_pattern_index** represents the index of the next pattern (superimposed) in the pattern array (see *FileStructureInternalGlobalData* Section 8.3.5.2).

Table 106 — *PRC_TYPE_GRAPH_HatchingPattern*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_HatchingPattern*
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_next_pattern_index	*Unsigned integer*	(Required) **next_pattern_index** + 1
number_of_hatching_lines	*UnsignedInteger*	(Required) number of pattern hatching lines
hatch	Array<groups of 5 Doubles and 1 Integer>[number of hatching lines]	(Required) (2 D vector **start point,** 2 D vector **end point,** Double **angle,** Index_of_line_style + 1)

8.5.13 *PRC_TYPE_GRAPH_SolidPattern*

This type defines a two-dimensional filling pattern with a particular style (color, material, texture).

— **next_pattern_index** represents the index of the next pattern (superimposed) in the pattern array (see *FileStructureInternalGlobalData* Section 8.3.5.2)

— **material_index** is the index into the material array. (see *FileStructureInternalGlobalData* Section 8.3.5.2)

— **color_index** is the index into the color array. (see *FileStructureInternalGlobalData* Section 8.3.5.2)

Table 107 — *PRC_TYPE_GRAPH_SolidPattern*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_SolidPattern*
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_next_pattern_index	*UnsignedInteger*	(Required) **next_pattern_index** + 1
is_material	*Boolean*	(Required) If **is_material** is **TRUE** then the fill is a material otherwiswise plain color.
biased_material_index	*UnsignedInteger*	(Optional; if **is_material** is TRUE) **material_index**+1
biased_color_index	*UnsignedInteger*	(Optional; if **is_material** is FALSE) **color_index**+1

8.5.14 *PRC_TYPE_GRAPH_VpicturePattern*

This type defines a two-dimensional filling pattern consisting of a vectorized picture. In this version a restricted version of *PRC_TYPE_TESS_Markup* is used. The allowed types are:

— Polyline

— Triangles

— Color

— Line Stipple

— Points

— Polygon

— Line Width

next_pattern_index represents the index of the next pattern (superimposed) in the pattern array (see *FileStructureInternalGlobalData* Section 8.3.5.2).

Table 108 — *PRC_TYPE_GRAPH_VpicturePattern*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_VpicturePattern*
base	*ContentPRCRefBase*	(Required) Base information associated with the entitiy
biased_next_pattern_index	*UnsignedInteger*>	(Required) **next_pattern_index** + 1
patern_dimensions	Array <*Double*>[2]	(Required) X and Y dimensions of the pattern one for x one for y
markup	*PRC_TYPE_TESS_Markup*	(Required) *PRC_TYPE_TESS_Markup* object (See MARKUP Section for types)

8.5.15 *PRC_TYPE_GRAPH_AmbientLight*

This type defines the ambient illumination of a scene.

Table 109 — *PRC_TYPE_GRAPH_AmbientLight*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_AmbientLight*
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_ambient_index	*UnsignedInteger*	(Required) **ambient_index** + 1
biased_diffuse_index	*UnsignedInteger*	(Required) **diffuse_index**+ 1
biased_emissive_index	*UnsignedInteger*	(Required) **emissive_index** + 1
biased_specular_index	*UnsignedInteger*	(Required) **specular_index** + 1

8.5.16 *PRC_TYPE_GRAPH_PointLight*

This type defines scene light from a point with attenuation factors.

Table 110 — *PRC_TYPE_GRAPH_PointLight*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_PointLight*
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_ambient_index	*UnsignedInteger*	(Required) **ambient_index** + 1
biased_diffuse_index	*UnsignedInteger*	(Required) **diffuse_index** + 1
biased_emissive_index	*UnsignedInteger*	(Required) **emissive_index** + 1
biased_specular_index	*UnsignedInteger*	(Required) **specular_index** + 1
location	*Vector3d*	(Required) location of light
constant_attenuation_factor	*Double*	(Required) constant light attenuation factor in the range [0.0,1.0]

Table 110 *(continued)*

linear_attenuation_factor	*Double*	(Required) linear light attenuation factor in the range [0.0,1.0]
quadratic_attenuation_factor	*Double*	(Required) quadratic light attenuation factor in the range [0.0,1.0]

The attenuation factor is defined (like OpenGL) as:

$F \leftarrow 1/(Cc + Cl*d + Cq*d*d)$

Where:

d = postive distance between the light's position and the vertex

Cc = constant light attenuation

Cl = linear light attenuation.

Cq = quadratic light attenuation

8.5.17 *PRC_TYPE_GRAPH_DirectionalLight*

This type defines scene directional illumination.

Table 111 — *PRC_TYPE_GRAPH_DirectionalLight*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_GRAPH_DirectionalLight*
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_ambient_index	*UnsignedInteger*	(Required) **ambient_index** + 1
biased_diffuse_index	*UnsignedInteger*	(Required) **diffuse_index** + 1
biased_emissive_index	*UnsignedInteger*	(Required) **emissive_index** + 1
biased_specular_index	*UnsignedInteger*	(Required) **specular_index** + 1
direction	*Vector3d*	(Required) **direction** of light
intensity	*Double*	(Required) light **intensity**, a coefficient for the light in the range [0.0,1.0]

8.5.18 *PRC_TYPE_GRAPH_SpotLight*

This type defines scene light from a spot illumination, a point, with angle, intensity and attenuation parameters.

Table 112 — *PRC_TYPE_GRAPH_SpotLight*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_GRAPH_SpotLight
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
biased_ambient_index	*UnsignedInteger*	(Required) **ambient_index** + 1
biased_diffuse_index	*UnsignedInteger*	(Required) **diffuse_index** + 1
biased_emissive_index	*UnsignedInteger*	(Required) **emissive_index** + 1
biased_specular_index	*UnsignedInteger*	(Required) **specular _index**+ 1
location	*Vector3d*	(Required) location of light
constant_attenuation_factor	*Double*	(Required) constant light attenuation factor in the range [0.0,1.0]
linear_attenuation_factor	*Double*	(Required) linear light attenuation factor in the range [0.0,1.0]
quadratic_attenuation_factor	*Double*	(Required) quadratic light attenuation factor in the range [0.0,1.0]
direction	*Vector3d*	(Required) direction of light
fall_off_angle	*Double*	(Required) **fall_off_angle**: the maximum spread angle of the light source in degrees in the range [0.0,90.0] or 180,0 degrees.
fall_off_exponent	*Double*	(Required) **fall_off_exponent**: intensity distribution of the light in the range [0.0,128.0]

The **fall_off_angle** is the angle between the axis of the cone and a ray along the edge of the cone. A value of 180 degrees specifies that the light is emitted in all directions.

8.5.19 *PRC_TYPE_GRAPH_SceneDisplayParameters*

Type defines parameters used for scene visualization, including ambient light and camera.

— **index_of_line_style**: index into the line style array stored in the *FileStructureInternalGlobalData* Section 8.3.5.2. This array contains a list of PRC_TYPE_GRAPH_Style objects.

— **is_active**: since there can be more than one object of this type, this boolen is used to specify if this object is the currently active scene.

— **rotation_center**: This defines the center of rotation of the scenegraph. In other words, all objects in the scenegraph shall turn around this point if this SceneDisplayParameters is activated.

Table 113 — *PRC_TYPE_GRAPH_SceneDisplayParameters*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_GRAPH_SceneDisplayParameters
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
is_active	*Boolean*	(Required) **is_active**
number_of_lights	*UnsignedInteger*	(Required) number of lights
lights	Array <*PRC_TYPE_GRAPH_light objects*>[**number_of_lights**]	(Required) (see the sections on the light objects for details)
camera_defined	*Boolean*	(Required) camera_defined is **TRUE if** a camera is defined.
camera	*PRC_TYPE_GRAPH_Camera*	(Optional; if **camera_defined** is TRUE) (see Section 8.5.20 for details)
rotation_center_defined	*Boolean*	(Required) **rotation_center_defined** is TRUE if a rotation_center is defined
rotation_center	*Vector3d*	(Optional; if **rotational_center_defined** is TRUE)
number_of_clipping_planes	UnsignedInteger	(Required) number of clipping planes
clipping_planes	Array <*PRC_TYPE_SURF_Plane*> [**number_of_clipping_planes**]	(Optional; if **number_of_clipping_planes** > 0) (see Section 7.11.13 for details)
index_of_line_style_background	*UnsignedInteger*	(Required) **index_of_line_style**+1 (background)
index_of_line_style_default	*UnsignedInteger*	(Required) **index_of_line_style**+1 (default)
number_default_styles	*UnsignedInteger*	(Required) number of default styles per type
styles	Array <*UnsignedInteger*>[2 **number_default_styles**]	List of (**type**, **line_style_index**+1) pairs (see section 7 for a list of base entities)
is_absolute	*Boolean*	(Required) If (**TRUE**), the position of lights, camera and clipping planes are absolute even when those parameters belong to a sub assembly.

8.5.20 *PRC_TYPE_GRAPH_Camera*

This type defines the camera used in scene visualization. It contains attributes such as its position, view angle, and zoom.

Table 114 — *PRC_TYPE_GRAPH_Camera*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_GRAPH_Camera
base	*ContentPRCRefBase*	(Required) Base information associated with the entity
is_orthographic	*Boolean*	(Required) If is_orthographic is TRUE then projection is orthographic, else perspective.
position	*Vector3D*	(Required) position of the camera (3D Position)
look	*Vector3D*	(Required) "look at" point (3D Position)
up	*Vector3D*	(Required) up vector (3D Vector)
x	*Double*	(Required) field of view angle in radian (X direction) if **is_orthographic** is FALSE, Scale X if **is_orthographic** is TRUE
y	*Double*	(Required) field of view angle in radian (Y direction) if **is_orthographic** is FALSE,, Scale Y if **is_orthographic** is TRUE
ratio	*Double*	(Required) ratio of X to Y
clip_near	*Double*	(Required) near clipping plane distance from the viewer (positive value)
clip_far	*Double*	(Required) far clipping plane distance from the viewer (positive value)
zoom	*Double*	(Required) zoom factor (default 1.0)

8.6 Representation items

8.6.1 Entity types

Table 115 — Representation items entity types

Type Name	Type Value	Referenceable
PRC_TYPE_RI	*PRC_TYPE_ROOT* + 230	
PRC_TYPE_RI_RepresentationalItem	*PRC_TYPE_RI* + 1	no
PRC_TYPE_RI_BrepModel	*PRC_TYPE_RI* + 2	yes
PRC_TYPE_RI_Curve	*PRC_TYPE_RI* + 3	yes
PRC_TYPE_RI_Directioni	*PRC_TYPE_RI* + 4	yes
PRC_TYPE_RI_Plane	*PRC_TYPE_RI* + 5	yes
PRC_TYPE_RI_PointSet	*PRC_TYPE_RI* + 6	yes
PRC_TYPE_RI_PolyBrepModel	*PRC_TYPE_RI* + 7	yes
PRC_TYPE_RI_PolyWire	*PRC_TYPE_RI* + 8	yes
PRC_TYPE_RI_Set	*PRC_TYPE_RI* + 9	yes
PRC_TYPE_RI_CoordinateSystem	*PRC_TYPE_RI* + 10	yes

8.6.2 *PRC_TYPE_RI*

This is an abstract base class. When PRC_TYPE_RI class is referenced in this documentation of the PRC File Format Specification, one of its constituent classes will be physically present in the file.

This is an abstract class to group the following classes:

— *PRC_TYPE_RI_RepresentationItem*

— *PRC_TYPE_RI_BrepModel*

— *PRC_TYPE_RI_Curve*

— *PRC_TYPE_RI_Direction*

— *PRC_TYPE_RI_Plane*

— *PRC_TYPE_RI_PointSet*

— *PRC_TYPE_RI_PolyBrepModel*

— *PRC_TYPE_RI_PolyWire*

— *PRC_TYPE-RI_Set*

— *PRC_TYPE_RI_CoordinateSystem*

8.6.3 *PRC_TYPE_RI_RepresentationItem*

8.6.3.1 General

This is an abstract class for all representation items. *PRC_TYPE_RI_RepresentationItem* denotes the abstract type from which any RI type derives and gathers all data common to any RI type.

8.6.3.2 *RepresentationItemContent*

This represents common data for all PRC_TYPE_RI entities.

— **index_local_coordinate_system** represents, if defined with a value other than -1, the index of the coordinate system as stored in *FileStructureInternalGlobalData*. The transformation is used to position geometry or tessellation. The general principal is that this transformation (LocalMatrix) shall be post multiplied by the global matrix to obtain the transformation using:

GlobalMatrix x LocalMatrix

— **index_tessellation represents, if defined with a value other that -1, the index in the *FileTessellation* section within a *FileStructure***

Table 116 — *RepresentationItemContent*

Name	Data Type	Data Description
base	*PRC_TYPE_ROOT_PRCBaseWithGraphics*	(Required)
biased_index_local_coordinate_system	*UnsignedInteger*	(Required) **index_local_coordinate_system + 1**
biased_index_tessellation	*UnsignedInteger*	(Required) **index_tessellation +1**

8.6.4 *PRC_TYPE_RI_BrepModel*

This type represents a brep model.

If the brep model has a body in the exact geometry section of the *FileStructure*, the index of the topological context and the index of the body within the topological context identify the body.

A boolean flag indicates if the body is open or closed. Even if there is no body in the exact geometry section, tessellation data may represent a closed body.

Table 117 — *PRC_TYPE_RI_BrepModel*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_BrepModel*
item_content	*RepresentationItemContent*	(Required)
exact_geometry	*Boolean*	(Required) **exact_geometry** is TRUE if brep model has a body in the exact geometry section of the File Structure ; else FALSE
index_topological_context	*UnsignedInteger*	(Optional; if **exact_geometry** is TRUE) Index of the topological context in the exact geometry section of the File Structure
index_body	*UnsignedInteger*	(Optional; if **exact_geometry** is TRUE) Index of the body within the topological context
is_closed	*Boolean*	(Required) **is_closed** is TRUE if the body is closed; else FALSE
user_data	*UserData*	(Required) User defined data

8.6.5 *PRC_TYPE_RI_Curve*

This type represents a curve.

If there is a wire body in the exact geometry section of the FileStructure, the index of the topological context and the index of the body within the topological context identify the wire body.

Table 118 — *PRC_TYPE_RI_Curve*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_Curve*
item_content	*RepresentationItemContent*	(Required)
exact_geometry	*Boolean*	(Required) **exact_geometry** is TRUE if curve has a wire body in the exact geometry section of the File Structure ; else FALSE
index_topological_context	*UnsignedInteger*	(Optional; if **exact_geometry** is TRUE) Index of the topological context in the exact geometry section of the File Structure
index_body	*UnsignedInteger*	(Optional; if **exact_geometry** is TRUE) Index of the wire body within the topological context
user_data	*UserData*	(Required) User defined data

8.6.6 *PRC_TYPE_RI_Direction*

This type represents a direction vector with an optional origin. This is used to define an axis.

This entity can be used to define infinite construction lines.

Table 119 — *PRC_TYPE_RI_Direction*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_Direction*
item_content	*RepresentationItemContent*	(Required)
has_orgin	*Bit*	(Required) TRUE if the direction has an origin; else FALSE
orgin	*Vector3d*	(Optional; if **has_orgin** is TRUE) Direction origin
direction	*Vector3d*	(Required) Direction vector
user_data	*UserData*	(Required) User defined data

8.6.7 *PRC_TYPE_RI_Plane*

This type represents a construction plane as opposed to a planar surface.

If the plane has an associated body in the exact geometry section of the FileStructure, the index of a topological context and an index of the body within the topological context identify the body.

Table 120 — *PRC_TYPE_RI_Plane*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_Plane*
item_content	*RepresentationItemContent*	(Required) Common data
exact_geometry	*Boolean*	(Required) **exact_geometry** is TRUE if plane has associated body in the B-rep model; else FALSE
index_topological_context	*UnsignedInteger*	(Optional; if **exact_geometry** is TRUE) Index of a topological context in the exact geometry section containing the body
index_body	*UnsignedInteger*	(Optional; if **exact_geometry** is TRUE) Index of a body within the topological context
user_data	*UserData*	(Required) Users defined data

8.6.8 *PRC_TYPE_RI_PointSet*

This type represents a set of 3D points.

Table 121 — *PRC_TYPE_RI_PointSet*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_PointSet*
item_content	*RepresentationItemContent*	(Required)
number_of_points	*UnsignedInteger*	(Required) Number of points
points	Array <*Vector3d*>[number_of_points]	(Required) Array of points in the set
user_data	*UserData*	(Required) User defined data

8.6.9 *PRC_TYPE_RI_PolyBrepModel*

This type represents a PolyBrepModel defined by the tessellation data stored in the *RepresentationItemContent*. A boolean flag indicates if the tessellation is closed or open.

Table 122 — *PRC_TYPE_RI_PolyBrepModel*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_PolyBrepModel*
item_content	*RepresentationItemContent*	(Required)
is_closed	*Boolean*	(Required) **is_closed** is TRUE if the tessellation is closed; else FALSE
user_data	*UserData*	(Required) User defined data

8.6.10 *PRC_TYPE_RI_PolyWire*

This type represents a PolyWire defined by the tessellation data stored in the *RepresentationItemContent*.

Table 123 — *PRC_TYPE_RI_PolyWire*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_PolyWire*
item_content	*RepresentationItemContent*	(Required)
user_data	*UserData*	(Required) User defined data

8.6.11 *PRC_TYPE_RI_Set*

This represents the logical grouping of an arbitrary number of representational items.

Table 124 — *PRC_TYPE_RI_Set*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_Set*
item_content	*RepresentationItemContent*	(Required)
number_of_items	*UnsignedInteger*	(Required) Number of representation items in the set
representation_items	Array <*PRC_TYPE_RI_RepresentationItem*> [**number_of_items**]	(Required) An array of any of the PRC_TYPE_RI_xx items
user_data	*UserData*	(Required) User defined data

8.6.12 *PRC_TYPE_RI_CoordinateSystem*

A coordinate system can have one of two distinct roles

— As a representation item belonging to the tree of a part definition.

— An entity to position other representation items. In this role, the coordinate system exists in the global section of the FileStructure (see *FileStructureInternalGlobalData* and *PRC_TYPE_RI* description).

Table 125 — *PRC_TYPE_RI_CoordinateSystem*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_RI_CoordinateSystem*
item_content	*RepresentationItemContent*	(Required)
transform	*Transformation*	(Required) *PRC_TYPE_MISC_GeneralTransformation* or *PRC_TYPE_MISC_CartesianTransformation*
user_data	*UserData*	(Required) User defined data

8.7 Markup

8.7.1 Entity types

Table 126 — Markup entity types

Type Name	Type Value	Referenceable
PRC_TYPE_MKP	*PRC_TYPE_ROOT* + 500	
PRC_TYPE_MKP_View	*PRC_TYPE_MKP* + 1	yes
PRC_TYPE_MKP_Markup	*PRC_TYPE_MKP* + 2	yes
PRC_TYPE_MKP_Leader	*PRC_TYPE_MKP* + 3	yes
PRC_TYPE_MKP_AnnotationItem	*PRC_TYPE_MKP* + 4	yes
PRC_TYPE_MKP_AnnotationSet	*PRC_TYPE_MKP* + 5	yes
PRC_TYPE_MKP_AnnotationReference	*PRC_TYPE_MKP* + 6	yes

8.7.2 *PRC_TYPE_MKP*

This is the basic type for all 3D markups (annotations). Markups are non-geometric entities that aid viewers in understanding PRC model geometry. Markup types and subtypes include notes, dimensional annotations, geometric tolerance blocks, and weld symbols. Markups are linked to items, such as part geometry or assemblies. Markups may be attached to linked items by leaders (leader lines) for clarity.

Markups may contain tesselation data as patterns to define a vectorized picture incorporated in the markup. In this version, only the following entities may incorporate tesselated data: polyline, triangles, color, line style, points, polygon, line width.

8.7.3 *PRC_TYPE_MKP_View*

3D markups can be grouped into views that are associated with planes in which markup annotations lie. A view contains an array of annotation entities. A view can also define visibilities and positions of entities.

Table 127 — *PRC_TYPE_MKP_View*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_MKP_View
base	*PRC_TYPE_ROOT_PRCBaseWithGraphics*	(Required) See Section 8.2.4 for details.
number_of_annotations	*UnsignedInteger*	(Required) Number of annotations
annotations	Array <*ReferenceUniqueIdentifiers*> [**number_of_annotations**]	(Required) Unique identifiers for annotation entities
annotation_plane	*PRC_TYPE_SURF_Plane*	(Required) See Section 8.11.13 for data definition.
has_parameters	*Boolean*	(Required) **scene_display_parameters**
scene_display_parameters:	*SceneDisplayParameters*	(Optional if **has_paramaters** is TRUE) See Section 8.5.19 for data definition
is_annotation_view	*Boolean*	(Required) If TRUE then view is an annotation view
is_default_view	*Boolean*	(Required) If TRUE the view is the default view

Table 127 *(continued)*

is_direction	*Boolean*	(Required) If TRUE the plane is only indicating a direction
number_of_linked_items	*UnsignedInteger*	(Required) Number of linked items in markup view
linked_items	Array <*ReferenceUniqueIdentifiers*> [**number_of_linked_items**]	(Required) Unique identifiers of linked items
number_of_filters	*UnsignedInteger*	(Required) Number of display filters
filters	Array <*PRC_TYPE_ASM_Filter*> [**number_of_filters**]	(Required) Display Filters
user_data	*UserData*	(Required) User defined data

Definition of *ReferenceUniqueIdentifier*: reference_in_same_file_structure indicates whether the object is in the same file structure.

Table 128 — *ReferenceUniqueIdentifier*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_MISC_ReferenceOnPRCBase
type	*UnsignedInteger*	(Required) Reference type
reference_in_same_file_structure	*Boolean*	(Required) **reference_in_same_file_structure**
target_file_structure:	*CompressedUniqueID*	(Optional; if **reference_in_same_file_structure** is TRUE) See Section 8.2.2 for details
unique_id	*UnsignedInteger*	(Required) Unique_identifier for entity

8.7.4 *PRC_TYPE_MKP_Markup*

This is the Basic type for simple markups. Each markup is defined by a type and a subtype. For instance, a markup may be of the type "dimension" and the subtype "dimension radius edge" indicating that this annotation points to the radius arc of the edge of an object.

Markup types are as follows:

Table 129 — Markup types

Enum label	Description (value)
KEPRCMarkupType_Unknown	Unknown value (0)
KEPRCMarkupType_Text	Plain text (1)
KEPRCMarkupType_Dimension	Dimension (2)
KEPRCMarkupType_Arrow	Arrow (3)
KEPRCMarkupType_Balloon	Balloon (4)
KEPRCMarkupType_CircleCenter	Center of Circle (5)
KEPRCMarkupType_Coordinate	Coordinate (6)
KEPRCMarkupType_Datum	Datum (7)
KEPRCMarkupType_Fastener	Fastener (8)

Table 129 *(continued)*

KEPRCMarkupType_Gdt	Geometric Dimensioning and Tolerance (GD&T) Block (9)
KEPRCMarkupType_Locator	Locator (10)
KEPRCMarkupType_MeasurementPoint	Point (11)
KEPRCMarkupType_Roughness	Roughness (12)
KEPRCMarkupType_Welding	Welding (13)
KEPRCMarkupType_Table	Table (15)
KEPRCMarkupType_Other	Other (16)

Markup subtypes are as follows:

Table 130 — Markup subtypes

Enum label	Description (value)
KEPRCMarkupSubType_Datum_Ident	Datum Identifier subtype (1)
KEPRCMarkupSubType_Datum_Target	Datum Target subtype (2)
KEPRCMarkupSubType_Dimension_Distance	Distance Dimension (1)
KEPRCMarkupSubType_Dimension_Distance_Offset	Dimension offset distance (2)
KEPRCMarkupSubType_Dimension_Distance_Cumulate	Dimension cumulative distance (3)
KEPRCMarkupSubType_Dimension_Chamfer	Dimension chamfer callout (4)
KEPRCMarkupSubType_Dimension_Slope	Dimension slope (5)
KEPRCMarkupSubType_Dimension_Ordinate	Dimension ordinate (6)
KEPRCMarkupSubType_Dimension_Radius	Dimension radius (7)
KEPRCMarkupSubType_Dimension_Radius_Tangent	Tangent radius dimension (8)
KEPRCMarkupSubType_Dimension_Radius_Cylinder	Cylinder radius dimension (9)
KEPRCMarkupSubType_Dimension_Radius_Edge	Radius edge dimension (10)
KEPRCMarkupSubType_Dimension_Diameter	Diameter dimension (11)
KEPRCMarkupSubType_Dimension_Diameter_Tangent	Tangent diameter dimension (12)
KEPRCMarkupSubType_Dimension_Diameter_Cylinder	Cylinder diameter dimension (13)
KEPRCMarkupSubType_Dimension_Diameter_Edge	Diameter edge dimension (14)
KEPRCMarkupSubType_Dimension_Diameter_Cone	Cone diameter dimension (15)
KEPRCMarkupSubType_Dimension_Length	Length dimension (16)
KEPRCMarkupSubType_Dimension_Length_Curvilinear	Curvilinear length dimension (17)
KEPRCMarkupSubType_Dimension_Length_Circular	Circular length dimension (18)
KEPRCMarkupSubType_Dimension_Angle	Angle Dimension (19)
KEPRCMarkupSubType_Gdt_Fcf	Geometric Dimensioning and Tolerancing (1)
KEPRCMarkupSubType_Welding_Line	Welding line (1)
KEPRCMarkupSubType_Welding_Spot	Welding Spot (2)

Table 130 *(continued)*

KEPRCMarkupSubType_Other_Symbol_User	Symbol User (1)
KEPRCMarkupSubType_Other_Symbol_Utility	(2)
KEPRCMarkupSubType_Other_Symbol_Custom	(3)
KEPRCMarkupSubType_Other_GeometricReference	Geometric Reference (4)

index_tessellation represents, if defined (by specifying a value other than -1), the index of the tessellation in the tessellation section of the file structure. This index should point to a **PRC_TYPE_TESS_Markup** type object associated with this markup.

Table 131 — *PRC_TYPE_MKP_Markup*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MKP_Markup*
base	*PRC_TYPE_ROOT_PRCBaseWithGraphics*	(Required) See Section 8.2.4 for details
maerkup_type	*UnsignedInteger*	(Required) type
markup_subtype	*UnsignedInteger*	(Required) sub_type
number_of_linked_items	*UnsignedInteger*	(Required) **number_of_linked_items**
linked_items	Array <*ReferenceUniqueIdentifiers*> [**number_of_linked_items**]	(Required) Unique identifiers for each linked item
number_of_leaders	*UnsignedInteger*	(Required) **number_of_leaders**
leaders	Array <*ReferenceUniqueIdentifiers*> [**number_of_leaders**]	(Optional; if **number_of_leaders** > 0) Unique identifiers for each leader
biased_index_tessellation	*UnsignedInteger*	(Required) **index_tessellation** + 1
user_data	*UserData*	(Required) User defined data

8.7.5 *PRC_TYPE_MKP_Leader*

This is the basic type for a 3D markups leader. Leaders attach the markup annotation item to the annotation reference.

Table 132 — *PRC_TYPE_MKP_Leader*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MKP_Leader*
base	*PRC_TYPE_ROOT_PRCBaseWithGraphics*	(Required)
first_linked_item	*ReferenceUniqueIdentifiers*	(Required) Unique identifiers for each linked item
is_second_linked_item	*Boolean*	(Required) **is_second_linked_item** is TRUE if there is a second linked item
second_linked_item	*ReferenceUniqueIdentifiers*	(Optional; if **is_second_linked_item** is TRUE) Unique identifiers for second linked item
biased_index_tessellation	*UnsignedInteger*	(Required) **index_tessellation** + 1
user_data	*UserData*	(Required) User defined data

8.7.6 *PRC_TYPE_MKP_AnnotationItem*

This section contains the data for a single annotation item.

Table 133 — PRC_TYPE_MKP_AnnotationItem

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MKP_AnnotationItem*
base	*PRC_TYPE_ROOT_PRCBaseWithGraphics*	(Required)
unique_id	*ReferenceUniqueIdentifier*	(Required) Unique identifier for the annotation item
user_data	*UserData*	(Required) User defined data

8.7.7 *PRC_TYPE_MKP_AnnotationSet*

An annotation set is a group of annotation items or subsets. For example, a tolerance defined by a datum and a feature control frame are described by an annotation set with two annotation items, where the items point respectively to a markup of type "datum" and a markup of type "feature control frame."

Table 134 — *PRC_TYPE_MKP_AnnotationSet*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_MKP_AnnotationSet
base	*PRC_TYPE_ROOT_PRCBaseWithGraphics*	(Required)
number_of_annotations	*UnsignedInteger*	(Required) Number of entities in the annotation set
annotations	Array <*AnnotationEntity*> [**number_of_annotations**]	(Optional; if **number_of_annotations** > 0) For each entity in the annotation set.
user_data	*UserData*	(Required) User defined data

The *AnnotationEntity* entry above will be one of the type: *PRC_TYPE_MKP_AnnotationItem* or *PRC_TYPE_MKP_AnnotationSet* or *PRC_TYPE_MKP_AnnotationReference*.

8.7.8 *PRC_TYPE_MKP_AnnotationReference*

An annotation reference stores explicit combinations of markup data with modifiers that can then be used to define other annotations. An example would be a feature control frame.

Table 135 — *PRC_TYPE_MKP_AnnotationReference*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MKP_AnnotationReference*
base	*PRC_TYPE_ROOT_PRCBaseWithGraphics*	(Required)
number_of_linked_items	*UnsignedInteger*	(Required) Number of linked items in the annotation reference
linked_items	Array <*ReferenceUniqueIdentifiers*> [**number_of_linked_items**]	(Optional; if **number_of_linked_items** > 0) List of the identifiers of the linked items in the reference

8.8 Tessellation

8.8.1 Entity Types

Table 136 — Tessellation entity types

Type Name	Type Value	Referenceable
PRC_TYPE_TESS	*PRC_TYPE_ROOT* + 170	no
PRC_TYPE_TESS_Base	*PRC_TYPE_TESS* + 1	no
PRC_TYPE_TESS_3D	*PRC_TYPE_TESS* + 2	no
PRC_TYPE_TESS_3D_Compressed	*PRC_TYPE_TESS* + 3	no
PRC_TYPE_TESS_Face	*PRC_TYPE_TESS* + 4	no
PRC_TYPE_TESS_3D_Wire	*PRC_TYPE_TESS* + 5	no
PRC_TYPE_TESS_Markup	*PRC_TYPE_TESS* + 6	no

8.8.2 *PRC_TYPE_TESS*

8.8.3 *PRC_TYPE_TESS_Base*

Abstract root type for any tessellated entity.

8.8.4 *ContentBaseTessData*

This base class stores the coordinates of the tessellated data.

The interpretation of the **coordinates** data depends upon the entity type containing this array. See *PRC_TYPE_TESS_3D*, *PRC_TYPE_TESS_3D_Compressed*, *PRC_TYPE_TESS_3D_Wire*, or *PRC_TYPE_TESS_Markup* for a description of the interpretation of the coordinates array within these contexts.

Table 137 — *ContentBaseTessData*

Name	Data Type	Data Description
is_calculated	*Boolean*	(Required) **is_calculated** is a flag denoting whether the tessellation was calculated during import or read directly from the native CAD file.
number_of_coordinates	*UnsignedInteger*	(Required) **number_of_coordinates** represents the number of doubles in the coordinate array.
coordinates	Array <Double> [**number_of_coordinates**]	(Required) **coordinates** is an array of doubles.

8.8.5 *PRC_TYPE_TESS_3D*

8.8.5.1 General

A *PRC_TYPE_TESS_3D* entity contains tessellation data for an ordered collection of faces (*PRC_TYPE_TESS_Face*) as well as tessellation data for the wire boundaries of the faces. The notion of face does not necessarily reflect that the data comes from geometrical faces; it is also possible to store tessellation data within this entity which are an unordered set of triangles.

The following is a description of the data in the file:

— The *ContentBaseTessData* class defines the **number_of_coordinates** and **coordinates** of the tessellation data. It also defines a flag, **is_calculated**, indicating whether the data was calculated during import or comes directly from a CAD system. Data in the **coordinates** array are interpreted as the x, y, and z coordinates of the 3D points for the entire tessellation.

— **number_of_normal_coordinates** is size of the normal_coordinates array

— **normal_coordinates** is an array of doubles. Data in the **normal_coordinates** array are interpreted as the (nx, ny, nz) values of a normal vector at a 3D point. A 3D point may have multiple normal values each associated with a different triangle within the tessellated data. The nomal vector is not required to be a unit normal and may be of arbitrary length.

— **number_of_triangulated_indices** is the size of the **triangulated_index_array**.

— **triangulated_index_array** is an array of integers which are an index into the **coordinates** or **normal_coordinates** arrays. Because these arrays represents triples of numbers of the (x, y, z) of a point or the (nx, ny, nz) values of a normal vector, the index is always a multiple of 3. The interpretation of the data in this array is described below.

— **number_of_wire_indices** is the size of the wire index array

— **wire_indices** are indices into the coordinates array. The indicies in this array are grouped into the indices for a wire contour of the face. The array **wire_index** within the *PRC_TYPE_TESS_Face* indicates the start of the wire for each of the wire contours within a specific face.

— **has_faces** is TRUE if this entity is built using geometrical faces.

— **has_loops** is TRUE if this entity is built using geometrical faces and loops (wires of faces denote the loops).

— **number_of_face_tessellation_data** is the faces in the array of **face_tessellation_data**

— **face_tessellation_data** an array of *PRC_TYPE_TESS_Face* objects

— **Number_of_texture_coordinates** is the size of the texture coordinate array

— **Texture_coordinates** texture coordinate (see *PRC_TYPE_GRAPH_TextureApplication*)

— **crease_angle** is the threshold angle between two faces.

When recalculating the normals at points, the angle between two adjacent triangles is calculated and compared to the **crease_angle**. If it is below **crease_angle**, the normal would be shared at this point for the two triangles; otherwise, two distinct normals will exist.

— If must_recalculate_normals is set to TRUE, the normals shall be recalculated at loading according to the **crease_angle**. In this case, no normal indices are stored in the triangulated_index_array and the normal_coordinate array size is set to 0.

However, all the indices stored in *PRC_TYPE_TESS_Face* are not affected by the value of **must_recalculate_normals**. Specifically, **used_entities_flag** and **start_triangulated** are set as if normal indices were stored.

NOTE When storing a tessellation data with two faces, with one triangle each that have a common edge, both with a flag used_entities_flag = *PRC_FACETESSDATA_Triangle*, and with must_recalculate_normals = TRUE, this is what will be stored :

— **number_normal_coordinates** = 0; (It Would be 12 with must_recalculate_normals = FALSE)

— **number_of_coordinates** = 12;

— **number_of_triangulated_indicies** = 6. (It Would be 12 with must_recalculate_normals = FALSE)

— all of the data for *PRC_TYPE_TESS_Face* is identical regardless of the **must_recalculate_normals** flag setting.

The basic tessellation data consists of

— an array **coordinates** representing the (x, y, z) coordinates of the 3D points ot the tessellation;

— an array **normal_coordinates** representing the (nx, ny, nz) components of normal vectors at the points; a given point may have multiple normal vectors, one for each vertex of the point in the triangularization data of the tessellation;

an array **triangulated_index_array** of indicies into either the coordinates or **normal_coordinates** array. The entries in this array are grouped into one of the types of triangularization data (**PRC Tessellation Type**).

— The type of triangularization defines the sequence and type of data (point or normal) of the triangularization data. For instance, a *PRC_FACETESSDATA_Triangle* is described with 6 indices (normal, point, normal, point, normal, point). Note that it is mandatory to specify at least one normal per triangularization data.

— The order of tessellated faces in **face_tessellation_data** defines the order of *PRC_TYPE_TESS_3D* triangularization data in the **triangulated_index_array**. The triangularization data for the first face is first in the **triangulated_index_array**, followed by the data for the second face, etc.

— The triangularization data within a face consists of multiple triangulations. Each triangulation is of one of the types described in **PRC Tessellation Types** and identical types are grouped together. The bit fields of the **used_entities_flag** indicates if that type of triangularization data is present in the triangularization data for the face and the order of the bit fields from low to high (0 to 31) indicate the order of data in the TriangulatedData array. See *PRC_TYPE_TESS_Face* for a description of the face data.

A face tessellation corresponds to a geometrical face if faces are used (as denoted by **has_faces**). Otherwise, it is a large container that can be used for any tessellated data.

Wire_indices are the indicies describing the face's wire contours. See *PRC_TYPE_TESS_3D_Wire* and *PRC_TYPE_TESS_Face* for respective descriptions of how to interpret the data in the **wire_indices** array.

Texture coordinates are also to be interpreted according to the final graphics of each **face_tessellation**. Those graphics are specified either in face_tessellation or by the representation item owning the *PRC_TYPE_TESS_3D*. Then, the graphics will correspond to a texture with an appropriate number of coordinates as explained in *PRC_TYPE_GRAPH_TextureApplication* type description.

