Inside the STEP File

Products, shapes and tolerances, one record at a time

Julian Chultarsky

Contents

Preface . . . 7
How the book is organised . . . 7
The running example . . . 7
What is left out . . . 7
Conventions . . . 8
About this book . . . 8

I The File and the Product

1 What STEP Is . . . 10
1.1 Three things called STEP . . . 10
1.2 How the standard is organised . . . 11
1.3 Why the entity names look the way they do . . . 11
1.4 The lineage of the application protocols . . . 12
1.5 What “conforming” means . . . 12

2 A First Look at a File . . . 13
2.1 The skeleton . . . 14
2.2 Reading a record . . . 15
2.3 The kinds of value . . . 15
2.4 `$` and `*` . . . 16
2.5 References point from specific to general . . . 16
2.6 Complex records . . . 16
2.7 What the block file contains . . . 17

3 The Header . . . 18
3.1 `FILE_DESCRIPTION` . . . 18
3.2 `FILE_NAME` . . . 18
3.3 `FILE_SCHEMA` . . . 19
3.4 The optional header records . . . 20
3.5 The record that repeats the claim . . . 20

4 Where Records Get Their Meaning: The Schema . . . 22
4.1 An entity declaration . . . 22
4.2 Inheritance and the flattened record . . . 23
4.3 Derived attributes and `*` . . . 24
4.4 Inverse attributes . . . 25
4.5 Complex instances and the external mapping . . . 25

4.6 Enumerations . . . 26
4.7 SELECT types and the typed wrapper . . . 26
4.8 Case . . . 27
4.9 What to skip, and where to find the rest . . . 27

5 Identity: The Product Chain . . . 28
5.1 The three-level core . . . 28
5.2 The contexts . . . 29
5.3 The category . . . 30
5.4 What exporters put where . . . 30
5.5 Several products in one file . . . 31

6 Measuring Things: Contexts and Units . . . 32
6.1 Representations and their contexts . . . 32
6.2 The context record . . . 32
6.3 The millimetre . . . 33
6.4 Inch files . . . 33
6.5 Uncertainty . . . 34
6.6 Derived units . . . 34
6.7 Contexts in assemblies . . . 35

7 From Product to Shape . . . 36
7.1 `PRODUCT_DEFINITION_SHAPE` . . . 36
7.2 `SHAPE_DEFINITION_REPRESENTATION` . . . 37
7.3 The shape representation . . . 37
7.4 The items . . . 38
7.5 Reading the whole path . . . 38

8 Assemblies . . . 40
8.1 Three product chains . . . 40
8.2 The tree: `NEXT_ASSEMBLY_USAGE_OCCURRENCE` . . . 41
8.3 The position: five records per occurrence . . . 42
8.4 The frame as a transform . . . 43
8.5 Walking the tree . . . 44
8.6 The other way: `MAPPED_ITEM` . . . 45
8.7 Deeper trees and other variations . . . 45

II Geometry

9 Boundary Representation from Scratch . . . 48
9.1 The topological ladder . . . 48
9.2 Sharing . . . 49
9.3 Orientation . . . 49

9.4 Closed and manifold . . . 49
9.5 What the file does not say . . . 50

10 Reading a Solid: The Block . . . 51
10.1 Vertices . . . 51
10.2 Edges . . . 51
10.3 Loops . . . 52
10.4 Faces . . . 53
10.5 Composing the orientation . . . 54
10.6 Shell and solid . . . 54
10.7 The count . . . 55

11 Curves and Surfaces . . . 56
11.1 The pin . . . 56
11.2 Closed edges and the seam . . . 57
11.3 The cylinder record . . . 58
11.4 Circles . . . 58
11.5 The hole in the bracket . . . 58
11.6 The sphere: a face with no edges . . . 61
11.7 The torus: two seams . . . 61
11.8 The cone . . . 62
11.9 The analytic vocabulary . . . 63

12 Free-Form Geometry: B-Splines . . . 65
12.1 A B-spline curve record . . . 65
12.2 A B-spline surface record . . . 66
12.3 Rational splines: the circle as a spline . . . 67
12.4 Recognising analytic geometry in spline clothing . . . 68
12.5 What the geometry chapters leave out . . . 68

13 Parameter Curves, and Meshes . . . 69
13.1 The edge that is written twice . . . 69
13.2 Meshes: tessellated geometry . . . 71

III Everything Else in the File

14 Appearance: Colours and Layers . . . 74
14.1 Reading the chain . . . 74
14.2 What gets coloured . . . 75
14.3 Named colours . . . 75
14.4 Layers . . . 75
14.5 Visibility . . . 75
14.6 Names on geometry . . . 76

15 Properties and Validation . . . 77
15.1 The property pattern . . . 77
15.2 Validation properties . . . 78
15.3 Where names hide . . . 79
15.4 What is not here . . . 80

16 Product Manufacturing Information . . . 81
16.1 Two kinds of PMI . . . 81
16.2 Graphical PMI . . . 82
16.3 Semantic PMI: features . . . 84
16.4 Semantic PMI: datums . . . 84
16.5 Semantic PMI: geometric tolerances . . . 85
16.6 Semantic PMI: dimensions . . . 86
16.7 Linking the two . . . 87
16.8 Saved views . . . 87

17 Application Protocols and the Real World . . . 89
17.1 Three protocols, one lineage . . . 89
17.2 Conformance classes . . . 90
17.3 Recommended practices . . . 90
17.4 Reading a file's origin from its habits . . . 90
17.5 When files are wrong . . . 90

18 Beyond Edition 2 . . . 92
18.1 What edition 3 adds . . . 92
18.2 How much of this is in use . . . 93
18.3 The edition 2 workaround for external files . . . 93
18.4 Other encodings . . . 93
18.5 The shape of things . . . 94

Appendices . . . 95

A The Block File, Complete . . . 96

B Entity Quick Reference . . . 100
B.1 Header . . . 100
B.2 Context and identity . . . 100
B.3 Shape binding . . . 101
B.4 Assembly . . . 101
B.5 Units and context . . . 101
B.6 Topology . . . 102
B.7 Geometry . . . 102
B.8 Tessellation (AP242) . . . 104
B.9 Appearance . . . 104

B.10 Properties **104**
B.11 PMI **105**

C Sources and Further Reading **107**

D Glossary **108**

Preface

This is a book about what is in a STEP file. Not how to parse one — that is a smaller problem than it looks, and there is a formal grammar for it — but what the records mean once you have them: why a single bracket takes three records to name, why a millimetre takes a record with three parts, why the placement of one part inside another takes five records and no numbers, and how a cylinder manages to be a closed solid with only three faces.

It is written for someone who is comfortable with data structures and databases, has opened a STEP file in a text editor, and found it opaque. No mechanical-engineering background is assumed. Where the format cannot be understood without some — boundary representation, NURBS, geometric dimensioning and tolerancing — the background is supplied in place, in boxed sections you can skip if you already have it.

How the book is organised

Part I is about the file and the product. It starts with a complete, real file of 170 records and works through it a record at a time: the syntax, the header, the schema that gives records their meaning, the product identity chain, units, and the path from a product to its shape. It ends with assemblies, which is where most of the interesting structure lives and where most tools get things wrong.

Part II is about geometry. It begins with what a boundary representation is and why topology and geometry are kept separate, then reads a block, a cylinder, a sphere and a spline part line by line. B-splines get a chapter of their own.

Part III covers everything else a file can carry: colours and layers, user properties and validation properties, product manufacturing information (PMI) in both its graphical and semantic forms, the differences between application protocols, and the third edition of the file format.

Each part builds on the last, but the chapters in Part III can be read in any order once Part I is done.

The running example

Most of the book follows one small assembly: a rectangular bracket, 60 by 20 by 10 millimetres with a 6 mm hole through it, and two identical pins, 6 mm in diameter and 30 mm long, standing on the bracket. It is deliberately trivial as a mechanical object. As a STEP file it exercises almost everything the format has: product structure, repeated components, placement transforms, planar and cylindrical faces, seam edges, inner loops, colours, and enough boilerplate to show you what to skim.

The files were produced with Open CASCADE Technology (OCCT), the open-source kernel behind FreeCAD and many commercial tools, rather than by hand; the one exception is the PMI in `bracket-pmi.step`, written by hand to the CAx Implementor Forum's recommended practice and checked by reading it back with OCCT. This matters. A hand-written example shows you what the standard permits; a real export shows you what a widely used writer actually does, including the quirks. Where other CAD systems do something different, the text says so.

Every example file is included with the book's source, in the `examples/` directory of the repository, so you can open them beside the text. The drawings of the parts are generated from those same files, so they show what the files contain: dark lines are edges, dots are vertices, and thin grey lines are the outlines of curved faces, which no file records. A listing shows only the records under discussion, but nothing it refers to is out of reach: beneath each one, the HTML edition has a panel with every record the listing points to, directly or through others, and the PDF and EPUB print them too, or, when a listing reaches a whole solid, the records it points to directly.

What is left out

Three things, deliberately.

Parsing. Tokens, lexers and the grammar are covered only as far as needed to read a record. ISO 10303-21 contains the full grammar, and it is not what makes STEP hard.

The constraint language. EXPRESS, the schema language, includes rules and functions for validating data. Readers of files rarely need them, and they would double the length of the schema chapter.

Implementation. This book describes the format. How to store it, index it or write it back out is a different book.

Conventions

Entity types appear exactly as they do in files, in upper case and in a monospaced face: `PRODUCT_DEFINITION`. Attribute names are in lower case as they appear in the schema: `product_definition.formation`. Instance names such as `#22` refer to the record with that number in whichever file the surrounding text is discussing; they are file-local and mean nothing across files.

Boxed sections headed *Engineering background* supply the mechanical-engineering or geometry context for the section that follows. Sections headed *In the wild* describe what real exporters do that the standard does not say, or says differently. Where those sections name a CAD system, they report what has been observed in files rather than anything the vendor publishes or guarantees, and exporters change. Where the text says a string or a pattern is prescribed, it says whether the schema enforces it or the recommended practices merely ask for it; the two are not the same, and only the first is checkable. Each chapter closes with a short summary headed *In brief*.

About this book

This book was written as a companion to a series of one-to-one lessons on the STEP format. Every file excerpt in it is real unless the surrounding text says otherwise: the records were written by Open CASCADE Technology 7.9, exporting small parts built for the purpose, and have been edited only in the header's file name, author and organisation fields, and in the identifier and description of each product. The excerpts for degree units, properties, validation properties, PMI, tessellation and edition-3 sections are written to the published patterns rather than exported, and are marked as such where they appear. The block file is reproduced complete in Appendix A.