Table 138 — *PRC_TYPE_TESS_3D*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_TESS_3D
tessellation_coordinates	*ContentBaseTessData*	(Required) **tessellation coordinates**
has_faces	*Boolean*	(Required) **has_faces**
has_loops	*Boolean*	(Required) **has_loops**
must_calculate_normals	*Boolean*	(Required) **must_calculate_cormals**
normal_recalculation_flags	*Character*	normal_recalculation_flags
crease_angle	*Double*	(Optional; if **must_calculate_normals is TRUE)** crease_angle
number_of_normal_coordinates	*UnsignedInteger*	(Required) **number_of_normal_coordinates**
normal_coordinates	Array<*Double*> [**number_of_normal_coordinates**]	(Required) **normal_coordinates**
number_of_wire_indices	*UnsignedInteger*	(Required) **number_of_wire_indices**
wire_indices	Array <*Unsigninteger*> [**number_of_wire_indices**]	(Required) **wire_indices**
number_of_triangulated_indicies	*UnsignedInteger*	(Required) **number_of_triangulated_indicies**
triangulated_index_array	Array<*UnsignedInteger*> [**number_of_triangulated_indicies**]	(Required) **triangulated_index_array**
number_of_face_tessellation	UnsignedInteger	(Required) **number_of_face_tessellation**
face _tessellation_data	Array <*PRC_TYPE_TESS_Face*> [**number_of_face_tessellation**]	(Required) **face_tessellation_data**
number_of_texture_coordinates	*UnsignedInteger*	(Required) **number_of_texture_coordinates**
texture coordinates	Array <*Double*> [**number_of_texture_coordinates**]	(Required) **texture coordinates**

8.8.5.2 PRC Tessellation Types

Table 139 — PRC tessellation types

Value	Enum	Description
0x40000000	PRC_FACETESSDATA_NORMAL_Single	If this flag is set, the corresponding OneNormal entity (see PRC Tessellation Types) is planar and only one normal is defined for the entity. Otherwise, one normal per point is defined. This flag is only used for PRC_FACETESSDATA_*OneNormal entities
0x0001	PRC_FACETESSDATA_Polyface	Not used
0x0002	PRC_FACETESSDATA_Triangle	described with 6 indices (normal,point,normal,point,normal,point).
0x0004	PRC_FACETESSDATA_TriangleFan	described with 2*n indices (normal,point,... ,normal,point).
0x0008	PRC_FACETESSDATA_TriangleStripe	described with 2*n indices (normal,point,... ,normal,point).
0x0010	PRC_FACETESSDATA_PolyfaceOneNormal	Not used
0x0020	PRC_FACETESSDATA_TriangleOneNormal	described with 4 indices (normal,point,point,point).
0x0040	PRC_FACETESSDATA_TriangleFanOneNormal	described with n+1 indices (normal,point,point,..., point) if PRC_FACETESSDATA_NORMAL_Single is set described with 2*n indices (normal,point,... ,normal,point) if *PRC_FACETESSDATA_NORMAL_Single* is not set, in which case normal is to be interpreted as triangle normal (last normal is repeated)
0x0080	PRC_FACETESSDATA_TriangleStripeOneNormal	Described with n+1 indices (normal,point,point,..., point) if PRC_FACETESSDATA_NORMAL_Single is set Described with 2*n indices (normal,point,... ,normal,point)if *PRC_FACETESSDATA_NORMAL_Single* is not set, in which case normal is to be interpreted as triangle normal (last normal is repeated)
0x0100	PRC_FACETESSDATA_PolyfaceTextured	Not used
0x0200	PRC_FACETESSDATA_TriangleTextured	This is the same as PRC_FACETESSDATA_Triangle except that there are texture coordinate indices between normal and point indices. The variable **number_of_texture_coordinate_indexes** in PRC_TYPE_TESS_Face specifies the number of indices. For example, a simple triangle with one texture coordinate index is described by (normal,texture,point,normal,texture,point,normal,texture,point).

Table 139 *(continued)*

0x0400	PRC_FACETESSDATA_TriangleFanTextured	This is the same as PRC_FACETESSDATA_TriangleFan except that there are texture coordinate indices between normal and point indices. The variable **number_of_texture_coordinate_indexes** in PRC_TYPE_TESS_Face specifies the number of indices. For example, a triangle fan with one texture coordinate index is described by (normal,texture,point,normal,texture,point,normal,texture,point).
0x0800	PRC_FACETESSDATA_TriangleStripeTextured	This is the same as PRC_FACETESSDATA_TriangleStripe except that there are texture coordinate indices between normal and point indices. The variable **number_of_texture_coordinate_indexes** in PRC_TYPE_TESS_Face specifies the number of indices. For example, a triangle stripe with one texture coordinate index is described by (normal,texture,point,normal,texture,point,...,normal,texture,point).
0x1000	PRC_FACETESSDATA_PolyfaceOneNormalTextured	Not used
0x2000	PRC_FACETESSDATA_TriangleOneNormalTextured	This is the same as PRC_FACETESSDATA_TriangleOneNormal except that there are texture coordinate indices between normal and point indexes. The variable **number_of_texture_coordinate_indexes** in PRC_TYPE_TESS_Face specifies the number of indices. For example, a simple triangle with one texture coordinate index is described by (normal,texture,point,texture,point,texture,point).

Table 139 *(continued)*

0x4000	PRC_FACETESSDATA_TriangleFanOneNormalTextured	This is the same as PRC_FACETESSDATA_TriangleFanOneNormal except that there are texture coordinate indices between normal and point indexes. The variable **number_of_texture_coordinate_indexes** in PRC_TYPE_TESS_Face specifies the number of indices. For example, a triangle fan with one texture coordinate index is described as follows: (normal,texture,point,... ,normal,texture,point) if PRC_FACETESSDATA_NORMAL_Single is not set. (normal,texture,point,... ,texture,point) if PRC_FACETESSDATA_NORMAL_Single is set.
0x8000	PRC_FACETESSDATA_TriangleStripeOneNormalTextured	This is the same as PRC_FACETESSDATA_TriangleStripeOneNormal except that there are texture coordinate indices between normal and point indexes. The variable **number_of_texture_coordinate_indexes** in PRC_TYPE_TESS_Face specifies the number of indices. For example, a triangle stripe with one texture coordinate index is described as follows: (normal,texture,point,... ,normal,texture,point) if PRC_FACETESSDATA_NORMAL_Single is not set. (normal,texture,point,... ,texture,point) if PRC_FACETESSDATA_NORMAL_Single is set

8.8.6 *PRC_TYPE_TESS_Face*

8.8.6.1 General

This represents tessellation data for a face. An entity of this type only exists in a PRC File because it is referenced by a PRC_TYPE_TESS_3D. The coordinates, normals, and indices of the triangulated data are found in the PRC_TYPE_TESS_3D which references this entity.

The following is a description of the variables in the file:

— **size_of_line_attributes** is the number of entries in line_attributes

— **line_attributes** is an array of line styles

— **start_of_wire_data** represents the starting index for the wire data in the array of **wire_indices** of the *PRC_TYPE_TESS_3D* entity. Using **sizes_wire**, and **start_of_wire_data** determines where to retrieve wire point coordinates.

— **size_of_sizes_wire** is the number of entries in **sizes_wire**

— **sizes_wire** is an integer array of the number of indices for each wire edge of this face. The indicies are stored in the array **wire_indicies** within the *PRC_TYPE_TESS_3D* entity containing this face.

— **used_entities_flag** is a flag that indicates the types of triangulated entities in the array **triangulateddata**; the various bits of this flag are defined in *PRC Tessellation Types*; the order is the same as the order defined in the table of PRC_TessellationTypes.

— **start_triangulated** represents the starting index for the triangulated data of this face within the array triangulated_index_array of the PRC_TYPE_TESS_3D entity containing this face.

— **size_of_triangulateddata** is the number of entries in the array **triangulateddata**.

— **triangulateddata** is an integer array describing the tessellation data for a face. See below for a description of the data within this array.

— **number_of_texture_coordinate_indexes** represents the number of texture coordinate indices (see *PRC Tessellation Types*).

— **has_vertex_colors** is a flag indicating if colors are stored directly in the vertices. Either there is no color for the vertices, or every vertex shall have a color.

— **behavior** denotes the graphics behaviour, such as inheritance, for the entity in the tree owning the face tessellation, as described in **behavior_bit_field** of *GraphicsContent* section. Note that this is not relevant if **size_of_line_attributes** is 0 (meaning that there are no graphic attributes for the face).

The tessellation data for a face consists of a number of triangulations. Each triangulation is of one of the types described in *PRC Tessellation Types*. The bit fields of the **used_entities_flag** indicate if that type of triangularization data is present and the order is the same as the oder defined in the table of PRC_TessellationTypesif such data is present.

The first entry of the **triangulateddata** array indicates the number of triangles (PRC_FACETESSDATA_Triangle); other entries will indicate the number of the entities of that type followed by number of points used by that entity type.

For example, consider a face whose tessellation data contains 5 triangles, two fans of 5 and 7 indices, and 1 stripe of 11 indices. In this case,

— **used_entities_flag** = *PRC_FACETESSDATA_Triangle* & *PRC_FACETESSDATA_TriangleFan* & *PRC_FACETESSDATA_TriangleStripe*

— **start_triangulated** = index into triangulated_index_array of the start of data for this face; this would be 0 for a single face in a *PRC_TYPE_TESS_3D* entity.

— **triangulateddata** = (5, 2, 5, 7, 1, 11)

size_of_line_attributes can have one of following values.

— 0 if there are no graphics. In this case, all graphics are inherited from the owner of the *PRC_TYPE_TESS_3D* data.

— 1 if there is one graphic associated with the whole face tessellation data.

— 2 or higher : in this case, the number of graphics entities must be equal to the number of entities stored in the current face. For instance, if the face contains 3 triangles, 2 fans and 7 stripes, this number shall be set to 12.

The size of a wire edge of a FaceTessData is limited to 16383 (0x3FFF) points. For wire edges, two flags denote the drawing **behaviour** (see Special flags for 3DwireTessData wire tessellation.).

For example, if there are two loops having 2 and 1 wire edges, respectively: For the first loop, the first edge would have 10 points and the second edge would have 20 points. For the second loop there would be 12 points. The array would be [10, 20 | *PRC_FACETESSDATA_WIRE_IsClosing*, 12 | *PRC_FACETESSDATA_WIRE_IsClosing*] Note that the indices for the edge extremes are always stored. Therefore, the 10th point of the first edge should be at the same location as the first point of the second edge.

In the cases where the tessellation type contains one normal, the number of points is combined with the flag *PRC_FACETESSDATA_NORMAL_Single*. Hence the number of points is always limited to 0x3FFFFFFF whatever the PRC tessellation type for FaceTessData.

Table 140 — *PRC_TYPE_TESS_Face*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TESS_Face*
size_of_line_attributes	*UnsignedInteger*	(Required) **size_of_line_attributes**
line_attributes	Array <*UnsignedInteger*> [size_of_line_attributes]	(Required) array of **line_attributes** where each entry is (**index_of_line_style**+1) into the array of line styles (see GraphicsContent).
start_of_wire_data	*UnsignedInteger*	(Required) **start_of_wire_data**
size_of_sizes_wire	*UnsignedInteger*	(Required) **size_of_sizes_wire**
sizes_wire	ArrayOf[*UnsignedInteger*> [**size_of_sizes_wire**]	(Required) **sizes_wire**
used_entities_flag	*UnsignedInteger*	(Required) **used_entities_flag**
start_triangulated	*UnsignedInteger*	(Required) **start_triangulated**
size_of_triangulateddata	*UnsignedInteger*	(Required) **size_of_TriangulatedData**
triangulateddata	Array*Of[UnsignedInteger*> [**size_of_triangulateddata**]	(Required) **triangulatedData**
number_of_textured_coordinate_indexes	*UnsignedInteger*	(Required) **number_of_textured_coordinate_indexes**
has_vertex_colors	*Boolean*	(Required) **has_vertex_colors**
vertex color data	*VertexColors*	(Required) **vertex color data**
behavior	*UnsignedInteger*	(Optional; if **size_of_line_attributes** > 0) behavior

8.8.6.2 *Face Wire Tessellation Flags*

Table 141 — *Face Wire Tessellation Flags*

Value	Enum	Description
0x4000	PRC_FACETESSDATA_WIRE_IsNotDrawn	Indicates that the edge should not be drawn
0x8000	PRC_FACETESSDATA_WIRE_IsClosing	Indicates that this is the last edge of a loop.

8.8.7 *PRC_TYPE_TESS_3D_Wire*

8.8.7.1 General

Tessellation for a 3D wire edge

The following is a description of the variables in the file:

— The *ContentBaseTessData* class defines the **number_of_coordinates** and **coordinates** of the tessellation data. It also defines a flag, **is_calculated**, indicating whether the data was calculated during import or comes directly from a CAD system. Data in the **coordinates** array is interpreted as the x, y, and z coordinates of the 3D points in the tessellation.

— **number_of_wire_indexes** is the number of integers in the wire_indexes array.

— **wire_indexes** is an array of integers which is defined below.

— **has_vertex_colors** is a flag indicating if colors are stored directly in the vertices. Either there is no color for the vertices, or every vertex shall have a color.

If **number_of_wire_indexes** is zero, the tessellation **coordinates** represents a single wire edge. If **number_of_wire_indexes** is not zero, the array **wire_indexes** defines a sequence of wire edges by specifying the number_of_indices_per_wire_edge followed by the indicies for that wire edge. The indicies define the index into the **coordinates** array for the (x, y, z) of a point along the wire edge. The indices shall be a multiple of 3.

The **number_of_indices_per_wire_edge** is an encoded 32 bit integer containing the following:

Flag	Number_of indicies_per_wire_edge

The flag is the leftmost 4 bits and is interpreted using **3D Wire Tess Flags** to indicate

— if the first point of this wire should be linked to the last point of the preceeding wire (PRC_3DWIRETESSDATA_IsContinuous)

— if the last point of this wire should be linked to the first point of this wire (PRC_3DWIRETESSDATA_IsClosing)

Table 142 — *PRC_TYPE_TESS_3D_Wire*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_type_tess_3D_wire
tessellation_coordinates	*ContentBaseTessData*	(Required) **tessellation_coordinates**
number_of_wire_indexes	*UnsignedInteger*	(Required) **number_of_wire_indexes**
wire_indexes	Array <*Integer*> [**number_of_wire_indexes**]	(Required) **wire_indexes**
has_vertex_colors	*Boolean*	(Required) **has_vertex_colors**
vertex color data	*VertexColors*	(Optional; if **has_vertex_colors** is TRUE) vertex color data

8.8.7.2 *VertexColors*

— **color_data** is a sequence of characters indicating the RGB or RGBA values for each of the vertexes or segments in the tessellation.

The **number_of_colors** stored in the **color_data** must be calculated from the number of point indices

— found in the **wire_indexes** array in the case of a PRC_TYPE_TESS_3D_Wire

— found in the **sizes_triangulated** in the case of a PRC_TYPE_TESS_Face

Table 143 — *VertexColors*

Name	Data Type	Data Description
is_rgba	*Boolean*	(Required) **is_rgba** isTRUE implies the color is 4 characters (RGBA); FALSE implies the color is 3 characters (RGB).
is_segment_color	*Boolean*	(Required) **is_segment_color**: . If **is_segment_color** is FALSE, there is a color for every point in the appropriate array; otherwise, there is a color for every segment in the array. It is important to remember that implicit points shall also have a color. An implicit point is a point that is implied in the sequence of wire points but is not stored in the file, such as when a wire is of type *PRC_3DWIRETESSDATA_IsClosing* (i.e. last point connects to first point, but the first point is not repeated in the file)
b_optimized	*Boolean*	(Required) **b_optimised**: reserved for future use; it should always be FALSE.
color_data	*ColorData*	(Optional; if b_optimized is FALSE)

8.8.7.3 *ColorData*

Table 144 — *ColorData*

Name	Data Type	Data Description
first_vertex	*Color*	(Required) Color of first vertex; **is_rgba** indicates either 3 characters (FALSE) or 4 characters (TRUE)
remaining_vertices	Array<*ColorDataRemainder*> [**number_of_colors** - 1]	(Required) Color of remaining vertexes

8.8.7.4 *ColorDataRemainder*

Table 145 — *ColorDataRemainder*

Name	Data Type	Data Description
is_same	*Boolean*	TRUE implies this entry has the same color as the previous one
color	*Color*	(Optional; if is_same is FALSE) Color of vertex; **is_rgba** indicates either 3 characters (FALSE) or 4 characters (TRUE)

8.8.7.5 *3D Wire Tess Flags*

Table 146 — *3D Wire Tess Flags*

Value	Enum	Description
0x10000000	**PRC_3DWIRETESSDATA_IsClosing**	Indicates that the first point is implicitly repeated after the last one to close the wire edge.
0x20000000	**PRC_3DWIRETESSDATA_IsContinuous**	Indicates that the last point of the preceding wire should be linked with the first point of the current one.

8.8.8 *PRC_TYPE_TESS_Markup*

8.8.8.1 General

Contains information describing the graphical behavior for the tessellation associated to a markup (*PRC_TYPE_MKP_Markup*).

The tessellation of a markup uses two arrays containing the codes and the coordinates.

The codes array contains a description of the entities used in the tessellation.

The coordinates array (*ContentBaseTessData*) contains point coordinates as well as other floating point values used by entities.

Each entity has at least two codes. The first code contains the entity type and the number of specific inner codes. The second code is the number of doubles (coordinates) for this entity. These doubles are located in the coordinates array.

— The *ContentBaseTessData* class defines the **number_of_coordinates** and **coordinates** of the tessellation data. In the case of markup, the flag **is_calculated**, is meaningless. Data in the **coordinates** array is normally interpreted as x,y,z data, but can also contain data such as the 16 elements of a matrix.

— **number_of_codes** specifies the size of the code array

— **code_numbers** is an integer array of code numbers for the markup entity

— **number_of_text_strings** specfies the size of the string array

— **text_strings** is an array that contains the text strings for any text entities contained current markup object.

— **tessellation label** is the name of the corresponding *PRC_TYPE_MKP_Markup*.

— **behavior** is the bit field describes the graphical behavior of the tessellation.

Table 147 — *PRC_TYPE_TESS_Markup*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_TESS_Markup
tessellation_coordinates	*ContentBaseTessData*	(Required) **tessellation_coordinates**
number_of_codes	*UnsignedInteger*	(Required) **number_of_codes**
code_numbers	Array <*UnsignedInteger*> [**number_of_codes**]	(Required) **code_numbers** associated with the current markup object
number_of_text_strings	*UnsignedInteger*	(Required) **number_of_text_strings**
text_strings	Array <*String*> [**number_of_text_strings**]	(Required) **text_strings**
tessellation_label	*String*	(Required) **tessellation_label**
behavior	*Character*	(Required) **behavior**

8.8.8.2 Markup Flags

Special flags for various markup conditions. These flags are used to extract the corresponding information from the integer code array as explained in **Markup tessellation codes**.

Table 148 — *PRC_TYPE_TESS_Markup flags*

Value	Enum	Description
0x08000000	*PRC_MARKUP_IsMatrix*	Bit to denote that the current markup entity is a matrix
0x04000000	*PRC_MARKUP_IsExtraData*	Bit to denote that the current markup entity is extra data (it is neither a matrix nor a polyline).
0x000FFFFF	*PRC_MARKUP_IntegerMask*	Integer mask to retrieve the number of inner codes for a given entity
0x03E00000	*PRC_MARKUP_ExtraDataType*	Mask to retrieve the integer type of the markup entity

8.8.8.3 Markup Tessellation Behavior

Special flags for handling the graphical behavior of the tessellation associated with the markup object. these flags are represented by bits in the variable **Behavior**.

Table 149 — Markup Tessellation Behavior

Value	Enum	Description
0x01	*PRC_MARKUP_IsHidden*	The tessellation is hidden
0x02	*PRC_MARKUP_HasFrame*	The tessellation has a frame
0x04	*PRC_MARKUP_IsNotModifiable*	tessellation is given and should not be modified
0x08	*PRC_MARKUP_IsZoomable*	tessellation has zoom capability
0x10	*PRC_MARKUP_IsOnTop*	The tessellation is on top of the geometry
0x20	*PRC_MARKUP_IsFlipable*	The text tessellation can be flipped to always be readable on screen.

8.8.8.4 Description of the first Markup code.

There are three masks needed to identify the entity type.

— PRC_MARKUP_IsMatrix

— PRC_MARKUP_IsExtraData

— PRC_MARKUP_ExtraDataType

If none of these masks is set, the entity is a polyline then PRC_MARKUP_IsMatrix should not be set if PRC_MARKUP_IsExtraData.

If PRC_MARKUP_IsExtraData is set then PRC_MARKUP_ExtraDataType mask should be used to retrieve the type of markup entity.

8.8.8.5 Description of the Second Markup Code.

The second code is the number of doubles needed by the entity.

The following table shows, for each defined entity, the extra data type, the number of inner codes , and the number doubles in the coordinate array.

The extra data type is set using the PRC_MARKUP_ExtraDataType mask.

8.8.8.6 Table of Entities

In the table below, [1] indicates entity types which are used to define blocks. The notion of block is discussed in next section. [2] indicates entity modes as discussed in further section as well.

Table 150 — Table of Entities

Entity	Extra Data Type	Number of inner codes	Number of Doubles
Polyline	None	0	Points*3
Matrix mode [1]	None	0 or number of entities in the block	0 or number of doubles used in the block (at least 16)
Pattern	0	3+number of loops	Points in loop*3
Picture	1	1	0
Triangles	2	0	Number triangle*9

Table 150 *(continued)*

Quads	3	0	Number of quads*12
Face view model[1]	6	0 or number of entities in the block	0 or number of doubles used in the block
Frame draw model[1]	7	0 or number of entities in the block	0 or number of doubles used in the block
Fixed Size Model[1]	8	0 or number of entities in the block	0 or number of doubles used in the block
Symbol	9	1	3
Cylinder	10	0	3
Color	11	1	0
Line stipple[2]	12	0	10
Font	13	1	0
Text	14	1	2
Points	15	0	Number Points*3
Polygon	16	0	Number points*3
Linewidth[2]	17	0	0 or 1

8.8.8.7 Block and Entity Modes

8.8.8.7.1 Description of a Block

Blocks are defined by face view, frame draw, fixed size and matrix modes (described below).

Each block is surrounded by the corresponding entity. At the start of a block, the entity modifies the state, which may include the line style, or current transformation matrix. The state is restored at the end of the block.

For example, a matrix mode starts by defining a matrix that will multiply the current transformation matrix, draws some entities, and ends with another matrix mode entity indicating the end of the mode.

8.8.8.7.2 Description of Modes Used in Block Definitions

Because the face view, frame draw, fixed size, and matrix modes start with the corresponding entity and end when the same entity is encountered, they define blocks.

The starting entity has a non-zero number of inner codes. It represents the number of codes until the end of the block, not counting the two mandatory codes for each entity. The same rule applies to the doubles. The ending entity has no inner codes and no doubles.

The number of inner codes makes it possible to skip a block when reading a tessellation. To treat the content of a block, use the numbers as shown in the following table.

Table 151 — Description of Modes Used in Block Definitions

Mode	Number of inner codes	Number of doubles
Face view (starting)	number of entities in the block	number of doubles in the block (at least 3)
Face view (ending)	0	0
Frame draw(starting)	number of entities in the block	number of doubles in the block (at least 3)
Frame draw(ending)	0	0
Fixed size(starting)	number of entities in the block	number of doubles in the block (at least 3)
Fixed size(ending)	0	0
Matrix(starting)	number of entities in the block	number of doubles in the block (at least 16)
Matrix(ending)	0	0

The following example shows the codes for defining a matrix mode and then 3 points in the block.

(0x08000000 + 3) (begin matrix block; 3 entities to follow), 16 + 3*3 (number of doubles in block)

(0x04000000 + (15 << 21) & 0x03E00000) (first point) , 3 (it uses 3 doubles)

(0x04000000 + (15 << 21) & 0x03E00000) (second point) , 3 (it uses 3 doubles)

(0x04000000 + (15 << 21) & 0x03E00000) (third point) , 3 (it uses 3 doubles)

(0x08000000) (end matrix block), 0 (matrix ending; no double)

8.8.8.7.3 Description of Entity Modes

The line stipple and line width modes operate identically to the modes used in block definitions, but the numbers correspond only to the entity and not to the block.

For the line stipple mode, the number of inner codes denotes the start (1) or the end (0) of the block.

For the line width mode, the number of doubles denotes the start (1) or the end (0) of the block.

8.8.8.8 Entity description

8.8.8.8.1 General

For each entity, the following tables show the mandatory codes and the inner codes, as well as the doubles needed by the entity.

8.8.8.8.2 Polyline

There is an (x,y,z) triplet for each point of the polyline.

Table 152 — Polyline

Extra data Type	Number of inner Codes	Number of Doubles
0x00000000	0	Number of points*3

8.8.8.8.3 Triangles

A list of triangles. There is an (x,y,z) triplet for each point of the triangle list.

Table 153 — Triangles

Extra data Type	Number of inner Codes	Number of Doubles
0x04400000	0	Number of triangles*9

8.8.8.8.4 Quads

A list of quads. There is an (x,y,z) triplet for each point of the quad list.

Table 154 — Quads

Extra data Type	Number of inner Codes	Number of Doubles
0x04600000	0	Number of quads*12

8.8.8.8.5 Polygon

There is an (x,y,z) triplet for each point of the polygon.

Table 155 — Polygon

Extra data Type	Number of inner Codes	Number of Doubles
0x06000000	0	Number of points*3

8.8.8.8.6 Points

A list of points. There is a (x,y,z) triplet for each point.

Table 156 — Points

Extra data Type	Number of inner Codes	Number of Doubles
0x05e00000	0	Number of points*3

8.8.8.8.7 Face View Mode

In this mode, all the drawing entities are parallel to the screen (billboard). The point given in the doubles corresponds to the origin of the new coordinate system in which entries are drawn parallel to the screen.

Table 157 — Face View Mode

Extra data Type	Number of inner Codes	Number of Doubles
0x04c00000	0 or number of entities in block	0 or number of doubles in block

8.8.8.8.8 Frame Draw Mode

In this mode, all the drawing entities are given in 2-dimensional space. The point given in the doubles corresponds to a 3D point projected onto the screen, providing the origin of the 2-dimensional coordinate system in which to draw (viewport).

Table 158 — Frame Draw Mode

Extra data Type	Number of inner Codes	Number of Doubles
0x04e00000	0 or number of entities in block	0 or number of doubles in block

8.8.8.8.9 Fixed Size Mode

In this mode, all the drawing entities are drawn at a fixed size, independent of zoom. The point given in the doubles corresponds to the origin of the new coordinate system in which to draw at fixed size.

Table 159 — Fixed Size Mode

Extra data Type	Number of inner Codes	Number of Doubles
0x05000000	0 or number of entities in block	0 or number of doubles in block

8.8.8.8.10 Matrix Mode

In this mode, all the drawing entities are transformed by the matrix given in the doubles post multiplied by the current transformation matrix. At the end of the mode, the transformation matrix that was previously active is restored.

Table 160 — Matrix Mode

Extra data Type	Number of inner Codes	Number of Doubles
0x08000000	0 or number of entities in block	0 or number of doubles in block: A(1,1), A(2,1), A(3,1), A(4,1),...A(4,4)

8.8.8.8.11 Symbol

The point given in the doubles corresponds to the position of the symbol in 3D. The pattern identifier is an index into the picture array stored in *FileStructureInternalGlobalData.* The symbol is a *VPicturePattern* type.

Table 161 — Symbol

Extra data Type	Number of inner Codes	Number of Doubles
0x05200001	1	3

8.8.8.8.12 Color

This entity defines a color that will be effective until a new one is defined. The color identifier is an index into the color array stored in *FileStructureInternalGlobalData*.

Table 162 — Color

Extra data Type	Number of inner Codes	Number of Doubles
0x05600001	1	0

8.8.8.8.13 Line Style Mode

This entity defines the line style that will be effective inside the block.

The first code is 1 for beginning the block and 0 for ending.

The line style identifier is an index into the line stype array store in *FileStructureInternalGlobalData*.

Table 163 — Line Style Mode

Extra data Type	Number of inner Codes	Number of Doubles
0x05800000	0 or 1	0

8.8.8.8.14 Font

This entity defines the font used for the next Text entity.

The font identifier is an index into the font array stored in *FileStructureInternalGlobalData*.

Table 164 — Font

Extra data Type	Number of inner Codes	Number of Doubles
0x05a00001	1	0

8.8.8.8.15 Text

This entity defines text to be rendered using the current font (defined by the Font entity).

The text index refers to the text number in the string array.

W and H correspond to the width and height, respectively, of the text.

Table 165 — Text

Extra data Type	Number of inner Codes	Number of Doubles
0x05c00001	1	2

8.8.8.8.16 Line Width Mode

This entity defines the line width that will be effective inside the block.

The number of doubles is 1 for the beginning of the block and 0 for the ending of the block.

W is the line width to use in the block. It is not used when ending the block.

Table 166 — Line Width Mode

Extra data Type	Number of inner Codes	Number of Doubles
0x06200000	1	2

8.8.8.8.17 Cylinder

The cylinder is positioned by a matrix mode, oriented with the z-axis, with the base at Z = 0 and the top at Z = Height.

Table 167 — Cylinder

Extra data Type	Number of inner Codes	Number of Doubles
0x05400000	0	3

8.8.8.8.18 Image

This entity defines an image.

The picture identifier is an index into the picture array stored in the *FileStructureInternalGlobalData* section of the file.

Table 168 — Image

Extra data Type	Number of inner Codes	Number of Doubles
0x04020000	1	0

8.8.8.8.19 Pattern

The pattern identifier is an index into the fill pattern array stored in the *FileStructureInternalGlobalData* section of the file.

The filled mode is one of the following values: 0 = OR, 1 = AND, 2 = XOR.

The behavior is a bit field, with the 0x1 bit indicating whether to ignore the view transformation. If it is true, the pattern is not transformed by the current view transformation. The other bits should be set to zero.There is an (x,y,z) triplet for each point in the loops, and they are listed in sequential order.

Table 169 — Pattern

Extra data Type	Number of inner Codes	Number of Doubles
0x04000000	3 + Number of loops	Points in loops * 3

8.8.9 *PRC_TYPE_TESS_3D_COMPRESSED*

A highly compressed tessellation which is a compact approximation of a *PRC_TYPE_TESS_3D* object.The starting point is a mesh described with points, normals and triangles, with implicit topology. Each triangle has 3 normals (one for each point). The triangle normal is determined by cross-product on its vertices, oriented in conjunction with one of its 3 normals (it is assumed that the calculation gives the same sign whatever the normal). As vertex ordering may be modified during encoding, additional data structures are used to determine the sign of the encoded normals. A tolerance for approximation is also given as input. All triangles are supposed to be not-degenerated relative to this tolerance: they shall have edge length and height greater than the tolerance. IEEE 754 compliant double precision computations must be used for mesh compression calculations.

The input non-compressed mesh is duplicated into a working structure which will be traversed by a compliant writer as described below. At each step, approximation on points, normals and textures occur and the results of these approximations are reinjected into this working structure and used in further calculations until traversal is completed, producing an output compressed mesh. The approximation algorithms on points and normals are described in the following sections. The code corresponding to the basic functions used in those algorithms is given as pseudo code in the Section 10.

8.8.9.1 *Mesh Traversal*

The input mesh is traversed as follows (see Figure 2):

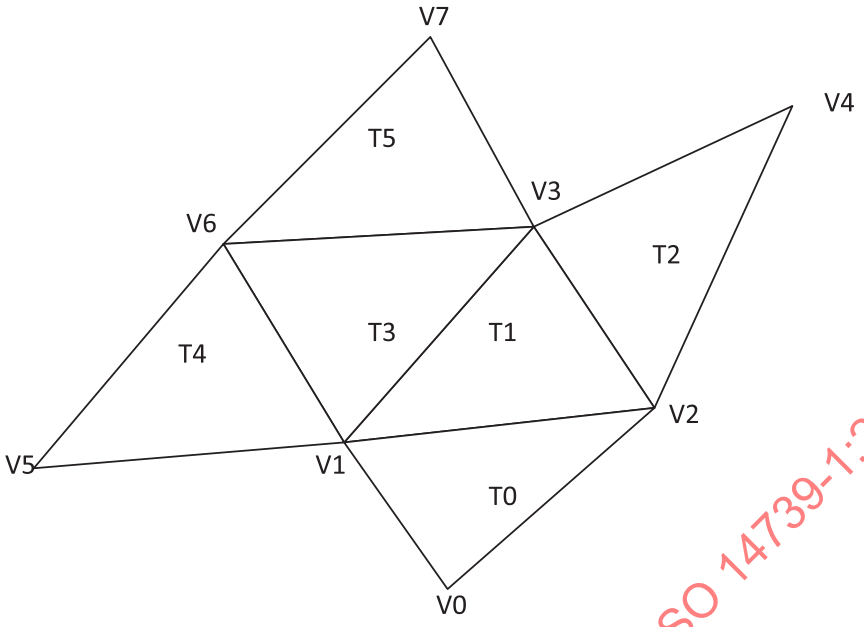


Figure 2 — Mesh Traversal

Let T0 [V0 V1 V2] be the first triangle and [V0 V1] be the first edge of T0 (this edge is arbitrarily chosen). Then, T1 is the left neighbor of T0 and the edge between V1 and V2 is the first edge of T1. T3 is the left neighbor of T1 and the edge between V1 and V3 is the first edge of T3. T2 is the right neighbor of T1 and the edge between V2 and V3 is the first edge of T2, and so on... Left / right characteristic for the neighbor is determined using the triangle normal.

To traverse the mesh structure.

— If the current triangle has only one neighbor, this neighbor is pushed on the stack.

— If the current triangle has both a left and a right neighbor which are not treated, the left triangle is first pushed on the stack. Then the right triangle is pushed on the stack.

Once done with the current triangle the stack is popped and it becomes the current triangle.

8.8.9.2 *Mesh Points and Triangles:*

The following table contains a description of the variables used while computing the compressed point mesh.

Table 170 — *Mesh Points and Triangles*

Name	Type	Description
tolerance	*Double*	(Required) 3D point tolerance
point_array	*CompressedIntegerArray*	(Required) Array of points
edge_status_array	*CharacterArray*	(Required) Flags to describe a Triangle neighbor
point_reference_array	*CompressedIndiceArray*	(Required) index of the point in the **point_array**
reference_array_size	*UnsignedInteger*	(Required) Size of **point_is_a_reference**
point_is_a_reference	Array <*Boolean*> [**reference_array_size**]	(Required) Indicates whether a point is a reference

point_array describes the vertex coordinates of each point. Coordinates are stored only if necessary, if the point has not been encountered before. As denoted in previous section, the first triangle [V_0 V_1 V_2] of a mesh, its first edge [V_0 V_1] and first point V_0 are chosen arbitrarily. This first triangle is stored in the following way : For V_0, its coordinates X,Y,Z are divided by the **tolerance** and rounded to the nearest integer. The result is stored as V_{0app} and V_0 is updated in the working structure to be V_{0app}. This assumes that those coordinates divided by tolerance do not overflow a 32 bit integer. This is a condition at every step of the compression process. For V_1, $DV_1 \leftarrow V_1\text{-}V_0$ is computed and the result is compressed and stored like the first point as DV1app; then V1 is updated in the working structure. For V_2 : $DV_2 \leftarrow V_2$ - (V_0+V_1) / 2 is computed, compressed and stored the same way as DV_{2app}; then V_2 is updated in the working structure. For subsequent triangles, they are always entered through an edge as explained in previous section. Let [V_0 V_1 V_2] be the current triangle to treat, [V_0 V_1] be the entering edge and Tn [V_0 V_1 V_3] the already-treated triangle which is the neighbor of the current towards [V_0 V_1]. If V_2 is not a reference as denoted in **point_is_a_reference** array, V_2 is stored following way: A coordinate system is defined using Tn. Origin O ← (V_1 + V_0) * 0.5. Others axis are defined below.

$$\vec{X} \leftarrow \frac{\left(\vec{V}_1 - \vec{V}_0\right)}{\left\|\vec{V}_1 - \vec{V}_0\right\|} \tag{1}$$

$$\vec{Z}_{temp} \leftarrow \vec{V}_3 - \vec{O} \text{ and } \vec{Z} \leftarrow \vec{Z}_{temp} \wedge \vec{X} \tag{2}$$

$$\vec{Y} \leftarrow \vec{Z} \wedge \vec{X} \tag{3}$$

In the equation (1), V_0 and V_1 are taken so that V_0 has a treatement index less than V_1 (which means that V_0 has been treated before V_1). A particular case occurs if the Z axis or Y are null (length less than FLT_EPSILON). FLT_EPSILON is the minimum positive floating point number of type float in accordance with the IEEE single-precision standard. In these cases, they are computed by the function MakeOrthoRep() described in annex, using the unit axis X as input. Then V_2 is expressed in this coordinate system, compressed the same way as before and updated in the working structure. Then the next triangle is traversed as explained above.

edge_status_array describes triangles' neighbors. Each triangle has a flag which is initialized to 0 and then set to:

|= 0x1 if Triangle has a Right Neighbor

|= 0x2 if Triangle has a Left Neighbor

point_reference_array is used to store treatment indexes of points which have been stored by processing a previous triangle.

point_is_a_reference indicates if a point has been already treated.

8.8.9.3 *Mesh Normal Description*

The following table contains a description of the variables used while computing the compressed normal mesh.

Table 171 — *Mesh Normal Description*

Name	Type	Description
normal_binary_data_size	*UnsignedInteger*	(Required) Size of **normal_binary_data**
normal_binary_data	Array <*Boolean*>	(Required) Information used to compute normal
normal_angle_array	*ShortArray*	(Required) Spherical coordinates of the normal
is_face_planar	Array <*Boolean*>	(Required) Is associated face planar The size of this array correspond to number of face stored in the mesh.

normal_binary_data is a bit field used to store information types on normals.

— Bit **has_multiple_normal** is TRUE if the current vertex has many normals. This bit is added only if the current vertex is encountered for the first time.

— Bit **triangle_normal_reversed** is TRUE if the computed triangle normal used to define a local coordinate system shall be reversed. See next paragraph for determination of the local coordinate system.

— Bit **is_a_reference** is TRUE if the current normal is stored as a reference on another normal of the current vertex. In this case, reference_index denotes the value of the reference. It is stored in **normal_binary_data** on a variable number of bit : **number_of_bits**. **Number_of_bits** is computed using number_of_stored_normals: number of already actually stored normals (without references) on the current vertex.

— Bit **x_is_reversed** is TRUE if the x-coordinate of the normal in the local coordinate system is reversed. (TRUE if x is reversed). Same for y_is_reversed.

Normal_angle_array describe spherical coordinates of normals (normals are unit vectors). Values stored are comprised between 0 and π / 2. For each triangle, a local coordinate system is computed and used to calculate these two angles. Finally, these two angles are compressed and temporarily stored in a short. The compressed value is computed using **normal_angle_number_of_bits**. This number shall be less than 16, (default value is 10) to be stored in an array of shorts.

Is_face_planar is TRUE if corresponding face is planar. A face is a group of triangles. In this case, only one normal is stored for all triangles of this face. It is stored when treating the first vertex of the first triangle of this face.

8.8.9.4 *Mesh Normal Construction*

For each triangle, 3 normals are computed and stored. The first one corresponds to the vertex that has the min treatment index in the first edge. The second one corresponds to the max treatment index in the first edge. Then, a local coordinate system X,Yand Z is defined from the triangles vertices in the working structure.

$$\vec{V}_1 \leftarrow \frac{\overrightarrow{secondVertex} - \overrightarrow{firstVertex}}{\left\| \overrightarrow{secondVertex} - \overrightarrow{firstVertex} \right\|} \qquad \vec{V}_2 \leftarrow \frac{\overrightarrow{thirdVertex} - \overrightarrow{firstVertex}}{\left\| \overrightarrow{thirdVertex} - \overrightarrow{firstVertex} \right\|}$$

$$\vec{V}_3 \leftarrow \frac{\overrightarrow{thirdVertex} - \overrightarrow{\sec ondVertex}}{\left\| \overrightarrow{thirdVertex} - \overrightarrow{\sec ondVertex} \right\|}$$

$$\theta_1 \leftarrow \left| \left(\vec{V}_1, \vec{V}_2 \right) - \frac{\pi}{2} \right| \qquad \theta_2 \leftarrow \left| \left(\vec{V}_3, -\vec{V}_1 \right) - \frac{\pi}{2} \right| \qquad \theta_3 \leftarrow \left| \left(-\vec{V}_2, -\vec{V}_3 \right) - \frac{\pi}{2} \right|$$

If $(\theta_1 < \theta_2)$ and $(\theta_1 < \theta_3) \Rightarrow \vec{X} \leftarrow \vec{V}_3 \qquad \vec{Z} \leftarrow \left(\vec{V}_1 \wedge \vec{V}_2 \right)$

Else if $(\theta_1 < \theta_3) \Rightarrow \vec{X} \leftarrow \vec{V}_3 \qquad \vec{Z} \leftarrow \left(-\vec{V}_3 \wedge \vec{V}_1 \right)$

Else $\Rightarrow \vec{X} \leftarrow -\vec{V}_2 \qquad \vec{Z} \leftarrow \left(\vec{V}_2 \wedge \vec{V}_3 \right)$

Z is reversed to have a scalar product positive or null with the current vertex normal. Note that this coordinate system is the same for the 3 vertices of the triangle as a consequence of the primary condition on triangle normal. The result is stored in **triangle_normal_reversed**.

$$\vec{Y} \leftarrow \frac{\vec{Z}}{\left\| \vec{Z} \right\|} \wedge \vec{X}$$

A particular case occurs if the Z axis or Y are null (length less than FLT_EPSILON). In these cases, they are computed by the function MakeOrthoRep() described in annex, using the unit axis X as input. For each vertex normal, the angles in **normal_angle_array** are computed as described below (n denotes the triangle normal) :

$$\phi \leftarrow a\sin(\vec{n} \cdot \vec{Z}) \text{ with } \vec{n} \cdot \vec{Z} \in [0,1] \text{ and } \phi \in \left[0, \frac{\pi}{2} \right]$$

$$\theta \leftarrow a\sin\left(\frac{\left(\vec{n} - (\vec{n} \cdot \vec{Z}) \times \vec{Z} \right)}{\left\| \vec{n} - (\vec{n} \cdot \vec{Z}) \times \vec{Z} \right\|} \cdot \vec{Y} \right) \text{ with } \frac{\left(\vec{n} - (\vec{n} \cdot \vec{Z}) \times \vec{Z} \right)}{\left\| \vec{n} - (\vec{n} \cdot \vec{Z}) \times \vec{Z} \right\|} \cdot \vec{Y} \in [0,1] \text{ and } \theta \in \left[0, \frac{\pi}{2} \right]$$

Spherical angles θ and ϕ are then compressed and stored the same way as follows (same formula for ϕ) :

$$\theta_{short} \leftarrow \frac{|\theta| (2^{normalAngleNumberOfBits} - 1)}{\frac{\pi}{2}}$$

These values are then written in unsigned short integers on 16 bits with a cast. Then, the nearest unsigned short values to these angles are stored in **normal_angle_array**.

$$\frac{\theta_{short} \frac{\pi}{2}}{\left(2^{normalAngleNumberOfBits} - 1 \right)} - |\theta| > \frac{1}{2} \Rightarrow \theta_{short} \leftarrow \theta_{short} + 1$$

For each vertex in each triangle, Theta and then Phi are added in **normal_angle_array** if the normal is not a reference. The pseudo code below describes how **normal_binary_data** and **normal_angle_array** are filled.

```
If (number_of_stored_normal == 0 || !has_multiple_normal ) {
    Add has_multiple_normal in normal_binary_data
    Add triangle_normal_reversed in normal_binary_data
    Add x_is_reversed in normal_binary_data
    Add y_is_reversed in normal_binary_data
    Add Angles in normal_angle_array
} else {
    Add is_a_reference in normal_binary_data
    if (is_a_reference) {
        for( i = 0; i < number_of_stored_normal; i++)
            Add reference_index&(1<<i) in normal_binary_data
    } else {
        Add triangle_normal_reversed in normal_binary_data
        Add x_is_reversed in normal_binary_data
        Add y_is_reversed in normal_binary_data
        Add Angles in normal_angle_array
    }
    }
```

After the compressed normal calculation, a compressed normal is computed and re-injected in the working structure as follow.

$$\theta_{comp} \leftarrow \frac{\theta_{short}\left(\frac{\pi}{2}\right)}{2^{normalAngleNumbeOfBits}-1}$$

$$\phi_{comp} \leftarrow \frac{\phi_{short}\left(\frac{\pi}{2}\right)}{2^{normalAngleNumbeOfBits}-1}$$

$$\vec{n}_{comp} \leftarrow \cos(\theta_{comp})\cos(\phi_{comp})\vec{X} + \sin(\theta_{comp})\cos(\phi_{comp})\vec{Y} + \sin(\phi_{comp})\vec{Z}$$

The cos and sinus functions are computed using a taylor expansion with 4 terms and the following expression.

if($\alpha > \pi/4$) => $\sin(\alpha) = \cos(\pi/2 - \alpha)$ $\cos(\alpha) = \sin(\pi/2 - \alpha)$

8.8.9.5 *Mesh Texture Structure*

The following table contains a description of the variables used for textures storage. This structure is used to describe the textures' UV parameters.

Table 172 — *Mesh Texture Structure*

Name	Type	Description
all_face_has_texture	*Boolean*	(Required) False if there is at least one face without texture
face_has_texture	Array <*Boolean*>	(Required) Does corresponding face have texture The size of this array correspond to number of face stored in the mesh
texture_data	*CompressedTextureParameter*	(Required) Information to retrieve UV texture parameters.

The combination of **all_face_has_texture** and **face_has_texture** determines whether a face has textures. **texture_data** contains information to retrieve UV textures' parameters. See *CompressedTextureParameter* for more details.

8.8.9.6 *Mesh Attribute Structure*

The following table contains a description of the variables used to contain mesh attribute data.

Table 173 — *Mesh Attribute Structure*

Name	Type	Description
is_point_color	*Boolean*	TRUE if there is at least one face with point color
is_point_color_on_face	Array <*Boolean*>	TRUE if corresponding face has point color The size of this array correspond to number of face stored in the mesh.
point_color_array	*CharacterArray*	RGB or RGBA
is_multiple_attribute	*Boolean*	TRUE if there is at least one face with multiple attributes
is_multiple_line_attribute_on_face	Array <*Boolean*>	True if the face has multiple line attributes. If there is one line attribute on the face, one graphic referenced in line_attributes_array is associated with the face. Otherwise, the number of graphics referenced in line_attributes_array is equal to the number of triangles in the face . The size of this array correspond to number of face stored in the mesh
line_attribute_array	*ShortArray*	Indexes in the graphics array

point_color_array describes colors on vertices for each triangle. For each triangle vertex with point color, 5 characters are stored. The first character describe if the vertex has got RGB or RGBA components. Then 4 components are used to stored R, G, B, and alpha.

line_attribute_array describe indexes in a graphic array. If a face contains multiple at¬tributes, one index per triangle is added in line_attribute_array. Otherwise, one index per face is added, when encountering the first triangle of this face.

8.8.9.6.1 *Description of the Data Written to the File*

The following is a description of the data in the file:

— **is_calculated** indicates whether the tessellation has been calculated during the import or has been read directly from a native file.

— **has_faces** is TRUE if the entity is built using geometrical faces.

— **tolerance** represents the tolerance of the approximation of the original tessellation.

— **origin_array** contains three floating point coordinates that describe the bounding boxcenter of the compressed 3D tessellation data.

— **points_array** contains the array of vertex points.

— **edge_status_array** for each triangle, used to describe the triangles neighbors.

point_is_referenced_array_size size of the reference array.

— **point_is_referenced_array** indicates whether a point is a reference.

— **number_of_referenced_points** size of the point reference array.

— **point_reference_array** relative point references.

— **triangle_face_array** represents, for each triangle, the index of the face to which it belongs.

— **character_array** is calculated as shown below.

— **character_array_compressed** is an integer array obtained by the Huffman algorithm, with 6 bits in character_array.

— **must_recalculate_normals** and crease_angle are described in 3D_TESS_FACE.

— **number_implicit_normal** is reserved for future use.

— **normal_is_reversed** is reserved for future use.

— **normal_binary_data** information used to compute normal.

— **normal_angle_array** spherical coordinates.

— **normal_angle_number_of_bits** is the number of bits used to approximate the triangles normals. It shall be lower than 16 and should be set to 10 bits to ensure good performance.

— **normal_angle_array** is an unsigned short array containing values less than (1 << **normal_angle_number_of_bit**) - 1). This array is optionally compressed with a Huffman algorithm using **normal_angle_number_of_bit** bits.

— **is_normal_angle_array_compressed** indicates whether **normal_angle_array** is compressed.

— **normal_angle_array_compressed** is integer array obtained by the Huffman algorithm on **normal_angle_array**.

— **face_number** is derived from the maximum value in triangle_face_array.

— **is_face_planar** is TRUE if the face is planar. In this case, only one normal per face is stored.

— **is_point_color** is TRUE if at least one face has vertices with colors (RGB or RGBA).

— **is_point_color_on_face** is TRUE if the corresponding face has vertices with colors (RGB or RGBA).

— **point_color_array** contains an RGB or RGBA component compressed using a Huffman algorithm with 8 bits.

— **is_multiple_line_attribute** indicates if there is at least one face with multiple line attributes.

— **is_multiple_line_attribute_on_face** is TRUE if the face has multiple line attributes. If there is one line attribute on the face, one graphic referenced in line_attributes_array is associated with the face. Otherwise, the number of graphics referenced in line_attributes_array is equal to the number of triangles in the face .

— **no_texture** is TRUE if there is no texture.

— **texture_data**. See corresponding chapter.

— **all_faces_have_texture** is TRUE if all faces have a texture.

— **face_has_texture** is a boolean array. It indicates which faces have texure when no_texture is TRUE and all_faces_have_texture is FALSE.

— **has_behaviours** is TRUE if special graphics behaviors or inheritances exist on faces or triangles. See 3D_TESS_FACE for more information.

— **behaviours_array** represents the behavior for each face.

Table 174 — Description of the Data Written to the File

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_TESS_3D_COMPRESSED
is_calculated	*Boolean*	(Required) **is_calculated**
has_faces	*Boolean*	(Required) **has_faces**
tolerance	*Double*	(Required) **tolerance**
origin_array	Array <*FloatAsBytes*>[3]	(Required) **origin_array** The data is compressed as follows: FloatAsBytes(**origin_array**[i]) for i = 0, 1, 2 (See **FloatAsBytes**)
point_array	*CompressedIntegerArray*	(Required) **point_array** (See 10.8)
edge_status_array	*CharacterArray*	(Required) **edge_status_array** (2 bits per character only) (See 10.6)
triangle_face_array	*CompressedIndiceArray*	(Required) **triangle_face_array** (See 10.9)
reference_array_size	*UnsignedInteger*	(Required) **reference array size**

Table 174 *(continued)*

points_is_reference_array	Array <*Boolean*> [**reference_array_size**	(Required) **points_is_reference_array**
point_reference_array	*CompressedIndiceArray*	(Required) **point_reference_array** *CompressedIndiceArray* (see 10.9) invokes *WriteCharacterArray* (see10.6); in this case, the boolean value which indicates whether the character array is compressed is not stored. Its value is implicit and set to **number_of_reference_points**=>3
must_recalculate_normals	*Boolean*	(Required) **must_recalculate_normals**
normal_is_reversed	Array <*Boolean*> []	(Optional; if **must_recalculate_normals** is TRUE) **normal_is_reversed** The number of normals is implicit, depending of the number of triangles and faces. Vertices have always as many normals as number of faces to which they belong.
crease_angle	*Double*	(Optional; if **must_recalculate_normals** is TRUE) **crease_angle**
normal_recalculation_flags	*Character*	(Optional; if **must_recalculate_normals** is TRUE) Normal recalculation flags (not used; should be zero)
normal_angle_number_of_bits	*Character*	(Optional; if **must_recalculate_normals** is FALSE) **normal_angle_number_of_bits**
normal_binary_data_size	*UnsignedInteger*	(Optional; if **must_recalculate_normals** is FALSE) **normal_binary_data_size**
normal_binary_data	Array <*Boolean*>[**normal_binary_data_size**]	(Optional; if **must_recalculate_normals** is FALSE) **normal_binary_data**
normal_angle_array	*ShortArray*	(Optional; if **must_recalculate_normals** is FALSE) **normal_angle_array** (size 16 bits)
is_face_planar	Array <*Boolean*>	(Optional; if **must_recalculate_normals** is FALSE) **is_face_planar**
is_point_color	*Boolean*	(Required) **is_point_color**

Table 174 *(continued)*

is_point_color_on_face	Array <*Boolean*>	(Optional; if **is_point_color** is TRUE) **is_point_color_on_face**
point_color_array	*CharacterArray*	(Optional; if **is_point_color** is TRUE) **point_color_array** (size 8 bits)
is_multiple_line_attribute	*Boolean*	(Required) **is_multiple_line_attribute**
is_multiple_line_attribute_on_face	Array <*Boolean*>	(Optional; if **is_multiple_line_attribute** is TRUE) **is_multiple_line_attribute_on_face**
line_attribute_array	*ShortArray*	(Required) **line_attribute_array** (size 16 bits)
no_texture	*Boolean*	(Required) **no_texture**
texture_data	*CompressedTextureParameter*	(Optional; if **no_texture** is FALSE) **texture_data**
all_faces_have_texture	*Boolean*	(Optional; if **no_texture** is FALSE) all_faces_have_texture
face_has_texture	Array <*Boolean*>	(Optional; if **no_texture** is FALSE and **all_faces_have_texture** is FALSE) face_has_texture
has_behaviors	*Boolean*	(Required) **has_behaviors**
behaviors_array	*CharacterArray*	(Optional; if **has_behaviors** is TRUE) **behaviours_array** (size 8 bits)

8.8.9.7 *CompressedTextureParameter*

The following table contains a description of the variables used to store UV textures' parameters.

binary_texture_data represents a bit field. During mesh traversal, if the current vertex has a texture, a bit texture_is_reference is set in this array.This bit is true if the same UV parameter has already been stored during mesh traversal for the same vertex. In this case, the reference index is stored in reference_array. Otherwise, this bit is set to FALSE and UV parameters are stored in Texture_parameters.

reference_array is used to reference UV parameters. The references on UV parameters are done per vertex. A UV parameter for the current vertex is referenced only if the same UV parameter has already been stored for the same vertex during the treatment of another triangle. This treatment is performed during mesh traversal, the same way as normals' treatment.

texture_parameters_tolerance is reserved for future use and should be set to zero.

texture_parameters is an array of float which contains UV textures' coordinates.This array is filled during mesh traversal as well.

Table 175 — *CompressedTextureParameter*

Name	Data Type	Data Description
binary_texture_data	*BinaryTextureData*	**(**Required) **binary_texture_data**
reference_array_size	*UnsignedInteger*	**(**Required) **reference_array_size**
reference_array	Array <*UnsignedIntegerWithVariableBitNumber*> [**reference_array_size**]	(Required) **reference_array**
texture_parameters_tolerance	*Double*	**(**Required) **texture_parameters_tolerance** reserved for future use. Should be set to 0.
texture_parameters_size	*UnsignedInteger*	**(**Required) **texture_parameters_size**
texture_parameters	Array <*FloatAsBytes*> [**texture_parameters_size**]	**(**Required) **texture_parameters**

8.8.9.8 *BinaryTextureData*

BinaryTextureData represents a bit field. It indicates during mesh traversal whether UV coordinates are referenced. (See previous chapter for more details). Then **last_integer_used_bit_number** bits are added to this array so that Texture_binary_data size becomes a multiple of 32. Consequently, 0 <= **last_integer_used_bit_number** < 32. Then the unsigned integer array is written byte by byte : each unsigned integer leads to 4 bytes obtained from *MakePortable32BitsUnsigned*. (see 10.2);

Table 176 — *BinaryTextureData*

Name	Data Type	Data Description
texture_binary_data_size	*UnsignedInteger*	**texture_binary_data_size** / 32
texture_binary_data	**Array <***bits(8)***> [texture_binary_data_size** / 8**]**	**texture_binary_data**. This size array is equal to **texture_binary_data_size** / 8.
last_integer_used_bit_number	UnsignedInteger	**last_integer_used_bit_number**

8.9 Topology

8.9.1 Entity Types

Table 177 — Topology entity types

Type Name	Type Value	Referenceable
PRC_TYPE_TOPO	*PRC_TYPE_ROOT* + 140	
PRC_TYPE_TOPO_Context	*PRC_TYPE_TOPO* + 1	no
PRC_TYPE_TOPO_Item	*PRC_TYPE_TOPO* + 2	no
PRC_TYPE_TOPO_MultipleVertex	*PRC_TYPE_TOPO* + 3	yes
PRC_TYPE_TOPO_UniqueVertex	*PRC_TYPE_TOPO* + 4	yes
PRC_TYPE_TOPO_WireEdge	*PRC_TYPE_TOPO* + 5	yes
PRC_TYPE_TOPO_Edge	*PRC_TYPE_TOPO* + 6	yes
PRC_TYPE_TOPO_CoEdge	*PRC_TYPE_TOPO* + 7	no
PRC_TYPE_TOPO_Loop	*PRC_TYPE_TOPO* + 8	yes
PRC_TYPE_TOPO_Face	*PRC_TYPE_TOPO* + 9	yes
PRC_TYPE_TOPO_Shell	*PRC_TYPE_TOPO* + 10	yes
PRC_TYPE_TOPO_Connex	*PRC_TYPE_TOPO* + 11	yes
PRC_TYPE_TOPO_Body	*PRC_TYPE_TOPO* + 12	no
PRC_TYPE_TOPO_SingelWireBody	*PRC_TYPE_TOPO* + 13	no
PRC_TYPE_TOPO_BrepData	*PRC_TYPE_TOPO* + 14	no
PRC_TYPE_TOPO_SingleWireBodyCompress	*PRC_TYPE_TOPO* + 15	no
PRC_TYPE_TOPO_BrepDataCompress	*PRC_TYPE_TOPO* + 16	no
PRC_TYPE_TOPO_WIreBody	*PRC_TYPE_TOPO* + 17	no

8.9.2 Basetopology

Base topology represents optional information for a topological entity. Such optional information is comprised of a name, attributes, and an identifier within the originating CAD system. The identifier is not used as an identifier within the PRC File, but is simply carried as information about the origin of this entity within the originating CAD system.

Table 178 — Basetopology

Name	Data Type	Data Description
has_base	*Boolean*	(Required) TRUE if base information is present; else FALSE
attribute_data	*AttributeData*	(Optional; if **has_base** is TRUE) Attributes attached to the topological entity
name	*Name*	(Optional; if **has_base** is TRUE) Name attached to the topological entity
id	*UnsignedInteger*	(Optional; if **has_base** is TRUE) Identifier in originating CAD system; this may not be used to reference entity within PRC File;

8.9.3 *PRC_TYPE_TOPO*

Abstract base class for topology.

8.9.4 *PRC_TYPE_TOPO_Context*

A topological context is a self-contained set of geometry and topology. Every geometrical and topological entity belongs to a single topological context. A topological context contains topological bodies represented as entry elements that point to topological items and geometry.

granularity represents the minimal size of an edge. This is a non-dimensional value.

tolerance represents the global base tolerance used in the context for topological elements. This is a non-dimensional value and can be superseded by looser local tolerances for particular topological elements. See Section 5.7.

smallest_thickness represents the smallest face thickness. It is used for loop algorithms, and its default value is 100 * granularity.

scale represents an optional scale that can be used to interpret the context data. This scale accommodates the different ranges of values of various CAD systems. The preceding values * scale yield dimensional values to be interpreted with the unit. For example, granularity * scale is dimensional granularity in part units. If scale is not specified no scaling is done.

The **behavior** field defines the behavior of *PRC_TYPE_TOPO_BrepData* bodies. It is a character of bits which define

— The order of outer loops within the list of loops on a face.

— Whether UV curves are clamped to the parameter domain boundaries for periodic surfaces or can extend past the boundary.

— Whether 3D edge curves (and faces) on closed or periodic surfaces are split along the seam or not.

The following table defines the bit values and behavior for this field:

Table 179 — PRC_TYPE_TOPO_Context behavior bit values

Value	Type Name	Type Description
0x0001	*PRC_CONTEXT_OuterLoops First*	Outer loops are first in the list of loops on a face.
0x0002	*PRC_CONTEXT_NoClamp*	UV curves can go beyond the domain of the bearing surface; this is used for interpreting UV curves on periodic-surfaces.
0x0004	*PRC_CONTEXT_NoSplit*	3d edge curves on closed or periodic surfaces are allowed to cross the seam of the surface.

Table 180 — *PRC_TYPE_TOPO_Context*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TYPO_Context*
base	*ContentPRCBase*	(Required) *Base information associated with the entity*
behavior	*Character*	(Required) ***behavior***
grandularity	*Double*	(Required) ***grandularity***
tolerance	*Double*	(Required) ***tolerance***
has_face_thickness	*Boolean*	(Required) **has_face_thickness** *is TRUE if smallest face thickness is present else FALSE*
Option: TRUE	*Double*	(Optional; if **has_face_thickness** is TRUE) Smallest face thickness
has_scale	*Boolean*	(Required) *TRUE if the scale factor is present; else FALSE*
Option: TRUE	*Double*	(Optional; if **has_scale** is TRUE) **scale**
number_of_bodies	*UnsignedInteger*	(Required) *Number of bodies*
bodies	Array <*PRC_TYPE_TOPO_Body*>[**number_of_bodies**]	(Required) *Array of bodies*

8.9.5 *PRC_TYPE_TOPO_Item*

Abstract root type for any topological entity (body or single item)

8.9.6 *PRC_TYPE_TOPO_MultipleVertex*

This represents a vertex whose position is the average of all edges' extremity positions which end at that vertex, that is,

```
Vertex_position = (points_for_vertex[0] + ... + point_for_vertex[number_of_points - 1]) / number_of_points;
```

Table 181 — *PRC_TYPE_TOPO_MultipleVertex*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_MultipleVertex*
base	*BaseTopology*	(Required) Common topology data (name, attributes, CAD identifier)
number_of_points	*UnsignedInteger*	(Required) Number of points
points	Array <*Vector3d*> [**number_of_points**]	(Required) Array of points for vertex

8.9.7 *PRC_TYPE_TOPO_UniqueVertex*

This represents a vertex whose position is specified by a 3D absolute postion and a tolerance. By default, the tolerance is the same as the tolerance of the topological context, but it can be over-ridden by a local one. The optional tolerance shall be either 0.0 or greater than the tolerance of the topological context of the vertex.

The tolerance is used to define a sphere around the vertex within which the vertex may lie. It is used to determine if a position is the same (within tolerance) as this vertex. See 5.7.

Table 182 — *PRC_TYPE_TOPO_UniqueVertex*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_UniqueVetex*
base	*BaseTopology*	(Required) Common topology data (name, attributes, CAD identifier)
vertex	*Vector3d*	(Required) Position of vertex
has_tolerance	*Boolean*	(Required) TRUE if there is an associated tolerance; else FALSE
tolerance	*Double*	(Optoinal; if **has_tolerance** is TRUE) tolerance

8.9.8 *PRC_TYPE_TOPO_WireEdge*

A WireEdge may belong to either a wire body or single wire body.

The geometry of a wire edge is a 3D curve which has an optional trim interval to limit the geometric definition of the curve. It is not bound by vertices. The sense of the WireEdge is the same as the underlying curve.

Table 183 — *PRC_TYPE_TOPO_WireEdge*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_WireEdge*
curve	*ContentWireEdge*	(Required) 3D curve defining the wire edge and an optional interval to restrict the wire edge to a subset of the curve.

8.9.9 *PRC_TYPE_TOPO_Edge*

This class represents an edge which is a bounded segment of a curve where the segment is not coincident or self-intersecting except possibly at the end points of the edge. The geometry of an edge is provided by a wire edge which has an optional trim interval to limit the geometric definition of the curve. The sense of the edge is the same as the sense of the wire edge which is the same as the sense of the underlying curve.

An optional tolerance may be provided which is either zero or greater than the tolerance of the topological context the edge lies in. The tolerance is used to define a pipe centered on the edge within which the edge may lie. It is used to determine if a position lies on (within tolerance of) the edge. See Section 5.7.

A start and end vertex, of type *PTR_TYPE_TOPO_UniqueVertex* or *PTR_TYPE_TOP_MultipleVertex*, represent the start and end positions on the edge. The vertices and curve trim interval are related by the tolerances associated with the vertex and edge

```
Distance(Vertex, Edge_end)<= Vertex.Tolerance() + Edge.Tolerance()
```

Table 184 — *PRC_TYPE_TOPO_Edge*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_Edge*
wire_edge	*ContentWireEdge*	(Required) Curve providing the geometric definition of the edge along with trimming information
start_vertex	*PtrTopology*	(Required) Start vertex
end_vertex	*PtrTopology*	(Required) End vertex
has_tolerance	*Boolean*	(Required) Has tolerance
tolerance	*Double*	(Optional; if **has_tolerance** is TRUE) Tolerance

8.9.10 *PRC_TYPE_TOPO_CoEdge*

A coedge represents the usage of an edge within a loop. The usage specifies the orientation of the coedge with respect to the edge:

— 0 Opposite direction

— 1 Same direction

— 2 Unknown

Normally, the orientation will be in the opposite or same direction. If the orientation is set to unknown, then *PRC_CONTEXT_OuterLoopsFirst* shall be set to TRUE to assist in the computation of the proper orientation.

A coedge may have a UV curve which may be NULL or of type *PRC_TYPE_CRV_NURBS*. The UV curve maps R^1 (the interval of the UV curve) to R^2 (the domain space of the surface defining the face the loop of coedges lie in). As with an edge, the UV curve has an orientation (opposite, same, unknown) with respect to the orientation of the coedge within the loop.

If orientation_with_loop is equal to orientation_uv_with_loop, the 3D curve orientation is the same as 2D UV curve, that is, the start point of coedge (base_surface.evaluate(curve_uv.evaluate(curve_uv.param.min)) is the same as the start point of edge (within the tolerance of edge).

Table 185 — *PRC_TYPE_TOPO_CoEdge*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_CoEdge*
base_tolopogy	*BaseTopology*	(Required) Common topology data (name, attributes, CAD identifier)
ptr_topology	*PtrTopology*	(Required) This shall be an edge (PRC_TYPE_TOPO_Edge) and shall not be NULL
ptr_curves	*PtrCurve*	(Required) UV Curve in the domain of the face this coedge lies in; may be NULL
coedge_orientation	*Character*	(Required) Orientation of coedge with respect to the loop
uv_orientation	*Character*	(Required) Orientation of the UV curve with respect to the loop

8.9.11 *PRC_TYPE_TOPO_Loop*

8.9.11.1 General

A loop is a list of coedges bounding a portion of a face in a B-rep entity. The loop may define an outer boundary of a face or it may define a hole within the face.

A loop has the following properties:

— A loop is an ordered array of references to coedges which define the boundary of the loop.

— The list of coedges form a closed boundary for the portion of the face delimited by the loop. None of the references may be null and all references shall be to *PTR_TYPE_TOPO_CoEdge*. The start vertex of one coedge shall be the end vertex of the next coedge in the list.

— The loop of coedges is oriented with respect to the surface normal using the rule of material to the left. That is, the cross-product of the tangent to the coedge at any position on the coedge with the face normal at that same position will point towards or opposite the material of the surface within the loop. The orientation of the loop might be

  — 0 Opposite direction

  — 1 Same direction

  — 2 Unknown

with respect to the normal of the face. If it is set to unknown, geometric tests shall be performed to determine the correct orientation of the loop (same or opposite).

Table 186 — *PRC_TYPE_TOPO_Loop*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_Loop*
base_topology	*BaseTopology*	(Required) Common topology data (name, attributes, CAD identifier)
loop_orientation	*Character*	(Required) Orientation of loop with respect to surface normal
number_of_coedges	*UnsignedInteger*	(Required) Number of coedges in the loop
coedge	Array <*CoedgeInLoop*> [**number_of_coedges**]	(Required) Coedges in loop

8.9.11.2 *CoedgeInLoop*

This represents a coedge around a loop. The PtrTopology shall not be NULL and shall point to a PTR_TYPE_TOPO_CoEdge.

Each coedge in the loop may index a neighboring coedge which shares the same edge but represents another usage of the edge in a boundary of a face, usually another face on another surface.

Table 187 — *CoedgeInLoop*

Name	Data Type	Data Description
next_coedge	*PtrTopology*	(Required) Next coedge in loop
neighbor_index	*UnsignedInteger*	(Requred) Index of neighboring coedge (i.e. coedge which points to the same edge as this coedge) or 0 if there is no neighboring coedge

8.9.12 *PRC_TYPE_TOPO_Face*

A face is a bounded portion of a surface where the surface is not coincident or self intersecting except possibly at the boundary of the face

It is defined by

— A surface providing the geometric definition of the face. The face always has the same orientation as the underlying surface.

— An optional Domain may restrict the definition of the face to a portion of the surface. Otherwise the parameter domain of the face is the domain of the surface.

— Like a vertex and an edge, a face has an associated tolerance which is the topological context tolerance unless an optional tolerance is specified. If the optional tolerance is specified, it shall be either 0.0 or greater than the topologial context tolerance. If the tolerance is 0.0 the topological context tolerance is used. See 5.7.

— An unordered list of loops delimiting the bounded portion (interior) of the face.

— One of the loops represents the exterior boundary of the face and the other loops (if any) represent interior loops (holes) within the face. If *PRC_CONTEXT_OuterLoopsFirst* is set to TRUE in the topological context the face is contained in, the index of the outer loop shall be defined.

Table 188 — *PRC_TYPE_TOPO_Face*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_TOPO_Face
base	*BaseTopology*	(Required) Common topology data (name, attributes, CAD identifier)
surface_geometry	*PtrSurface*	(Required) Surface geometry
is_trimmed	*Boolean*	(Required) TRUE if the suface definition is to be trimmed to a specific domain; else FALSE
trimmed_surface	*Domain*	(Optional; if **is_trimmed** is TRUE) UV domain of trimmed surface
has_tolerance	*Boolean*	(Required) TRUE if there is a tolerance associated with this face; else FALSE
tolerance	*Double*	(Optional; if **has_tolerance** is TRUE) Tolerance

Table 188 *(continued)*

number_of_loops	*UnsignedInteger*	(Required) Number of loops in this face; shall be 1 or more
index_of_outer_loop	*Integer*	(Required) Index of outer loop; it shall be set to -1 if it is not defined
loops	Array <*PtrTopology*> [**number_of_loops**]	(Required) Array of loops within this face; each pointer shall be of type *PRC_TYPE_TOPO_Loop*

8.9.13 *PRC_TYPE_TOPO_Shell*

8.9.13.1 General

A shell is a collection of faces which form either a closed or open boundary.

Table 189 — *PRC_TYPE_TOPO_Shell*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_Shell*
base	*BaseTopology*	(Required) Common topology data (name, attributes, CAD identifier)
is_closed	*Boolean*	(Required) **is_closed** is TRUE if the shell is closed; else FALSE
number_of_faces	*UnsignedInteger*	(Required) Number of faces in shell
faces	Array <*FacesInShell*> [**number_of_faces**]	(Required) Faces within shell

8.9.13.2 *FacesInShell*

This represents a face within a shell. Each face is oriented with respect to the underlying surface so that the shell normal points outside the material of the shell if the shell is closed and is arbitrary otherwise.

The orientation of the surface with respect to the shell may be

— 0 Opposite direction

— 1 Same direction

— 2 Unknown

If the orientation is unknown, geometric tests shall be performed to determine the correct orientation (within the shell) of the face (same or opposite) with respect to the surface.

Table 190 — *FacesInShell*

Name	Data Type	Data Description
face	*PtrTopology*	(Required) Face within this shell; this shall be non-NULL and of type *PRC_TYPE_TOPO_Face*
orientation	*Character*	(Required) Orientation of face with respect to the underlying surface

8.9.14 *PRC_TYPE_TOPO_Connex*

This represents a region of space delimited by one or more shells. The shells may be open or closed, may touch at a vertex, an edge, or a face, or may be contained within another shell if the interior shell represents a void within the exterior shell.

The region

— may represent a skin if the shells are open

— may represent a manifold solid if all of the shells are closed but not touching

— may represent a non-manifold solid where all of the shells are closed but some touch at a vertex, edge, or face

If the connex is delimiting material, it is mandatory that it be bounded by one or more closed shells.

Table 191 — *PRC_TYPE_TOPO_Connex*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_Connex*
base	*BaseTopology*	(Required) Common topology data (name, attributes, CAD identifier)
number_of_shells	*UnsignedInteger*	(Required) Number of shells in connex
shells	Array <*PtrTopology*> [**number_of_shells**]	(Required) Shells within this connex; each entry shall be a shell, be non-NULL, and be of type PRC_TYPE_TOPO_Shell

8.9.15 *PRC_TYPE_TOPO_Body*

This represents an abstract type for any topological body

— *PRC_TYPE_TOPO_SingleWireBody*

— *PRC_TYPE_TOPO_BrepData*

— *PRC_TYPE_TOPO_SingleWireBodyCompress*

— *PRC_TYPE_TOPO_BrepDataCompress*

8.9.16 *ContentBody*

ContentBody provides additional infromation about base topological entities (*PRC_TYPE_TOPO_SingleWireBody* and *PRC_TYPE_TOPO_BrepData*) such as its name, attributes, and CAD identifier and how the bounding box for a *PRC_TYPE_TOPO_BrepData* has been calculated.

The following table shows the possible values for bounding box behavior:

Table 192 — ContentBody bounding box behavior

Value	Type Name	Type Description
0x001	*PRC_BODY_BBOX_Evaluation*	Bounding box based on tessellation
0x002	*PRC_BODY_BBOX_Precise*	Bounding box based on geometry
0x004	*PRC_BODY_BBOX_CADData*	Bounding box given by a CAD data file

Table 193 — ContentBody

Name	Data Type	Data Description
base	*BaseTopology*	(Required) Optional topology information (name, attributes, CAD identifier)
bounding_box_behavior	*Character*	(Required) Bounding box behavior; relevent only for *PRC_TYPE_TOPO_BrepData*; otherwise shall be set to 0

8.9.17 ContentWireEdge

This represents the data defining a wire edge. It points to a 3D curve defining the geometrical shape of the edge. Any curve, including curves on a surface may be used. An optional interval may be used to limit the portion of the curve used to define the geometry of the edge. This interval shall lie within the interval of the underlying curve. If no triming interval is specified, the edge is defined by the interval defining the curve.

Table 194 — ContentWireEdge

Name	Data Type	Data Description
base	*BaseTopology*	(Required) Common topology data (name, attributes, CAD identifier)
ptr_curve	*PtrCurve*	(Required) 3D curve defining the geometry of the wire edge
is_trimmed	*Boolean*	(Required) TRUE if the wire edge restricts the 3D curve to a subset
trim_interval	*Interval*	(Optional; if **is_trimmed** is TRUE) Interval defining the subset of the 3D curve represented by the wire edge

8.9.18 *PRC_TYPE_TOPO_SingleWireBody*

PRC_TYPE_TOPO_SingleWireBody is the topological equivalent of a single curve.