The standard itself is ISO 10303. Part numbers cited in the text (Part 21, Part 42, AP242) refer to its parts. Quotations from the standard are paraphrased, not reproduced.

The book is open source. Its text is written in Quarto Markdown and lives at https://github.com/jchultarsky/step-book, where corrections and contributions are welcome.

I The File and the Product

1 What STEP Is 10

2 A First Look at a File 13

3 The Header 18

4 Where Records Get Their Meaning: The Schema 22

5 Identity: The Product Chain 28

6 Measuring Things: Contexts and Units 32

7 From Product to Shape 36

8 Assemblies 40

1. What STEP Is

> *Three things share the name, and people switch between them without noticing. The standard, the application protocol and the file; how the parts are numbered; and why the entity names look the way they do.*

Open a STEP file in a text editor and the first thing you see is that it is text, and the second is that the text does not help. Here are three lines from the middle of a small one:

```
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
  GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169))
  GLOBAL_UNIT_ASSIGNED_CONTEXT((#166,#167,#168))
  REPRESENTATION_CONTEXT('Context #1','3D Context with UNIT and UNCERTAINTY') );
```

Referred to above: 4 records from `block.step`. The complete file.

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#167 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#168 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
```

The first two are almost readable: a vertex, at a point, at the origin. The third looks like four records that have collided. By the end of Part I you will read all three fluently, and you will know why the third one has to look like that. But before any of that, it is worth being precise about what the word “STEP” refers to, because it refers to three different things and people switch between them without noticing.

1.1 Three things called STEP

The standard. ISO 10303, *Industrial automation systems and integration — Product data representation and exchange*, is a family of several hundred documents developed since the mid-1980s and still growing. It defines a data modelling language, several ways of encoding data, a library of shared data definitions for everything from geometry to process plans, and a set of *application protocols* that assemble those definitions into models for particular industries. “STEP” is its informal name — the *STandard for the Exchange of Product model data*. Almost nobody has read all of it. A working knowledge of a dozen parts is enough for anything in this book.

The application protocol. When a CAD vendor says they “support STEP,” they mean they can read and write files conforming to one or more *application protocols* (APs). An AP is a scoped data model: it picks entities from the shared library, adds a few of its own, and says which combinations are legal for its domain. Three matter for mechanical CAD: AP203 (*configuration-controlled design*, the original), AP214 (*core data for automotive mechanical design processes*, which added colours, layers and more), and AP242 (*managed model-based 3D engineering*), which merged the two and added product manufacturing information, tessellation and much else. AP242 is the current one; the live publication as this is written is ISO 10303-242:2025. Files claiming AP203 and AP214 will be in circulation for decades.

The file. The text format with the `#22 = ...` records is defined in one part, ISO 10303-21, and is properly called the *clear text encoding of the exchange structure* or, universally, a *Part 21 file*. It is one of several encodings — there is also an XML form and, in principle, a binary one — but it is the one everyone uses, and when this book says “a STEP file” it means a Part 21 file.

The distinction matters in practice. “STEP can’t carry colours” is true of AP203’s first edition and false of its second, of AP214 and of AP242. “STEP files are huge” is true of Part 21 and not intrinsic to the data model. “Our STEP import failed” almost always means the *reader* did not support something the *AP* allowed and the *writer* used.

1.2 How the standard is organised

Every ISO 10303 document has a part number, and the numbering scheme is the only index you will need. Learn the ranges once and you can find anything.

Range	Contents	You will meet
1	Overview and fundamental principles	—
11–19	Description methods	**11**, the EXPRESS language
21–29	Implementation methods	**21**, the clear-text file; 26, 28
31–39	Conformance testing	rarely
41–99	Integrated generic resources	**41, 42, 43, 44, 46, 47**
101–199	Integrated application resources	101 (draughting), 105 (kinematics)
201–299	Application protocols	**203, 214, 242**
301–399	Abstract test suites	rarely
401–499	Application protocol modules	**442**, which is AP242 itself in modular form
501–599	Application interpreted constructs	when tracing where an entity came from
1001–1999	Application modules	the building blocks of AP242

The *integrated resources* in the 41–99 and 101–199 ranges are the heart of it. These are the ranges the standard sets aside, not the parts published: ISO has reached Part 62 in the first and Part 113 in the second, and both keep growing. Part 41 defines products, versions and contexts; Part 42 geometry and topology; Part 43 the notion of a *representation*; Part 44 product structure and assemblies; Part 46 visual presentation; Part 47 shape tolerances. Every entity in a mechanical STEP file comes from one of these. The application protocols do not define geometry; they *use* Part 42's.

1.3 Why the entity names look the way they do

The name `CONTEXT_DEPENDENT_SHAPE_REPRESENTATION` does not suggest "the position of a part in an assembly," yet that is what it means. Nobody chose that name for that purpose. Understanding how it came about explains most of what is strange about STEP.

Each application protocol is written twice. The first version is the **Application Reference Model** (ARM), written in the vocabulary of the domain: *part*, *assembly component occurrence*, *datum feature*. It is what the committee actually designed, and it is readable. The second version is the **Application Interpreted Model** (AIM), produced by *mapping* every ARM concept onto entities from the integrated resources — the same pool of entities that has to serve shipbuilding, process plants, electrical harnesses and finite-element meshes. The AIM is what appears in the file.

So the file does not contain a CAD data model. It contains the residue of one, after it has been dissolved into a general-purpose vocabulary designed by committee to be reusable across industries with nothing in common. When the ARM says "an occurrence of component B inside assembly A, at this position," the mapping says: represent A's shape and B's shape as `REPRESENTATION`s (Part 43), relate them with a `REPRESENTATION_RELATIONSHIP` carrying a transformation (Part 43 again), and tie that relationship to the *usage* of B in A via a `PRODUCT_DEFINITION_SHAPE` (Part 41). The names fall out of the mapping.

Two consequences follow, and both will recur throughout this book.

First, the AIM is *lossy in the direction you care about*. Several distinct ARM concepts can land on the same AIM entities, distinguished only by what they are connected to and by the values of certain fixed strings. Reading a file well means reconstructing intent the file never states directly.

Second, the file is designed to be *readable without the schema*. Every record carries its own type name and its own attribute values. This is why the four-way record at the top of the chapter spells out every

one of its types instead of letting you infer them — and it is the design decision that makes the format verbose, self-describing and thirty years durable all at once.

1.4 The lineage of the application protocols

A quick history, because the three APs you will meet are best understood as one line of descent.

AP203, published in 1994, was the first to see wide use. It carried product structure and B-rep solids and almost nothing else: no colours, no layers, no tolerances. It was the format that made cross-vendor exchange of solid models routine, and its limitations are why so many STEP files look grey. A second edition in 2011 added most of what was missing.

AP214, driven by the German automotive industry and published in 2001, took a broader scope from the start — colours, layers, drawings, kinematics, configuration management — and became the default export of most CAD systems for a decade. Its `FILE_SCHEMA` name, `AUTOMOTIVE_DESIGN`, appears on files for products that have never been near a car.

AP242, first published in 2014, merged AP203 and AP214 into a single protocol and added the things model-based engineering needs: semantic product manufacturing information (tolerances and datums as data rather than pictures), tessellated geometry, and a business-object XML form for product data management. Editions 2 (2020), 3 (2022) and 4 (2025) extended it further. A file claiming AP242 can carry everything an AP214 file can and more; a reader that handles AP242 handles the older two almost for free, since the geometry and product-structure entities are the same.

For the purposes of this book the differences are small until Part III. The block, the cylinder and the clamp assembly are written as AP214 files because that is what the exporter chose; the same records, with a different header and three changed values in the two opening records, would be AP242.

1.5 What “conforming” means

A STEP file makes three claims about itself, and each can be false.

It claims to follow the *file format*: Part 21, at a stated edition. A file that violates the grammar — an unterminated string, a lower-case keyword — is not a Part 21 file, and most readers will still open it.

It claims to follow a *schema*: the `FILE_SCHEMA` header names the AP. Every record must be an entity type in that schema, with the right number of attributes of the right types. Exporters get this wrong more often than you would expect, usually by naming one AP and writing entities from another.

And it claims to satisfy the schema’s *rules*: constraints such as AP214’s requirement that every application context be named by an `APPLICATION_PROTOCOL_DEFINITION` carrying the right schema string, or that a geometric context be two- or three-dimensional. Some things you would expect to be rules are not: that a closed shell encloses a volume is stated in Part 42’s prose as an informal proposition, and no EXPRESS constraint checks it. Almost no writer checks these, almost no reader enforces them, and the tools that do — validators such as NIST’s STEP File Analyzer — exist precisely because they are so often violated.

None of this makes the format unreliable. It makes it a format whose *documents* are reliable and whose *files* are as good as the software that wrote them, which is true of every exchange format ever designed. The difference with STEP is that the standard is precise enough that when a file is wrong, you can say exactly how.

> ! In brief
>
> Three meanings: the standard (ISO 10303, hundreds of parts), the application protocol (AP203, AP214, AP242 — a scoped data model), and the file (Part 21, the clear-text encoding). The part numbers are the index. Entity names are the residue of mapping a domain model onto a cross-industry library, which is why they are long, abstract and self-describing.

2. A First Look at a File

> *A complete file of 170 records, and how to read any one of them. Instance names, entity types, parameters; strings, references, lists; the two symbols that mean "nothing"; and the one kind of record that breaks the pattern.*

The running example begins with the simplest thing worth exporting: a rectangular block, 60 by 20 by 10 millimetres, sitting with one corner at the origin (Figure 2.1).

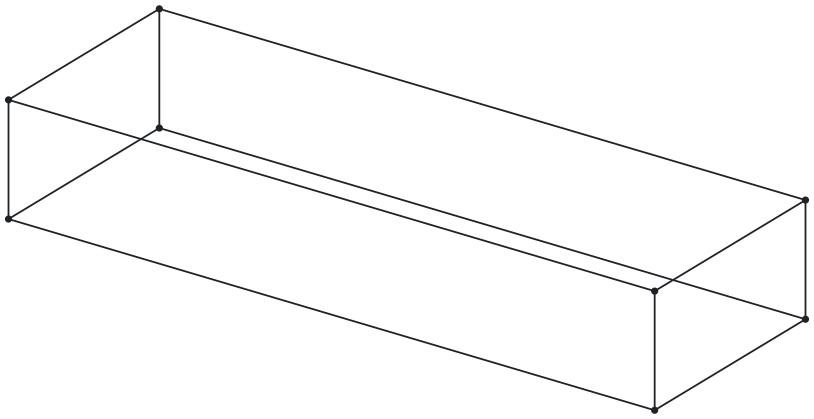


Figure 2.1: `block.step`, drawn from its records: twelve edges and eight vertices.

Open CASCADE writes it as 170 records. Here is the whole file with the middle cut out; the complete listing is in Appendix A.