Table 195 — *PRC_TYPE_TOPO_SingleWireBody*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_TOPO_SingleWireBody
base	*ContentBody*	(Required) Common data for PRC base entities
wire_body	*PtrTopology*	(Required) wire edge shall be of type *PRC_TYPE_TOPO_WireEdge* or *PRC_TYPE_TOPO_Edge*

8.9.19 *PRC_TYPE_TOPO_BrepData*

This is the main representation of solid and surface topology (which is not highly compressed).

Table 196 — *PRC_TYPE_TOPO_BrepData*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_BrepData*
base	*ContentBody*	(Required) Common data for PRC base entities
number_of_connex	*UnsignedInteger*	(Required) Number of connex entities in this B-rep
connex	Array <*PtrTopology*> [**number_of_connex**]	(Required) Array of connex entities in this B-rep; each entry shall be non-NULL and reference a PRC_TYPE_TOPO_Connex entity
bounding_box	*BoundingBox*	(Optional; if **ContentBody: bounding_box_behavior** is PRC_BODY_BBOX_CADData) Optional bounding box; the required field ContentBody defines the Boolean flag indicating the presence of this bounding box

8.9.20 *PRC_TYPE_TOPO_SingleWireBodyCompress*

This represents a single wire body stored in compressed format.

Curve_tolerance is the tolerance used to approximate the curve of a single wire body. See 5.7.

Table 197 — *PRC_TYPE_TOPO_SingleWireBodyCompress*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_TOPO_SingleWireBodyCompress*
base	*ContentBody*	(Required) Common data for PRC base entities
curve_tolerance	*Double*	(Required) **curve_tolerance** is the tolerance that has been used to approximate the curve of a single wire body.
compressed_curve	*CompressedCurve*	(Required)

8.9.21 *PRC_TYPE_TOPO_BrepDataCompress*

8.9.21.1 General

This represents manifold brep data stored in compressed format. In contrast to PRC_TYPE_TOPO_BrepData, geometrical and topological entities are not shared with other bodies even if they belong to the same topological context.

— **brep_data_compressed_tolerance** represents the tolerance used for the brep data approximation.

— **number_of_bits_to_store_reference** represents the number of bits written in the file for the following integers

— **number_vertex_references** represents the number of referenced verticies in the brep; see CompressedVertex

— **number_edge_references** represents the number of referenced edges in the brep; see CompressedCurve

The number of faces in the compressed brep is calculated as the number of faces in all of the shells in all of the connex entities.

Table 198 — *PRC_TYPE_TOPO_BrepDataCompress*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_TOPO_BrepDataCompress
base	*ContentBody*	(Required) Common data for PRC base entities
brep_data_compressed_tolerance	*Double*	(Required) **brep_data_compressed_tolerance**
number_of_bits_to_store_reference	*NumberOfBitsThenUnsignedInteger*	(Required) **number_of_bits_to_store_reference**
number_vertex_references	*UnsignedIntegerWithVariableBitNumber*	(Required) **number_vertex_references**
number_edge_references	*UnsignedIntegerWithVariableBitNumber*	(Required) **number_edge_references**
single_connex	*Boolean*	(Required) TRUE if this brep consists of one connex entity with one shell
single_connex	*CompressedShell*	(Optional; if **single_connex** is TRUE) Single compressed shell
multi_connex	*MultipleCompressedConnex*	(Optional; if **single_connex** is FALSE) Multiple compressed connex stored in file
base_topology_data	Array <*BaseTopology*>[	(Required) Base topology data for each of the faces in the compressed brep data; the order of elements in this array corresponds to the order the faces are encountered in the scanning of connex/shell data within the compressed brep

8.9.21.2 *MultipleCompressedConnex*

This represents the data stored when the compressed brep data contains multiple connex entities or multiple shells within a single connex entity.

Table 199 — *MultipleCompressedConnex*

Name	Data Type	Data Description
number_of_connex	*UnsignedInteger*	(Required) Number of connex entities
connex	Array <*CompressedConnex*> [**number_of_connex**]	(Required) Array of compressed connex entities

8.9.21.3 *CompressedConnex*

This represents all of the shells within a compressed connex entity.

Table 200 —*CompressedConnex*

Name	Data Type	Data Description
number_of_shells	*UnsignedInteger*	(Required) Number of shell entities
shells	Array <*CompressedShell*> [**number_of_shells**]	(Required) Array of compressed shells within a connex entity

8.9.21.4 *CompressedShell*

This represents the compressed data for a single shell.

Table 201 — *CompressedShell*

Name	Data Type	Data Description
single_face	*Boolean*	(Required) True if there is only a single face in the shell
number_of_faces	*NumberOfBitsThenUnsignedInteger*	(Optional); if **single_face** is TRUE) Number of faces in the shell if there is more than a single face
faces	Array <*CompressedFace*> [number_of_faces]	(Required) Array of faces within the shell..
Required	Array <*Boolean*> [**number_of_faces**]	(Required) Array of Boolean values indicating if the face is an iso face (TRUE) or not (FALSE). This is in the same order as the previous array of compressed faces.

8.9.21.5 *Compressed Face*

8.9.21.5.1 General

This represents the data for a single compressed face. There are two types of compressed faces: iso-faces and ana-face. An iso-face is a surface trimmed by four iso-parametric curves. If a face is not an iso-face, it is an ana-face.

The types of iso-faces include *ISO_PLANE*, *ISO_CYLINDER*, *ISO_CONE*, *ISO_SPHERE*, *ISO_TORUS*, and *ISO_NURBS*. Except for an *ISO_NURBS*, an iso-face is described using two curves. For example, an *ISO_CYLINDER* is described with 4 trimming curve, 2 lines and 2 circle. Using the first line and the first circular arc of the trimming loops enable to deduce the cylindrical surface and the two other trimming edge curves.

For both types of faces, all curves used to define or trim them are 3D. Therefore, surface parameterizations are not described and are set arbitrarily.

A curve is implicit if it is computed using iso-face properties. For example, the third and fourth curves in an *ISO_CYLINDER* are implicit curves. Other curves are explicit. Explicit curves are stored with the same orientation as the loop that references them first in the compressed B-rep data.

If the face belongs to a shell and the curve was already serialized by a neighbor's face, it is referenced with an index.

8.9.21.5.2 Enumeration of Compressed Entity Types

The following lists the types for compressed entities.

Table 202 — Enumeration of Compressed Entity Types

Value	Type Name	Type Description
0	*PRC_HCG_NewLoop*	Intermediate trimming loop on an AnaFace
1	*PRC_HCG_EndLoop*	Last trimming loop on an AnaFace
2	*PRC_HCG_IsoPlane*	Plane trimmed by iso parametric curves
3	*PRC_HCG_IsoCylinder*	Cylinder trimmed by iso parametric curves
4	*PRC_HCG_IsoTorus*	Torus trimmed by iso parametric curves
5	*PRC_HCG_IsoSphere*	Sphere trimmed by iso parametric curves
6	*PRC_HCG_IsoCone*	Cone trimmed by iso parametric curves
7	*PRC_HCG_IsoNurbs*	Nurbs trimmed by iso parametric curves
8	*PRC_HCG_AnaPlane*	Plane trimmed with non-iso parametric curves
9	*PRC_HCG_AnaCylinder*	Cylinder trimmed with non-iso parametric curves
10	*PRC_HCG_AnaTorus*	Torus trimmed with non-iso parametric curves
11	*PRC_HCG_AnaSphere*	Sphere trimmed with non-iso parametric curves
12	*PRC_HCG_AnaCone*	Cone trimmed with non-iso parametric curves
13	*PRC_HCG_AnaNurbs*	Cone trimmed with non-iso parametric curves
14	*PRC_HCG_AnaGenericFace*	ana face lying on an uncompressed surface which can be of any type under *PRC_TYPE_SURF*
0	*PRC_HCG_Line*	Compressed line
1	*PRC_HCG_Circle*	Compressed circle
2	*PRC_HCG_BsplineHermiteCurve*	Compressed hermite bspline
12	*PRC_HCG_Ellipse*	Compressed ellipse; reserved for future use
13	*PRC_HCG_CompositeCurve*	Compressed composite

8.9.21.5.3 *PRC_HCG_IsoPlane*

The origin of the plane is the first vertex of the loop.

Table 203 — *PRC_HCG_IsoPlane*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_IsoPlane*
x	*Double*	(Required) X coordinate of unit plane normal
y	*Double*	(Required) Y coordinate of unit plane normal
positive_z	*Boolean*	(Required) TRUE if Z coordinate of unit plane normal is greater than 0.0; else FALSE
face	*ContentCompressedFace*	(Required) Boundary of compressed face

8.9.21.5.4 *PRC_HCG_IsoCylinder*

A conforming PRC Reader should recognize accordingly lines and circles from ContentCompressedFace to reconstruct a cylinder surface.

Table 204 — *PRC_HCG_IsoCylinder*

Name	Data Type	Data Description
Required	*CompressedEntityType*	(Required) *PRC_HCG_IsoCylinder*
face	*ContentCompressedFace*	(Required) Boundary of compressed face

8.9.21.5.5 *PRC_HCG_IsoTorus*

is_major_radius is TRUE if the first serialized circle corresponds to the major radius.

A conforming PRC Reader should recognize and classify circles from ContentCompressedFace to reconstruct a torus surface.

Table 205 — *PRC_HCG_IsoTorus*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_IsoTorus*
is_major_radius	*Boolean*	(Required) is_major_radius
face	*ContentCompressedFace*	(Required) Boundary of compressed face

8.9.21.5.6 *PRC_HCG_IsoSphere*

A conforming PRC Reader should recognize and classify two first circles of the loop from ContentCompressedFace to reconstruct radius and center of a sphere surface. The sphere is computed from two circles made by two iso parametric curves in two different directions. The curves are on the surface of the sphere.

Table 206 — *PRC_HCG_IsoSphere*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_IsoSphere*
face	*ContentCompressedFace*	(Required) Boundary of compressed face

8.9.21.5.7 *PRC_HCG_IsoCone*

A conforming PRC Reader should recognize accordingly lines and circles from ContentCompressedFace to reconstruct a cone surface.

Table 207 — *PRC_HCG_IsoCone*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_IsoCone*
face	*ContentCompressedFace*	(Required) Boundary of compressed face

8.9.21.5.8 *PRC_HCG_AnaPlane*

The origin of the plane is the first vertex of the loop.

Table 208 — *PRC_HCG_AnaPlane*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_AnaPlane*
x	*Double*	(Required) X coordinate of unit plane normal
y	*Double*	(Required) Y coordinate of unit plane normal
positive_z	*Boolean*	(Required) TRUE if Z coordinate of unit plane normal is greater than 0.0; else FALSE
face	*ContentCompressedFace*	(Required) Boundary of compressed face

8.9.21.5.9 *PRC_HCG_AnaCylinder*

The analytic cylinder axis is defined from point_on_axis and cylinder_axis_direction. The cylinder radius is computed using loop vertices to obtain an average radius when projected onto the axis.

Table 209 — *PRC_HCG_AnaCylinder*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_AnaCylinder*
face	*ContentCompressedFace*	(Required) Boundary of compressed face
point	*CompressedPoint*	(Required) Point on cylinder axis
direction	*CompressedPoint*	(Required) Direction of cylinder axis

8.9.21.5.10 *PRC_HCG_AnaTorus*

x_axis and **y_axis** define the torus placement.

x_axis length is equal to the major torus radius.

y_axis length is equal to the minor torus radius.

Table 210 — *PRC_HCG_AnaTorus*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_AnaTorus*
face	*ContentCompressedFace*	(Required) Boundary of compressed face
center	*CompressedPoint*	(Required) Torus center
x_axis	*CompressedPoint*	(Required) Torus x_axis
y_axis	*Vector3D*	(Required) Torus y_axis

8.9.21.5.11 *PRC_HCG_AnaSphere*

The radius of the sphere is computed using the first vertex of the loop and **sphere_center**.

Table 211 — *PRC_HCG_AnaSphere*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_AnaSphere*
face	*ContentCompressedFace*	(Required) Boundary of compressed face
sphere_center	*CompressedPoint*	(Required) Sphere center

8.9.21.5.12 *PRC_HCG_AnaCone*

axis_point and apex_point are used to compute the z-axis. The cone origin and the semi-angle correspond to the loop vertex that is furthest from the z-axis.

Table 212 — *PRC_HCG_AnaCone*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_AnaCone*
face	*ContentCompressedFace*	(Required) Boundary of compressed face
axis_point	*CompressedPoint*	(Required) Axis point
apex_point	*CompressedPoint*	(Required) Apex point

8.9.21.5.13 *PRC_HCG_AnaGenericFace*

This represents the data stored for any analytic face where the surface data of the face are not compressed. In this case, the trimming data for the face is saved and a regular surface description is saved, if there is one.

In cases where no surface data has been saved, the entity type *PRC_TYPE_ROOT* is used instead of PRC_TYPE_SURF.

Table 213 — *PRC_HCG_AnaGenericFace*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_AnaGenericFace*
Face	*ContentCompressedFace*	(Required) Boundary of compressed face
surface_definition	*PRC_TYPE_SURF or PRC_TYPE_ROOT*	(Required) Surface definition

8.9.21.5.14 *PRC_HCG_IsoNurbs*

8.9.21.5.14.1 General

An iso-NURBS surface is a face trimmed by four iso-parametric curves. This type of iso-face is special because it is not stored using the first two curves (see 8.9.21.5).

In this case, the surface is stored using the following information:

— **orientation_surface_with_shell** is defined in FacesInShell.

— **orientation_loop_with_surface** is defined in *PRC_TYPE_TOPO_Loop*.

— **sense_array** is the correspondence between the surface natural boundaries, as described in CompressedNurbs, and the trim curves.

The two first boolean values describe where the first curve is. Their possible values are:

— FALSE FALSE : the first curve is on Umin.

— FALSE TRUE : the first curve is on Vmin.

— TRUE FALSE : the first curve is on Umax.

— TRUE TRUE : the first curve is on Vmax.

If the last boolean value is FALSE, the sense is the same as if the first curve is on Umin, the second curve is on Vmin, and the third curve is on Umax. If the last boolean value is TRUE, the reverse sense is applied.

The four curve types are stored as reference (identifier), line, circle, or other (iso boundary of surface).

number_of_bits_to_store_reference is described in *PRC_TYPE_TOPO_BrepDataCompress*.

reference_indice is described in *RefOrCompressedCurve*.

Table 214 — *PRC_HCG_IsoNurbs*

Name	Data Type	Data Description
	CompressedEntityType	(Required) PRC_HCG_IsoNurbs
orientation_surface_with_shell	*Boolean*	(Required) Orientation of surface with shell
orientation_loop_with_surface	*Boolean*	(Required) Orientation of loop with surface
sense_array	Array <*Boolean*>[3]	(Required) Three values of sense array
Surface	*CompressedNurbs*	(Required) Compressed Nurbs surface

Table 214 *(continued)*

Curves	Array<*IsoNurbsTrimCurve*>[4]	(Required) Trimming information for four boundary curves
LoopVertex0	*CompressedVertex*	(Optional if **Curves[0] is_referenced** AND **Curves[1] is_referenced**) The shared vertex of the first and second boundry curves
LoopVertex1	*CompressedVertex*	(Optional if **Curves[1] is_referenced** AND **Curves[2] is_referenced**) The shared vertex of the second and third boundry curves
LoopVertex2	*CompressedVertex*	(Optional if **Curves[2] is_referenced** AND **Curves[3] is_referenced**) The shared vertex of the third and fourth boundry curves
LoopVertex3	*CompressedVertex*	(Optional if **Curves[3] is_referenced** AND **Curves[0] is_referenced**) The shared vertex of the fourth and first boundry curves

8.9.21.5.14.2 *IsoNurbsTrimCurve*

Table 215 — *IsoNurbsTrimCurve*

Name	Data Type	Data Description
is_referenced	*Boolean*	(Required) Is referenced
trim_curve_index	*UnsignedIntegerWithVariableBitNumber*	(Optional if **is_referenced** is TRUE) Index of trim curve
trim_curve	*IsoNurbsTrimCrv*	(Optional if **is_referenced** is FALSE) trim curve

8.9.21.5.14.3 *IsoNurbsTrimCrv*

8.9.21.5.14.4 General

Save the actual trim curve data on an iso NURBS surface.

Table 216 — *IsoNurbsTrimCrv*

Name	Data Type	Data Description
iso_boundary	*Boolean*	(Required) boolean with value is set to TRUE when the trimming curve is not a circle or a line.
is_a_circle	*Boolean*	(Optional; if **iso_boundary** is FALSE) boolean value is set to TRUE if the trimming curve is a circle. Else the trimming curve is a line
compressed_circle	*CompressedCircle*	(Optional; if **is_a_circle** is FALSE) compressed circle

8.9.21.5.15 *PRC_HCG_AnaNurbs*

Table 217 — *PRC_HCG_AnaNurbs*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_AnaNurbs*
compressed_face	*ContentCompressedFace*	(Required) Boundary of compressed face
compressed_surface	*CompressedNurbs*	(Required) Compressed nurbs surface

8.9.21.6 *CompressedNurbs*

8.9.21.6.1 General

This defines the storage of a compressed NURBS surface.

A compressed NURBS surface is defined by the following data.

See *CompressedControlPoints*, *CompressedKnotVector*, and *CompressedMultiplicities* for a description of converting from a NURBS surface to a compressed NURBS surface.

Table 218 — Input data in Nurbs

Name	Description
Number_ccpt_in_u	Number of control points in u
Number_ccpt_in_v	Number of control points in v
Number_knots_in_u	Number of knots in U
Number_knots_in_v	Number of knots in V
Is_closed_in_u	Boolean flag indicating a surface closed in u
Is_closed_in_v	Boolean flag indicating a surface closed in v
Ccpt	Two dimensional array of control points
Ccpt_type	Two dimensional array of integers defining the type of control point; see **type_param** for legal values
Cknot_u	Array of knots in U
Cknot_v	Array of knots in V
Mult_u	Array of multiplicities at the knots in U
Mult_v	Array of multiplicities at the knots in V
Is_rational	Boolean flag indicating if the surface is a rational surface (TRUE) and thus has an optional array of weights at the control points
Weight	Array of weights if the surface is rational

brep_data_compressed_tolerance is the tolerance for approximation as described in *PRC_TYPE_TOPO_BrepDataCompress*.

The following are used for stored compressed control points:

— **number_of_bits_for_isomin** is the number of bits used to store first row and column of control points

— **number_of_bits_for_rest** is the number of bits to store the remainder of the control points

The following are used for stored compressed knot values in U or V:

— **number_bit_parameter** is the number of bits used to store knots

— **tolerance_parameter** is the tolerance used to store knots

The following are used for stored weights of rational NURBS surfaces:

— **number_bit_weight** is the number of bits to store weights

— **weight_tolerance** is the tolerance used to store weights

A conforming PRC Writer shall ensure that all these numbers and tolerances used to store control points, knots and weights are appropriately chosen so that the overall brep_data_compressed_tolerance is respected. The algorithms to ensure this are not part of this specification.

type_param can have one of the following values:

— 0 for uniform parameterization.

— 1 for non-uniform parameterization.

— 2 for pseudo-uniform parameterization, meaning that the parameterization is uniform except for extremities, mostly coming from a trim applied uniformly.

The following are used in the definition of a compressed nurbs surface:

Nurbs_tolerance = **brep_data_compressed_tolerance** / 5.0

Number_stored_knots_in_u = **number_of_knots_in_u** - 2

Number_stored_knots_in_v = number_of_knots_in_v – 2

Number_bits_u = (degree_in_u ? ceil[log(degree_in_u + 2) / log(2)] : 2)

Number_bits_v = (degree_in_v ? ceil[log(degree_in_v + 2) / log(2)] : 2)

tolerance_parameter = 1./ 2^(number_bit_parameter -1)

Table 219 — *CompressedNurbs*

Name	Data Type	Data Description
degree_in_u stored	*UnsignedIntegerWithVariableBitNumber*	(Required) **degree_in_u** stored with 5 bits
degree_in_v	*UnsignedIntegerWithVariableBitNumber*	(Required) **degree_in_v** stored with 5 bits
number_stored_knots_in_u	*UnsignedIntegerWithVariableBitNumber*	(Required) **number_stored_knots_in_u** stored using 16 bits; the integer represents the number of knots in u that are stored in the **knot_u** array
mult_u	Array <*CompressedMultiplicities*> [**number_stored_knots_in_u**]	(Required) Array mult_u of data describing the knot multiplicities in U for **number_stored_knots_in_u** knots

Table 219 *(continued)*

number_stored_knots_in_u	*UnsignedIntegerWithVariableBitNumber*	(Required) **number_stored_knots_in_v** stored using 16 bits; the integer represents the number of knots in u that are stored in the **knot_u** array
mult_v	ArrayOf[*CompressedMultiplicities*> [**number_stored_knots_in_u**]	(Required) Array **mult_v** of data describing the knot multiplicities in V for **number_stored_knots_in_v** knots
is_closed_in_u	*Boolean*	(Required) **is_closed_in_u**
is_closed_in_v	*Boolean*	(Required) **is_closed_in_v**
number_of_bits_for_isomin	*UnsignedIntegerWithVariableBitNumber*	(Required) **number_of_bits_for_isomin** stored using 20 bits
number_of_bits_for_rest	*UnsignedIntegerWithVariableBitNumber*	(Required) **number_of_bits_for_rest** stored using 20 bits
compressed_control_points	*CompressedControlPoints*	(Required) Compressed control points for the surface
knot_vector_u	*CompressedKnotVector*	(Required) Save type_param_u, number_knots_u, and knots_u
knot_vector_v	*CompressedKnotVector*	(Required) Save type_param_v, number_knots_v, and knots_v
is_rational	*Boolean*	(Required) **is_rational**
weights	*CompressedWeights*	(Optional; if **is_rational** is TRUE) Save weights

8.9.21.6.2 *CompressedMultiplicities*

This defines an array of data stored to define the multiplicity of knots at each knot in the knot array for either U or V parameter.

number_stored_knots is either **Number_stored_knots_in_u** or **Number_stored_knots_in_v**

number_bits is either **number_bits_u** or **number_bits_v**.

Multiplicity is either **mult_u** or **mult_v**.

For each of the knots (0 <= i < number_stored_knots), the following data is stored

— A Boolean flag indicating if additional data is stored

  — If i > 0 **multiplicity_is_stored** = (multiplicity[i] == multiplicity[i-1]);

  — if i is 0 **multiplicity_is_stored** = (multiplicity[i] == 1)

— Optionally store the multiplicity at this knot using number_bit bits

Table 220 — *CompressedMultiplicities*

Name	Data Type	Data Description
multiplicity_is_stored	*Boolean*	(Required) **multiplicity_is_stored**
multiplicity	*UnsignedIntegerWithVariableBitNumber*	(Optional; if **multiplicity_is_stored** is TRUE) **multiplicity**[i] stored using number_bits

8.9.21.6.3 *CompressedControlPoints*

8.9.21.6.3.1 General

nurbs_tolerance describes the tolerance used to approximate the original nurbs surface. It ensures that each point on the compressed nurbs surface is at a distance of the original surface less than nurbs_tolerance.

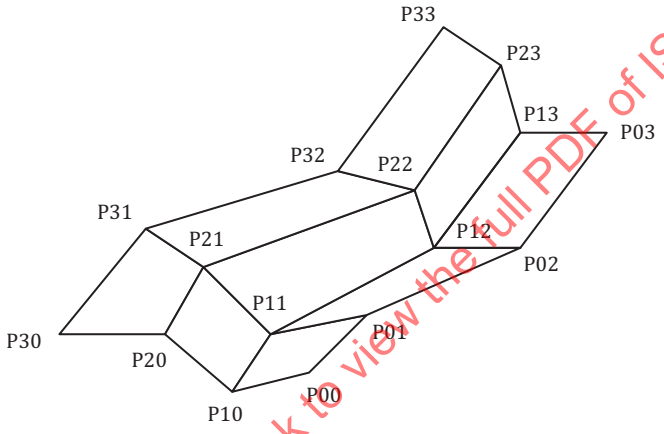


Figure 3 — CompressedControlPoints

compressed_control_point P is a two dimension array containing double values that allow to recompute x, y, z values of consecutive points. At start, the surface is copied into a working structure which is updated step by step. $P_{0,0}$ coordinates are pushed in compressed_control_point and into the working structure, thus $P_{0,0(compressed)} = P_{0,0(working\_structure)} = P_{0,0}$. Then $P_{1,0} - P_{0,0}$ is computed and each vector component is approximated to the nearest multiple of the nurbs_tolerance. This truncated vector $(P_{1,0} - P_{0,0(working_structure)})$_truncated is pushed into compressed_control_point. Then $P_{1,0(working\_structure)}$ is computed as follows : $P_{1,0(working_structure)} = (P_{1,0} - P_{0,0(working_structure)})$_truncated + $P_{0,0(working\_structure)}$. This point is re-injected into the working structure to avoid error propagation. In the same way, $(P_{2,0} - P_{1,0(working_structure)})$_truncated is pushed into compressed_control_point. $P_{2,0(working\_structure)}$ is computed and re-injected in the working structure. The same formula applies for each $P_{i,0}$ and then $P_{0,j}$ control points.

Internal compressed control points are also computed with previously stored points. for each i and for each j, $P_{i,j(compressed)}$ is computed

$$\vec{V} \leftarrow P_{i-1,j} - P_{i-1,j-1}$$

$$\vec{U} \leftarrow P_{i,j-1} - P_{i-1,j-1}$$

$$\vec{N} \leftarrow \vec{U} \wedge \vec{V}$$

$$P_{i,j(compressed)} \leftarrow P_{i,j} - \left(P_{i-1,j-1} + \vec{V} + \vec{U}\right)$$

Four cases are considered:

— If $P_{i,j(compressed)}$ length is less than nurbs_tolerance, control_point_type is set to zero and no value is pushed into compressed_control_point.

$\left(P_{i-1,j-1} + \vec{V} + \vec{U}\right)$ is reinjected in the working structure to replace $P_{i,j}$.

Otherwise, $P_{i,j}$(compressed) is evaluated in the following local coordinate system :

$$P_{i,j} \leftarrow \left(P_{i,j(compressed)} \cdot \frac{\vec{U}}{\|\vec{U}\|}, P_{i,j(compressed)} \cdot \left(\frac{\vec{N}}{\|\vec{N}\|} \wedge \frac{\vec{U}}{\|\vec{U}\|} \right), P_{i,j(compressed)} \cdot \frac{\vec{N}}{\|\vec{N}\|} \right)$$

— if $x^2 + y^2$ value is less than nurbs_tolerance^2, control_point_type is set to 1 and z is added in compressed_control_point.

— else if the z component length is less than nurbs_tolerance, control_point_type is set to 2 and the coordinates x and y are added in compressed_control_point.

— else x, y and z are stored in compressed_control_point. control_point_type is set to 3. If one of the vectors V, U, or N has a length less that 1e-12, this case is systematically used.

P is a two dimensional array of the 3D control points for a compressed Nurbs surface.

When linearized, the data is stored in the following order

— The corner point P[0][0] is stored as a Vector_3d (i.e. three doubles);

— The remainder of the first row of control points is stored as Point3DWithVarBitNumber

— The remainder of the first column of control points is stored as Point3DwithVarBitNumber

—The remainder of the matrix is stored by row (i.e. for a given i from 1 to number_ccpts_in_u, save the row of control points from j=1 to number_ccpts_in_v. For each point

— Save the type of control point

— If the type is 1, save the z coordinate of the control point

— If the type is 2 save the x and y coordinates of the control point

— If the type is 3 save the x, y, and z coordinates of the contol point

All Point3DWithVariableBitNumber data is written using Nurbs_tolerance and (number_of_bits_for_isomin + 1).

Table 221 — *CompressedControlPoints*

Name	Data Type	Data Description
p00	*Vector3d*	(Required) P[0][0]
ccpt_in_v	Array <*Point3DwithVariableBitNumber*> [**number_ccpt_in_v**]	(Required) P[0][j] 1 <= j < **number_ccpt_in_v**
ccpt_in_u	Array <*Point3DwithVariableBitNumber*> [**number_ccpt_in_u**]	(Required) P[i][0] 1 <= i < **number_ccpt_in_u**
ccpt_interior	Array <*InteriorCompressedControlPoints*> []	(Required) Array of data describing interior compressed control points. The points are saved using the order in the previous pseudo code

8.9.21.6.3.2 *InteriorCompressedControlPoints*

This represents the data stored for each interior compressed control point.

All DoubleWithVariableBitNumber data is written using Nurbs_tolerance and (number_of_bits_for_rest + 1).

The array of interior compressed control points is a two dimensional array written as

```
for ( i = 1; i < number_ccpt_in_u; i++) {
    for (j  = 1; j < number_ccpt_in_v; j++) {
        write P[i][j]
    }
}
```

Table 222 — *InteriorCompressedControlPoints*

Name	Data Type	Data Description
type	*UnsignedIntegerWithVariableBitNumber*	(Required) Ccpt_Type[i][j] Is written with 2 bits
P[i][j].z	*DoubleWithVariableBitNumber*	(Optional; if type is 1) P[i][j].z
P[i][j].x	*DoubleWithVariableBitNumber*	(Optional; if type is 2) P[i][j].x
P[i][j].y	*DoubleWithVariableBitNumber*	(Optional; if type is 2) P[i][j].y
P[i][j].x	*DoubleWithVariableBitNumber*	(Optional; if type is 3) P[i][j].x
P[i][j].y	*DoubleWithVariableBitNumber*	(Optional; if type is 3) P[i][j].y
P[i][j].z	*DoubleWithVariableBitNumber*	(Optional; if type is 3) P[i][j].z

8.9.21.6.4 *CompressedKnotVector*

8.9.21.6.4.1 General

The knot vectors are always between 0 and 1. The multiplicities are stored as described in the PRC File Format Specification. See 8.9.21.6.2. U knots are treated first, then V. Three types of knot parameterization are considered.

— If it is uniform, no parameter is stored in compressed_knot. This corresponds to type_param = 0.

— If it is pseudo uniform, the interval length between the two first parameters is computed and truncated using tolerance_parameter, and then stored into compressed_knot. Same for the two last parameters. This corresponds to type_param = 2.

— Otherwise, internal parameters are stored. For each internal parameter, a difference between the precedent compressed parameter is computed, truncated using tolerance_parameter and stored into compressed_knot. This corresponds to type_param = 1.

This represents the data stored for the knot vector of a compressed NURBS surface. The type_param and knot vector is saved for either the U or V parameter values.

Table 223 — *CompressedKnotVector*

Name	Data Type	Data Description
number_bit_parameter	*UnsignedIntegerWithVariableBitNumber*	(Required) **number_bit_parameter** is saved using 6 bits
is_uniform	*Boolean*	(Required) **is_uniform** is TRUE if type_param == 0
knots	*CompressedKnots*	(Optional; if **is_uniform** is FALSE)

8.9.21.6.4.2 *CompressedKnots*

This represents saving the knots (either knots_u or knots_v) for a compressed NURBS.

Table 224 — *CompressedKnots*

Name	Data Type	Data Description
???	*Boolean*	(Required) Type_param == 1
???	*Boolean*	(Optional; ??? is FALSE) Type_param == 2
compressed_knots	Array<*CompressedKnot*>[]	(Required) Save the array of compressed knots

8.9.21.6.4.3 *CompressedKnot*

This represents a single entry in an array of compressed knots.

— The number of elements in the array is given by number_knots

— The *DoubleWithVariableBitNumber* is written using tolerance_parameter and **number_bit_parameter** + 1

— The format of the data is either Double or DoubleWithVariableBitNumber in the file depending upon the Boolean test (**number_bit_parameter** > 30).

The array may be either an array of knot_u or knot_v defining the compressed NURBS surface.

Table 225 — *CompressedKnot*

Name	Data Type	Data Description
knot	*Double*	(Optional; if **number_bit_parameter** > 30) **knot**[i]
knot	*DoubleWithVariableBitNumber*	(Optional; if **number_bit_parameter** <= 30) **knot**[i]

8.9.21.6.5 *CompressedWeights*

This represents the data stored for the weights of a compressed NURBS surface.

— The number of entries to store is (number_ccpt_in_u * number_ccpt_in_v)

— The DoubleWithVariableBitNumber is written using weight_tolerance and **number_bit_weight** + 1

— The format of the data is either Double or DoubleWithVariableBitNumber in the file depending upon the Boolean test (number_bit_weight > 30)

Table 226 — *CompressedWeights*

Name	Data Type	Data Description
number_bit_weight	*UnsignedIntegerWithVariableBitNumber*	**number_bit_weight** save with 6 bits
weights	Array <*Double*>[number_ccpt_in_u * number_ccpt_in_v]	(Optional; if number_bit_weight > 30) Save weight array (without compression)
tolerance	*Double*	(Optional; if number_bit_weight <= 30) Weight tolerance
compressed_weights	Array <*DoubleWithVariableBitNumber*> [number_ccpt_in_u * number_ccpt_in_v]	(Optional; if number_bit_weight <= 30) Save weight array using compression

8.9.21.7 *ContentCompressedFace*

8.9.21.7.1 General

This represents the data for a compressed face. A compressed face is further classified by its trimming curves. An IsoFace is trimmed by four iso-parametric trimming curves. An AnaFace is trimmed by any other combination of trimming curves.

Vertex loops are used to represent a loop consisting of a single vertex, such as might exist on the apex of a cone, or a sphere touching a plane. They are represented by a degenerate line which has identical start and end vertices.

orientation_surface_with_shell describes the orientation of the surface normal with respect to the shell. See PRC_TYPE_TOPO_Shell.

orientation_loop_with_surface describes the orientation of a loop with respect to a surface normal. See PRC_TYPE_TOPO_Loop.

Table 227 — *ContentCompressedFace*

Name	Data Type	Data Description
orientation_surface_with_shell	*Boolean*	(Required) Orientation_surface_with_shell
iso_face	*ContentCompressedIsoFace*	(Optional; if **is_an_iso_face** is TRUE) Save data for a compressed iso face
ana_face	*ContentCompressedAnaFace*	(Optional; if **is_an_iso_face** is FALSE) Save data for a face trimmed by analytic curves

8.9.21.7.2 *ContentCompressedIsoFace*

In the case of an IsoFace, the first and second trim curves are stored explicitly either by reference or actual data. Reference to the third or fourth trim curve is stored in case the actual data has been stored by an adjacent face. Otherwise, the third or fourth trim curve will have to be deduced from the first and second trim curves and a vertex representing the join between the third and fourth trim curves. The loop of an IsoFace can not be degenerate.

Table 228 — *ContentCompressedIsoFace*

Name	Data Type	Data Description
orientation_loop_with_surface	*Boolean*	(Required) orientation_loop_with_surface
first_trim_curve	*RefOrCompressedCurve*	(Required) first trim curve on face
second_trim_curve	*RefOrCompressedCurve*	(Required) second trim curve on face
third_trim_curve_is_not_yet_saved	*Boolean*	(Required) TRUE if **third_trim_curve_is_not_yet_saved**
third_trim_curve	*UnsignedIntegerWithVariableBitNumber*	(Optional; if **third_trim_curve_is_not_yet_saved** if FALSE) Index of the third trim curve
fourth_trim_curve_is_not_yet_saved	*Boolean*	(Required) TRUE if fourth_trim_curve_is_not_yet_saved
fourth_trim_curve	*UnsignedIntegerWithVariableBitNumber*	(Optional; if fourth_trim_curve_is_not_yet_saved is FALSE) Index of the fourth trim curve
common_third_fourth_vertex	*CompressedVertex*	(Optional; if third_trim_curve_is_not_yet_saved is TRUE and fourth_trim_curve_is_not_yet_saved is TRUE) Save the common_third_fourth_vertex

8.9.21.7.3 *ContentCompressedAnaFace*

8.9.21.7.3.1 General

If the AnaFace has trimming boundaries, all of the trimming loops are stored as an array of loops with the last loop in the array having type PRC_HCG_EndLoop and all other loops having type PRC_HCG_NewLoop.

If the AnaFace is defined by a torus and all of the trimming loop are vertex trimming loops (e.g. the loop consists of a single degenerate line), a point on the torus far from degeneracy is stored to indicate the interior of the face (this corresponds to the interior part of the spindle“ side versus the exterior part of the spindle).

Table 229 — *ContentCompressedAnaFace*

Name	Data Type	Data Description
is_trimmed	*Boolean*	(Required) TRUE if surface is trimmed; else FALSE
trim_loop	Array <*AnaFaceTrimLoop*>[]	(Optional; if is_trimed is TRUE) Array of trimming loops on the face. The last loop in the array is of type *PRC_HCG_EndLoop*; all others are of type *HPC_HCG_TOPO_NewLoop*
point_on_torus	*CompressedPoint*	(Optional; if **all_loops_are_vertex_loops** is TRUE AND **surface_type** is *PRC_HCG_AnaTorus* AND is_trimmed is TRUE) Point_on_torus

8.9.21.7.3.2 *AnaFaceTrimLoop*

This represents the trimming curves for a loop on a compressed AnaFace.

The last loop on the face is of type of *PRC_HCG_EndLoop*. Other loops on the face are of type *PRC_HCG_NewLoop*.

Table 230 — *AnaFaceTrimLoop*

Name	Data Type	Data Description
loop_surface_orientation	*Boolean*	(Required) Orientation of loop with surface
curves	*ArrayOf[RefOrCompressedCurve]*	(Required) Array of curves in the trimming loop
curve_is_not_already_stored	*Boolean*	(Required) Boolean value is always TRUE; this represents a boolean flag that means curve_is_NOT_already_stored is TRUE; it is used when reading a PRC File to signal the end of the curves in a loop and the end of all loops.
type	*CompressedEntityType*	(Required) Will be PRC_HCG_EndLoop for last loop and PRC_HCG_NewLoop for all other loops

8.9.21.8 *RefOrCompressedCurve*

The flag **curve_is_not_already_stored** indicates if the trim curve has already been stored in the compressed brep data. If the curve has already been stored, the index of the curve is stored in the file; otherwise, a compressed version of the trim curve is stored.

The index is stored with a variable number of bits indicated by **number_of_bits_to_store_reference**. See 8.9.20 for a definition of this number.

Table 231 — *RefOrCompressedCurve*

Name	Data Type	Data Description
curve_is_not_already_stored	*Boolean*	(Required) curve_is_not_already_stored
index_compressed_curve	*UnsignedIntegerWithVariableBitNumber*	(Optional; if **curve_is_not_already_stored** is FALSE) Index to the already stored compressed curve data
compressed_curve	*CompressedCurve*	(Optional; if **curve_is_not_already_stored** is TRUE) Store the compressed curve data.

8.9.21.9 *CompressedCurve*

8.9.21.9.1 General

A compressed curve is one of *PRC_HCG_Line*, *PRC_HCG_Circle*, *PRC_HCG_BsplineHermiteCurve* or *PRC_HCG_CompositeCurve*.

The curve type PRC_HCG_Ellipse is reserved for future use.

8.9.21.9.2 *PRC_HCG_Line*

The representation of a compressed line (*PRC_HCG_Line*) is context dependent.

If a compressed line is part of a compressed face, the compressed line is represented by a pair of start/end vertices; otherwise it is represented by a pair of start/end points.

curve_trimming_face is TRUE if this compressed line is part of a *PRC_TYPE_TOPO_BrepDataCompress*; it is FALSE if this compressed line is a part of a *PRC_TYPE_TOPO_SingleWireBodyCompress*.

Table 232 — *PRC_HCG_Line*

Name	Data Type	Data Description
	CompressedEntityType	(Required) *PRC_HCG_Line*
start_end_data	*StartEndData*	(Required) Save the start/end trim data

8.9.21.9.3 *PRC_HCG_Circle*

8.9.21.9.3.1 General

The representation of a compressed circle (*PRC_HCG_Circle*) is context dependent.

The data stored for a compressed circle depends upon the context it is used in:

— **curve_trimming_face** is TRUE to indicate that the circle is used as part of the trimming data for a face (i.e as part of a *PRC_TYPE_TOPO_BrepDataCompress*) or is FALSE to indicate that the circle is not part of a face (i.e. it is used as part of *PRC_TYPE_TOPO_SingleWireBodyCompress*)

— **compressed_iso_spline** is TRUE if the circle is being used as the trim boundary of an *PRC_HCG_IsoNurbs*; otherwise it is FALSE

In addition, the data stored for a compressed circle depends on the geometry of the circle. A circular arc with angle 0.0, π , or 2π will have different data. The **particular_circle** Boolean flag indicates this.

Table 233 — *PRC_HCG_Circle*

Name	Data Type	Data Description
type	*CompressedEntityType*	(Optional; if compressed_iso_spline is FALSE) PRC_HCG_Circle
is_particular_circle	*Boolean*	(Required) is_particular_circle
particular_circle	*ParticularCircle*	(Optional; if **is_particular_circle** is TRUE) Circular arc with angle 0, π, or 2 π
general_circle	*GeneralCircle*	(Optional; if **is_particular_circle** is FALSE) Not a special circle

8.9.21.9.3.2 *ParticularCircle*

This is the data that is stored if the compressed circle is a special case (circular arc with angle 0, π, or 2 π).

Full_circle is TRUE if the start point and end point of the trim curve are identical (to within tolerance).

Table 234 — *ParticularCircle*

Name	Data Type	Data Description
full_circle	*Boolean*	(Required) full_circle
start_end_data	*StartEndData*	(Optional; if **compressed_iso_spline** is FALSE) Save the start/end trim data
center	*CompressedPoint*	(Optional; if **full_circle** is TRUE) Center of circle
normal_plane	*CompressedPoint*	(Optional; if **full_circle** is TRUE) Normal to plane of circle
middle_of_arc	*CompressedPoint*	(Optional; if **full_circle** is FALSE) Middle point on circular arc

8.9.21.9.3.3 *GeneralCircle*

This is the data that is stored for a general circle or circular arc.

Table 235 — *GeneralCircle*

Name	Data Type	Data Description
start_end_data	*StartEndData*	(Required) Save the start/end trim data
center	*CompressedPoint*	(Required) Center of circle
circle_angle	*Boolean*	(Required) circle_angle > π

8.9.21.9.4 *PRC_HCG_BsplineHermiteCurve*

A compressed Hermite curve structure contains information to store a compact representation of a Bspline curve with degree 3.

— A Hermite curve is defined by the following data:

— Compression_tolerance is the tolerance that was used to approximate the original nurbs curve; see 8.9.21 or 8.9.20.

— **compressed_points** are the control points representing the Hermite curve

— **compressed_tangents** describe how internal points computed from tangents are compressed

— **point_number_bits** is the number of bits used to store each compressed control point coordinate if control points are stored using a variable number of bits; a conforming PRC Writer will obtain this number through the routine *GetNumberOfBitUsedToStoreUnsignedInteger (See 10.1)*

— **tangent_number_bits** is the number of bits used to store each compressed tangent coordinate if the tangents are stored using a variable number of bits

Start and end curve points are explicitly stored in the PRC File. Compressed_points (Ptc) and **compressed_tangents** (Tgtc) allow computation of the control polygon. **compressed_points** contains points on the curve stored with difference :

$$P_0 \leftarrow StartPt \qquad P_{3i} \leftarrow P_{3(i-1)} + \left(Pt_c[3i], Pt_c[3i+1], Pt_c[3i+2]\right) \qquad P_n \leftarrow EndPt$$

compressed_tangent contains curves' tangents at each Ptc. It is used to determine two controls points between each point on curve (P_i)

$$P_1 \leftarrow P_0 + \frac{\left(Tgt_c[0], Tgt_c[1], Tgt_c[2]\right)}{\left\|P_3 - P_0\right\|}$$

$$P_2 \leftarrow P_3 - \frac{\left(Tgt_c[3], Tgt_c[4], Tgt_c[5]\right)}{\left\|P_3 - P_0\right\|}$$

$$P_4 \leftarrow P_3 + \frac{\left(Tgt_c[3], Tgt_c[4], Tgt_c[5]\right)}{\left\|P_6 - P_3\right\|}$$

The Bspline knot values are implicit and computed using control points.

$$U_0 \leftarrow 0 \qquad U_i \leftarrow U_{i-1} + \left\|\vec{P}_{3i} - \vec{P}_{3(i-1)}\right\|$$

Multiplicities are implicitly 4 for start and end knot and 3 for internal knots.

Table 236 — *PRC_HCG_BsplineHermiteCurve*

Name	Data Type	Data Description
type	*CompressedEntityType*	(Required) PRC_HCG_BsplineHermiteCurve
start_end_data	*StartEndData*	(Required) Save the start and end trimming data as either vertices or points
number_bits	*UnsignedIntegerWithVariableBitNumber*	(Required) Number_bits is the number of bits used to store number_points; this number is stored with 4 bits
number_points	*UnsignedIntegerWithVariableBitNumber*	(Required) Number_points is the number of compressed control points
point_number_bits	*UnsignedIntegerWithVariableBitNumber*	(Required) Point_number_bits is the number of bits used to store compressed points; this number is stored with 6 bits
points	Array <*Vector3d*> [**number_points - 2**]	(Optional; if **point_number_bits**>30) Array of (**number_points** -2) compressed points
compressed-points	Array <*Point3DWithVariableBitNumber*> [**number_points - 2**]	(Optional; if **point_number_bits**<=30) Array of (**number_points** – 2) compressed points with variable number of bits
tangent_number_bits	*UnsignedIntegerWithVariableBitNumber*	(Required) tangent_number_bits is the number of bits used to store compressed tangents; this number is stored with 6 bits
tangents	Array <*Vector3d*>[**number_points**]	(Optional; if tangent_number_bits > 30) Save number_points compressed tangents
compressed_tangents	Array <*Point3DWithVariableBitNumber*> [**number_points**]	(Optional; if tangent_number_bits <= 30) Save number_points compressed tangents with variable number of bits

8.9.21.9.5 *PRC_HCG_CompositeCurve*

This entity define a compressed composite curve. It must not be used on edges or coedges.

Table 237 — *PRC_HCG_CompositeCurve*

Name	Data Type	Data Description
type	*CompressedEntityType*	(Optional; if is_compressed_iso_spline is FALSE) *PRC_HCG_CompositeCurve*
start_end_data	*StartEndData*	(Required) Save the start and end trimming data as either vertices or points
dimension	*UnsignedInteger*	(Required) Dimension of the compressed composite curve (either 2 or 3)
is_closed	*Boolean*	(Required) TRUE if the curve is closed; else FALSE

Table 237 *(continued)*

number_of_curves	*UnsignedInteger*	(Required) Number of curves in the composite
curves	Array <*CompressedCurve*> [**number_of_curves**]	(Required) Array of compressed curves

8.9.21.9.6 *StartEndData*

This data defines the start and end verticies/positions of a trim curve. It is context dependent (see 8.9.21.9.2 or 8.9.21.9.3).

Table 238 — *StartEndData*

Name	Data Type	Data Description
start_vertex	*CompressedVertex*	(Optional; if **curve_trimming_face** is TRUE) Start vertex
end_vertex	*CompressedVertex*	(Optional; if **curve_trimming_face** is TRUE) End vertex
start_point	*CompressedPoint*	(Optional; if **curve_trimming_face** is FALSE) Start point
end_point	*CompressedPoint*	(Optional; if **curve_trimming_face** is FALSE) End point

8.9.21.10 *CompressedVertex*

This represents a compressed vertex either as a compressed point or a reference to an already compressed point. Each compressed brep data serialization maintains an array of previously written vertices, starting at index 0.

number_of_bits_to_store_reference is the number of bits used to define a reference to compressed data and is described in PRC_TYPE_TOPO_BrepDataCompress.

Table 239 — *CompressedVertex*

Name	Data Type	Data Description
already_stored	*Boolean*	(Required) TRUE if vertex is NOT already stored
point_index	*UnsignedIntegerWithVariableBitNumber*	(Optional; if **already_stored** is FALSE) Index to the already stored compressed point data
point_data	*CompressedPoint*	(Optional; if **already_stored** is TRUE) Compressed point data.

8.9.21.11 *CompressedPoint*

This represents a compressed point. The representation of the compressed point in the PRC File may be either as a Point3DwithVariableBitNumber or as a *PRC_TYPE_TOPO_UniqueVertex* depending upon how many bits are necessary to represent the data.

When a PRC File Writer writes a compressed point, the number of bits necessary to store the point is calculated using the formula

Double dTol = **brep_data_compressed_tolerance** / 100.0;

Unsigned int uMaxCoordinate = MAX(fabs(x), fabs(y), fabs(z)) / dTol + 1;

Unsigned int uNbBits = GetNumberOfBitUsedToStoreUnsignedInteger(uMaxCoordinate);

If uNbBits is greater than 30, the compressed point is stored as a *PRC_TYPE_UniqueVertex*; otherwise it is stored as a Point3DwithVariableBitNumber.

Table 240 — *CompressedPoint*

Name	Data Type	Data Description
uNbBits	*UnsignedIntegerWithVariableBitNumber*	(Required) **uNbBits** is stored using 6 bits
point	*Point3DWithVariableBitNumber*	(Optional; if **uNbBits** <= 30) Compressed point data is stored as a 3D point with **uNbBits** bits and dTol tolerance
point	*Vector3d*	(Optional; if **uNbBits** > 30) Compressed point data is stored as a unique vertex

8.9.22 *PRC_TYPE_TOPO_WireBody*

8.9.23 References

8.9.23.1 General

Each curve, surface, or topological entity within an individual topological context is assigned an identifier which can be used to refer to it from other entities within the same topological context.

The first reference to an entity stores the actual data and generates an identifier for that entity. Subsequent references to that entity store only the identifier.

Note that when the actual data is stored, the first entry is an UnsignedInteger which indicates the type of data stored. *PRC_TYPE_ROOT* (0) is used to indicate that the entity corresponds to a NULL pointer and no additional data is saved. Otherwise, the integer will be one of the subtypes of curve, surface, or topology.

8.9.23.2 *PtrCurve*

If the Boolean flag is TRUE, the actual curve data is stored. Otherwise, the identifier of the curve is stored. The only legal curves are those represented by the base class *PRC_TYPE_CRV*.

Table 241 — *PtrCurve*

Name	Data Type	Data Description
is_referenced	*Boolean*	(Required) **is_referenced** is TRUE if identifier of the entity is stored; else FALSE (i.e. actual entity data stored)
curve	*PRC_TYPE_CRV*	(Optional; if **is_referenced** is FALSE) The actual data for the stored entity.
curve_identifier	*UnsignedInteger*	(Optional; if **is_referenced** is TRUE) Identifier of stored entity

8.9.23.3 *PtrSurface*

If the Boolean flag is TRUE, the actual surface data is stored. Otherwise, the identifier of the surface is stored. The only legal surfaces are those represented by the base class PRC_TYPE_SURF.

Table 242 — *PtrSurface*

Name	Data Type	Data Description
is_referenced	*Boolean*	(Required) **is_referenced** is TRUE if identifier of the entity is stored; else FALSE (i.e. actual entity data stored)
surface	*PRC_TYPE_SURF*	(Optional; if **is_referenced** is FALSE) The actual data for the stored entity.
surface_identifier	*UnsignedInteger*	(Optional; if **is_referenced** is TRUE) Identifier of stored entity

8.9.23.4 *PtrTopology*

If the Boolean flag is TRUE, the actual topological entity is stored. Otherwise, the identifier of the topological entity is stored. The only legal topological entities are those represented by the base class PRC_TYPE_TOPO.

Table 243 — *PtrTopology*

Name	Data Type	Data Description
is_stored	*Boolean*	(Required) **is_referenced** is TRUE if identifier of the entity is stored; else FALSE (i.e. actual entity data stored)
topo	*PRC_TYPE_TOPO*	(Optional; if **is_stored** is FALSE) The actual data for the stored entity.
topo_identifier	*UnsignedInteger*	(Optional; if **is_stored** is TRUE) Identifier of stored entity

8.10 Curve

8.10.1 Entity Types

Table 244 — Curve entity types

Type Name	Type Value	Referenceable
PRC_TYPE_CRV	*PRC_TYPE_ROOT* + 10	
PRC_TYPE_CRV_Base	*PRC_TYPE_CRV* + 1	
PRC_TYPE_CRV_Blend02Boundary	*PRC_TYPE_CRV* + 2	
PRC_TYPE_CRV_NURBS	*PRC_TYPE_CRV* + 3	
PRC_TYPE_CRV_Circle	*PRC_TYPE_CRV* + 4	
PRC_TYPE_CRV_Composite	*PRC_TYPE_CRV* + 5	
PRC_TYPE_CRV_OnSurf	*PRC_TYPE_CRV* + 6	
PRC_TYPE_CRV_Ellipse	*PRC_TYPE_CRV* + 7	
PRC_TYPE_CRV_Equation	*PRC_TYPE_CRV* + 8	
PRC_TYPE_CRV_Helix01	*PRC_TYPE_CRV* + 9	
PRC_TYPE_CRV_Hyperbola	*PRC_TYPE_CRV* + 10	
PRC_TYPE_CRV_Intersection	*PRC_TYPE_CRV* + 11	
PRC_TYPE_CRV_Line	*PRC_TYPE_CRV* + 12	
PRC_TYPE_CRV_Offset	*PRC_TYPE_CRV* +1 3	
PRC_TYPE_CRV_Parabola	*PRC_TYPE_CRV* + 14	
PRC_TYPE_CRV_PolyLine	*PRC_TYPE_CRV* + 15	
PRC_TYPE_CRV_Transform	*PRC_TYPE_CRV* + 16	

8.10.2 *PRC_TYPE_CRV*

Abstract type for curves.

8.10.3 *PRC_TYPE_CRV_Base*

8.10.3.1 General

Abstract type for all geometric curves. The following data is stored for all curve types.

8.10.3.2 *ContentCurve*

ContentCurve provides additional information about a curve such as its name and attributes, how it extends past its boundary (start and end points), and if it is a 2D or 3D curve.

is_3d_flag: this flag is set to TRUE if the curve is a 3D curve; otherwise it is FALSE. If a curve has a transformation, this flag is used to determine if it a 2D transformation or a 3D transformation (See 8.4.11)

Table 245 — *ContentCurve*

Name	Data Type	Data Description
has_base_geometry	*Boolean*	(Required) TRUE if base information is present; else FALSE
attribute_data	*AttributeData*	(Optional; if **has_base_geometry** is TRUE) Attributes attached to the geometric entity
name	*Name*	(Optional; if **has_base_geometry** is TRUE) Name attached to the geometric entity
id	*UnsignedInteger*	(Optional; if **has_base_geometry** is TRUE) Identifier in originating CAD system; this may not be used to reference entity within PRC File;
extend_type	*UnsignedInteger*	(Required) Indicates how the curve is extended; see 8.9.11
is_3d flag	*Boolean*	(Required) **is_3d flag**; TRUE if the curve is 3D, otherwise FALSE

8.10.4 *PRC_TYPE_CRV_Blend02Boundary*

This entity represents a U iso-parametric curve of a Blend02 surface (along its center curve direction) at either the surfaces v minimum or v maximum value. The parameterization of the curve is inherited from the Blend02 surface.

blend represents the Blend02 surface and shall not be NULL; it shall be of type *PRC_TYPE_SURF_Blend02*.

bound indicates which blend U iso-parametric boundary is used:

— 0 represents the first blend bound (v minimum).

— 1 represents the second blend bound (v maximum).

bounding_surface is the bounding surface that the *Blend02Boundary* curve lies on.

sense_of_bounding_surface is equal to the sense of the bounding surface used in the intersection curve.

A *Blend02Boundary* curve is always a 3D curve (i.e. is_3d must be TRUE).

A Transformation positions the curve in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 246 — *PRC_TYPE_CRV_Blend02Boundary* transformation flags

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization enables a reparameterization and trim of the curve.

A Blend02Boundary curve can also be considered to be the intersection between one of the bounding surfaces of the blend and a construction surface, which is an implicit surface intersecting the blend and

is orthogonal to the blend surface along its bounding curve. Since the shape of this surface is not significant for the curve's geometry, it is not described.

In practice, a Blend02Boundary curve is created as an intersection curve but is interpreted as an iso curve on a Blend02 surface. The parameters of the intersection curve which are calculated during the construction process are saved are described below (see 8.10.13 for a description of the crossing points).

intersection_order is TRUE if the first intersection surface is the implicit surface and the second one is the bounding surface; otherwise it is FALSE.

number_of_crossing_points and **crossing_point_positions** give an approximation to the curve through an ordered set of spatial positions.

chordal_error is an estimate of the maximum distance between the curve and the set of segments given by the array of crossing points.

angular_error is the maximum angle between the tangents of two sequential crossing points.

bounding_surface is the adjacent surface.

base_parameter is the parameter at the first crossing point.

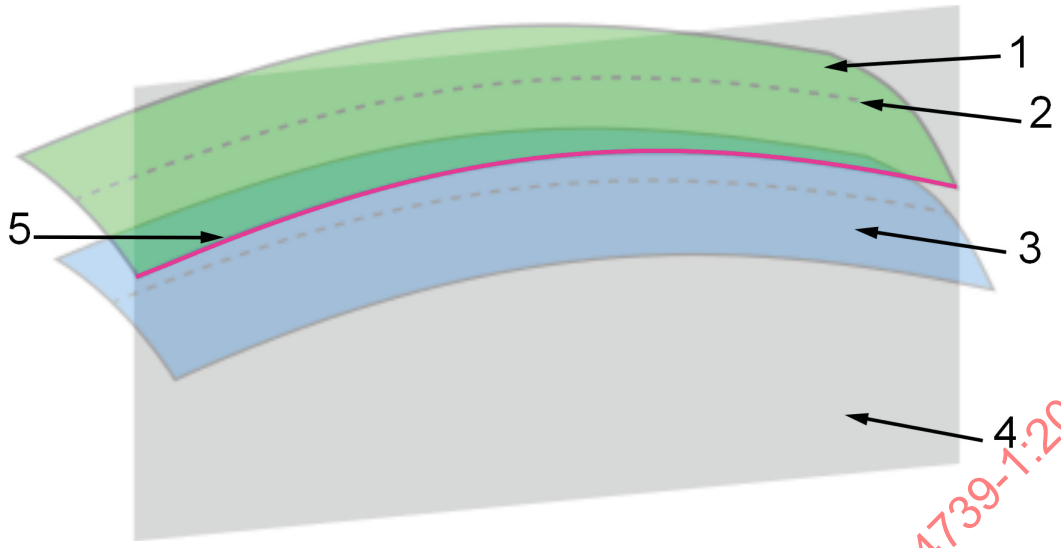
base_scale is the scale at the first crossing point.

start_limit_point and **end_limit_point** define the bounded portion of the curve; each point is further described by **start_limit_type** and **end_limit_type** (see 8.10.13.3).

The evaluation formula at a parameter value on a Blend02Boundary curve is

Calculate the implicit_parameter from the given parameter using this curves‘s Parameterization data.

```
If (bound == 0)
    v = minimum V value of the blend surface
}Else
    v = maximum V value of the blend surface
}
XYZ = blend.evaluate( implicit_parameter, v )
```



Key

1 blend02
2 blend02 spline
3 implicit surface intersecting the blend orthogonally
4 bounding surface
5 blend02 boundary

Figure 4 — *PRC_TYPE_CRV_Blend02Boundary*

Table 247 — *PRC_TYPE_CRV_Blend02Boundary*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_CRV_Blend02Boundary
curve_data	*ContentCurve*	(Required) Common curve data
transform	*Transformation*	(Required) Transformation positioning curve in model coordinate system
paramaterization	*Parameterization*	(Required) Redefine the parameterization
surface	*PtrSurface*	(Required) Blend surface shall be of type PRC_TYPE_SURF_Blend02
bound	*Integer*	(Required) bound
number_of_crossing_points	*UnsignedInteger*	(Required) Number of crossing points
crossing_points	Array <*Vector3d*> [**number_of_crossing_points**]	(Required) Crossing point positions
chord_error	*Double*	(Required) Chordal error
angle_error	*Double*	(Required) Angular error
bound_surface	*PtrSurface*	(Required) Bounding surface
sense_bound_surface	*Boolean*	(Required) Sense of bounding surface
intersection_order	*Boolean*	(Required) Intersection order

Table 247 *(continued)*

sense_intersection_order	*Boolean*	(Required) Sense of intersection curve
base_parameter	*Double*	(Required) Base parameter
base_scale	*Double*	(Required) Base scale
start_limit_point	*Vector3d*	(Required) Starting limit point
start_limit_type	*UnsignedInteger*	(Required) Starting limit type
end_limit_point	*Vector3d*	(Required) Ending limit point
end_limit_type	*UnsignedInteger*	(Required) Ending limit type

8.10.5 *PRC_TYPE_CRV_NURBS*

8.10.5.1 General

This class represents a non-uniform rational bspline curve. The curve may be either 2D or 3D and it may be either rational or non-rational.

A NURBS curve is defined by the following data:

— **d** is the degree of the curve and is restricted to the range 1 <= degree <= 25

— **P** is an array of control points.

— **np** (the number of control points) = **highest_index_of_control_points** + 1

— **U** is the knot vector

— the knots shall be a non-decreasing sequence, that is, U[i] <= U[i+1]

— The number of times a knot value u occurs in the knot vector is called its multiplicity; knot values are compared (to determine multiplicity) by : U[i+1] <= nextafter(U[i],DBL_MAX), nextafter being the IEEE-754 standard function returning the next representable neighbor of a double-precision floating point (see Bibliography).

— multiple end knots are required; for non-periodic curves, the multiplicity of the end knots is degree+1.

— Interior knots may have multiplicity up to degree+1. Thus, the interior of NURBS curves may be C0 or G1 for instance.

— **knot_type** shall be set in the EPRCKnotType range value

— **nu** (number of knots in the knot vector) = highest_index_of_knots + 1; it shall satisfy Nu = d + Np + 1.

— **is_rational** is TRUE if the curve is rational and has an optional array of weights

— **W** is an optional weight at each control point; W(i) shall be within [0.001, 1000]; all the coordinates x,y,z are weighted.

— **curve_form** shall be set in the EPRCBsplineCurveForm range value.

The evaluation formula at a parameter value on a Nurbs curve is

The curve C(u) at a parameter value u is given by:

$$C(u) = \frac{\sum_{i=0}^{k} W_i P_i N_i(u)}{\sum_{i=0}^{k} W_i N_i(u)}$$

Where

k + 1	=number of control points,
P_i	=control points,
W_i	=weights,
d	=degree.N_i are the normalized B-spline basis functions of degree d defined on the knot set:
$U_{i-d},\ldots,U_{i+1}$	U_{j+1} >= U_j(i.e. non-decreasing).

Table 248 — *PRC_TYPE_CRV_NURBS*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_CRV_NURBS*
curve_data	*ContentCurve*	(Required) Common curve data
is_rational	*Boolean*	(Required) **is_rational** is TRUE if this is a rational NURBS curve; else FALSE
d	*UnsignedInteger*	(Required) **d** is the degree of curve
highest_index_of_control_points	*UnsignedInteger*	(Required) **highest_index_of_control_points**
highest_index_of_knots	*UnsignedInteger*	(Required) **highest_index_of_knots**
p	Array <*ControlPointsNurbsCrv*>[np]	(Required) **P** is an array of control points defining curve.
u	Array <*Double*>[nu]	(Required) **U** is an array of knots
knot_type	*UnsignedInteger*	(Required) **Knot_type** (EPRCKnotType)
curve_form	*UnsignedInteger*	(Required) **Curve_form** (EPRCBsplineCurveForm)

8.10.5.2 *ControlPointsNurbsCrv*

Table 249 — *ControlPointsNurbsCrv*

Name	Data Type	Data Description
x	*Double*	(Required) X coordinate of control point
y	*Double*	(Required) Y coordinate of control point
z	*Double*	(Optional; if **is_3d_flag** is TRUE) Z coordinate of control point; the Boolean flag is_3d comes from the ContentCurve data
w	*Double*	(Optoinal; if **is_rational** is TRUE) W coordinate of control point

8.10.5.3 *EPRCKnotType*

This enumeration is used to characterize a knot vector.

NOTE This value is currently unused and should be set to KEPRCKnotTypeUnspecified.

Table 250 — *EPRCKnotType*

Value	Type Name	Data Description
0	*KEPRCKnotTypeUniformKnots*	Uniform knot vector
1	*KEPRCKnotTypeUnspecified*	Unspecified knot type
2	*KEPRCKnotTypeQuasiUniformKnots*	Quasi-uniform knot vector
3	*KEPRCKnotTypePiecewiseBezierKnots*	Extrema with multiplicities of degree + 1

8.10.5.4 *EPRCBsplineCurveForm*

This enumerated type defines the possible NURBS curve forms.

NOTE This value is currently not used and should be set to KEPRCBsplineCurveFormUnspecified.

Table 251 — *EPRCBsplineCurveForm*

Value	Type Name	Data Description
0	*KEPRCBSplineCurveFormUnspecified*	Unspecified curve form
1	*KEPRCBSplineCurveFormPolylinel*	Polygon
2	*KEPRCBSplineCurveFormCircularArc*	Circular arc
3	*KEPRCBSplineCurveFormEllipticArc*	Elliptical arc
4	*KEPRCBSplineCurveFormParabolicArc*	Parabolic arc
5	*KEPRCBSplineCurveFormHyperbolicArc*	Hyperbolic arc

8.10.5.5 *EPRCExtendType*

This enumerated type defines the possible methods for curve and surface extentions. Extensions may be either C or G continuous.

The first bit is reserved for future use and shall be set to 0 in any variable representing this type

Table 252 — *EPRCExtendType*

Value	Type Name	Data Description
0	*KEPRCExtendTypeNone*	Discontinuous position
2	*KEPRCExtendTypeExt1*	Same as KEPRCExtendTypeCInfinity
4	*KEPRCExtendTypeExt2*	Same as KEPRCExtendTypeG1R for surface and KEPRCExtendTypeG1 for curve
6	*KEPRCExtendTypeG1*	Continuous in direction but not magnitude of first derivative
8	*KEPRCExtendTypeG1R*	Surface extended with a ruled surface that connects with G1 continuity
10	*KEPRCExtendTypeG1_G2*	Extended by reflection, yielding a G2 continuous extension
12	*KEPRCExtendTypeCInfinity*	Unlimited continuity

8.10.6 *PRC_TYPE_CRV_Circle*

A cannonical circle is defined by its radius and lies on the XY plane centered at the origin. It is parameterized in radians on the interval [0.0, 2 π] where 0.0 lies on the x-axis and the mapping is defined by the right-hand rule relative to the z-axis normal to the plane of definition.

A Transformation positions the curve in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11)

Table 253 — *PRC_TYPE_CRV_Circle*

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization enables a reparameterization and trim of the circle.

The evaluation formula at a parameter value on a circle is

Calculate the implicit_parameter from the given parameter using this circle's Parameterization data.

X = Radius * cos(implicit_parameter);

Y = Radius * sin(implicit_parameter);

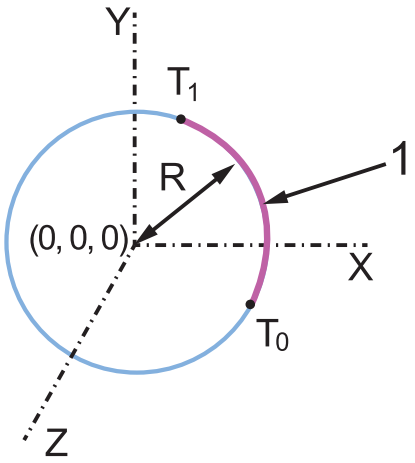
Z = 0.0

The following examples illustrate possible uses of the Parameterization class:

— To specify the interval in radians, set Coeff_a to 1.0, Coeff_b to 0.0, and interval to [0.0,2 π]. These parameter values specify an identity conversion.

— To specify the interval in degrees, set Coeff_a to PI/180, Coeff_b to 0.0, and interval to [0.0, 360.0]. Coeff_a is the ratio of radians to degrees.

— To reparameterize the circle so the parameter values are in the interval [0.0, 1.0], set Coeff_a to 2 π, Coeff_b to 0.0 and interval to [0.0, 1.0]



Key

1 circular arc in the X-Y plane
R radius of the circle
T_0 start of arc in 3D
T_1 end of arc

Figure 5 — Example of a circular arc

In the above example, the circular arc is in the XY plane (and therefore has an identity transformation), has radius R, and is restricted to the [t0 , t1] interval

Table 254 — *PRC_TYPE_CRV_Circle*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_CRV_Circle
curve_data	*ContentCurve*	(Required) Common curve data
has_transform	*Boolean*	(Required) **has_transform** is TRUE if there is a transformation; else FALSE
transform	*Transformation*	(Optional; if **has_transform** is TRUE) Transformation positioning circle in model coordinate system
paramaterization	*Parameterization*	(Required) Redefine the parameterization
radius	*Double*	(Required) Radius

8.10.7 PRC_*TYPE_CRV_Composite*

This represents a composite curve consisting of one or more subcurves.

The following restrictions apply:

Each subcurve shall be of the same dimensionality as the composite (i.e. all 2D or all 3D).

The subcurves shall define a piecewise continuous curve, that is, they shall define a G0 continuous curve.

A Transformation positions the composite curve in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section 7.4.11).

Table 255 — *PRC_TYPE_CRV_Composite* types

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization enables a reparameterization and composite curve.

The implicit parameterization of the composite is the interval [0.0, NumberOfSubCurves]. The subinterval from [i, i+1] corresponds to the ith subcurve either in the same or opposite direction according to the sense of the composite curve.

To evaluate a composite curve at a parameter value:

Calculate the implicit_parameter from the given parameter using this composite curve's Parameterization data. The implicit_parameter must lie in the interval[0.0, NumberOfSubCurves].

Find the sub-interval that the implicit_parameter lies in. Say it lies in the ith subinterval [i, i+1], that is, i <= implicit_parameter <= i + 1.

Get the interval defining the ith SubCurve. Say it is the interval [a, b].

Calculate delta = implicit_parameter – i;

If the sense of the ith SubCurve is the same as the sense of the composite

Parameter_OnSubCurve = a + delta * (b-a)

Else

Parameter_OnSubCurve = b – delta * (b-a)

Position = SubCurve.evaluate(Parameter_OnSubCurve)

Table 256 — *PRC_TYPE_CRV_Composite*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_CRV_Composite
curve_data	*ContentCurve*	(Required) Common curve data
transform	*Transformation*	(Required) Transformation positioning composite curve in model space
parameterization	*Parameterization*	(Required) Define parameterization
number_of_subcurves	*UnsignedInteger*	(Required) Number of subcurves
subcurves	Array <*CompositeSubCurve*> [number_of_subcurves]	(Required) Array of subcurves comprising the composite
is_closed	*Boolean*	(Required) **is_closed** isTRUE if the composite curve is closed; else FALSE

8.10.7.1 *CompositeSubCurve*

A subcurve of a composite curve consists of a pointer to the definition of the subcurve and a flag indicating if the subcurve is in the same direction as the composite curve (TRUE) or in the opposite direcection (FALSE).

Table 257 — *CompositeSubCurve*

Name	Data Type	Data Description
subcurve	*PrtCurve*	(Required) A subcurve in the composite
sense	*Boolean*	(Required) **sense** is TRUE if the subcurve is in the same direction as the composite curve; FALSE if it is in the opposite direction

8.10.8 *PRC_TYPE_CRV_OnSurf*

This represents a 3D curve defined as a UV curve lying in the domain of a surface.

The specified domain is currently ignored and the underlying surface domain is used to define the domain of the surface that the UV curve shall lie within.

A Transformation positions the curve in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 258 — *PRC_TYPE_CRV_OnSurf* flags

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization will enable this curve to be reparameterized and trimmed.

The tolerance is used internally but does not take part of the definition of the curve on surface. It indicates an appropriate tolerance that can be used to obtain a “representative” 3D NURBS approximation of the curve to aid in various operations. If not known it shall be set to 0.0. The unit of this tolerance is the same as that used to store CrvOnSurf Data. See Section 5.7.

To evaluate this curve at a parameter value:

Calculate the implicit_parameter from the given parameter using this CurveOnSurf‘s Parameterization data.

uv_position = uv_curve.evaluate(implicit_parameter)

XYZ = Surface.evaluate(uv_position)

Table 259 — *PRC_TYPE_CRV_OnSurf*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_CRV_onSurf*
curve_data	*ContentCurve*	(Required) Common curve data
tranform	*Transfomation*	(Required) Postition curve in model space
parameterization	*Parmeterization*	(Required) Define parameterization and trimming information
tolerance	*Double*	(Required) Tolerance; default is 0.0
uv_curve	*PtrCurve*	(Required) UV curve in parameter domain of surface; this shall be a 2D curve in the UV space of the surface.
surface	*PtrSurface*	(Required) Surface the curve lies on
uv_domain	*Domain*	(Required) UV domain on the surface; this is currently ignored and the surface domain is used

8.10.9 *PRC_TYPE_CRV_Ellipse*

A canonical ellipse is centered at the origin with radius Rx along the x-axis and radius Ry along the y-axis and lies in the XY-plane. It is parameterized in radians on the interval [0.0, 2 π] with 0.0 on the positive x-axis and the mapping is defined by a right-hand rule about the z-axis normal to the plane of definition.

A Transformation positions the curve in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 260 — *PRC_TYPE_CRV_Ellipse* entity

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A parameterization will enable the ellipse to be reparameterized and trimmed.

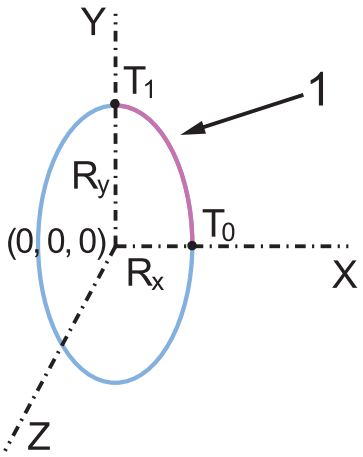
The evaluation formula at a parameter value on an ellipse is

Calculate the implicit_parameter from the given parameter using this ellipse‘s Parameterization data.

X = rx * cos(implicit_parameter);

Y = ry * sin(implicit_parameter);

Z = 0.0



Key

1 elliptic arc in X-Y plane
R_x radius in X dimension
R_y radius in Y dimension
T_0 starting point for arc in 3D
T_1 ending point for arc

Figure 6 — Example of an elliptic arc

In this example, the ellipse is in the XY plane (and therefore has an identity transformation), with radii rx and ry and is restricted to the [t_0 , t_1] interval. Assuming Coeff_a is 1.0 and Coeffb 0.0 (which indicates a parameterization in radians), then t_0=0 and t_1= π /2, t0 corresponds to the Cartesian coordinates (rx,0,0) and t_1 to (0,ry,0).

Table 261 — *PRC_TYPE_CRV_Ellipse*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_CRV_Ellipse
curve_data	*ContentCurve*	(Required) Common curve data
tranform	*Transformation*	(Required) Position ellipse into model space
parameterization	*Parameterization*	(Required) Define parameterization and trimming information
rx	*Double*	(Required) Radius along x axis
ry	*Double*	(Required) Radius along y axis

8.10.10 *PRC_TYPE_CRV_Equation*

This defines a curve by 1D mathematical functions in X, Y, and optionally, Z (see 8.12.3).