```
ISO-10303-21;
HEADER;
FILE_DESCRIPTION(('Open CASCADE Model'),'2;1');
FILE_NAME('block.step','2026-09-17T16:14:23',('J. Chultarsky'),('Physna'),
  'Open CASCADE STEP processor 7.9','Open CASCADE 7.9','Unknown');
FILE_SCHEMA(('AUTOMOTIVE_DESIGN { 1 0 10303 214 1 1 1 1 }'));
ENDSEC;
DATA;
#1 = APPLICATION_PROTOCOL_DEFINITION('international standard',
  'automotive_design',2000,#2);
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#3 = SHAPE_DEFINITION_REPRESENTATION(#4,#10);
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);
#12 = CARTESIAN_POINT('',(0.,0.,0.));
#13 = DIRECTION('',(0.,0.,1.));
#14 = DIRECTION('',(1.,0.,-0.));
```

```
#15 = MANIFOLD_SOLID_BREP('',#16);
#16 = CLOSED_SHELL('',(#17,#57,#97,#119,#141,#153));
#17 = ADVANCED_FACE('',(#18),#52,.F.);
#18 = FACE_BOUND('',#19,.F.);
#19 = EDGE_LOOP('',(#20,#30,#38,#46));
#20 = ORIENTED_EDGE('',*,*,#21,.F.);
#21 = EDGE_CURVE('',#22,#24,#26,.T.);
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
    /* ... 141 records of faces, edges, vertices and points ... */
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
  GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169))
  GLOBAL_UNIT_ASSIGNED_CONTEXT((#166,#167,#168))
  REPRESENTATION_CONTEXT('Context #1','3D Context with UNIT and UNCERTAINTY') );
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#167 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#168 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
#170 = PRODUCT_RELATED_PRODUCT_CATEGORY('part',$,(#7));
ENDSEC;
END-ISO-10303-21;
```

Referred to above: 141 records from `block.step`; the 11 it points to directly are shown. The complete file.

```
#24 = VERTEX_POINT('',#25);
#26 = LINE('',#27,#28);
#30 = ORIENTED_EDGE('',*,*,#31,.T.);
#38 = ORIENTED_EDGE('',*,*,#39,.T.);
#46 = ORIENTED_EDGE('',*,*,#47,.F.);
#52 = PLANE('',#53);
#57 = ADVANCED_FACE('',(#58),#92,.T.);
#97 = ADVANCED_FACE('',(#98),#114,.F.);
#119 = ADVANCED_FACE('',(#120),#136,.T.);
#141 = ADVANCED_FACE('',(#142),#148,.F.);
#153 = ADVANCED_FACE('',(#154),#160,.T.);
/* ... and 130 more: 25 CARTESIAN_POINT, 24 DIRECTION, 20 ORIENTED_EDGE, 12 VECTOR, ... */
```

Everything that follows in this book is a closer look at some piece of this. For now the aim is to be able to read any single line.

2.1 The skeleton

A Part 21 file written to edition 2, which is nearly all of them, has a fixed outer structure and two sections:

```
ISO-10303-21;
HEADER;
  /* three fixed records describing the file itself */
ENDSEC;
DATA;
  /* any number of numbered records: the actual content */
ENDSEC;
END-ISO-10303-21;
```

The `HEADER` says what the file is: who wrote it, when, and — crucially — which schema its records follow. It has its own chapter. The `DATA` section is the payload: a set of records, each with a number, that reference one another to form a graph. There is nothing else in an edition 2 file; the sections edition 3 added are in Chapter 18. Line breaks and tabs are ignored everywhere, even inside a string, so a long record or a long string may be split across lines without changing what it means. The space character is the only whitespace that separates tokens, and inside a string it is data. The exporter above wraps long records at 72 columns because an old convention says so, and many do not.

Comments are allowed anywhere whitespace is, delimited `/* ... */`. Real files almost never contain them; the ones above were added for the listing.

2.2 Reading a record

Take one line:

```
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
```

Referred to above: 2 records from `block.step`. *The complete file.*

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
```

Read it as a row from a database dump.

- `#7` is the record's *instance name* — its ID. It is unique within this file and means nothing outside it. Other records refer to this one by writing `#7`. There is no significance to the number itself; exporters assign them in whatever order they happen to write things, and a file that renumbered every record would be the same file.
- `PRODUCT` is the *entity type* — the table name. Somewhere in the schema is a definition of `product` listing its attributes in order.
- Everything inside the parentheses is the *parameter list*: the column values, in schema order. Here there are four.
- `;` ends the record.

So `#7` says: there is a `PRODUCT` whose four attribute values are the string `'BLOCK'`, the string `'block 60x20x10'`, an empty string, and a list containing a reference to record `#8`. What those four columns *mean* — identifier, name, description, list of contexts — is in the schema, and Chapter 4 shows how to look it up. For now, the shape.

A note on terminology, since two words are used for the same slots. In the *file*, the values in the parentheses are **parameters**. In the *schema*, the slots they fill are **attributes**. `PRODUCT` has four attributes; record `#7` supplies four parameters. Same slots, seen from the two ends.

2.3 The kinds of value

Every parameter is one of a small number of things. The block file contains all of them but the binary.

Strings are single-quoted: `'BLOCK'`, `'design'`, `''`. An apostrophe inside a string is doubled: `'Julian''s bracket'`. A file declaring `'2;1'` or `'3;1'` must spell anything above ASCII with an escape sequence — `\X2\00C5\X0\` for Å. Edition 3 made UTF-8 the encoding of the whole file, so a file declaring `'4;1'` may write Å directly. Many exporters write raw UTF-8 while still declaring `'2;1'`, and most readers accept it. Nearly every entity in a mechanical file begins with a `name` attribute, and nearly every exporter leaves it empty, which is why `''` opens so many records.

References are instance names used as values: the `#8` in `(#8)`, the `#16` in `MANIFOLD_SOLID_BREP('',#16)`. A reference is a pointer to another record in the same file. They can point forward to records not yet seen — `#3` refers to `#10`, seven lines later — and the file is a set of records, not a sequence, so this is normal.

Lists are parenthesised, comma-separated: `(#11,#15)`, `(0.,0.,0.)`, `(#17,#57,#97,#119,#141,#153)`. A list with one element is still a list: `(#8)`. Lists nest: `((#11,#12),(#13,#14))` is a list of two lists. Whether a given list is a `SET` or a `LIST` or an `ARRAY` — ordered, unordered, bounded — is the schema's business; the file writes them all the same way.

Reals always contain a decimal point: `0.`, `1.`, `-0.`, `1.E-07`. The trailing-dot form `10.` is standard and ubiquitous. `1.E-07` is one ten-millionth. `-0.` is negative zero, which the exporter wrote because that is what came out of its floating-point arithmetic; it is harmless and you will see a lot of it.

Integers have no decimal point: the `3` in `GEOMETRIC_REPRESENTATION_CONTEXT(3)`, the `2000` in `#1`. A parameter that is a real is *never* written as an integer, so `3` and `3.` are different tokens for different attribute types.

Enumerations are upper-case names between dots: `.T.`, `.F.`, `.MILLI.`, `.METRE.`, `.RADIAN.`. The two most common are the Booleans, `.T.` and `.F.`. EXPRESS counts `BOOLEAN` as a simple type, but Part 21 encodes it as an enumeration value and fixes the spelling at the single character `T` or `F`, so `.TRUE.` is not a legal spelling of one. Enumeration values come from a fixed list in the schema — `.MILLI.` is one of the sixteen SI prefixes the schema defines.

Binary values are hexadecimal in double quotes, with one extra leading digit: `"0F3A"`. That first digit is not data — it counts the zero bits added on the left so the bits divide evenly into groups of four, so `"0F3A"`

is the twelve bits `F3A`. They do not occur in this file, and are rare in mechanical files generally; you meet them in PMI for encoded text and in a few presentation entities.

Wrapped values are a type name with one value in parentheses: the `LENGTH_MEASURE(1.E-07)` in `#169`. Part 21 requires the wrapper whenever an attribute's schema type is a choice and the value chosen is a number, a string or an enumeration rather than a reference, so that a reader knows which choice was taken. Chapter 4 gives the rule.

Two symbols remain, and they are the ones that trip people up.

2.4 $ and *

`$` means **no value**. The attribute is optional in the schema and this record does not supply it. In `SI_UNIT($,.RADIAN.)` the first attribute is the prefix — `.MILLI.` in the millimetre record on the line above — and a plain radian has none. In `PRODUCT_RELATED_PRODUCT_CATEGORY('part',$,(#7))` the description is absent.

`*` holds the place of an attribute that the entity inherited as a stored value but that its own type recomputes. An attribute that is derived from the start takes no slot at all and is simply not written; it is only a *redeclared* one that leaves a gap to fill. In `NAMED_UNIT(*)` the attribute is `dimensions`, and an SI unit knows its own dimensions from its name. In `ORIENTED_EDGE('',*,*,#21,.F.)` the second and third attributes — the edge's start and end vertices — are derived from the edge it refers to and the orientation flag.

`$` and `''` are different. `''` is a string with no characters in it; `$` is the absence of a string. `PRODUCT_DEFINITION('design','',#6,#9)` has an empty description; `PRODUCT_DEFINITION('design',$,#6,#9)` would have no description. The schema treats these differently, exporters are inconsistent about which they write for a blank field, and a tool that cares about descriptions has to handle both. Most tools do not care.