A Transformation positions the curve in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 262 — *PRC_TYPE_CRV_Equation* transform entty

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A parameterization will enable the curve to be reparameterized and trimmed.

The evaluation formula at a parameter value on this curve is

Calculate the implicit_parameter from the given parameter using this curve equation‘s Parameterization data.

X = X_Function.evaluate(implicit_parameter)

Y = Y_Function.evaluate(implicit_parameter)

If (is_3d_flag)

Z = Z_Function.evaluate(implicit_parameter)

Else

Z = 0.0

Table 263 — *PRC_TYPE_CRV_Equation*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_CRV_Equation
curve_data	*ContentCurve*	(Required) Common curve data
tranform	*Transformation*	(Required) Position curve in model space
parameterization	*Parameterization*	(Required) Reparameterize and trim
interval	*Interval*	(Required) This interval should be set to the same interval as in the Parameterization data
x_function	*PRC_TYPE_MATH_FCT_1D*	(Required) X function
y_function	*PRC_TYPE_MATH_FCT_1D*	(Required) Y function
z_function	*PRC_TYPE_MATH_FCT_1D*	(Optional; if **is_3d_flag** is TRUE) Z funtion if this is a 3D curve; the Boolean flag comes from the ContentCurve field

8.10.11 *PRC_TYPE_CRV_Helix01*

8.10.11.1General

This curve type defines a helix defined on the inteval [- infinite_param , infinite_param]. A Helix is always a 3D curve.

A Transformation positions the helix in model space. This transformation is capable of translation, rotation, and scaling only the following flags are acceptable (see section 8.4.11).

Table 264 — *PRC_TYPE_CRV_Helix01* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization will enable the helix to be reparameterized and trimmed.

A Type variable indicates which of two kinds of helix is defined by this curve:

— Constant pitch (type 0)

— Variable pitch (type 1)

Each has unique data and a unique evaluation formula.

The Start variable represents a 3D position which is used to define the starting position of the helix.

Table 265 — *PRC_TYPE_CRV_Helix01*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_CRV_Helix01
curve_data	*ContentCurve*	(Required) Common curve data
tranform	*Transformation*	(Required) Position helix in model coordinate system
parameterization	*Parameterization*	(Required) Reparameterize and trim helix
type	*Character*	(Required) Type of helix; shall be 0 or 1
orientation	*Boolean*	(Required) Trigonometric orientation (TRUE if helix turns in a clockwise direction and FALSE if it turns in a counter-clockwise direction
start	*Vector3d*	(Required) start
type0_helix	*Type0HelixData*	(Optional; if **type** is 0) Data for type 0 helix
type1_helix	*Type1HelixData*	(Optional; if **type** is 1) Data for type 1 helix

8.10.11.2 *Type0HelixData*

A type 0 helix represents a constant radius helix.

The origin and direction define the axis of the helix. The axis of the helix is denoted as the z-axis.The projection of the Start position onto the helix axis determines an origin_on_axis. The x-axis is then defined as the vector from the origin_on_axis to the start point. This defines a coordinate system orienting and defining the constant radius helix.

The pitch of the helix is the width of one complete helix turn measured along the helix axis.

The radius evolution is used to define a linear evolution of radius, such as a conic helix.

The following is the evaluation formula for a helix of type 0 at a parameter value:

```
origin = point_3d ( origin[0], origin[1], origin[2] );
z_axis = vector_3d ( direction[0], direction[1], direction[2]);
origin_on_axis = project_point( origin, z_axis, start );
x_axis = vector_3d( start - origin_on_axis);
radius = x_axis.length + param * radius_evolution;
if (trignometric_orientation) {
    tmp_point.x = radius * cos( param );
    tmp_point.y = radius * sin( param );
    tmp_point.z = pitch * param;
} else {
    tmp_point.x = radius * cos( - param );
    tmp_point.y = radius * sin( - param );
    tmp_point.z = pitch * param;
}
eval_point = transform_point( origin_on_axis, x_axis, z_axis, tmp_point );
```

Table 266 — *Type0HelixData*

Name	Data Type	Data Description
orgin[0]	*Double*	(Required) Origin[0]
direction[0]	*Double*	(Required) Direction[0]
orgin[1]	*Double*	(Required) Origin[1]
direction[1]	*Double*	(Required) Direction[1]
orgin[2]	*Double*	(Required) Origin[2]
direction[2]	*Double*	(Required) Direction[2]
pitch	*Double*	(Required) Pitch
radius	*Doubel*	(Required) Radius_evolution

8.10.11.3 *Type1HelixData*

For a variable pitch helix (type 1) the following restrictions apply:

— The radius_law shall be of type PRC_TYPE_MATH_FCT_1D_Polynom.

— The theta_law shall be of type PRC_TYPE_MATH_FCT_1D_Polynom.

— The following values are reserved for future use but shall be initialized to the specified values:

  — reserved_double_0 shall be set to 1.

  — reserved_double_1 shall be set to 1.

  — reserved_double_2 shall be set to 1.

  — reserved_double_3 shall be set to 0.

A coordinate system whose

— origin is at the start point

— z-axis is the unit_z vector

— x-axis is the unit_u vector

orients the helix.

The radius and theta laws are used to change the radius according to the angle around the helix.

The z law is used to change the pitch of the helix along its z-axis.

The following is the evaluation formula for a helix of type 1 at a parameter value:

```
r1 = radius_law.coeffient[0];

r2 = radius_law.coeffient[1];

t1 = theta_law.coeffient[0];

t2 = theta_law.coeffient[1];

param = (param / ( r1 + r2 )) * 2;

radius = r1 + ( param - t1 ) * ( r2 - r1 ) / ( t2 - t1 );


if (trigonometric_orientation) {

    tmp_point.x = radius * cos( param );

    tmp_point.y = radius * sin( param );

    tmp_point.z = z_law.evaluate( param );

} else {

    tmp_point.x = radius * cos( - param );

    tmp_point.y = radius * sin( - param );

    tmp_point.z = z_law.evaluate( param );

}
```

x_axis = vector_3d (unit_u[0], unit_u[1], unit_u[2]);

z_axis = vector_3d (unit_z[0], unit_z[1], unit_z[2]);

eval_point = transform_point(start, x_axis, z_axis, tmp_point);

Table 267 — *Type1HelixData*

Name	Data Type	Data Description
unit_z[0]	*Double*	(Required) Unit_z[0]
unit_u[0]	*Double*	(Required) Unit_u[0]
unit_z[1]	*Double*	(Required) Unit_z[1]
unit_u[1]	*Double*	(Required) Unit_u[1]
unit_z[2]	*Double*	(Required) Unit_z[2]
unit_u[2]	*Double*	(Required) Unit_u[2]
reserved_double_0	*Double*	(Required) Reserved_double_0; shall be set to 1
reserved_double_1	*Double*	(Required) Reserved_double_1; shall be set to 1
reserved_double_2	*Double*	(Required) Reserved_double_2; shall be set to 1
reserved_double_3	*Double*	(Required) Reserved_double_3; shall be set to 0
radius_law	*PRC_TYPE_MATH_FCT_1D*	(Required) Radius law
z_law	*PRC_TYPE_MATH_FCT_1D*	(Required) z law
theta_law	*PRC_TYPE_MATH_FCT_1D*	(Required) Theta law

8.10.12 *PRC_TYPE_CRV_Hyperbola*

A canonical hyperbola is centered at the origin with semi_axis length along the x-axis and semi_image_axis length along the y-axis and lies in the XY-plane. It is parameterized on the interval [-infinite_param, infinite_param].

A Transformation positions the hyperbola in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 268 — *PRC_TYPE_CRV_Hyperbola* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization will enable the hyperbola to be reparameterized and trimmed.

The type of hyperbola defines how the parameterization of the hyperbola is to be interpreted:

— 0 The parameterization shall be changed so that param represents the value of the coordinate on the y-axis

— 1 The nominal parameterization formula applies based on cosh and sinh respectively for x and y;

The evaluation formula for a hyperbola at a parameter value is:

Calculate the implicit_parameter from the given parameter using this hyperbola's Parameterization data.

If this is a type 0 hyperbola {

Eval_point.x = semi_image_axis * sqrt(1.0 + pow(implicit_parameter/semi_axis, 2));

Eval_point.y = implicit_parameter;

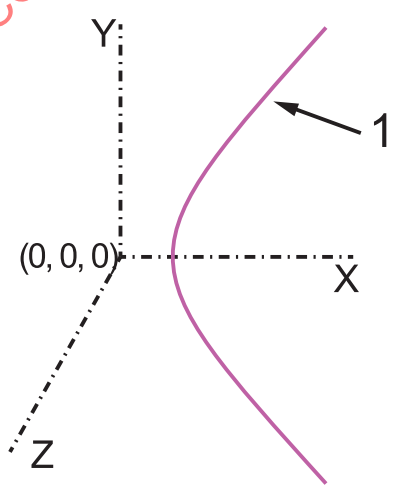
} else { // this is a type 1 hyperbola

Eval_point.x = semi_image_axis * cosh(implicit_parameter);

Eval_point.y = semi_axis * sinh(implicit_parameter);

}

Eval_point.z = 0.0;



Key

1 hyperbola in X-Y plane (right half of hyperbola in the positive X sub-plane)

Figure 7 — Example of a hyperbola

Table 269 — *PRC_TYPE_CRV_Hyperbola*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_CRV_Hyperbola*
curve_data	*ContentCurve*	(Required) Common curve data
tranform	*Transformation*	(Required) Position hyperbola in model coordinate system
parameterization	*Parameterization*	(Required) Reparameterize and trim hyperbola
semi_axis	*Double*	(Required) Semi_axis
semi_image_axis	*Double*	(Required) Semi_axis_image
type	*Character*	(Required) Type of hyperbola; shall be 0 or 1

8.10.13 *PRC_TYPE_CRV_Intersection*

8.10.13.1General

This represents a curve which is the exact intersection of two surfaces.

A Transformation positions the curve in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section8.4.11).

Table 270 — *PRC_TYPE_CRV_Intersection* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization will enable the curve to be reparameterized and trimmed.

A piecewise linear approximation to the true intersection curve is defined by a sequence of crossing points where each of the crossing points lie on the true intersection. At each crossing point the following in known

— the spacial position of the crossing point;

— the UV parameter value of the crossing point on each surface;

— the unit tangent of the intersection curve at this point (defined as the cross product of the surface normals (surface 1 cross surface 2) or reversed (surface 2 cross surface 1) depending on the sense of the intersection curve with an optional sense applied to each surface normal);

— the parameter value (which should satisfy the parameterization requirements described in 8.10.13.2);

— a scale value (which should satisfy the parameterization requirements described in 8.10.13.2).

The NumberOfCrossingPoints shall be sufficient so that evaluation of the curve at a parameter value results in a unique solution (see evaluation method below). The ChordalError and AngularError are used to indicate when more crossing points shall be added to the definition to ensure a unique solution when evaluating an intersection curve at a parameter value (see below).

The intersection curve is limited by two points (start_limit_point and end_limit_point) each characterized with a type of limit (start_limit_type and end_limit_type) as described in 8.10.13.3.

In the case of KEPRCIntersectionLimitTypeTerminator, the limit position is present in the crossing points array (as the first or last point).

ChordalError is an estimate of the maximum distance between the curve and the set of segments given by the crossing points array.

AngularError is the maximum angle between the tangents of two sequential crossing points.

parameterization_definition_respected indicates whether the parameters of the crossing points array are compliant with the parameterization requirements (see 8.10.13.2).

This corresponds to a valid geometry in the sense of the crossing_point_flags, which should be set to TRUE.

An intersection curve is always a 3D curve (i.e. is_3d shall be TRUE).

The evaluation formula for an intersection curve at a parameter value t is:

If t matches a crossing point parameter, the crossing point position is the intersection curve point.

If not, find two consecutive crossing points P1 and P2 such that the parameter t is included in the interval [t1, t2] (t1= parameter of P1 and t2 = parameter of P2]. The intersection curve point will be the intersection of surface 1, surface 2 and the plane defined by the origin O and the normal N where :

$O = P1 + [(t-t1)/(t2-t1)] * (P2-P1)$

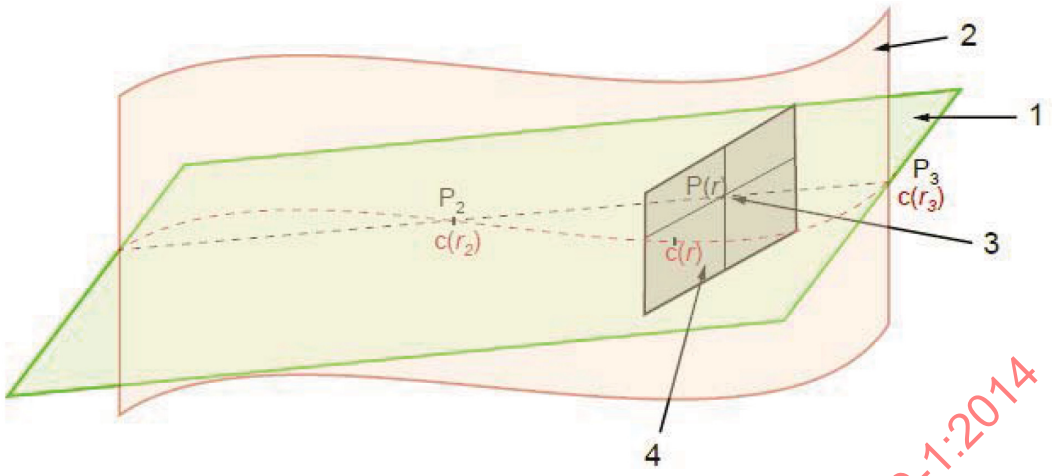
$N = P2 – P1$

In fact, to evaluate a point at a given parameter, an iterative process is used to find the intersection of the three surfaces : surface 1, surface 2 and the plane defined above (the plane depends on parameter t).

Hints on how to ensure a good intersection curve definition:

To ensure that there is a unique solution, additional conditions have to be added on crossing point tangent definition. Theoretically, the tangents of two consecutive points shall have angle smaller than 180 degree. In practice an angle smaller than 40 degree should be used to avoid numerical problems during the computation of the 3 surfaces intersection. This is the AngularError defined above and should be in radians.

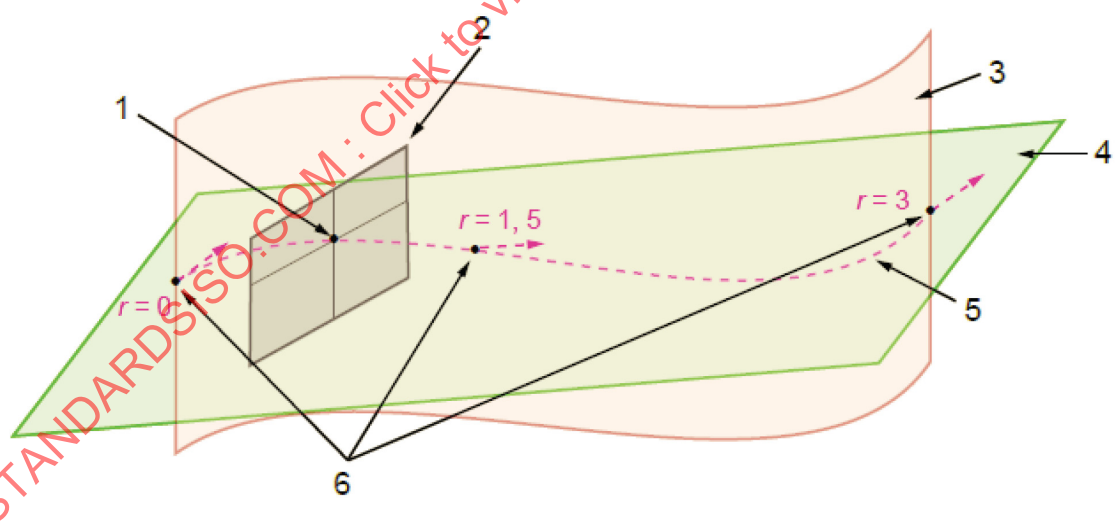
Also to ensure that there is a unique solution, the plane defined above can't cross the intersection curve many times within the ChordalError associated with the intersection curve. The more crossing points there are, the smaller the ChordalError will be. Therefore, in this case, more crossing points have to be added in the intersection curve definition to avoid such situations.



Key

1 surface 1
2 surface 2
3 P(r) – point of the piecewise linear approximation at parameter r
4 the parameter plane(r)
c(r) will be found at the intersection of the three surfaces:
- the parameter plane
- surface 1
- surface 2

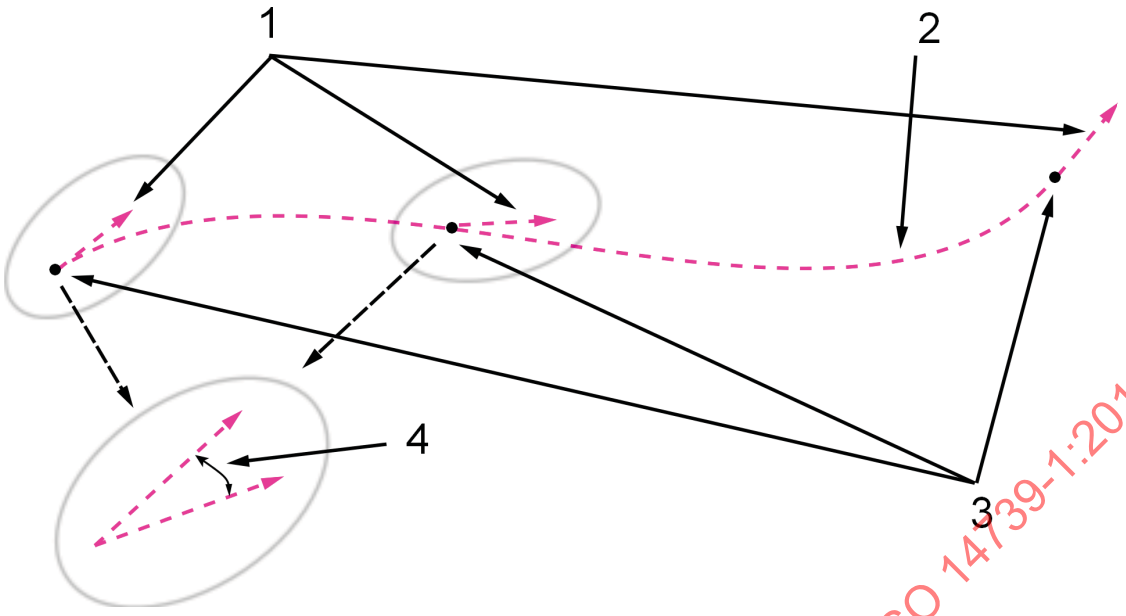
Figure 8 — Example of curve defined by intersection of 2 planes



Key

1 intersection curve point at parameter t = 1
2 plane defined for t = 1
3 surface 1
4 surface 0
5 intersection curve
6 crossing points

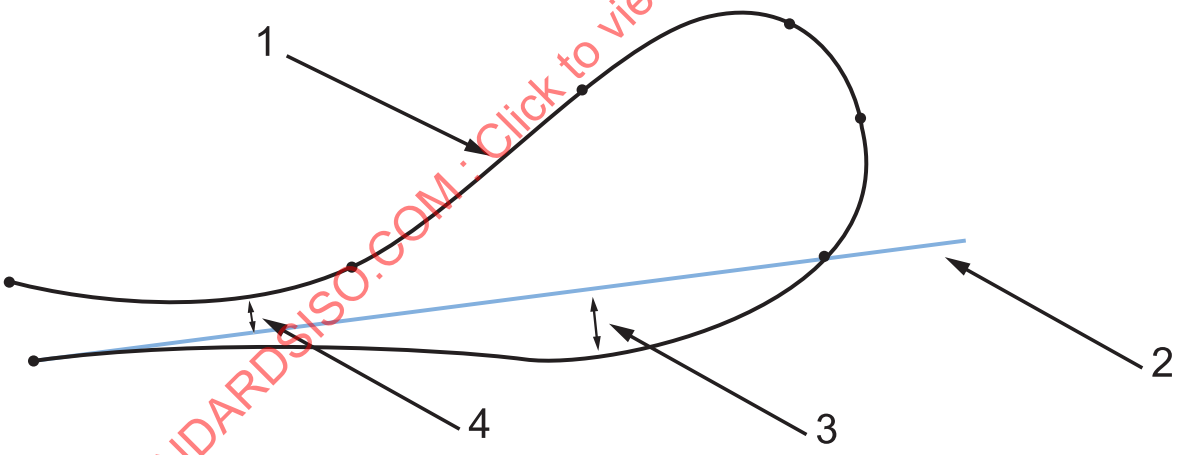
Figure 9 — Example of a curve defined by use of crossing points



Key

1 crossing point tangents
2 intersection curve
3 crossing points
4 angle < 40°

Figure 10 — Example of ensuring a good curve definition



Key

1 curve
2 cord
3 chord error
4 distance (< chord error)

Figure 11 — Example of a unique curve

Table 271 — *PRC_TYPE_CRV_Intersection*

Name	Data Type	Data Description
	UnsignedInteger	(Required) PRC_TYPE_CRV_Intersection
curve_data	*ContentCurve*	(Required) Common curve data
has_transform	*Boolean*	(Required) **has_transform** is TRUE if there is a transform; else FALSE
tranform	*Transformation*	(Optional; if **has_transform** is TRUE) Positions curve in model coordinate system
parameterization	*Parameterization*	(Required) Reparameterize and trim curve
surface_1	*PtrSurface*	(Required) Surface 1
surface_2	*PtrSurface*	(Required) Surface 2
sense_1	*Boolean*	(Required) TRUE if sense is the same as surface 1; FALSE otherwise
sense_2	*Boolean*	(Required) TRUE if sense is the same as surface 2; FALSE otherwise
sense_cross	*Boolean*	(Required) TRUE if the sense of the intersection sense is surface 1 cross surface 2; FALSE otherwise
number_of_crossings	*UnsignedInteger*	(Required) Number of crossing points
crossings	ArrayOf <*CrossingPointsCrvIntersection*> [**number_of_crossings**]	(Required) Array of crossing points
start_limit	*Vector3d*	(Required) Start limit point
start_limit_type	*UnsignedInteger*	(Required) Start limit type; EPRCIntersectionLimitTypes
end_limit	*Vector3d*	(Required) End limit point
end_limit_type	*UnsignedInteger*	(Required) End limit type; EPRCIntersectionLimitTypes
chord_error	*Double*	(Required) Chordal error
angle_error	*Double*	(Required) Angular error
param_respected	*Boolean*	(Required) Parameterization definition respected

8.10.13.2 *CrossingPointsCrvIntersection*

Each crossing point is described by the following:

— The spatial position (crossing_point_position).

— The parametric position on surface_1 (crossing_point_uv_1).

— The parametric position on surface_2 (crossing_point_uv_2).

— The normalized tangent on the curve, crossing_point_tangent, is given by the cross product of two surface normals, taking into account the senses of surfaces surface_1_sense and surface_2_sense.

— Parameter value associated with the crossing point.

— Scale associated with the crossing point.

— The flag shall be set to

PRC_INTERSECTION_CROSS_POINT_SURFACE1 |
PRC_INTERSECTION_CROSS_POINT_SURFACE2 |
PRC_INTERSECTION_CROSS_POINT_INSIDE_CURVE_INTERVAL

to indicate that

— the uv position on surface 1 is filled

— the uv position on surface 2 is filled

— The crossing point is inside the curve interval.

At the ith crossing point, the parameter and scale should adhere to the following Parameterization Requirements

$$scale[i] \leftarrow \frac{\text{tangent}[i].(position[i] - position[i-1])}{\text{tangent}[i].(position[i+1] - position[i])} scale[i-1]$$

$$parameter[i] \leftarrow parameter[i-1] + scale[i-1].\|position[i] - position[i-1]\|$$

parameter[0] is the parameter at first crossing point and the minimum parameter of the curve interval;

scale[0] should be set to 1.0 if not known.

This method is useful to get a more or less curvilinear parameterization without too much computation (function integration for example...).

The intersection curve has a boolean flag to indicate if these parameterization requirements are met.

The . (period) in the scale formula is the dot product while in the parameter formula it is simple multiplication.

Between two crossing points, the parameter of an intersection curve point is given by its projection onto the previous crossing point tangent line.

Table 272 — *CrossingPointsCrvIntersection*

Name	Data Type	Data Description
position	*Vector3d*	(Required) Crossing point position
uv_surface_1	*Vector2d*	(Required) Crossing point uv on surface 1
uv_sruface_2	*Vector2d*	(Required) Crossing point uv on surface 2
tangent	*Vector3d*	(Required) Crossing point tangent
parameter	*Double*	(Required) Crossing point parameter
scale	*Double*	(Required) Crossing point scale
flags	*Character*	(Required) Crossing point flags

8.10.13.3 *EPRCIntersectionLimitType*

This enumeration is used to classify an endpoint of a bounded portion (curve segment) of an intersection curve defined by the intersection of two surfaces (*PRC_ TYPE_CRV_Intersection*). This

classification takes into consideration the nature and relationship of the surface normals of the point on each of the two surfaces as well as the shape of the intersection curve (finite/infinite, open/closed).

The endpoint is classified *KEPRCIntersectionLimitTypeTerminator* if one (or both) of the surface normals is degenerate or if both of the surface normals are well defined but the normals are collinear.

EXAMPLE 1 Consider the following intersection of two cylinders. The intersection point where the two surface normals become collinear will be a limit point of type Terminator which is used to define two separate intersection curves (i.e. each branch of the intersection results in an intersection curve).

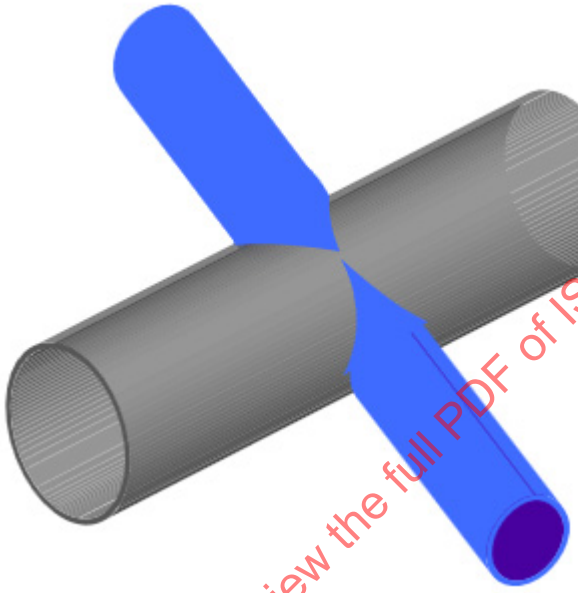


Figure 12 — Example of an intersection limit type

The endpoint is classified *KEPRCIntersectionLimitTypeBoundary* if the intersection curve is used as a center curve of a blend02 surface that becomes degenerate but it is not relevant to the intersection curve.

EXAMPLE 2 Consider a blend02 surface with a radius that becomes equal to its center curve curvature radius (an intersection curve).

The endpoint is classified *KEPRCIntersectionLimitTypeLimit*, if it lies on an infinite intersection curve. In this case, arbitrary endpoints are chosen to limit the curve segment to avoid having an infinite curve.

EXAMPLE 3 Consider the intersection of two cylinders which result in two parallel lines. Two endpoints of limit type Limit are picked to define a finite line segment on each of the branches of the surface/surface intersection.

The endpoint is classified *KEPRCIntersectionLimitTypeHelp*, if it lies on a finite, closed intersection curve. In this case, an arbitrary point is chosen to represent the end point of the curve segement.

EXAMPLE 4 Consider the following intersection of a plane and cone which results in an elliptical intersection curve. Any point on the intersection curve may be used as the limit point with limit type Help.

Table 273 — *EPRCIntersectionLimitType*

Value	Type Name	Data Description
0	*KEPRCIntersectionLimitTypeHelp*	Arbitary limit on a closed intersection curve.
1	*KEPRCIntersectionLimitTypeTerminator*	Limit where one of the two intersection surface normals is degenerate or where they become colinear.
2	*KEPRCIntersectionLimitTypeLimit*	Artificial limit to avoid an infinite curve.
3	*KEPRCIntersectionLimitTypeBoundary*	Limit of the curve if a PRV_TYPE_SURF_Blend02 surface (that uses the intersection curve as its center curve) becomes degenerate.

8.10.14 *PRC_TYPE_CRV_Line*

The cannonical line is defined along the x-axis. The implicit parameterization is [-infinite_param, infinite_param] with 0.0 being the origin and positive values along the positive x-axis.

The Transformation can reposition the cannonical representation in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 274 — *PRC_TYPE_CRV_Line* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The Parameterization enables the line to be reparameterized and trimmed.

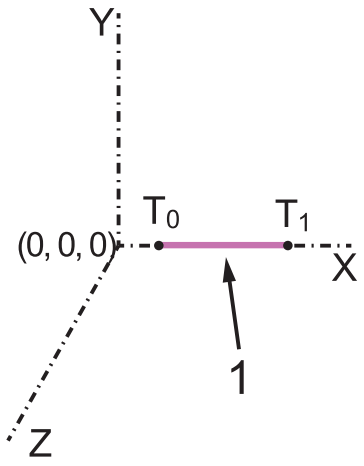
The evaluation formula for a line at a parameter value is:

Calculate the implicit_parameter from the given parameter using this line's Parameterization data.

X = implicit_parameter;

Y = 0.0;

Z = 0.0;



Key

1 line segment from T_0 to T_1 along X-axis

Figure 13 — Example of a line segment

In the above illustration, the line is restricted to [t0 , t1] interval on the X vector of its Cartesian transformation.

Table 275 — *PRC_TYPE_CRV_Line*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_CRV_Line*
curve_data	*ContentCurve*	(Required) Common curve data
has_transform	*Boolean*	(Required) **has_transform** is TRUE if there is a transform; else FALSE
tranform	*Transformation*	(Optional; if **has_transform** is TRUE) Position line in model coordinate system
parameterization	*Parameterization*	(Required) Reparameterize and trim line

8.10.15 *PRC_TYPE_CRV_Offset*

This represents the offset of a 3D curve following the binormal defined by the tangent of the curve and the offset plane normal.

The curve shall be 3D, that is, the Is_3d Boolean flag of the ContentCurve shall be TRUE.

The curve shall not have a transformation, that is, the Has_transformation Boolean flag of the Transformation shall be FALSE.

Parameterization shall have Coeff_a = 1.0, Coeff_b = 0.0, and the interval shall lie within the base curve interval.

You can use an existing offset curve entity as the base curve used to create a new offset curve.

The evaluation formula for an offset curve at a parameter value is:

Calculate the implicit_parameter from the given parameter using this curves‘s Parameterization data.

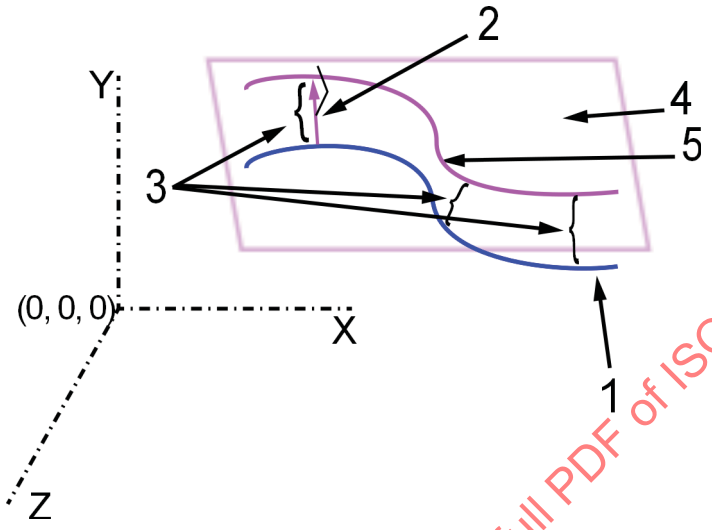
base_point = base_curve.evaluate(implicit_param)

base_deriv = base_curve.evaluate_derivative(implicit_param)

offset_dir = normalize(base_deriv ^ offset_plane_normal)

eval_point = base_point + offset_distance * offset_dir

Where X ^ Y is the cross product of the vectors X and Y



Key

1 base curve
2 offset vector v (defines offset distance and offset normal)
3 offset distance (constant)
4 offset plane
5 offset curve

Figure 14 — Example of an offset curve

In this example the base curve is offset by a 3D vector V which specifies the combination of the offset plane normal and the offset distance.

Table 276 — *PRC_TYPE_CRV_Offset*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_CRV_Offset*
curve_data	*ContentCurve*	(Required) Common curve data
tranform	*Transformation*	(Required) Postion offset curve in model space
parameterization	*Parmeterization*	(Required) Reparameterize and trim curve
base_curve	*PtrCurve*	(Required) Base curve to offset
offset_plane_normal	*Vector3d*	(Required) Offset plane normal; this should be a unit vector
offset	*Double*	(Required) Offset distance

8.10.16 *PRC_TYPE_CRV_Parabola*

A canonical parabola has its focus at (focal_length, 0, 0), its directrix at x = - focal_length and lies in the XY-plane. It is parameterized on the interval [-infinite_param, infinite_param].

A Transformation positions the parabola in model space. This transformation is capable of translation, rotation, and scaling Only the following flags are acceptable (see section 8.4.11).

Table 277 — *PRC_TYPE_CRV_Parabola* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization will enable the parabola to be reparameterized and trimmed.

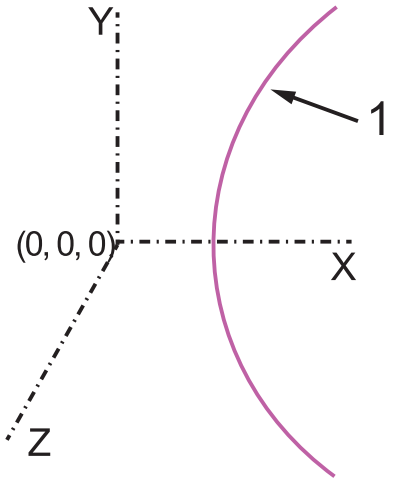
The type of parabola defines how the parameterization of the parabola is to be interpreted:

— 0 The parameter represents the value of the coordinate on the x-axis; the y-axis is the axis of the parabola

— 1 The nominal parameterization formula applies; the parameter is proportional to the value of the coordinate on the y axis; the x-axis is the axis of the parabola

The nominal evaluation formula for a parabola at param value is:

```
If type is 1 {
    eval_point.x = focal_length * param * param;
    eval_point.y = 2.0 * focal_length * param;
} else {
    param2 = param * param;
    p2 = param2/2.0 +
    sqrt(param2*param2/4.0 + 16.0*focal_length*focal_length*param2);
    param2 = (param < 0.0)sqrt(p2) : -sqrt(p2);
    if (param2 < 0) {
        eval_point.x  = - param2
        eval_point.y = - 2.0 * focal_length * sqrt( - param2/ focal_length)
    } else {
        eval_point.x  = param2
        eval_point.y = 2.0 * focal_length * sqrt( param2/ focal_length)
    }
}
Eval_point.z = 0.0
```



Key

1 parabola in the X-Y plane

Figure 15 — Example of a parabolic arc

Table 278 — *PRC_TYPE_CRV_Parabola*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_CRV_Parabola*
curve_data	*ContentCurve*	(Required) Common curve data
tranform	*Transformation*	(Required) Position curve in model coordinate system
parameterization	*Parameterization*	(Required) Reparameterize and trim curve
focal_length	*Double*	(Required) Focal length
type	*Character*	(Required) Parameterization type; shall be 0 or 1

8.10.17 *PRC_TYPE_CRV_PolyLine*

8.10.17.1General

This represents a PolyLine curve defined by a sequence of 2D or 3D points.

The implicit parameterization of a polyline is the interval [0.0, number of points]. The interval [i, i+1] corresponds to the segment between point[i] and point[i+1]. The curve between consecutive points is a straight Added transformation info line.

A Transformation positions the polyline in model space. This transformation is capable of translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 279 — *PRC_TYPE_CRV_PolyLine* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A Parameterization will enable the polyline to be reparameterized and trimmed.

Table 280 — *PRC_TYPE_CRV_PolyLine*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_CRV_PolyLine*
curve_data	*ContentCurve*	(Required) Common curve data
has_transform	*Boolean*	(Required) **has_transform** is TRUE if there is a transform; else FALSE
tranform	*Transformation*	(Optional; if **has_transform** is TRUE) Postion curve in model space
parameterization	*Parmeterization*	(Required) Reparameterize and trim curve
number_of_points	*UnsignedInteger*	(Required) Number of points in polyline
points	Array <*PolyLinePoint*> [**number_of_points**]	(Required) Array of points defining polyline

8.10.17.2 *PolyLinePoint*

Table 281 — *PolyLinePoint*

Name	Data Type	Data Description
point_3d	*Vector3d*	(Optional; if **is_3d_flag** is TRUE) 3D point; the is_3d_flag Boolean flag comes from the ContentCurve
point_2d	*Vector2d*	(Optional; if **is_3d_flag** is TRUE) 2D point

8.10.18 *PRC_TYPE_CRV_Transform*

A Transform curve represents a curve defined by applying a 3D mathematical function to a base curve.

Both the transform curve and the base_curve shall be 3D curves.

The Transformation can reposition the curve in model space using a translation, rotation, and scaling. Only the following flags are acceptable (7.4.11).