2.5 References point from specific to general

Look again at the first nine data records and note the *direction* of every reference.

```
#3 = SHAPE_DEFINITION_REPRESENTATION(#4,#10);
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
```

Referred to above: 161 records from `block.step`; the 2 it points to directly are shown. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
/* ... and 159 more: 27 CARTESIAN_POINT, 26 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
```

`#6` points at `#7`; `#5` points at `#6`; `#4` points at `#5`. The *version* knows which *product* it is a version of. The *product* has no idea that any versions exist. References in a STEP file run this way: from the more specific or dependent thing toward the more general or independent one, a shell to the faces it is made of as much as a version to its product. A face points at its surface; the surface does not list its faces. An edge points at its vertices; the vertex does not know which edges meet there.

The consequence is that almost every interesting question — "what does this product look like?", "which faces use this surface?", "which assemblies contain this part?" — starts at the general end and has to walk *against* the arrows. Nothing in the file helps with that; it is the reader's job to build the reverse index. This is the single most important structural fact about the format, and the reason a STEP file cannot be usefully processed one line at a time.

2.6 Complex records

Every record so far has had one entity type. Four records near the end have several:

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
```

The pattern: after `#166 =` comes an opening parenthesis, then a sequence of *type-name-with-parameters* groups, with no commas between them, then a closing parenthesis. This is a **complex entity instance**, and it describes **one object that is several entity types at once**. `#166` is a `LENGTH_UNIT` *and* a `NAMED_UNIT` *and* an `SI_UNIT`. Each group carries the attributes that its own type declares: `LENGTH_UNIT` declares none, `NAMED_UNIT` declares one, `dimensions`, which `SI_UNIT` redeclares as derived — hence the `*` — and `SI_UNIT` declares two of its own, prefix and name.

Why an object would need to be three types at once, why the groups are in alphabetical order, and why the file does not simply say `SI_UNIT` and let you infer the rest, are questions about the schema, and they are answered in Chapter 4. The point here is to recognise the form when you see it. Two rules of thumb: a complex record is always about something that has *both* a general classification and a specific one — a unit that is both "a named unit" and "a length unit"; and the records that most often take this form are units, representation contexts, and the relationship records that position parts in assemblies.

2.7 What the block file contains

Standing back from the syntax, the 170 records fall into five groups, and the same five groups appear in every single-part STEP file in roughly the same proportions.

Records	Group	What it says
`#1–#2`	Application context	"This is mechanical design data under AP214."
`#3–#9`	Product identity	"There is a part called BLOCK, one version, design view, and it has a shape."
`#10–#16`	Shape representation and solid	"The shape is this B-rep solid, in this coordinate system."
`#17–#164`	Topology and geometry	Six faces, twelve edges, eight vertices, and their planes, lines and points.
`#165–#170`	Context, units, category	"Millimetres, radians; points closer than 10^{-7} mm coincide; it's a part."

The proportions are the thing to notice. Nine records establish what the part *is*; a hundred and fifty describe its shape; six say how to measure it. In a real part with a few hundred faces the geometry becomes 99% of the file, and the product-identity records — the ones that carry the part number, the version and the assembly structure — become a few dozen lines scattered through a hundred thousand.

! In brief

A record is `#id = TYPE(parameters);`. Parameters are strings, references, lists, reals, integers, enumerations or binaries; `$` is an absent optional value and `*` a derived one. References point from the dependent thing to the one it depends on, so finding what is attached to a product means walking against the arrows. A record beginning `#id = (` is one object with several types.

3. The Header

> *Three records that say what the file is, who made it, and which schema it claims to follow — and the one record in the data section that repeats the claim.*

The header is short, its shape is fixed, and it is the only part of the file whose meaning does not depend on the schema. It is also the part you will read most often by eye, because it answers the two questions that come up before any other: *what wrote this?* and *what does it claim to be?*

```
HEADER;
FILE_DESCRIPTION(('Open CASCADE Model'),'2;1');
FILE_NAME('block.step','2026-09-17T16:14:23',('J. Chultarsky'),('Physna'),
  'Open CASCADE STEP processor 7.9','Open CASCADE 7.9','Unknown');
FILE_SCHEMA(('AUTOMOTIVE_DESIGN { 1 0 10303 214 1 1 1 1 }'));
ENDSEC;
```

Notice first what is missing: no `#n`. Header records have no instance names, because nothing refers to them and they refer to nothing. They are not part of the graph. There are exactly three mandatory ones, they must appear in this order, and each has a fixed set of parameters.

3.1 FILE_DESCRIPTION

```
FILE_DESCRIPTION(('Open CASCADE Model'),'2;1');
```

Two parameters. The first is a list of free-text strings describing the file's contents; exporters put their own name there, or a product name, or leave it as `('')`. Nothing reads it.

The second is the **implementation level**, and it matters, though not in the way the digits suggest. The first number is the registered version of Part 21 whose features the file uses; the second is the syntactical conformance class. It is not the edition of the document the writer worked from. `'2;1'` says the file stays inside what edition 1 defined: one data section, no optional header records, no short names for enumeration values. That is what virtually every file in existence declares, including files written long after edition 2 appeared in 2002. A file that uses edition 2's multiple data sections or optional header records declares `'3;1'`, and a file that uses anything edition 3 added declares `'4;1'`, `'4;2'` or `'4;3'` according to which of the three conformance classes it needs. Chapter 18 covers what changed; until then, `'2;1'`.

3.2 FILE_NAME

```
FILE_NAME('block.step',                          /* name */
          '2026-09-17T16:14:23',                 /* time_stamp */
          ('J. Chultarsky'),                     /* author */
          ('Physna'),                            /* organization */
          'Open CASCADE STEP processor 7.9',     /* preprocessor_version */
          'Open CASCADE 7.9',                    /* originating_system */
          'Unknown');                            /* authorization */
```

Seven parameters, and two of them are the most useful strings in the file.

The **name** is whatever the writer chose — often the original filename, often with a full Windows path, sometimes empty. The **time stamp** is ISO 8601, local time with no zone unless the writer adds one. **Author** and **organization** are lists of strings and are usually `('')` and `('')`. **Authorization** is a free string that almost nobody fills.

Preprocessor version names the software component that wrote the STEP file — the translator library. **Originating system** names the application it was running inside. They are distinct because most CAD

systems do not write STEP themselves; they license a translator. When the two strings are the same product, as here, the vendor wrote their own. When they differ, you learn something:

Preprocessor	Originating system	What it tells you
`Open CASCADE STEP processor 7.9`	`Open CASCADE 7.9`	OCCT, or a tool built on it (FreeCAD, CadQuery, many others)
`ST-DEVELOPER v20`	`NX 2312`	Siemens NX, writing through STEP Tools' library
`SwSTEP 2.0`	`SolidWorks 2024`	SolidWorks' own translator
`CATIA V5 STEP AP214`	`CATIA Version 5-6 ...`	Dassault, own translator
`Pro/ENGINEER by Parametric ...`	`Creo Parametric 10.0`	PTC, own translator

These are the strings as they characteristically appear; version numbers change every release and the exact wording drifts. The point is that once you know a file came from NX via ST-Developer, or from SolidWorks, you know a great deal about which of the structures in Part I and Part III it will contain and how it will spell them. Every experienced STEP user reads these two fields first.

3.3 FILE_SCHEMA

```
FILE_SCHEMA(('AUTOMOTIVE_DESIGN { 1 0 10303 214 1 1 1 1 }'));
```

One parameter, a list of schema names, almost always with one element. This is the line that says which application protocol the file follows. Every record in the `DATA` section must be an entity type from this schema.

The string has two parts. The name before the brace is the schema's EXPRESS identifier, upper-cased. The braces hold an *object identifier* — a sequence of integers registered with ISO that pins the exact schema and edition. The ones you will see:

Schema name	Protocol
`CONFIG_CONTROL_DESIGN`	AP203, first edition
`AP203_CONFIGURATION_CONTROLLED_3D_DESIGN_OF_MECHANICAL_PARTS_AND_ASSEMBLIES_MIM_LF`	AP203, second edition
`AUTOMOTIVE_DESIGN`	AP214
`AP242_MANAGED_MODEL_BASED_3D_ENGINEERING_MIM_LF`	AP242

The `_MIM_LF` suffix on the newer names means *module interpreted model, long form* — the AP's schema flattened into a single file rather than assembled from modules, which is the form tools actually load. The object identifier's digits differ between AP242 editions, which is how a reader can tell an edition 1 file from an edition 3 file when the name is the same.

The same block, exported by the same OCCT build with the schema set to AP242, has this header and a slightly different pair of opening records:

```
FILE_SCHEMA(
  ('AP242_MANAGED_MODEL_BASED_3D_ENGINEERING_MIM_LF {1 0 10303 442 1 1 4 }'));
ENDSEC;
DATA;
#1 = APPLICATION_PROTOCOL_DEFINITION('international standard',
  'ap242_managed_model_based_3d_engineering',2013,#2);
#2 = APPLICATION_CONTEXT('Managed model based 3d engineering');
```

Every other data record is identical, apart from `#7`, whose identifier and name were left as the exporter wrote them in this file. For a plain solid there is no difference between an AP214 and an AP242 file beyond the header and the three values in `#1` and `#2` — the schema name, the year and the context text — which is

why a reader that handles one handles the other with little extra work. The differences begin with colours, PMI and tessellation.

> ⚠ In the wild: When the schema lies
>
> `FILE_SCHEMA` is a claim, and exporters break it in two ways. Some declare `CONFIG_CONTROL_DESIGN` (AP203, which has no colour entities) and then write AP214 colour records anyway, because that is what the user asked for. Others declare AP214 and write entities that exist only in AP242. Strict readers reject these files; practical readers treat the header as advisory and accept any entity type they recognise from any of the three protocols. There is no rule in the standard that permits this, and it is what every widely used reader does.

3.4 The optional header records

Edition 2 defines three more header entities that may follow `FILE_SCHEMA`:

```
FILE_POPULATION('AUTOMOTIVE_DESIGN','SECTION_BOUNDARY',$);
SECTION_LANGUAGE($,'eng');
SECTION_CONTEXT($,('...'));
```

`FILE_POPULATION` says which instances, gathered from which data sections and by which of three named methods, are to be checked against a schema; `SECTION_LANGUAGE` declares the natural language of the strings, as a three-letter ISO 639-2 code; `SECTION_CONTEXT` is a free-form context list. They are rare enough that many readers do not implement them and a few reject files that contain them. Edition 3 adds more. The standard allows user-defined header records as well, provided the keyword begins with an exclamation mark, as in `!A_SPECIAL_ENTITY('ABC',123);`. Anything else in the header section is an error — and some exporters put things there regardless.

3.5 The record that repeats the claim

Strictly, the header is finished at `ENDSEC`. But the first two records of the data section belong with it in spirit, because they exist to say the same thing the header said:

```
#1 = APPLICATION_PROTOCOL_DEFINITION('international standard',
  'automotive_design',2000,#2);
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
```

`APPLICATION_CONTEXT` is the root from which the product-identity records hang: every `PRODUCT_CONTEXT` and `PRODUCT_DEFINITION_CONTEXT` in the file points back at it, and it says, in prose, what domain the data is in. `APPLICATION_PROTOCOL_DEFINITION` attaches to it and records the AP by status (`'international standard'`), name (`'automotive_design'`) and year (`2000`).

Why repeat what `FILE_SCHEMA` already said? Because the header belongs to the *file* and these two records belong to the *data*. A file might in principle carry data from several contexts; and when data is moved out of Part 21 into a database or another encoding, the header does not travel with it but the records do. In practice there is one application context per file and it agrees with the header. When it does not, believe the header, and note that the exporter was careless.

Each AP's schema has a global rule fixing the schema-name string in `#1`, and it differs by edition. AP203 edition 2's rule requires `'ap203_configuration_controlled_3d_design_of_mechanical_parts_and_assemblies'` — the long name, but *without* the `_mim_lf` that its schema name carries. Edition 1 files carry `'config_control_design'` instead. AP214 files write `'automotive_design'`, as above, which is the lower-cased schema name. AP242 requires `'ap242_managed_model_based_3d_engineering_mim_lf'` — and the export shown earlier writes it without the `_mim_lf`, which is a rule violation of exactly the kind Chapter 1 warned about. The year and the `APPLICATION_CONTEXT` text are convention: nothing in the schema checks either. They are the second place, after the header, where you can identify the protocol — and a place where files that have been converted between protocols often still show their origin.

! In brief

Three header records, no instance names, fixed order. `FILE_DESCRIPTION` gives the implementation level: the Part 21 version whose features the file uses and its syntactical conformance class, not the edition the writer worked from. `FILE_NAME` names the translator and the CAD system, which predict the file's quirks. `FILE_SCHEMA` names the AP, and is a claim rather than a guarantee. `APPLICATION_CONTEXT` and `APPLICATION_PROTOCOL_DEFINITION`, the first two data records, repeat that claim from inside the data.

4. Where Records Get Their Meaning: The Schema

> *The file says `ADVANCED_FACE('',(#18),#52,.F.)`. Nothing in the file says what those four values are. The schema does — and reading enough of the schema language to look them up is the one prerequisite for everything else in this book.*

Every entity type in a STEP file — `PRODUCT`, `ADVANCED_FACE`, `SI_UNIT` — is defined in a **schema** written in a language called **EXPRESS**, which is ISO 10303-11. The schema for AP214 is a single text file of about 19,000 lines; AP242's is nearly three times that. It lists every entity, its attributes in order, its relationship to other entities, and a body of constraints. The Part 21 file is a dump of instances of those entities, and it deliberately omits everything the schema already says: attribute names, attribute types, inheritance. To read a record you look its type up in the schema and lay the parameters against the attributes.

This chapter is EXPRESS as a *reader* needs it. The language also has functions, rules and procedures for validating data; they are skipped here, and you will not miss them.

4.1 An entity declaration

Here is the definition of `product`, from Part 41, exactly as it appears in the AP214 schema apart from spacing:

```
ENTITY product;
  id                : identifier;
  name              : label;
  description       : OPTIONAL text;
  frame_of_reference : SET [1:?] OF product_context;
END_ENTITY;
```

Lay record `#7` from the block file against it:

```
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
```

Referred to above: 2 records from `block.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
```

Attribute	Type	Value in `#7`
`id`	`identifier` (a string)	`'BLOCK'`
`name`	`label` (a string)	`'block 60x20x10'`
`description`	`text`, optional	`''`
`frame_of_reference`	set of 1 or more `product_context`	`(#8)`

Four attributes, four parameters, in order. Several things in the declaration are worth naming.

`identifier`, `label` and `text` are **defined types** — names given to a base type so the schema can say what a string is *for*. All three are declared elsewhere as `TYPE label = STRING;` and so on. The file does not distinguish them; a string is a string.

`OPTIONAL` means the attribute may be absent, in which case the file writes `$`. An attribute without `OPTIONAL` must have a value. `#7` supplies `''` for its optional description, which is a value (an empty one), not an absence.

`SET [1:?] OF product_context` is an **aggregate**: an unordered collection of at least one, at most any number, of references to `product_context` instances. EXPRESS has four aggregate kinds — `SET` (unordered, no duplicates), `BAG` (unordered, duplicates allowed), `LIST` (ordered) and `ARRAY` (ordered, fixed bounds) — and the file writes all four as a parenthesised list. The bounds are constraints on valid data, not storage instructions.

4.2 Inheritance and the flattened record

Most entities are not declared from scratch. `ADVANCED_FACE`, the workhorse of every B-rep file, is the bottom of a chain:

```
ENTITY representation_item;
  name : label;
END_ENTITY;

ENTITY topological_representation_item
  SUBTYPE OF (representation_item);
END_ENTITY;

ENTITY face
  SUBTYPE OF (topological_representation_item);
  bounds : SET [1:?] OF face_bound;
END_ENTITY;

ENTITY face_surface
  SUBTYPE OF (face, geometric_representation_item);
  face_geometry : surface;
  same_sense    : BOOLEAN;
END_ENTITY;

ENTITY advanced_face
  SUBTYPE OF (face_surface);
END_ENTITY;
```

(Constraints and supertype clauses removed; they return below.) `advanced_face` adds no attributes of its own. It is a `face_surface` with extra rules about what kinds of surface and edge it may use. `face_surface` has *two* supertypes — it is both a `face` and a `geometric_representation_item`, and both of those descend from `representation_item`. EXPRESS allows multiple inheritance, including the diamond.

Now the record:

```
#17 = ADVANCED_FACE('',(#18),#52,.F.);
```

Referred to above: 39 records from `block.step`. The complete file.

```
#18 = FACE_BOUND('',#19,.F.);
#19 = EDGE_LOOP('',(#20,#30,#38,#46));
#20 = ORIENTED_EDGE('',*,*,#21,.F.);
#21 = EDGE_CURVE('',#22,#24,#26,.T.);
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(0.,0.,10.));
#26 = LINE('',#27,#28);
#27 = CARTESIAN_POINT('',(0.,0.,0.));
#28 = VECTOR('',#29,1.);
#29 = DIRECTION('',(0.,0.,1.));
#30 = ORIENTED_EDGE('',*,*,#31,.T.);
#31 = EDGE_CURVE('',#22,#32,#34,.T.);
#32 = VERTEX_POINT('',#33);
#33 = CARTESIAN_POINT('',(0.,20.,0.));
#34 = LINE('',#35,#36);
#35 = CARTESIAN_POINT('',(0.,0.,0.));
#36 = VECTOR('',#37,1.);
#37 = DIRECTION('',(0.,1.,0.));
#38 = ORIENTED_EDGE('',*,*,#39,.T.);
#39 = EDGE_CURVE('',#32,#40,#42,.T.);
```

```
#40 = VERTEX_POINT('',#41);
#41 = CARTESIAN_POINT('',(0.,20.,10.));
#42 = LINE('',#43,#44);
#43 = CARTESIAN_POINT('',(0.,20.,0.));
#44 = VECTOR('',#45,1.);
#45 = DIRECTION('',(0.,0.,1.));
#46 = ORIENTED_EDGE('',*,*,#47,.F.);
#47 = EDGE_CURVE('',#24,#40,#48,.T.);
#48 = LINE('',#49,#50);
#49 = CARTESIAN_POINT('',(0.,0.,10.));
#50 = VECTOR('',#51,1.);
#51 = DIRECTION('',(0.,1.,0.));
#52 = PLANE('',#53);
#53 = AXIS2_PLACEMENT_3D('',#54,#55,#56);
#54 = CARTESIAN_POINT('',(0.,0.,0.));
#55 = DIRECTION('',(1.,0.,0.));
#56 = DIRECTION('',(0.,0.,1.));
```

Four parameters, and `advanced_face` declares zero attributes. They come from the ancestors, **flattened into one list**: each entity's attributes appear once, in the order given by walking the supertype graph depth-first, supertypes before subtypes, and multiple supertypes in the order they are listed.

Parameter	Attribute	Declared in
`''`	`name`	`representation_item`
`(#18)`	`bounds`	`face`
`#52`	`face_geometry`	`face_surface`
`.F.`	`same_sense`	`face_surface`

`geometric_representation_item` and `topological_representation_item` declare nothing, and `representation_item` is visited once even though there are two paths to it. This is the rule that makes ordinary records readable, and it is the rule that must be got exactly right: shift one attribute and every value lands in the wrong slot.

A record like this — one leaf type name, all inherited attributes flattened in — is what Part 21 calls the **internal mapping**. It is the normal case.

4.3 Derived attributes and *

Some attributes are declared with a formula instead of a stored value. From Part 41:

```
ENTITY named_unit;
  dimensions : dimensional_exponents;
END_ENTITY;

ENTITY si_unit
  SUBTYPE OF (named_unit);
  prefix : OPTIONAL si_prefix;
  name   : si_unit_name;
DERIVE
  SELF\named_unit.dimensions : dimensional_exponents
                             := dimensions_for_si_unit(name);
END_ENTITY;
```

`named_unit` stores its `dimensions` — length^{1}, mass^{0}, and so on. `si_unit` inherits that attribute and *redeclares it as derived*: an SI unit knows its dimensions from its name, so the value is computed by a function rather than stored. Attribute slots are positional, so the file cannot simply omit it; it writes `*` in the slot:

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
```

`*` is "there is a value here, but you compute it." `$` is "there is no value." What sits behind a `*` is not always unimportant: in `ORIENTED_EDGE('',*,*,#21,.F.)` the two starred slots are the edge's start and end vertices, which every reader of a B-rep needs. The point of a derived attribute is that the value is

always recoverable from what the record does store — here from the edge `#21` and the orientation flag — so nothing is lost by not writing it.

4.4 Inverse attributes

Declarations may also contain an `INVERSE` clause:

```
ENTITY representation_context;
  context_identifier : identifier;
  context_type       : text;
INVERSE
  representations_in_context : SET [1:?] OF representation
                               FOR context_of_items;
END_ENTITY;
```

An inverse attribute is the schema saying: *you can find the representations that use this context by looking for `representation` instances whose `context_of_items` points here.* It declares the reverse of a reference that exists elsewhere. **Inverse attributes never appear in the file.** `REPRESENTATION_CONTEXT('Context #1','3D Context ...')` has two parameters, not three. They are worth knowing about for two reasons: the schema uses them in constraints, and they are a map of exactly the reverse lookups a reader will end up building for itself.