Table 282 — *PRC_TYPE_CRV_Transform* transform

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The Parameterization enables the curve to be reparameterized and trimmed.

The nominal evaluation formula for a transform curve at param value is:

Calculate the implicit_parameter from the given parameter using this transform curve's Parameterization data.

tmp_point = base_curve.evaluate(implicit_parameter);

If (math_transformation != NULL)

eval_point =math_transformation.evaluate(tmp_point);

Else

eval_point = tmp_point;

Table 283 — *PRC_TYPE_CRV_Transform*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_CRV_Transform*
curve_data	*ContentCurve*	(Required) Common curve data
tranform	*Transformation*	(Required) Position curve in model space
parameterization	*Parmeterization*	(Required) Reparameterize and trim curve
base_curve	*PtrCurve*	(Required) Base curve
math_transformation	*PRC_TYPE_MATH_FCT_3D*	(Required) 3D mathematical transformation to apply to base curve

8.11 Surface

8.11.1 Entity Types

Table 284 — Surface entity types

Type Name	Type Value	Referenceable
PRC_TYPE_SURF	*PRC_TYPE_ROOT* + 75	
PRC_TYPE_SURF_Base	*PRC_TYPE_SURF* + 1	
PRC_TYPE_SURF_Blend01	*PRC_TYPE_SURF* + 2	
PRC_TYPE_SURF_Blend02	*PRC_TYPE_SURF* + 3	
PRC_TYPE_SURF_Blend03	*PRC_TYPE_SURF* + 4	
PRC_TYPE_SURF_NURBS	*PRC_TYPE_SURF* + 5	
PRC_TYPE_SURF_Cone	*PRC_TYPE_SURF* + 6	
PRC_TYPE_SURF_Cylinder	*PRC_TYPE_SURF* + 7	
PRC_TYPE_SURF_Cyldrical	*PRC_TYPE_SURF* + 8	
PRC_TYPE_SURF_Offset	*PRC_TYPE_SURF* + 9	
PRC_TYPE_SURF_Pipe	*PRC_TYPE_SURF* + 10	
PRC_TYPE_SURF_Plane	*PRC_TYPE_SURF* + 11	
PRC_TYPE_SURF_Ruled	*PRC_TYPE_SURF* + 12	
PRC_TYPE_SURF_Sphere	*PRC_TYPE_SURF* + 13	
PRC_TYPE_SURF_Revolution	*PRC_TYPE_SURF* +14	
PRC_TYPE_SURF_Extrusion	*PRC_TYPE_SURF* + 15	
PRC_TYPE_SURF_FromCurves	*PRC_TYPE_SURF* + 16	
PRC_TYPE_SURF_Torus	*PRC_TYPE_SURF* + 17	
PRC_TYPE_SURF_Transform	*PRC_TYPE_SURF* + 18	
PRC_TYPE_SURF_Blend04	*PRC_TYPE_SURF* + 19	

8.11.2 *PRC_TYPE_SURF*

Abstract type for surfaces. If this appears in the documentation defining a field in the PRC File, it means that any surface type may be used.

8.11.3 *PRC_TYPE_SURF_Base*

8.11.3.1 General

Abstract type for surfaces. The following data is stored for all surface types.

8.11.3.2 *ContentSurface*

ContentSurface provides additional information about a surface such as its name and attributes and how it extends past its boundary.

Table 285 — *ContentSurface*

Name	Data Type	Data Description
has_base_geometry	*Boolean*	(Required) TRUE if base information is present; else FALSE
attribute_data	*AttributeData*	(Optional; if **has_base_geometry** is TRUE) Attributes attached to the geometric entity
name	*Name*	(Optional; if **has_base_geometry** is TRUE) Name attached to the geometric entity
id	*UnsignedInteger*	(Optional; if **has_base_geometry** is TRUE) Identifier in originating CAD system; this may not be used to reference entity within PRC File;
extension_type	UnsignedInteger	(Required) Indicates how the surface is extended; see *EPRCExtendType*

8.11.4 *PRC_TYPE_SURF_Blend01*

A Blend01 surface is defined by three curves, a center curve, an origin curve and an optional tangent curve, all defined over the same parameter interval. If the tangent curve is not defined (NULL), the first derivative of the origin curve is used instead. The implicit parameterization of a Blend01 surface is [0, 2 π] x [center_curve.interval.min, center_curve.interval.max].

A Blend01 surface represents a variable radius pipe surface centered on the center curve with the origin curve defining both the radius and 0.0 location of the u parameter and the tangent curve defining the normal of the cross section plane of the Blend01 surface along the center curve.

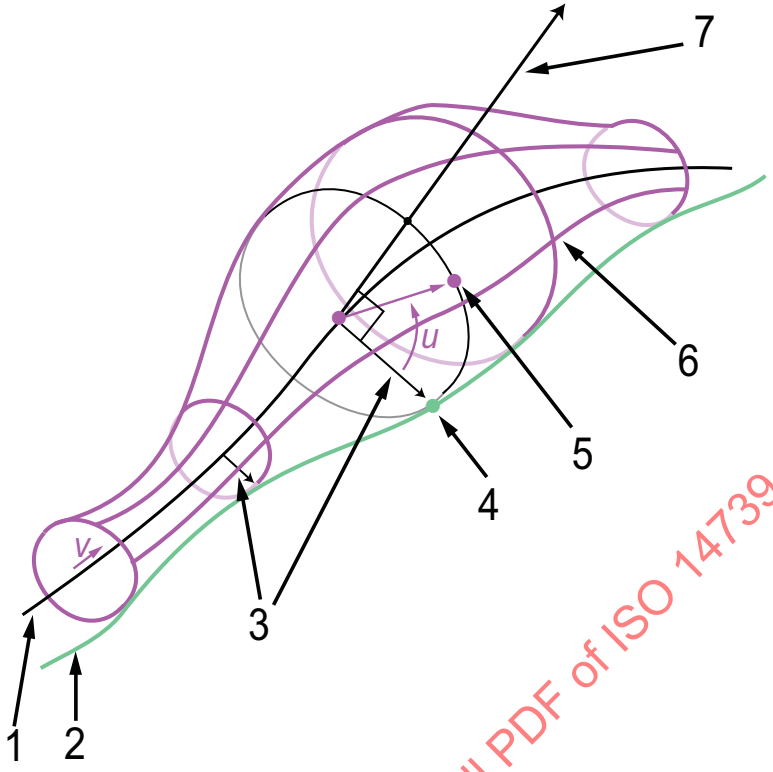
A Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see8.4.11).

Table 286 — *PRC_TYPE_SURF_Blend01* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

A UVParameterization enables the surface to be reparameterized and trimmed.

The following diagram illustrates the relationship between the curves defining the Blend01 surface.



Key

1 center(v) curve
2 origin(v) curve
3 radius(v)
4 origin(v)
5 XYZ(u, v) (on Blend01 surface)
6 Blend01 surface
7 tangent(v)

Figure 16 — Example of a Blend01 surface

To evaluate a Blend01 surface at a parameter value:

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

R(implicit_param_v) = origin_curve(implicit_param_v) - center_curve(implicit_param_v)

XYZ = origin_curve(implicit_param_v) + cos(implicit_param_u) * R(implicit_param_v) + sin(implicit_param_u) * [tangent_curve(implicit_param_v) ^ R(implicit_param_v)]

(where ^ is the cross product)

If the tangent curve is NULL, use the unitized first derivative of origin curve instead of the tangent curve.

Table 287 — *PRC_TYPE_SURF_Blend01*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Blend01*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
center_curve	*PtrCurve*	(Required) Center curve; shall not be NULL
orgin_cruve	*PtrCurve*	(Required) Origin curve; shall not be NULL
tangent_curve	*PtrCurve*	(Required) Tangent curve; may be NULL in which case the first derivative of the center curve is used as the tangent curve.

8.11.5 *PRC_TYPE_SURF_Blend02*

A Blend02 surface is an exact rolling ball blend defined by rolling a ball of a constant radius along a center curve while maintaining tangential contact with two bounding surfaces (or curves in the case of a “cliff hanging“ blend). A point on the center curve is projected into the two bounding geometries and the Blend02 surface is defined by the circular arc between these two points (from point1 to point2).

The Blend02 surface is defined by

— A rolling ball center curve (center_curve) which defines the u parameter of the surface.

— A first bounding surface or a first bounding curve (bound_surface 1 or bound_curve 1). Either bound_surface 1 or bound_curve 1 may be given and the other shall be NULL.

— A second bounding surface or a second bounding curve (bound_surface 2 or bound_curve 2). Either bound_surface 2 or bound_curve 2 may be given and the other shall be NULL.

— A radius and two senses. Radius 1 and radius 2 have the same absolute value but may have different signs. A positive sign indicates that the blend is on the side of the surface normal after taking into account the senses of the surfaces bound_surface_sense 1 and bound_surface_sense 2. A negative sign indicates that the blend is on the opposite side as the bounding surface normal.

— Two instances of cliff_supporting_surface. If one of the bounds is a curve, the surface is a cliff edge blend and its two supporting surfaces are the surfaces of the faces adjacent to the cliff edge (NULL otherwise).

— The type of parameterization (parameterization_type):

  — 0 indicates that the v parameter is zero at the first bound and one at the second bound.

  — 1 indicates that the v parameter is zero at the first bound and the angular value, in radians, at the second bound.

— The implicit parameterization is

  — [center_curve.interval.min, center_curve.interval.max] x [0, 1] if the parameter_type is 0

— [center_curve.interval.min, center_curve.interval.max] x [0, 2 π] if the parameter_type is 1

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 288 — *PRC_TYPE_SURF_Blend02* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

A boundary surface may be replaced by a boundary curve, in that case, known as “cliff edge blending”, the center curve point shall be projected onto this curve and not onto the missing boundary surface.

To evaluate a Blend02 surface at a parameter value:

Calculate the implicit_parameter from the given parameter using this surface‘s UVParameterization data.

Radius is the absolute value of radius 1 and radius 2

P1 = center(u) projected onto Bound Surface 1 (or Bound Curve 1)

P2 = center(u) projected onto Bound Surface 2 (or Bound Curve 2)

Where this is a perpendicular projection and the distance between Center(u) and P1 (and P2) must equal the blend radius

X(u) = (P1 – center(u)) / ‖ P1 – center(u) ‖

Y(u) = (P2 – center(u)) / ‖ P2 – center(u) ‖

A(u) = angle between X(u) and Y(u)

Y2(u) = ((X(u) ^ Y(u)) ^ X(u)) / ‖ ((X(u) ^ Y(u)) ^ X(u)) ‖

So that X(u) . Y2(u) = 0 and Y2(u) is a unit vector

If (parameter_type == 0)

XYZ = center(u) + Radius * (cos(A(u) * v) * X(u) + sin(A(u)*v) * Y2(u))

Else

XYZ = Center(u) + Radius * (cos(v).X(u) + sin(v).Y2(u))

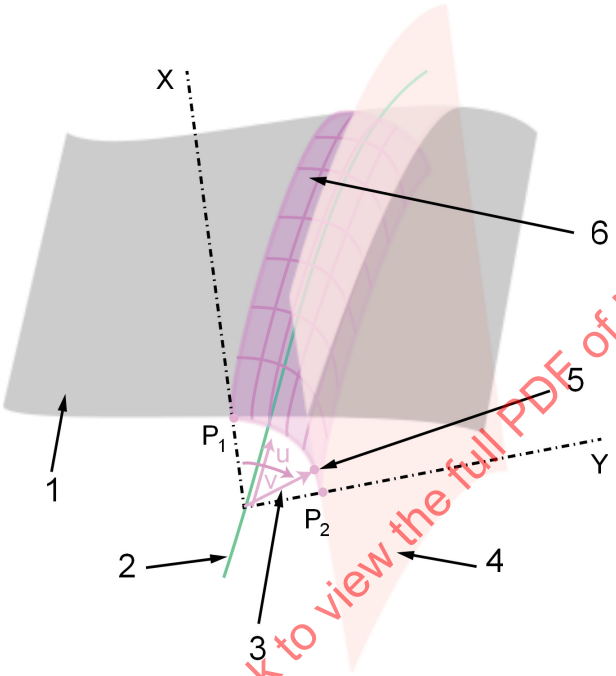
where

u and v mean implicit_param_u and implicit_param_v in all descriptions

|| X || is the length of vector X

X^Y is the cross product of X and Y vectors

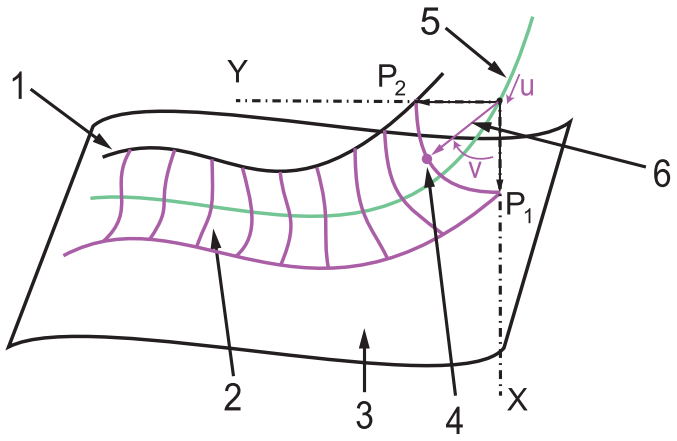
X . Y is the dot product of X and Y vectors



Key

1	bound surface 1
2	center(u) curve
3	Blend02 radius
4	bound surface 2
5	Blend02(u,v)
6	Blend02 surface
P_1	center(u) projected onto bound surface 1
P_2	center(u) projected onto bound surface 2

Figure 17 a) — Example of Blend02 surface



Key

1 bound curve 2
2 Blend02 surface
3 bound surface 1
4 Blend02 (u,v) (on Blend02 surface)
5 centre (u) curve
6 radius

NOTE The same as a cliff blend except P2 = centre (u) projected onto bound curve 2.

Figure 17 b)— Example of Blend02 surface

Table 289 — *PRC_TYPE_SURF_Blend02*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Blend02*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
bound_surface_0	*PtrSurface*	(Required) Bound Surface 0
bound_curve_0	*PtrCurve*	(Required) Bound curve 0
bound_surface_1	*PtrSurface*	(Required) Bound surface 1
bound_curve_1	*PtrCurve*	(Required) Bound curve 1
center_curve	*PtrCurve*	(Required) Center curve
center_curve_sense	*Boolean*	(Required) Center curve sense
bound_surface_0_sense	*Boolean*	(Required) Bound surface 0 sense
bound_surface_1_sense	*Boolean*	(Required) Bound surface 1 sense
radius_0	*Double*	(Required) Radius 0
readius_1	*Double*	(Required) Radius 1
cliff_suface_0	*PtrSurface*	(Required) Cliff supporting surface 0
cliff_surface_1	*PtrSurface*	(Required) Cliff supporting surface 1
parameterization_type	*Character*	(Required) Parameterization type

8.11.6 *PRC_TYPE_SURF_Blend03*

A Blend03 surface is a fillet surface defined by four curves: a center curve, two rail curves, and an angle curve which defines the V parameterization of the surface. All curves are quintic splines defined over the same nodal vector (Number_of_elements, Parameters and Multiplicities).

A center_curve, rail_curve_1, and rail_curve2 are defined using point, tangent, and second derivative data from Number_of_element entries in the Points, Tangents, and SecondDerivative arrays:

— center_curve uses data at indicies i*3 + 0;

— rail_curve_1 uses data at indicies i*3 + 1;

— and rail_curve_2 uses data at indicies i*3 + 2.

where 0 <= i <= Number_of_elements.

A rail2_anglesV_curve is defined using the data in the arrays Rail2AnglesV, Rail2DerivativesV, and Rail2SecondDerivativesV. This curve defines the V parameterization of the Blend03 surface by controlling the V parameterization along the isoparametric U curves.

Each isoparametric U curve is a circle defined on a plane centered on a point evaluated on the center_curve, where the point on rail_curve_1 evaluated at the same parameter gives the x-axis, and the point on rail_curve_2 gives the y-axis. For each circle, rail_curve_1 corresponds to Parameter[0], and the parameter corresponding to the point on rail_curve_2 is found using the angle curve function rail2_anglesV_curve. The same parameter is divided by Rail2ParameterV.

The implicit parameterization is [Parameter[0], Parameter[Number_of_elements - 1]] x [trim_v_min, trim_v_max].

If trim_v_max is less than trim_v_min, the V parameterization is set to [0, 1].

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11)

Table 290 — *PRC_TYPE_SURF_Blend03* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

To evaluate a Blend03 surface at a parameter value:

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

X(u) = (rail_curve_1(u) – center_curve(u)) / || rail_curve_1(u) – center_curve(u) || (so that X(u) is a unit vector)

Y(u) = rail_curve_2(u) – center_curve(u)

Y2(u) = [X(u)^Y(u)]^X(u) / || [X(u)^Y(u)]^X(u) || (so that X(u).Y2(u)=0 and Y2(u) is a unit vector)

A(u) = rail2_anglesV_curve(u) / Rail2ParameterV

Radius(u) = || rail_curve_2(u) - center_curve(u) ||

XYZ = center _curve (u) + Radius(u) * (cos(A(u)*v).X(u) + sin(A(u)*v).Y2(u))

Where

u and v mean implicit_param_u and implicit_param_v.

|| X || is the length of vector X

X^Y is the cross product of X and Y vectors.

The following values are reserved for future use:

— reserved_int[0] should be set to 5.

— reserved_int[1] should be set to 0.

— reserved_int[2] should be set to 0.

— reserved_int[3] should be set to number_of_element.

— reserved_int[4] should be set to 0.

— reserved_int[5] should be set to 1.

— reserved_chars_0 should be set to 1.

— reserved_chars_1 should be set to 0.

— reserved_chars_2 should be set to 0.

— reserved_supplemental_doubles[i] and number_of_supplemental_doubles should be set to 0.

Table 291 — *PRC_TYPE_SURF_Blend03*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Blend03*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
number_of_elements	*Integer*	(Required) Number_of_elements
parameters	Array <*Double*> [**number_of_elements**]	(Required) Parameters
multiplicities	Array <*Integer*> [**number_of_elements**]	(Required) Multiplicities
points	Array <*Vector3d*> [**number_of_elements**]	(Required) Array of Points
rail_2_angles_v	Array <*Double*> [**number_of_elements**]	(Required) Array of Rail2AnglesV
tangents	Array <*Vector3d*> [**number_of_elements**]	(Required) Array of Tangents
rail_2_derivatives_v	Array <*Double*> [**number_of_elements**]	(Required) Array of Rail2DerivativesV
second_derivatives	Array <*Vector3d*> [**number_of_elements**]	(Required) Array of SecondDerivatives
rail_2_second_derivatives	Array <*Double*> [**number_of_elements**]	(Required) Array of Rail2SecondDerivativesV
rail_2_parameter_v	*Double*	(Required) Rail2Parameter V
trim_v_min	*Double*	(Required) Trim v min
trim_v_max	*Double*	(Required) Trim v max
reserved_int	Array <*Integer*>[6]	(Required) Reserved_int
reserved_char_0	*Character*	(Required) Reserved_char_0
reserved_char_1	*Character*	(Required) Reserved_char_1
reserved_char_1	*Character*	(Required) Reserved_char_2
number_of_supplimental_doubles	*Integer*	(Required) Number of reserved supplimental doubles
supplimental_doubles	Array <*Double*>[**number_of_supplimental_doubles**]	(Required) Reserved supplimental doubles

8.11.7 *PRC_TYPE_SURF_NURBS*

8.11.7.1 General

This class represents a non-uniform rational bspline surface.

A NURBS surface is defined by the following data:

— **Du** is the degree of the surface in u and is restricted to the range 1 <= degree <= 25

— **Dv** is the degree of the surface in v and is restricted to the range 1 <= degree <= 25

— **P** is a two dimensional array of control points.

— **Npu** (number of control points in u) = **highest_index_of_control_points_in_u** + 1

— **Npv** (number of control points in v) = **highest_index_of_control_points_in_v** + 1

— **Ku** is the knot vector in u

  — the knots shall be a non-decreasing sequence , that is, Ku[i] <= Ku[i+1]

  — multiple end knots are required; for non-periodic surfaces, the multiplicity of the end knots is Du+1.

  — Interior knots may have multiplicity up to Du+1.

— **Nku** (number of knots in the u knot vector) = **highest_index_of_knots_in_u** + 1; it shall satisfy Nku = Du + Npu + 1.

— **Kv** is the knot vector in v

  — the knots shall be a non-decreasing sequence , that is, Kv[i] <= Kv[i+1]

  — multiple end knots are required; for non-periodic surfaces, the multiplicity of the end knots is Dv+1.

  — Interior knots may have multiplicity up to Du+1.

— **Nkv** (number of knots in the v knot vector) = **highest_index_of_knots_in_v** + 1; it shall satisfy Nkv = Dv + Npv + 1.

— **knot_type** shall be set in the EPRCKnotType range value

— **Rational** is TRUE if the surface is rational and has an optional array of weights

— **W** is an optional weight at each control point; W(i,j) shall be within [0.001, 1000]; all the coordinates x,y,z are weighted.

— **surface_form** shall be set in the EPRCSplineSurfaceForm range value.

The evaluation formula at a parameter value on a Nurbs surface is

The surface S(u,v)) at a parameter value u and v is given by:

$$S(u,v) \leftarrow \frac{\sum_{i=0}^{npu}\sum_{j=0}^{npv} W_{i,j}P_{i,j}N_i(u)N_j(v)}{\sum_{i=0}^{npu}\sum_{j=0}^{npv} W_{i,j}N_i(u)N_j(v)}$$

Where

npu =number of control points in u

npv =number of control points in v

P_{ij} =control points,

W_{ij} =weights,

Du =degree in u

Dv =degree in v

N_i are the normalized B-spline basis functions of degree d defined on the knot set:

$U_{i-d},...,U_{i+1}$ $U_{j+1} >= U_j$ (i.e. non-decreasing).

Table 292 — *PRC_TYPE_SURF_NURBS*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_NURBS*
curve_data	*ContentSurface*	(Required) Common surface data
is_rational	*Boolean*	(Required) is_**rational** is TRUE if this is a rational NURBS surface; else FALSE
du	*UnsignedInteger*	(Required) **Du** is the degree of surface in u
dv	*UnsignedInteger*	(Required) **Dv** is the degree of surface in v
highest_index_of_control_points_in_u	*UnsignedInteger*	(Required) **highest_index_of_control_points_in_u**
highest_index_of_control_points_in_v	*UnsignedInteger*	(Required) **highest_index_of_control_points_in_v**
highest_index_of_knots_in_u	*UnsignedInteger*	(Required) **highest_index_of_knots_in_u**
highest_index_of_knots_in_v	*UnsignedInteger*	(Required) **highest_index_of_knots_in_u**
p	2DimArray <ControlPointsNurbsSurf> **[highest_index_of_control_points_in_u] [highest_index_of_control_points_in_v]**	(Required) **P** is a two dimensional array of control points defining surface.
ku	Array <Double> **[highest_index_of_knots_in_u]**	(Required) **Ku** is an array of knots in u

Table 292 *(continued)*

kv	Array <Double> [**highest_index_of_knots_in_v**]	(Required) **Kv** is an array of knots in v
knot_type	*UnsignedInteger*	(Required) **Knot_type** shall be set to a value in EPRCKnotType
surface_form	*UnsignedInteger*	(Required) **Surface_form** shall be set to a value in EPRCSplineSurfaceForm

8.11.7.2 *ControlPointsNurbsSurf*

An array of control points for a Nurbs surface are stored in a two dimensional array

For (i=0; i<= highest_index_of_control_points_in u; i++)

For (j=0; j<=highest_index_of_control_points_in_v; j++)

Store the x, y, z and optional w value

Table 293 — *ControlPointsNurbsSurf*

Name	Data Type	Data Description
x	*Double*	(Required) X coordinate of control point
y	*Double*	(Required) Y coordinate of control point
z	*Double*	(Required) Z coordinate of control point
w	*Double*	(Optional; if is_rational is TRUE) W coordinate of control point

8.11.7.3 *EPRCSplineSurfaceForm*

This enumerated type defines the possible NURBS surface forms.

NOTE This value is currently not used and should be set to KEPRCBsplineSurfaceFormUnspecified.

Table 294 — *EPRCSplineSurfaceForm* types

Value	Type Name	Type Description
0	*KEPRCBSplineSurfaceFormPlane*	Planar surface
1	*KEPRCBSplineSurfaceFormCylindrical*	Cylindrical surface
2	*KEPRCBSplineSurfaceFormConical*	Conical surface
3	*KEPRCBSplineSurfaceFormSpherical*	Spherical surface
4	*KEPRCBSplineSurfaceFormRevolution*	Surface of revolution
5	*KEPRCBSplineSurfaceFormRuled*	Ruled surface
6	*KEPRCBSplineSurfaceFormGeneralizedCone*	Cone
7	*KEPRCBSplineSurfaceFormQuadric*	Quadric surface
8	*KEPRCBSplineSurfaceFormLinearExtrusion*	Surface of extrusion
9	*KEPRCBSplineSurfaceFormUnspecified*	Unspecified surface
10	*KEPRCBSplineSurfaceFormPolynomial*	Polynomial surface

8.11.8 *PRC_TYPE_SURF_Cone*

This represents a canonical definition of a conical surface where the axis of the cone lies along the z-axis. The x-axis represents the 0.0 value of the u parameter interval [0.0, 2 π] with positive values counter-clockwise about the z-axis using the right hand rule. The z-axis represents the v parameter interval [-infinite_param, infinite_param]. The 0.0 value of the v parameter is indicated by the bottom radius. The semi-angle is the half angle of the cone in radians.

The implicit parameterization of the cone is [0.0, 2 π] x [-infinite_param, infinite_param].

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 295 — *PRC_TYPE_SURF_Cone transformation*

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

To evaluate this surface at a parameter value:

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

radius= bottom_radius + implicit_param_v * tan(semi-angle)

tmp_point.x = cos(implicit_param_u) * radius

tmp_point.y = sin(implicit_param_u) * radius

tmp_point.z = implicit_param_v

Table 296 — *PRC_TYPE_SURF_Cone*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Cone*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
radius	*Double*	(Required) Bottom radius
semi_angle	*Double*	(Required) Semi angle in radians

8.11.9 *PRC_TYPE_SURF_Cylinder*

This represents a canonical definition of a cylinder where the axis of the cylinder lies along the z-axis. The x-axis represents the 0.0 value of the u parameter (radians) interval [0.0, 2 π] with positive values

counter-clockwise about the z-axis using the right hand rule. The z-axis represents the v parameter interval [-infinite_param, infinite_param]. The 0.0 value of the v parameter is at the origin.

The implicit parameterization of the cylinder is [0.0, 2π] x [-infinite_param, infinite_param].

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 297 — *PRC_TYPE_SURF_Cylinder* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

To evaluate this surface at a parameter value:

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

tmp_point.x = cos(implicit_param_u) * radius

tmp_point.y = sin(implicit_param_u) * radius

tmp_point.z = implicit_param_v

Table 298 — *PRC_TYPE_SURF_Cylinder*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Cylinder*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
radius	*Double*	(Required) Radius

8.11.10 *PRC_TYPE_SURF_Cylindrical*

This represents a cylindrical surface expressed in cylindrical coordinate system where (R, Theta, h). A base surface defines the mapping from UV to (R, Theta, h) = (x, y, z). The axis of the cylinder lies along the z-axis.

The implicit parameterization of the cylindrical surface is the same as the UV domain of the base surface.

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 299 — *PRC_TYPE_SURF_Cylindrical* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

The tolerance is used internally but does not take part of the definition of the surface. It indicates an appropriate tolerance that can be used to obtain a "representative" 3D NURBS approximation of the surface to aid in various operations. If not known it shall be set to 0.0.

To evaluate this surface at a parameter value

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

base_point = base_surface.evaluate(implicit_param_u, implicit_param_v)

tmp_point.x = base_point .x * cos(base_point.y)

tmp_point.y = base_point .x * sin(base_point.y)

tmp_point.z = base_point.z

Table 300 — *PRC_TYPE_SURF_Cylindrical*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Cylindrical*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
base_surface	*PtrSurface*	(Required) Base surface
tolerance	*Double*	(Required) Tolerance

8.11.11 *PRC_TYPE_SURF_Offset*

This represents a surface defined by offsetting a given surface along its normal by a specified distance. The implicit parameterization is the same as the UV domain of the base surface.

The Transformation can reposition the surface in model space using a translation, rotation, and scaling.Only the following flags are acceptable (see section 8.4.11).

Table 301 — *PRC_TYPE_SURF_Offset* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

To evaluate this surface at a parameter value

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

base_point = base_surface.evalaute(implicit_param_u, implicit_param_v)

base_normal = base_surface.evaluate_normal(implicit_param_u, implicit_param_v)

tmp_point = base_point + (offset_distance * base_normal)

Table 302 — *PRC_TYPE_SURF_Offset*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Offset*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
base_surface	*PtrSurface*	(Required) Base surface
offset_distance	*Double*	(Required) Offset distance

8.11.12 *PRC_TYPE_SURF_Pipe*

This surface type is currently not supported and reserved for future use.

Table 303 — *PRC_TYPE_SURF_Pipe*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Pipe*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
center_curve	*PtrCurve*	(Required) Center curve
orgin_curve	*PtrCurve*	(Required) Origin curve
radius	*Double*	(Required) Radius of pipe

8.11.13 *PRC_TYPE_SURF_Plane*

The represents the canonical definition of a planar surface which is defined as the XY plane

The canonical representation of a plane has

— the x-axis set to (1, 0, 0)

— the y axis set to (0, 1, 0)

— the z axis set to (0,0,1)

the implicit parameterization is the uv domain [-infinite_param, infinite_param] x [-infinite_param, infinite_param]

The implicit parameter value for a plane is calculated using

— Implicit_param.u = u_parameter_coeff_a * param.u + u_parameter_coeff_b;

— Implicit_param.v = v_parameter_coeff_a * param.v + v_parameter_coeff_b

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 304 — *PRC_TYPE_SURF_Plane* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

To evaluate this surface at a parameter value

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

tmp_point.x = implicit_param_u

tmp_point.y = implicit_param_v

tmp_point.z = 0

Table 305 — *PRC_TYPE_SURF_Plane*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Plane*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*Domain*	(Required) Define parameterization and trimming information
u_parameter_coeff_a	*Double*	(Required) U parameter coeff_a
v_parameter_coeff_a	*Double*	(Required) V parameter coeff_a
u_parameter_coeff_b	*Double*	(Required) U parameter coeff_b
v_parameter_coeff_b	*Double*	(Required) V parameter coeff_b

8.11.14 *PRC_TYPE_SURF_Ruled*

This represents a ruled surface defined by connecting points on each of two curves by a straight line. It is required that both curves are defined over the same interval since points at equal parameter value along each curve are connected by a straight line.

The implicit parameterization of the ruled surface is [0, 1] x [first_curve.interval.min, first_curve.interval.max].

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 306 — *PRC_TYPE_SURF_Ruled* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

To evaluate this surface at a parameter value

Calculate the implicit_parameter from the given parameter using this surface‘s UVParameterization data.

base_point1 = first_curve.evaluate(implicit_param_v)

base_point2 = second_curve.evaluate(implicit_param_v)

tmp_point.x = (1.0-implicit_param_u) * base_point1.x + implicit_param_u * base_point2.x

tmp_point.y = (1.0-implicit_param_u) * base_point1.y + implicit_param_u * base_point2.y

tmp_point.z = (1.0-implicit_param_u) * base_point1.z + implicit_param_u * base_point2.z

Table 307 — *PRC_TYPE_SURF_Ruled*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Ruled*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
first_curve	*PtrCurve*	(Required) First curve
second_curve	*PtrCurve*	(Required) Second curve

8.11.15 *PRC_TYPE_SURF_Sphere*

This represents the canonical definition of a spherical surface centered at the origin. The u parameter corresponds to a circle in the XY plane with 0.0 being the x-axis and positive angles measured around the z-axis using the right hand rule. The v parameter corresponds to a semi-circle in the plane defined by the u parameter and passing through the z-axis with the XY plane being 0.0 and positive angles above the XY plane and negative angles below the XY plane. The implicit parameterization of a sphere is $[0, 2\pi]$ x $[-\pi/2, \pi/2]$.

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 308 — *PRC_TYPE_SURF_Sphere* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

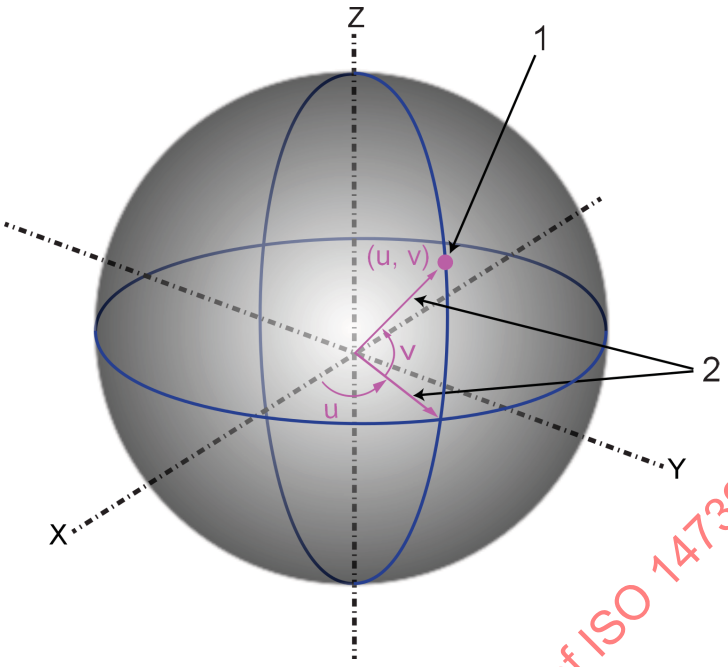
To evaluate this surface at a parameter value

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

tmp_point.x = radius * cos(implicit_param_v) * cos(implicit_param_u)

tmp_point.y = radius * cos(implicit_param_v) * sin(implicit_param_u)

tmp_point.z = radius * sin(implicit_param_v)



Key

1 point (u, v) on surface of the sphere

2 radius of the sphere

u is measured in the X-Y plane in radians determining a plane which includes the Z-axis on which v is measured in radians from the X-Y plane

Figure 18 — Example of a sphere

Table 309 — *PRC_TYPE_SURF_Sphere*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Sphere*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
radius	*Double*	(Required) Radius

8.11.16 *PRC_TYPE_SURF_Revolution*

This represents a surface of revolution defined as revolving a base curve around an axis of revolution.

The implicit parameterization is $[0, 2\pi]$, [base_curve.interval.min, base_curve.interval.max]. The u value is in radians and the 0.0 value corresponds to a point on the base_curve. Positive angles are in the direction determined by the axis direction using the right hand rule.

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 310 — *PRC_TYPE_SURF_Revolution* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

The axis of revolution is defined by an origin and the cross product of x-axis and y-axis.

The tolerance is used internally but does not take part of the definition of the surface. It indicates an appropriate tolerance that can be used to determine if the base_curve lies in a plane passing by the axis of revolution. If not known it shall be set to 0.0. See Section 5.7.

To evaluate this surface at a parameter value

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

base_point = Base_curve.evaluate(implicit_param_v)

point_on_axis = axis_of_revolution.project_on_DirectionZ(base_point)

tmp_axis_x = base_point – point_on_axis

tmp_axis_y = axis_of_revolution.DirectionZ ^ tmp_axis_x

tmp_point = point_on_axis + cos(implicit_param_u) * tmp_axis_x +sin(implicit_param_u) * tmp_axis_y

where ^ indicates the cross product

Table 311 — *PRC_TYPE_SURF_Revolution*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Revolution*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
tolerance	*Double*	(Required) Tolerance
orgin	*Vector3d*	(Required) Origin
x_axis	*Vector3d*	(Required) X axis
y_axis	*Vector3d*	(Required) Y axis
base_curve	*PtrCurve*	(Required) Base curve

8.11.17 *PRC_TYPE_SURF_Extrusion*

This represents an extruded surface where a base curve is extruded along a sweep vector.

The implicit parameterization of the extruded surface is

[base_curve.interval.min, base_curve.interval.max] x [-infinite_param, infinite_param].

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 312 — *PRC_TYPE_SURF_Extrusion* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

The evaluation at a parameter value param is

Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

XYZ = base_curve.evaluate(implicit_param_u) + implicit_param_v * sweep_vector;

Table 313 — *PRC_TYPE_SURF_Extrusion*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Extrusion*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
sweep_vector	*Vector3d*	(Required) Sweep vector shall be a unit vector
base_curve	*PtrCurve*	(Required) base curve

8.11.18 *PRC_TYPE_SURF_FromCurves*

The implicit parameterization is [first_curve.interval.min, first_curve.interval.max] x [second_curve.interval.min, second_curve.interval.max].

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 314 — *PRC_TYPE_SURF_FromCurves* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

The evaluation at a parameter value param is

Eval_point = first_curve.evaluate(param.u) + second_curve.evaluate(param.v) – origin;

Table 315 — *PRC_TYPE_SURF_FromCurves*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_FromCurves*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
orgin	*Vector3d*	(Required) Origin
first_curve	*PtrCurve*	(Required) First curve
second_curve	*PtrCurve*	(Required) Second curve

8.11.19 *PRC_TYPE_SURF_Torus*

This represents the cannonical definition of a torus centered at the origin with the major axis in the XY plane. The implicit parameterization is [0, 2 π] x [0, 2 π] where the u parameter is 0.0 corresponding to a circle on the XZ plane with radius = minor_radius and the v parameter corresponds to a circle on the XY plane with radius = major_radius + minor_radius.