4.5 Complex instances and the external mapping

Chapter 2 introduced records with several type names and promised the explanation. It is in the `SUPERTYPE OF` clause, which I removed from the declarations above. Here is `named_unit` in full:

```
ENTITY named_unit
  SUPERTYPE OF (ONEOF (si_unit, conversion_based_unit, context_dependent_unit)
                ANDOR
               ONEOF (length_unit, mass_unit, time_unit, electric_current_unit,
                      thermodynamic_temperature_unit, amount_of_substance_unit,
                       luminous_intensity_unit, plane_angle_unit,
                       solid_angle_unit, ratio_unit));
  dimensions : dimensional_exponents;
END_ENTITY;
```

A supertype clause constrains how an entity's subtypes may be combined in a single instance. `ONEOF (a, b, c)` means at most one of them: they are mutually exclusive, and only if the supertype is declared `ABSTRACT` must one of them be chosen. `a ANDOR b` means either, or both. `a AND b` means both. An entity with no supertype clause allows any combination, as if every pair were `ANDOR`.

Read `named_unit`'s clause: an instance is *one of* {SI, conversion-based, context-dependent} **and/or** *one of* {length, mass, time, …}. A millimetre is an SI unit *and* a length unit. There is no entity named `si_length_unit`; the schema deliberately does not enumerate the thirty combinations. So there is no leaf type to name in an internal-mapping record, and the file falls back to the **external mapping**:

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
```

The rules of the external mapping:

- Every entity type the instance *is* appears as a block: the leaves, and every supertype above them, up to the root. `NAMED_UNIT` is present because both leaves descend from it.
- Each block carries **only the attributes that entity declares itself**, not inherited ones. `SI_UNIT` gets `prefix` and `name`; `NAMED_UNIT` gets `dimensions`; `LENGTH_UNIT` declares nothing and gets an empty pair of parentheses.
- Blocks are in **alphabetical order** of entity name. Not hierarchical order — alphabetical. L before N before S.
- The set of leaves must be one the supertype clauses permit. `(LENGTH_UNIT() MASS_UNIT() ...)` would violate the second `ONEOF`.

The other complex record in the block file works the same way with a different reason. `representation_context` has *no* supertype clause, so any combination of its subtypes is legal, and a 3D context that also carries units and an uncertainty is three of them at once:

```
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
  GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169))
  GLOBAL_UNIT_ASSIGNED_CONTEXT((#166,#167,#168))
  REPRESENTATION_CONTEXT('Context #1','3D Context with UNIT and UNCERTAINTY') );
```

Referred to above: 4 records from `block.step`. The complete file.

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#167 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#168 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
```

Each subtype adds one attribute — dimension count, uncertainty set, unit set — and the root carries the two it declares.

Why list the supertypes at all, when a reader with the schema could derive them from the leaves? Partly because the supertype blocks are carrying attribute values that have to go somewhere. Mostly because the format was designed to be readable by a program that does *not* have the schema: a tool that only cares about `REPRESENTATION_CONTEXT` can find that block and take its two values without knowing the other three types exist. The redundancy is the price of that. It is also a free consistency check — if the supertypes listed do not match what the schema derives from the leaves, the file was written against a different schema than it claims.

Exporters are supposed to use the internal mapping whenever a leaf name exists and the external mapping only when it does not. A few use the external form more liberally, so a reader must accept either form for any entity.

4.6 Enumerations

An enumeration is a closed list of names:

```
TYPE si_prefix = ENUMERATION OF
  (exa, peta, tera, giga, mega, kilo, hecto, deca,
   deci, centi, milli, micro, nano, pico, femto, atto);
END_TYPE;

TYPE si_unit_name = ENUMERATION OF
  (metre, gram, second, ampere, kelvin, mole, candela, radian, steradian,
   hertz, newton, pascal, joule, watt, coulomb, volt, farad, ohm, siemens,
   weber, tesla, henry, degree_celsius, lumen, lux, becquerel, gray, sievert);
END_TYPE;
```

In the file the chosen name is upper-cased and dotted: `.MILLI.`, `.METRE.`. The base type `BOOLEAN` is written as an enumeration too — `.T.` or `.F.`, and only those — and its cousin `LOGICAL` adds a third value, `.U.` for unknown.

4.7 SELECT types and the typed wrapper

One record in the block file has a value that is wrapped in a type name:

```
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
```

Referred to above: 1 record from `block.step`. The complete file.

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
```

The first attribute, `value_component`, is declared with a **SELECT** type — EXPRESS's tagged union:

```
TYPE measure_value = SELECT
  (amount_of_substance_measure, area_measure, celsius_temperature_measure,
```

```
   context_dependent_measure, count_measure, descriptive_measure,
   electric_current_measure, length_measure, luminous_intensity_measure,
   mass_measure, numeric_measure, non_negative_length_measure, parameter_value,
   plane_angle_measure, positive_length_measure, positive_plane_angle_measure,
   positive_ratio_measure, ratio_measure, solid_angle_measure,
   thermodynamic_temperature_measure, time_measure, volume_measure);
END_TYPE;

TYPE length_measure      = REAL; END_TYPE;
TYPE plane_angle_measure = REAL; END_TYPE;
```

A `measure_value` is any one of those twenty-two; AP242's list runs to forty-two. But `length_measure` and `plane_angle_measure` are both plain `REAL` underneath, so a bare `1.E-07` would be ambiguous — a length or an angle? The file resolves this by wrapping the value in the name of the alternative chosen: `LENGTH_MEASURE(1.E-07)`.

The rule for when a wrapper appears is exact. A parameter is wrapped when **both** hold:

1. the attribute's declared type is a `SELECT`, and
2. the alternative chosen is *not* an entity type.

The second attribute of `#169`, `unit_component`, is also a SELECT — `TYPE unit = SELECT (derived_unit, named_unit)` — but both alternatives are entities, so the value is a bare reference: `#166`. The record at `#166` already says what it is. Wrappers exist only for values that have no record of their own to carry a type.

You will meet wrappers most often with measures (`LENGTH_MEASURE`, `PLANE_ANGLE_MEASURE`, `POSITIVE_LENGTH_MEASURE`), with curve parameters (`PARAMETER_VALUE(0.5)` in trimmed curves), and in PMI, where `DESCRIPTIVE_MEASURE('...')` carries text. A SELECT can include another SELECT, so wrappers can in principle nest; in practice they rarely do.

4.8 Case

EXPRESS identifiers are case-insensitive, and published schemas are written in lower case: `representation_context`. Part 21 keywords — entity names, enumeration values, the wrapper names — are required to be **upper case**; the grammar admits nothing else. So the mapping from schema to file is "upper-case the name," and the mapping back is the reverse. With one caveat: Part 21 also lets an AP define **short names**, abbreviations that a file may use in place of entity type names and enumeration values. Nothing in mechanical CAD writes them, and a file declaring `'2;1'` is promising it has not used them for enumeration values, but the abbreviation cannot be derived from the name, so a reader that meets one needs the AP's short-name table. Strings, being data, keep their case: `'BLOCK'` and `'block'` are different values.

4.9 What to skip, and where to find the rest

The schema also contains `WHERE` clauses on entities (per-instance constraints), global `RULE`s (constraints across instances), `FUNCTION`s used by both, `ABSTRACT SUPERTYPE` declarations (an entity that can only appear as part of a subtype), `UNIQUE` clauses, and the `SCHEMA`/`USE FROM` machinery that assembles AP242 from hundreds of modules. A validator needs all of it. A reader of files needs none of it, and this book will not use any of it again except to explain, in passing, why a particular record has the shape it has.

The schemas themselves are free even though the standard is not. STEP Tools publishes the long-form EXPRESS for AP203, AP214 and AP242 at `steptools.com/stds/step/`, and that is the text every declaration in this book was checked against. Opening the AP242 long form and searching for `ENTITY next_assembly_usage_occurrence` is the best twenty minutes you can spend before Chapter 8.

> ! In brief
>
> A record's parameters are its entity's attributes, flattened from the root of the inheritance chain down, supertypes first. `$` is an absent optional; `*` is a derived value. A record with several type names in parentheses is one object whose combination of subtypes has no single name; blocks are alphabetical and carry only their own attributes. A value is wrapped in a type name only when its attribute is a SELECT and the chosen alternative is not an entity.

5. Identity: The Product Chain

> *Eight records that say "there is a part called BLOCK" — and why a CAD file needs the version-and-view structure of a product lifecycle system to say it.*

Strip the geometry out of the block file and what remains is this:

```
#1 = APPLICATION_PROTOCOL_DEFINITION('international standard',
  'automotive_design',2000,#2);
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#3 = SHAPE_DEFINITION_REPRESENTATION(#4,#10);
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#170 = PRODUCT_RELATED_PRODUCT_CATEGORY('part',$,(#7));
```

Referred to above: 160 records from `block.step`; the 1 it points to directly are shown. The complete file.

```
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
/* ... and 159 more: 27 CARTESIAN_POINT, 26 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
```

Two of these (#3, #4) are the bridge to geometry and belong to Chapter 7. The rest are the **product chain**: the records that establish what thing the file is about. Every STEP file has this structure, once per product, and it is the same in AP203, AP214 and AP242.

5.1 The three-level core

The chain has a spine of three records, and understanding why there are three rather than one is most of the chapter.

```
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
```

Referred to above: 3 records from `block.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
```

`PRODUCT` is the thing as a catalogue item: a part number and a name, and nothing else. Not a version, not a shape. `id` is `'BLOCK'`; `name` is `'block 60x20x10'`; the optional `description` is empty; `frame_of_reference` points at a `PRODUCT_CONTEXT`. Schema:

```
ENTITY product;
  id                : identifier;
  name              : label;
  description       : OPTIONAL text;
  frame_of_reference : SET [1:?] OF product_context;
END_ENTITY;
```

`PRODUCT_DEFINITION_FORMATION` is a **version** of that product. "Formation" is the standard's word for revision. It has an `id` (the revision letter or number), a `description`, and a reference to the product it is

a version of. This exporter has left both strings empty — it does not know about revisions — and pointed at #7.

```
ENTITY product_definition_formation;
  id          : identifier;
  description : OPTIONAL text;
  of_product  : product;
END_ENTITY;
```

PRODUCT_DEFINITION is that version as seen from one **discipline** or **life-cycle stage**. The id here is 'design'; the frame_of_reference points at a PRODUCT_DEFINITION_CONTEXT that names the stage. The same version of the same part could have a second PRODUCT_DEFINITION for manufacturing, with different geometry hanging off it — stock material, no chamfers, tapped holes shown as drilled — and a third for inspection.

```
ENTITY product_definition;
  id                : identifier;
  description       : OPTIONAL text;
  formation         : product_definition_formation;
  frame_of_reference : product_definition_context;
END_ENTITY;
```

So the three levels are **item → revision → view**. This is the data model of a product lifecycle management (PLM) system, and it is in the file because the integrated resources were designed to serve PLM systems as much as CAD exporters. A CAD tool writing a single part fills two of the three levels with empty strings and moves on; the structure is there for someone else.

> **i** Engineering background: Item, revision, view
>
> In a manufacturing organisation a *part number* identifies a design intent — "the bracket." A *revision* identifies a specific released state of that design — "bracket, rev C" — and revisions are not interchangeable: rev C may have a different hole spacing from rev B, and a drawing or an assembly refers to a specific one. A *view* (or *definition*) is the same revision described for a purpose: the design view carries nominal geometry and tolerances; the manufacturing view carries the as-machined model, stock, and process notes; the as-built view carries what was actually made. PLM systems track all three levels and the relationships between them. STEP inherits the model so that a file can carry, say, an assembly whose components are at specific revisions, or a manufacturing definition alongside a design definition, without inventing new structure. Most CAD exports use none of this, which is why the middle level is nearly always blank.

5.2 The contexts

Two records exist only to anchor the chain in a domain:

```
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
```

> *Referred to above: 1 record from `block.step`. The complete file.*
>
> ```
> #2 = APPLICATION_CONTEXT(
>   'core data for automotive mechanical design processes');
> ```

PRODUCT_CONTEXT says what discipline the product belongs to; its discipline_type is 'mechanical' and its frame_of_reference is the application context #2. PRODUCT_DEFINITION_CONTEXT says what life-cycle stage the definition is in; life_cycle_stage is 'design', again pointing at #2. Both strings are convention rather than rule. AP203 and AP242 fix them on the subtypes MECHANICAL_CONTEXT and DESIGN_CONTEXT, which both schemas list as deprecated and no exporter writes; AP214, the schema of this file, does not define those subtypes at all, and leaves both values as unconstrained labels. What AP214 does say runs the other way: a definition whose context is named 'part definition' or 'part occurrence' must belong to a product categorised 'part', 'raw material' or 'tool'. Both records are boilerplate that you will learn to skim. Some exporters write one pair per product, as OCCT does; others share one pair across every product in the file. Both are legal.

5.3 The category

The last non-geometric record in the file classifies the product:

```
#170 = PRODUCT_RELATED_PRODUCT_CATEGORY('part',$,(#7));
```

Referred to above: 3 records from `block.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
```

A `PRODUCT_RELATED_PRODUCT_CATEGORY` has a `name`, an optional `description`, and a list of the products in the category. The name is an unconstrained string, but the APs constrain it from outside, and differently. AP214 requires every product to carry exactly one category named from a fixed list: `'part'`, `'raw material'`, `'tool'`, `'document'`, and five more for requirements, functions, concepts, alternative solutions and physically realised products. AP203 and AP242 impose no such list; they tie particular names to particular kinds of definition — `'assembly'` on a product whose definition is declared an assembly definition, `'part'`, `'raw material'` or `'tool'` on a part occurrence, and a handful more for collections and support structures. Either way the category does not tell a leaf from an assembly: OCCT writes `'part'` for the assembly as well as for its components. The assembly structure is in the `NEXT_ASSEMBLY_USAGE_OCCURRENCE` records of Chapter 8, and nowhere else.

Notice the reference direction: the *category* points at the *product*. A product does not know its own category; you find it by looking for categories that list it. This is the pattern from Chapter 2 again — specific to general — and it will recur every time a classification, a property or a style is attached to something.

5.4 What exporters put where

The schema says what the slots are for. It does not say what a CAD system should write in them, and they differ. What follows is what these exporters have been seen to write, not what their vendors promise.

- **`product.id`** is meant to be the part number. OCCT writes whatever name it was given; SolidWorks writes the document's filename without extension; NX and Creo write the part name, which in a well-run shop *is* the part number; CATIA writes the part number field from the part's properties, which is often the same string as the name.
- **`product.name`** is meant to be the human name. Most exporters write the same string as `id`. OCCT, if given only one name, duplicates it, which is why unedited OCCT files say `PRODUCT('Open CASCADE STEP translator 7.9 1','Open CASCADE STEP translator 7.9 1', ...)`.
- **`product_definition_formation.id`** is the revision. It is empty in almost every CAD export. When PLM-integrated systems write it, it holds `'A'`, `'B'`, `'01'`.
- **`product_definition.id`** is nearly always `'design'`, sometimes `''`, occasionally the part name again.

The practical rule for finding "the name of this part" is: `product.name`, falling back to `product.id`, falling back to the filename. The practical rule for finding the part *number* is `product.id`, and it is only as good as the exporter's mapping.

> ⚠ In the wild: Formations with a source
>
> All three protocols define a subtype, `PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE`, which adds a `make_or_buy` enumeration: `.MADE.`, `.BOUGHT.` or `.NOT_KNOWN.`. AP203's first edition required it; its second edition and AP242 both list it as deprecated, which is why it turns up in AP203 exports and almost nowhere else. It is written mostly in AP203 exports, almost always with `.NOT_KNOWN.`, so a file's formation records look like `PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE('','',#7,.NOT_KNOWN.)`. It is the same record with one more attribute, and a reader should treat it as a formation.

5.5 Several products in one file

A single-part file has one chain. An assembly file has one chain *per product* — the assembly and each distinct component — each with its own `PRODUCT`, formation, definition and category, and usually its own pair of contexts. The clamp assembly in Chapter 8 has three: `CLAMP-ASSY`, `bracket` and `pin-1`. What connects them is not in the product chain at all; it is a separate set of records that relate one `PRODUCT_DEFINITION` to another. The chain establishes *that* things exist. Structure is layered on top.

One more thing the chain does not do: it does not point at geometry. `#5`, the design definition of the block, knows its version and its context and nothing else. The shape is attached from the other side, by records that point *at* `#5`, and that is the subject of Chapter 7.

! In brief

`PRODUCT` → `PRODUCT_DEFINITION_FORMATION` → `PRODUCT_DEFINITION` is item → revision → view, the PLM model built into the format. Each level points at the one above. `PRODUCT_CONTEXT` and `PRODUCT_DEFINITION_CONTEXT` anchor the chain in a discipline and life-cycle stage with conventional strings. `PRODUCT_RELATED_PRODUCT_CATEGORY` classifies the product and points at it, not the other way round, and it does not tell you whether the product is an assembly.

6. Measuring Things: Contexts and Units

> *Five records that say "millimetres, radians, and anything closer than a ten-millionth is the same point." Why a millimetre is three types at once, how an inch file smuggles in a millimetre anyway, and what a kernel's tolerance is doing in an exchange file.*

Every number in the geometry of the block file — the `60.` in a corner point, the `1.` in a direction, the `3.` in a circle's radius later on — is a bare real. Nothing on the record says what unit it is in. That information lives in one place, and every geometric representation points at it:

```
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
    ...
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
  GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169))
  GLOBAL_UNIT_ASSIGNED_CONTEXT((#166,#167,#168))
  REPRESENTATION_CONTEXT('Context #1','3D Context with UNIT and UNCERTAINTY') );
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#167 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#168 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
```

Referred to above: 155 records from `block.step`; the 3 it points to directly are shown. The complete file.

```
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);
#15 = MANIFOLD_SOLID_BREP('',#16);
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#166,#167,#168)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
/* ... and 152 more: 27 CARTESIAN_POINT, 26 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
```

Read top down: *a three-dimensional coordinate space, measured in millimetres, radians and steradians, in which two points closer than* 10^{-7} *mm are the same point.*

6.1 Representations and their contexts

The word *representation* is Part 43's, and it is worth a moment because it organises half the file. A **representation** is a set of **representation items** together with a **representation context** that says how to interpret them. `#10` is a representation: its items are `#11` (an axis placement) and `#15` (a solid), and its context is `#165`. Every point, curve, surface, face and solid in the file is a representation item, and every one belongs to some representation, and every representation has exactly one context.

The context is where units live. It is shared: in a single-part file one context carries the units and every geometric representation points at it — though the file may still hold a unitless 2D context per pcurve, as Chapter 13 shows. In an assembly file there may be several — OCCT writes one per product, all identical — and each representation points at its own. A reader that assumes one context per file will be right most of the time and wrong in ways that are hard to notice.

6.2 The context record

`#165` is a complex instance of four types, and Chapter 4 explained why: `representation_context` has no supertype constraint, so any combination of its subtypes is legal, and this one is three at once.

- `REPRESENTATION_CONTEXT('Context #1','3D Context with UNIT and UNCERTAINTY')` — the root, with an identifier and a free-text type. Both strings are the writer's choice and mean nothing to a reader.

- `GEOMETRIC_REPRESENTATION_CONTEXT(3)` — this context is for geometry, in three dimensions. Two-dimensional contexts, `(2)`, appear in files that carry drawing views or the 2D parameter-space curves of Chapter 13.
- `GLOBAL_UNIT_ASSIGNED_CONTEXT((#166,#167,#168))` — the units in force for every item in this context.
- `GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169))` — the tolerance in force.

AP203's first edition is the strictest: the unit set must hold exactly three units, one length, one plane angle and one solid angle. AP214 requires exactly one length unit and exactly one plane angle unit and says nothing about solid angle; AP203 second edition and AP242 state no rule at all. Nothing in a mechanical file ever measures a solid angle, but the oldest and most widely implemented of the three demanded one, so exporters emit a steradian everywhere and you will see `#168` in nearly every file.

6.3 The millimetre

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
```

Three blocks, alphabetical. `LENGTH_UNIT()` declares no attributes and says only *this measures length*. `NAMED_UNIT(*)` carries the derived `dimensions`. `SI_UNIT(.MILLI.,.METRE.)` gives the prefix and the base unit. A metre would be `SI_UNIT($,.METRE.)`; a micron `SI_UNIT(.MICRO.,.METRE.)`. The radian and steradian at `#167` and `#168` follow the same pattern with `$` for no prefix, and with `PLANE_ANGLE_UNIT()` and `SOLID_ANGLE_UNIT()` in place of `LENGTH_UNIT()`.

This exact record — the millimetre as a three-way complex instance — is probably the most frequently written line in all of STEP.