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 316 — *PRC_TYPE_SURF_Torus* transformation

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

To evaluate this surface at a parameter value

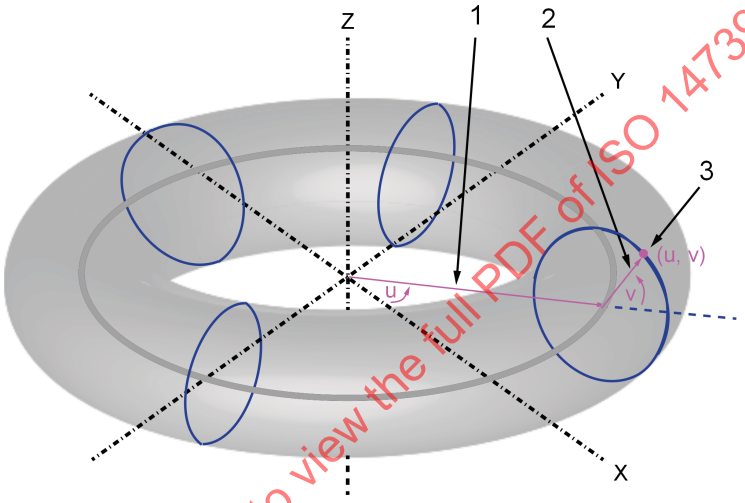
Calculate the implicit_parameter from the given parameter using this surface's UVParameterization data.

radius = major_radius + minor_radius * cos(implicit_param_v);

tmp_point.x = radius * cos(implicit_param_u)

tmp_point.y = radius * sin(imiplicit_param_u)

tmp_point.z = minor_radius * sin(implicit_param_v)



Key

1 major radius

2 minor radius

3 point on torus surface (u, v)

Major torus axis is in the X-Y plane

Figure 19 — Example of a torus

Table 317 — *PRC_TYPE_SURF_Torus*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Torus*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
major_radius	*Double*	(Required) Major radius
minor_radius	*Double*	(Required) Minor radius

8.11.20 *PRC_TYPE_SURF_Transform*

A Transform surface is defined by applying a 3D mathematical transformation to a base surface.

The implicit parameterization is the same as the base surface.

The Transformation can reposition the surface in model space using a translation, rotation, and scaling. Only the following flags are acceptable (see section 8.4.11).

Table 318 — *PRC_TYPE_SURF_Transform* transformations

Value	Type Name	Data Description
0x00	*PRC_TRANSFORMATION_Identity*	Identity
0x01	*PRC_TRANSFORMATION_Translate*	Translation
0x02	*PRC_TRANSFORMATION_Rotate*	Rotation
0x08	*PRC_TRANSFORMATION_Scale*	Uniform scale

The UVParameterization enables the surface to be reparameterized and trimmed.

The mathematical transformation can be NULL.

The nominal evaluation formula for a transform curve at param value is:

Calculate the implicit_parameter from the given parameter using this transform surface's Parameterization data.

Tmp_point = base_surface.evaluate(implicit_parameter);

If (math_transformation != NULL)

Eval_point =math_transformation.evaluate(tmp_point);

Else

Eval_point = tmp_point;

Table 319 — *PRC_TYPE_SURF_Transform*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_SURF_Transform*
curve_data	*ContentSurface*	(Required) Common surface data
tranform	*Transformation*	(Required) Position surface into model space
parameterization	*UVParameterization*	(Required) Define parameterization and trimming information
base_surface	*PtrSurface*	(Required) Base surface
math_transformation	*PRC_TYPE_MATH_FCT_3D*	(Required) 3D mathematical transformation

8.11.21 *PRC_TYPE_SURF_Blend04*

This type is currently not supported and is reserved for future use.

8.12 Mathematical Operator

8.12.1 Entity Types

Table 320 — Mathematical operator entity types

Type Name	Type Value	Referenceable
PRC_TYPE_MATH	*PRC_TYPE_ROOT* + 900	
PRC_TYPE_MATH_FCT_1D	*PRC_TYPE_MATH* + 1	
PRC_TYPE_MATH_FCT_1D_Polynom	*PRC_TYPE_MATH_FCT_1D* + 1	
PRC_TYPE_MATH_FCT_1D_Trigonometric	*PRC_TYPE_MATH_FCT_1D* + 2	
PRC_TYPE_ MATH_FCT_1D_Fraction	*PRC_TYPE_MATH_FCT_1D* + 3	
PRC_TYPE_ MATH_FCT_1D_ArctanCos	*PRC_TYPE_MATH_FCT_1D* + 4	
PRC_TYPE_ MATH_FCT_1D_Combination	*PRC_TYPE_MATH_FCT_1D* + 5	
PRC_TYPE_MATH_FCT_3D	*PRC_TYPE_MATH* + 10	
PRC_TYPE_MATH_FCT_3D_Linear	*PRC_TYPE_MATH_FCT_3D* + 1	
PRC_TYPE_MATH_FCT_3D_NonLinear	*PRC_TYPE_MATH_FCT_3D* + 2	

8.12.2 *PRC_TYPE_MATH*

Abstract class for mathematical operators.

8.12.3 *PRC_TYPE_MATH_FCT_1D*

Base type for a equation of one variable. The following are legal types of equations;

— Polynomial equation *PRC_TYPE_MATCH_FCT_1D_Polynom*

— Cosine based equation *PRC_TYPE_MATCH_FCT_1D_Trigonometric*

— Fraction of two 1D equations *PRC_TYPE_MATCH_FCT_1D_Fraction*

— Specific equation *PRC_TYPE_MATCH_FCT_1D_ArctanCos*

— Combination of 1D equation *PRC_TYPE_MATCH_FCT_1D_Combination*

8.12.4 *PRC_TYPE_MATH_FCT_1D_Polynom*

This represents 1D polynomial equation.

The evaluation formula for a given parameter value is

```
output = 0.0

For (i=0; i<number_of_coefficients; i++) {

    output = output + coefficient[i] * pow(param, i);

}
```

Table 321 — *PRC_TYPE_MATH_FCT_1D_Polynom*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MATH_FCT_1D_Polynom*
number_of_coefficients	*UnsignedInteger*	(Required) Number of coefficients in polynomial
coefficient	Array <*Double*> [**number_of_coefficients**]	(Required) Array of coefficients

8.12.5 *PRC_TYPE_MATH_FCT_1D_Trigonometric*

This represents a 1D trigonometric equation.

The evaluation formula for param value is

Output = dc_offset + amplitude * cos(param*freq – phase)

Table 322 — *PRC_TYPE_MATH_FCT_1D_Trigonometric*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MATH_1D_Trigonometric*
amplitude	*Double*	(Required) Amplitude
phase	*Double*	(Required) Phase
freq	*Double*	(Required) Frequency
dc_offset	*Double*	(Required) Dc_offset

8.12.6 *PRC_TYPE_MATH_FCT_1D_Fraction*

This represents a 1D equation that is a fraction of two 1D equations.

The evaluation formula for param value is

Output = Numerator / Denominator

Table 323 — *PRC_TYPE_MATH_FCT_1D_Fraction*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MATH_FCT_1D_Fraction*
numerator	*PRC_TYPE_MTH_FCT_1D*	(Required) Numerator
denominator	*PRC_TYPE_MTH_FCT_1D*	(Required) Denominator

8.12.7 *PRC_TYPE_MATH_FCT_1D_ArctanCos*

This represents a 1D trignometric arcfunction.

The evaluation formula for param is

Output = atan((amplitude * cos((param * frequency) + phase))) * a

Where

	$\pi g/2 <$ amplitude $< \pi g/2$
	263 < (param * frequency) < 263

E Reserved_double shall be set to 0.0

Table 324 — *PRC_TYPE_MATH_FCT_1D_ArctanCos*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MATH_FCT_1D_ArctanCos*
a	*Double*	(Required) A
amplitude	*Double*	(Required) Amplitude
frequency	*Double*	(Required) Frequency
phase	*Double*	(Required) Phase
e	*Double*	(Required) E ; note that this is not used

8.12.8 *PRC_TYPE_MATH_FCT_1D_Combination*

8.12.8.1 General

This represents a function that is a combination of several 1D functions

The evaluation formula at a param value is

Output = 0.0

For (i=0; i<number_of_coefficients; i++) {

Output = output + coefficient[i] * function[i];

}

Table 325 — *PRC_TYPE_MATH_FCT_1D_Combination*

Name	Data Type	Data Description
Required	*UnsignedInteger*	(Required) *PRC_TYPE_MATH_FCT_1D_Combination*
number_of_coefficients	*UnsignedInteger*	(Required) Number of coefficients (or functions)
coefficient	Array <*CombinationFunctions*> [**number_of_coefficients**]	Array of coefficients and functions

8.12.8.2 *CombinationFunctions*

Table 326 — CombinationFunctions

Name	Data Type	Data Description
coefficient	*Double*	(Required) Coefficient
function	*PRC_TYPE_MATH_FCT_1D*	(Required) Any 1D mathematical function

8.12.9 *PRC_TYPE_MATH_FCT_3D*

Abstract class for 3D mathematical functions.

The following are legal 3D mathematical functions:

PRC_TYPE__MATH_FCT_3D_Linear

PRC_TYPE_MATH_FCT_3D_NonLinear

8.12.10 *PRC_TYPE_MATH_FCT_3D_Linear*

The represents a 3D linear function.

The evaluation formula at a param value is

output.x= mat[0] [0] * param.x + mat[1] [0] * param.y + mat[2][0] * param.z + vect[0]

output.y= mat[0] [1] * param.x + mat[1] [1] * param.y + mat[2][1] * param.z + vect[1]

output.z= mat[0] [2] * param.x + mat[1] [2] * param.y + mat[2][2] * param.z + vect[2]

Table 327 — *PRC_TYPE_MATH_FCT_3D_Linear*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MATH_FCT_3D_Linear*
mat	Array *Double*[3] [3]	(Required) Matrix[3][3] stored row by row
vect	Array *Double*[3]	(Required) Vector of 3 coordinates

8.12.11 *PRC_TYPE_MATH_FCT_3D_NonLinear*

This represents a 3D non-linear mathematical function.

The evaluation formula for param is as follows:

tmp_result = left_transformation.evaluate(param);

output.x = tmp_result.x * cos(tmp_result.y * d2);

output.y = tmp_result.x * sin(tmp_result.y *d2);

output.z = tmp_result.z;

output = right_transformation.evaluate(output);

Note that the following are reserved for future use but should be initialized to default values:

— Reserved_double shall be set to π

— Reserved_int_1 shall be set to 2

— Reserved_int_2 shall be set to 2

— Reserved_int_3 shall be set to 0

Table 328 — *PRC_TYPE_MATH_FCT_3D_NonLinear*

Name	Data Type	Data Description
	UnsignedInteger	(Required) *PRC_TYPE_MATH_FCT_3D_NonLinear*
left_transformation	*PRC_TYPE__MATH_FCT_3D*	(Required) Left 3D non-linear transformation
right_transformation	*PRC_TYPE__MATH_FCT_3D*	(Required) Rignt 3D non-linear transformation
d2	*Double*	(Required) d2
reserved_double	*Double*	(Required) Reserved_double (not used)
reserved_int_1	*Integer*	(Required) Reserved_int_1 (not used)
reserved_int_2	*Integer*	(Required) Reserved_int_2 (not used)
reserved_int_3	*Integer*	(Required) Reserved_int_3 (not used)

9 Schema Definition

9.1 General

The PRC File Format Specification provides a mechanism for describing data stored within a PRC File. A schema definition language can describe changes between versions of the PRC File Format Specification. See 5.2 for a basic description of versioning in PRC.

This mechanism allows new entity types to be defined and information to be added for an existing entity type and written to a PRC File which is readable by PRC File Reader software written to conform to previous versions of the PRC File Format Specification.

New data for an existing entity type shall be added at the end and before UserData (if any); this rule applies to all PRC_TYPE entities.

The schema definition language contains tokens to

— describe the primary data in an entity (Boolean, Integer, UnsignedInteger, Double, Character, String, Vector 2D, Vector3D, Interval, Domain, BoundingBox);

— define a block of data which may indicate a version number;

— describe conditional processing (if/then/else, relational tests such as less than, less than or equal to, greater than, greater than or equal to, equal to, not equal to);

— define variables which may be assigned values;

— perform simple binary mathemathical operations such as multiplication, division, addition, subtraction;

— test the type of geometry present in a file;

— read a curve or surface.

Using these schema tokens, new entity types or additions to an existing entity type may be described. This capability is used to describe changes between versions of the PRC File Format Specification.

The following describes processing the data in a PRC File from the point of view of a PRC File Writer and a PRC File Reader:

— The current data in the PRC File (indicated by the authoring_version) consists of base entity data (indicated by the mimimal_version_for_read) plus a delta describing new fields within base entities plus new entities (indicated by schema definitions in the PRC File).

— A PRC File Writer

  — Is based on a current_version of the PRC File Format Specification.

  — Writes **minimal_version_for_read** to indicate the structure and content of the base data that the Writer is based on.

  — Writes authoring_version to indicate the structure and content of the current file.

  — If **minimal_version_for_read** = **authoring_version**, the Writer does not have to write any extra data.

  — If mimimal_version_for_read < **authoring_version**, the PRC File Writer shall include schema definitions in the PRC File describing the delta: the new fields of base entities and all new entities, but only if such entities actually exist in the file.

— A PRC File Reader

  — Is based on a current_version of the PRC File Format Specification.

  — If the current_version is less that the **minimal_version_for_read**, an error has occurred and the Reader should not continue to process the file. The Reader is built on a version of the specification that preceeds the (base) version of the specification that the Writer was built on.

  — If the **current_version** is the same as the authoring_version, the PRC File Reader may read all of the data in the file without reference to any schema definitions. Both the Writer and Reader have the same view of the data in the PRC File.

  — Otherwise, the Reader is built on a version of the specification that knows the structure of the base entities in the file and will use the schema definitions to process new data from the PRC File. When reading an entity

    — If the entity type is a base entity type, the reader will read all of the base data for the entity from the file. It will then use the schema to skip new information in the file.

    — If the entity type is a new entity, the Reader will use the schema to skip the information in the PRC File

9.2 Enumeration Of Schema Tokens

The following table provides the value, name, and short description of the schema tokens that are used to describe a PRC entity. See 9.3 for a description of how these tokens are used.

Table 329 — Enumeration Of Schema Tokens

Value	*Token Name*	**Description**
0	*EPRCSchema_Data_Boolean*	Read a Boolean
1	*EPRCSchema_Data_Double*	Read a Double
2	*EPRCSchema_Data_Character*	Read a Character
3	*EPRCSchema_Data_Unsigned_Integer*	Read an UnsignedInteger
4	*EPRCSchema_Data_Integer*	Read an Integer
5	*EPRCSchema_Data_String*	Read a String
6	*EPRCSchema_Parent_Type*	Parent type of the current object
7	*EPRCSchema_Vector_2D*	Read a Vector2d
8	*EPRCSchema_Vector_3D*	Read a Vector3d
9	*EPRCSchema_Extent_1D*	Read an Interval
10	*EPRCSchema_Extent_2D*	Read a Domain
11	*EPRCSchema_Extent_3D*	Read a BoundingBox
12	*EPRCSchema_Ptr_Type*	Read a specified typed object
13	*EPRCSchema_Ptr_Surface*	Read a surface
14	*EPRCSchema_Ptr_Curve*	Read a curve
15	*EPRCSchema_For*	Loop of instructions
16	*EPRCSchema_SimpleFor*	Loop of instructions
17	*EPRCSchema_If*	Condition block
18	*EPRCSchema_Else*	Condition block
18	*EPRCSchema_Block_Start*	Define an instruction block
20	*EPRCSchema_Block_Version*	Define a versioned instruction block
21	*EPRCSchema_Block_End*	End of a block
22	*EPRCSchema_Value_Declare*	Declare a global value
23	*EPRCSchema_Value_Set*	Set a global value
24	*EPRCSchema_Value_DeclareAndSet*	Declare and set a global value
25	*EPRCSchema_Value*	Access a global value
26	*EPRCSchema_Value_Constant*	Value constant
27	*EPRCSchema_Value_For*	Value of the for-loop
28	*EPRCSchema_Value_CurveIs3D*	Specific value (9.3)
29	*EPRCSchema_Operator_MULT*	* operator
30	*EPRCSchema_Operator_DIV*	/ operator
31	*EPRCSchema_Operator_ADD*	+ operator
32	*EPRCSchema_Operator_SUB*	– operator
33	*EPRCSchema_Operator_LT*	< operator
34	*EPRCSchema_Operator_LE*	<= operator

Table 329 *(continued)*

35	*EPRCSchema_Operator_GT*	> operator
36	*EPRCSchema_Operator_GE*	>= operator
37	*EPRCSchema_Operator_EQ*	== operator
38	*EPRCSchema_Operator_NEQ*	!= operator

9.3 Schema Processing

9.3.1 EPRCSchema_Data_Boolean

The next field in the file is a **Boolean**.

9.3.2 EPRCSchema_Data_Double

The next field in the file is a **Double**.

9.3.3 EPRCSchema_Data_Character

The next field in the file is a **Character**.

9.3.4 EPRCSchema_Data_Unsigned_Integer

The next field in the file is an **UnsignedInteger**.

9.3.5 EPRCSchema_Data_Integer

The next field in the file is an **Integer**.

9.3.6 EPRCSchema_Data_String

The next field in the file is a **String**.

9.3.7 EPRCSchema_Parent_Type

This token indicates that the next token in the token array is the parent type. Legal values of the parent types are ***PRC_TYPE_SURF*** and ***PRC_TYPE_CRV***.

Parent type is used in case of inheritance from another object. There is no specific field in the PRC File corresponding to this type of token. In this case, the parent object (abstract curve or surface) is read before the object.

9.3.8 EPRCSchema_ Vector_2D

The next field in the file is a **Vector2d**.

9.3.9 EPRCSchema_ Vector_3D

The next field in the file is a **Vector3d**.

9.3.10 EPRCSchema_Extent_1D

The next field in the file is an **Interval**.

9.3.11 EPRCSchema_Extent_2D

The next field in the file is a **Domain**.

9.3.12 EPRCSchema_Extent_3D

The next field in the file is a **BoundingBox**.

9.3.13 EPRCSchema_Ptr_Type

This token indicates that the next token in the token array corresponds to the type of object that is next in the file. The legal types that may be read from the file are

— PRC_TYPE_SURF

— PRC_TYPE_CRV

— PRC_TYPE_TOPO

— PRC_TYPE_RI

— PRC_TYPE_MKP_Markup

— PRC_TYPE_MATH_FCT_1D

— PRC_TYPE_MATH_FCT_3D

— The integer number representing a new entity type defined in the schema sections of a PRC File.

9.3.14 EPRCSchema_Ptr_Surface

The next field in the file is a surface (**PRC_TYPE_SURF**). This is equivalent to a **EPRCSchema_Ptr_Type** with an implicit **PRC_TYPE_SURF** object type.

9.3.15 EPRCSchema_Ptr_Curve

The next field in the file is a curve (**PRC_TYPE_CRV**). This is equivalent to a **EPRCSchema_Ptr_Type** with an implicit **PRC_TYPE_CRV** object type.

9.3.16 EPRCSchema_For

An EPRCSchema_For defines a for loop as three tokens

— The token EPRCSchema_For indicating a for loop follows.

— A schema block defining the number of times to execute the block.

— A schema block defining the block to be executed.

Table 330 — EPRCSchema_For example:

Token Sequence	Pseudo Code

EPRCSchema_For
EPRCSchema_Operator_ADD
EPRCSchema_Data_Integer
EPRCSchema_Value_Constant
6
EPRCSchema_Data_Double
	int loop_end = read_integer() + 6;
	for (int i=0; i<loop_end; i++) {
	read_double();
	}

9.3.17 EPRCSchema_SimpleFor

An EPRCSchema_SimpleFor defines a for loop as two tokens

— The token EPRCSchema_For indicating a for loop follows.

— The number of time to execute the block is not defined in the schema. For a SimpleFor loop the loop counter shall be a simple integer in the PRC File and not defined by a more complex schema block.

— A schema block defining the block to be executed.

The next token is the loop counter number and the token after that is the token to execute counter number of times.

Table 331 — EPRCSchema_SimpleFor example:

Token Sequence	Pseudo Code

EPRCSchema_SimpleFor
EPRCSchema_Block_Start
EPRCSchema_Data_Integer
EPRCSchema_Data_Double
EPRCSchema_If
EPRCSchema_Operator_EQ
EPRCSchema_Value_For
EPRCSchema_Value_Constant
0
EPRCSchema_Data_Double
EPRCSchema_Block_End
	int loop_end = read_unsigned_integer();
	for (int i=0; i<loop_end; i++)
	{
	read_integer();
	read_double()
	if (i == 0)
	read_double();
	}

9.3.18 EPRCSchema_If and EPRCSchema_Else

This token indicates the start of a conditional token clause. The next token instruction corresponds to the conditional value, and executes the following token instruction if the conditional value is TRUE. If there is an EPRCSchema_Else token.

Table 332 — EPRCSchema_If and EPRCSchema_Else example:

Token Sequence	Pseudo Code

EPRCSchema_If
EPRCSchema_OperatorEQ
EPRCSchema_Data_Integer
EPRCSchema_Value_Constant
3
EPRCSchema_Vector_3D
EPRCSchema_Else
EPRCSchema_Vector_2D
	tmp_integer = read_integer();
	If (tmp_integer == 3)
	read_vector_3d();
	Else
	read_vector_2d();

9.3.19 EPRCSchema_Block_Start

This token indicates a block of tokens terminated by an EPRCSchema_Block_End. The block of tokens is not versioned.

9.3.20 EPRCSchema_Block_Version

This token indicates a versioned block of tokens terminated by an EPRCSchema_Block_End. The token following the EPRCSchema_Block_Version is the version number. If the version number is less than or equal to the current version, the block version is ignored.

Table 333 — EPRCSchema_Block_Version example:

Token Sequence	Pseudo Code

EPRCSchema_Block_Version
150
EPRCSchema_Data_Integer
EPRCSchema_Vector_3D
EPRCSchema_Block_End
	If (current_version < 150) {
	read_integer();
	read_vector_3d();
	}

9.3.21 EPRCSchema_Block_End

This token indicates the end of a token block.

9.3.22 EPRCSchema_Value_Declare

This token declares and sets to zero a new value indexed by the next instruction value. All values are local to the block instruction.

Table 334 — EPRCSchema_Value_Declare example:

Token Sequence	Pseudo Code
EPRCSchema_Value_Declare 5	int value_5 = 0;

9.3.23 EPRCSchema_Value_Set

Sets a value indexed by the next token instruction value. Before it can be used, a variable shall be declared.

Table 335 — EPRCSchema_Value_Set Example:

Token Sequence	Pseudo Code
EPRCSchema_Value_Declare 5 EPRCSchema_Value_Set 5 EPRC_Schema_Data_Integer	int value_5 = 0; value_5 = read_integer();

9.3.24 EPRCSchema_Value_DeclareAndSet

Declares and sets a new value indexed by the next instruction value.

Table 336 — EPRCSchema_Value_DeclareAndSet example:

Token Sequence	Pseudo Code
EPRCSchema_Value_DeclareAndSet 5 EPRC_Schema_Data_Integer	int value_5 = read_integer();

9.3.25 EPRCSchema_Value

Retrieves the content of the value indexed by the next instruction value. Before it can be used, a variable shall be declared.

Table 337 — EPRCSchema_Value example:

Token Sequence	Pseudo Code

EPRCSchema_Value_DeclareAndSet
5
EPRCSchema_Value_Constant
2
EPRCSchema_Value_DeclareAndSet
2
EPRCSchema_Data_Integer
EPRCSchema_DeclareAndSet
6
EPRCSchema_Operator_ADD
EPRCSchema_Value
5
EPRCSchema_Value
2
	int value_5 = 2;
	int value_2 = read_integer();
	int value_6 = value_5 + value_2;

9.3.26 EPRCSchema_Value_Constant

Retrieves a constant value given by the next token.

Table 338 — EPRCSchema_Value_Constant example:

Token Sequence	Pseudo Code
EPRCSchema_Value_DeclareAndSet 5 EPRC_Schema_Value_Constant 10	int value_5 = 10;

9.3.27 EPRCSchema_Value_For

Retrieves the value of the current for-loop.

Example: See 10.3.17 example.

9.3.28 EPRCSchema_Value_CurveIs3D

If the current object is a curve, return TRUE if it is a 3D curve, FALSE if it is a 2D curve.

9.3.29 EPRCSchema_Operator_MULT

Binary operator for multiplication.

9.3.30 EPRCSchema_Operator_DIV

Binary operator for division.

9.3.31 EPRCSchema_Operator_ADD

Binary operator for addition.

9.3.32 EPRCSchema_Operator_SUB

Binary operator for subtraction.

9.3.33 EPRCSchema_Operator_LT

Relational operator for less than comparison.

Table 339 — EPRCSchema_Operator_LT example:

Token Sequence	Pseudo Code

EPRCSchema_Value_DeclareAndSet
5
EPRCSchema_Data_Integer
EPRC_Schema_Value_DeclareAndSet
10
EPRCSchema_Data_Integer
EPRCSchema_If
EPRC_Schema_Operator_LT
EPRCSchema_Value
5
EPRCSchema_Value
10
EPRCSchema_Block_Start
.....
EPRCSchema_Block_End
	int value_5 = read_integer();
	int value_10 = read_integer();
	if (value_5 < value_10) {
	...
	}

9.3.34 EPRCSchema_Operator_LE

Relational operator for less than or equal to comparison.

9.3.35 EPRCSchema_Operator_GT

Relational operator for greater than comparison.

9.3.36 EPRCSchema_Operator_GE

Relational operator for greater than or equal to comparison.

9.3.37 EPRCSchema_Operator_EQ

Relational operator for equal to comparison.

9.3.38 EPRCSchema_Operator_NEQ

Relational operator for not equal to comparison.

9.4 Schema Requirements and Examples

9.4.1 General

The following are some requirements of the schema definition language and its usage

— All of the examples have an enclosing block which is not a version block. This is a requirement.

— New curves (or surfaces) have either the PRC_TYPE_CRV or a particular curve entity type as the EPRCSchema_FatherParent_Type. The FatherParent_Type indicates what data will be inherited by the entity in the schema.

— For example, for PRC_TYPE_CRV and PRC_TYPE_SURFACE, attributes, transform, and parametrization are inherited from the FatherParent_Type.

— The example to add a field to an existing entity did not define the previous data. All new data shall be added to the end of an existing entity.

9.4.2 An Existing Entity

This is the schema definition of PRC_TYPE_MISC_GeneralTransformation which was defined in version 8137 of the PRC File Format Specification.

Table 340 — An Existing Entity

Data Type	Token Sequence	Comments
UnsignedInteger	*PRC_TYPE_MISC_GeneralTransformation*	Value indicating type of entity
UnsignedInteger	*8*	N umber of tokens that follow
UnsignedInteger	*EPRCSchema_Block_Start*	Start of all blocks
UnsignedInteger	*EPRCSchema_Block_Version*	Beginning of a versioned block
UnsignedInteger	*8137*	Version number
UnsignedInteger	*EPRCSchema_SimpleFor*	For loop to get the data
UnsignedInteger	*16*	Read 16
UnsignedInteger	*EPRCSchema_Data_Double*	Doubles in the transformation matrix
UnsignedInteger	*EPRCSchema_Block_End*	End of new versioned block
UnsignedInteger	*EPRCSchema_Block_End*	End of all blocks

9.4.3 Existing *PRC_TYPE_CRV_Polyline*

The following is the schema for the existing PRC_TYPE_CRV_Polyline.

Table 341 — Existing *PRC_TYPE_CRV_Polyline*

Data Type	Token Sequence	Comments
UnsignedInteger	*PRC_TYPE_CRV_Polyline*	Value indicating type of entity
UnsignedInteger	*14*	Number of tokens that follow
UnsignedInteger	*EPRCSchema_Block_Start*	Start of all blocks
UnsignedInteger	*EPRCSchema_Block_Version*	Beginning of a versioned block
UnsignedInteger	*Version_0*	Version number
UnsignedInteger	*EPRCSchema_Parent_Type*	The class of the parent entity
UnsignedInteger	*PRC_TYPE_CRV*	Is a curve
UnsignedInteger	*EPRCSchema_SimpleFor*	For loop to get the data
UnsignedInteger	*Integer value*	Number of points in the polyline
UnsignedInteger	*EPRCSchema_If*	if the
UnsignedInteger	*EPRCSchema_Value_CurveIs3D*	Curve is a 3D curve
UnsignedInteger	*EPRCSchema_Vector_3D*	The data field will be a 3D vector
UnsignedInteger	*EPRCSchema_Else*	otherwise
UnsignedInteger	*EPRCSchema_Vector_3D*	The data field will be a 2D vector
UnsignedInteger	*EPRCSchema_Block_End*	End of new versioned block
UnsignedInteger	*EPRCSchema_Block_End*	End of all blocks

9.4.4 Add a Field to Existing Entity

In this example, a string field is added to the existing curve *PRC_TYPE_CRV_Line*.

For an existing entity, new data shall be added to the end.

Table 342 — Add a Field to Existing Entity

Data Type	Token Sequence	Comments
UnsignedInteger	*PRC_TYPE_CRV_Line*	This is the entity type of the existing type of curve (line).
UnsignedInteger	*6*	Number of tokens to follow
UnsignedInteger	*EPRCSchema_Block_Start*	Start of all blocks
UnsignedInteger	*EPRCSchema_Block_Version*	Beginning of a versioned block
UnsignedInteger	*9149*	File version number 149th day of 2009
UnsignedInteger	*EPRCSchema_Data_String*	Data field
UnsignedInteger	*EPRCSchema_Block_End*	End of new versioned block
UnsignedInteger	*EPRCSchema_Block_End*	End of all blocks

9.4.5 Add a New Curve

In this example, a new curve type with 2 doubles and a pointer to a curve is added to the schema.

Table 343 — Add a New Curve

Data Type	Token Sequence	Comments
UnsignedInteger	*PRC_TYPE_CRV_NewCurve*	This is the entity type of the new curve.
UnsignedInteger	*10*	Number of tokens to follow
UnsignedInteger	*EPRCSchema_Block_Start*	Start of all blocks
UnsignedInteger	*EPRCSchema_Block_Version*	Beginning of a versioned block
UnsignedInteger	*Version_1*	File version number
UnsignedInteger	*EPRCSchema_Parent_Type*	The class of the parent entity
UnsignedInteger	*PRC_TYPE_CRV*	Is a curve
UnsignedInteger	*EPRCSchema_Double*	Data field is a double
UnsignedInteger	*EPRCSchema_Double*	Data field is a double
UnsignedInteger	*EPRCSchema_Ptr_Curve*	Data field is a curve
UnsignedInteger	*EPRCSchema_Block_End*	End of new versioned block
UnsignedInteger	*EPRCSchema_Block_End*	End of all blocks

9.4.6 Multiple Revisions to an Entity Type

Table 344 — Multiple Revisions to an Entity Type

Data Type	Token Sequence	Comments
UnsignedInteger	*PRC_TYPE_CRV_NewCurve*	This is the entity type of the new curve.
UnsignedInteger	*14*	Number of tokens to follow
UnsignedInteger	*EPRCSchema_Block_Start*	Start of all blocks
UnsignedInteger	*EPRCSchema_Block_Version*	Start of
UnsignedInteger	*Version_1*	Version number of Version_1 block
UnsignedInteger	*EPRCSchema_Parent_Type*	The parent type
UnsignedInteger	*PRC_TYPE_CRV_Line*	Is a PRC_TYPE_CRV_Line
UnsignedInteger	*EPRCSchema_Data_Double*	Data field 1 is a double
UnsignedInteger	*EPRCSchema_Data_Double*	Data field 2 is a double
UnsignedInteger	*EPRCSchema_Block_End*	End of version 1 block
UnsignedInteger	*EPRCSchema_Block_Version*	Start of
UnsignedInteger	*Version_2*	Version number of Version_2 block
UnsignedInteger	*EPRCSchema_SimpleFor*	Simple for loop to read
UnsignedInteger	*EPRCSchema_Data_Integer*	An integer
UnsignedInteger	*EPRCSchema_Block_End*	End of version 2 block
UnsignedInteger	*EPRCSchema_Block_End*	End of all blocks

10 I/O Algorithms

10.1 Getnumberofbitsusedtostoreunsignedinteger

Computes the number of bits needed to serialize an unsigned integer. **uValue** must be less than (UINT_MAX / 2).

The following code example shows how to compute the number of bits needed for an unsigned integer.

unsigned GetNumberOfBitsUsedToStoreUnsignedInteger(unsigned uValue)

```
{
    unsigned uNbBit = 1;
    unsigned uTemp = 1;
    while(uValue > uTemp)
    {
        uTemp=uTemp<<1 | 1;
        uNbBit++;
    }
    return uNbBit;
}
```

10.2 Makeportable32bitsunsigned

Builds an array of unsigned characters from an unsigned 32-bit integer. The final value is therefore independent of the machine byte-ordering.

The following code example shows how to make a portable array of unsigned character values.

void MakePortable32BitsUnsigned(unsigned uValue, unsigned char pcValue[4])

```
{
    pcValue[0] = (unsigned char)(uValue & 0xFF);
    uValue >>= 8;
    pcValue[1] = (unsigned char)(uValue & 0xFF);
    uValue >>= 8;
    pcValue[2] = (unsigned char)(uValue & 0xFF);
    uValue >>= 8;
    pcValue[3] = (unsigned char)(uValue & 0xFF);
}
```

10.3 Writebits

Writes bits from left to right.

The following code example shows how to write bits.

iBitsCount specifies the number of bits to be written.

The bits are added to a single byte (uRemainder), which is then flushed to the final device each time 8 bits are filled.

At the end of the serialization, uRemainder is padded with zeros before writing the last byte to the device.

void WriteBits(unsigned uValue, int iBitsCount)

```
{
    static unsigned uRemainder = 0;
    static int iRemainderBits = 0;
    while (iBitsCount > 0)
```

```
    {
        int iBitsDelta = iBitsCount + iRemainderBits - 8 ;
        if (iBitsDelta == 0)
        {
            uRemainder |= uValue;
            iRemainderBits = 0 ;
            iBitsCount = 0 ;
        }
        else if (iBitsDelta < 0)
        {
            uRemainder |= uValue << (-iBitsDelta);
            iRemainderBits += iBitsCount;
            iBitsCount = 0 ;
        }
        else
        {
            int loc = uValue >> iBitsDelta;
            uRemainder |= loc ;
            uValue -= loc << iBitsDelta;
            iBitsCount -= (8 - iRemainderBits) ;
            iRemainderBits = 0 ;
        }
        if (iRemainderBits == 0)
        {
            // writing 1 byte (uRemainder) to the device

        }
    }
}
```

10.4 Writestring

Writes a UTF8-encoded string. The number of characters does not include a terminating null character.

The following code example shows how to write a string.

void WriteString(const char* pcString)

```
{
    if (pcString == 0 )
        WriteBit(0);
    else
    {
        unsigned i, iNumberOfCharacters = strlen(pcString);
        WriteBit(1);
        WriteUnsignedInteger(iNumberOfCharacters);
        for (i=0;i<iNumberOfCharacters;i++)
            WriteCharacter(pcString[i]);
    }
}
```

10.5 Writefloatasbytes

Write a float as 4 bytes.

void WriteFloatAsBytes(float fValue)

```
{
```

```
    union
    {
        float fFloat;
        unsigned int uInt;
    }unionFloatunsignedInt ;

    unionFloatunsignedInt.fFloat = (float) fValue;
    unsigned uTemp = unionFloatunsignedInt.uInt;
    WriteBits(uTemp & 0xFF, 8 );
    uTemp >>= 8 ;
    WriteBits(uTemp & 0xFF, 8 );
    uTemp >>= 8 ;
    WriteBits(uTemp & 0xFF, 8 );
    uTemp >>= 8 ;
    WriteBits(uTemp & 0xFF, 8 );
}
```

10.6 Writecharacterarray

This function writes an array of characters.

The following code example shows how to write a character array.

This function allows for both direct storage, or storage after Huffman compression (see section 11.2, Huffman Algorithm), as denoted by variable bIsCompressed. The strategy whether or not compressing is left outside the scope of the standard and can vary between implementations to try to optimize size of output file.

For instance, compression can be skipped systematically when the array size is lower than 5, since compressed strategy leads to write at least one unsigned integer.

bWriteCompressStrategy is TRUE by default except for calls from WriteCompressedIndiceArray (See 10.9) when writing Point_reference_array in Entity description (See 7.8.8.8).

```
void Huffman(
    char* pcArray,unsigned uCharArraySize,
    unsigned*& puHuffmanArray,unsigned& uHuffmanArraySize);

// gives Huffman compression strategy on case-by-case basis
bool HuffmanCompression();

void WriteCharacterArray (
    char* pcArray,unsigned uCharArraySize,
    unsigned uBitNumber,bool bWriteCompressStrategy=true)
{
    bool bIsCompressed = HuffmanCompression();
    if (bWriteCompressStrategy)
        WriteBoolean ( bIsCompressed );

    if( bIsCompressed )
    {
        // calling Huffman to create
        // puHuffmanArray and uHuffmanArraySize
        unsigned* puHuffmanArray;
        unsigned u,uHuffmanArraySize;
        Huffman(pcArray,uCharArraySize,
            puHuffmanArray,uHuffmanArraySize);

        WriteUnsignedInteger ( uHuffmanArraySize );
```