6.4 Inch files

Imperial units cannot be spelled with an SI prefix and name, so an inch is a **conversion-based unit**: a named unit defined as a multiple of another unit. The block, exported by the same OCCT build with its output unit set to inches:

```
#166 = ( CONVERSION_BASED_UNIT('INCH',#168) LENGTH_UNIT() NAMED_UNIT(#167) );
#167 = DIMENSIONAL_EXPONENTS(1.,0.,0.,0.,0.,0.,0.);
#168 = LENGTH_MEASURE_WITH_UNIT(25.4,#169);
#169 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#172 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(3.937007874016E-09),#166,
  'distance_accuracy_value','confusion accuracy');
```

`#166` is still a length unit and still a named unit, but `SI_UNIT` has been replaced by `CONVERSION_BASED_UNIT`, which carries a `name` and a `conversion_factor`. Because the blocks are alphabetical it now sorts first. The factor, `#168`, is a *measure with unit*: 25.4 of unit `#169`. And `#169` is the millimetre. So an inch file contains the millimetre after all — as the thing the inch is defined in terms of — and every coordinate in the file is then in inches: the block's far corner is at `(2.362204724409,0.787401574803,0.393700787402)`, and the uncertainty has been converted too, to 3.9×10^{-9} inches.

Three details. First, `NAMED_UNIT(#167)` is an explicit reference where the SI record had `*`. An SI unit's dimensions are always derived from its name, so that slot is `*`. For a conversion-based unit AP203 and AP242 require the exponents to be written out and to agree with the factor's unit; AP214 edition 3 derives them instead, so a conversion-based unit written to that edition carries `NAMED_UNIT(*)` and no `DIMENSIONAL_EXPONENTS` record at all. This file, an older AP214, writes them out. Here `DIMENSIONAL_EXPONENTS(1.,0.,0.,0.,0.,0.,0.)` says length^{1} mass^{0} time^{0} current^{0} temperature^{0} substance^{0} luminosity^{0} — a length. Second, the string `'INCH'` is a label. What makes it an inch is the `25.4`. A reader that keys on the name will meet `'inch'`, `'IN'` and `'Inch'` before long; the factor is the reliable part. Third — and this is a real defect in a widely used exporter — the `25.4` in `#168` is written bare. Chapter 4's rule says a SELECT-typed value that is not an entity must be wrapped, and the uncertainty record two lines below wraps its value correctly as `LENGTH_MEASURE(...)`; the conversion factor should read `LENGTH_MEASURE(25.4)`. Every mainstream reader accepts the bare form, which is why it has never been fixed, and it is a fair specimen of the gap between the standard and the files.

Angles are treated the same way. A file that measures angles in degrees carries (this excerpt written to the same pattern rather than exported)

```
#900 = ( CONVERSION_BASED_UNIT('DEGREE',#901) NAMED_UNIT(#902)
         PLANE_ANGLE_UNIT() );
#901 = PLANE_ANGLE_MEASURE_WITH_UNIT(PLANE_ANGLE_MEASURE(0.01745329252),#903);
#902 = DIMENSIONAL_EXPONENTS(0.,0.,0.,0.,0.,0.,0.);
#903 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
```

— a degree is 0.0174533 radians, and an angle is dimensionless, so every exponent is zero.

6.5 Uncertainty

```
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
```

Referred to above: 1 record from `block.step`. The complete file.

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
```

Four attributes: a value (wrapped, because `measure_value` is a SELECT), a unit (bare, because it is an entity), a name, and a description. The name is the important one. `'distance_accuracy_value'` is the string the recommended practices call for, and it means: *two points closer together than this value are to be treated as coincident.* Here that is one ten-millionth of a millimetre.

> **i** Engineering background: Why a tolerance travels with the geometry
>
> A boundary representation says that a solid's faces meet along shared edges and its edges meet at shared vertices. But the faces are defined by surfaces — planes, cylinders, splines — and the edges by curves, and those are computed in floating point. Where a cylinder meets a plane, the curve of intersection does not lie *exactly* on either; it lies within some small distance of both. Whether a shell is "closed" is therefore a question about a threshold: are the gaps smaller than the tolerance? Every geometric kernel has such a threshold, and they differ — 10^{-7} in Open CASCADE, 10^{-6} or 10^{-8} elsewhere, sometimes scaled to the model's size. The file records the writer's value so that a reader can know how tightly the geometry was made to fit. If the reader's kernel is stricter, gaps that were invisible at export become holes at import, and the solid falls apart into a surface model. Much of the "healing" that STEP importers do is repairing exactly that.

The uncertainty is what makes the same file behave differently in different tools. A `'confusion accuracy'` of 10^{-7} mm from one kernel and 10^{-4} mm from another describe geometry fitted to very different thresholds, and the reader that ignores this value is the one whose imports mysteriously fail. The value is a declaration, not a measurement: OCCT, for one, can be set to write its default precision, a user-chosen value, or the smallest, average or largest tolerance among the shape's sub-shapes, and its reader can be told to ignore the file's value and use its own. It says nothing about topology either: vertices are shared by reference, and nothing in a file is merged because its coordinates are close.

6.6 Derived units

Length, angle and solid angle cover geometry. Properties such as area, volume and mass need units that are products of the base ones, and Part 41 provides `DERIVED_UNIT`:

```
#300 = DERIVED_UNIT((#301));
#301 = DERIVED_UNIT_ELEMENT(#166,3.);
```

Referred to above: 1 record from `block.step`. The complete file.

```
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
```

`#300` is "millimetres to the power three" — a cubic millimetre — built from one element: the length unit `#166` raised to the exponent `3.`. An area would use `2.`; a density would combine a mass unit with exponent `1.` and a length unit with exponent `-3.`. These appear with validation properties in Chapter 15 and almost nowhere else.

6.7 Contexts in assemblies

The clamp assembly of Chapter 8 has three geometric contexts, one per product, all textually identical to `#165`. Each product's shape representation points at its own. One context for the whole file is not even legal here: the transformation record that places a component requires its two representations to sit in different context records, so the assembly's context cannot also be a component's. Writing one per product, as this exporter does, is the simplest way to satisfy that. It has a consequence that is easy to state and easy to forget: **units are a property of the representation, not of the file.** A file may, in principle, hold one part in millimetres and another in inches, each in its own context, and an assembly that places both. Readers that read the first context they find and apply it to everything will get the second part wrong by a factor of 25.4. Follow the reference from each representation to its context; never cache "the" unit.

> ! In brief
>
> Units live in the representation context, which every geometric representation references. A geometric context carries a length unit and a plane angle unit — required by AP214, conventional elsewhere — and by convention a solid angle unit too; each is a complex instance. SI units derive their dimensions; conversion-based units (inch, degree) define themselves as a multiple of another unit — this inch file carries 25.4 millimetres, another could carry 0.0254 metres — and whether their dimensions are written out or derived depends on the schema edition. The uncertainty is the writing kernel's coincidence tolerance, named `'distance_accuracy_value'`: a declaration of the writer's threshold, not a measurement of the geometry. A file can have several contexts; units belong to representations, not files.

7. From Product to Shape

> *Two records bridge the product chain to the geometry, and they point in opposite directions. Following them is the first thing every reader does and the first place the arrows run the wrong way.*

The product chain ends at #5, the design definition of the block. The geometry begins at #10, a shape representation. Nothing in either record refers to the other. What joins them is a pair of records that Part 41 and Part 43 supply for exactly this purpose:

```
#3 = SHAPE_DEFINITION_REPRESENTATION(#4,#10);
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
    ...
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);
#15 = MANIFOLD_SOLID_BREP('',#16);
```

Referred to above: 163 records from `block.step`; the 7 it points to directly are shown. The complete file.

```
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
#12 = CARTESIAN_POINT('',(0.,0.,0.));
#13 = DIRECTION('',(0.,0.,1.));
#14 = DIRECTION('',(1.,0.,-0.));
#16 = CLOSED_SHELL('',(#17,#57,#97,#119,#141,#153));
/* ... and 156 more: 26 CARTESIAN_POINT, 24 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
```

Drawn with the arrows as the references actually point:

```
#5    PRODUCT_DEFINITION ('design')              the part, design view
 ↑
#4    PRODUCT_DEFINITION_SHAPE ('','',#5)         "this part has a shape"
 ↑
#3    SHAPE_DEFINITION_REPRESENTATION (#4,#10)   "...and one representation of it"
 │
 ↓
#10   ADVANCED_BREP_SHAPE_REPRESENTATION          the container: items + context
 │
 ↓
#15   MANIFOLD_SOLID_BREP                          the solid
#11   AXIS2_PLACEMENT_3D                          the part's own coordinate frame
```

From the part, the first two hops go *against* the references — #4 points at #5, #3 points at #4 — and from #3 on they run forward. Every walk from a product to its geometry has this shape: two steps backward, then forward all the way down. It is the concrete reason a reader needs an index of "who points at me," because the file offers no other way to get from #5 to #3.

7.1 PRODUCT_DEFINITION_SHAPE

```
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
```

Referred to above: 6 records from `block.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
```

```
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
```

The record looks like pure bureaucracy — a name, a description, and "the thing this is the shape of" — and it is, but it is a hook that a surprising amount hangs from. Its schema:

```
ENTITY property_definition;
  name        : label;
  description : OPTIONAL text;
  definition  : characterized_definition;
END_ENTITY;

ENTITY product_definition_shape
  SUBTYPE OF (property_definition);
END_ENTITY;
```

`PRODUCT_DEFINITION_SHAPE` is a *property definition* — the same entity family that carries mass, material and user-defined attributes in Chapter 15 — specialised to mean "the shape property." Its `definition` is a SELECT that may point at a `product_definition`, as here, or at a `next_assembly_usage_occurrence`, which is how a *placement* becomes a shape property of an assembly relationship in the next chapter. When PMI is present, the tolerance features are `shape_aspects` whose `of_shape` points back at this record. It is the one node in the product graph that stands for "the geometry of this part, abstractly," and everything that needs to talk about that geometry without naming a specific face attaches here.

7.2 SHAPE_DEFINITION_REPRESENTATION

```
#3 = SHAPE_DEFINITION_REPRESENTATION(#4,#10);
```

Referred to above: 167 records from `block.step`; the 2 it points to directly are shown. The complete file.

```
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
/* ... and 165 more: 27 CARTESIAN_POINT, 26 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
```

Two attributes: `definition`, the property being represented, and `used_representation`, the representation that does it. It says: *the shape of this part is represented by `#10`*.

The word *one* matters. A part may have several `SHAPE_DEFINITION_REPRESENTATION`s, each pointing at the same `PRODUCT_DEFINITION_SHAPE` and at a different representation: a precise B-rep, a tessellated mesh of the same body, a simplified envelope, or — commonly — a representation containing nothing but an axis placement, so that the part has a coordinate frame even when its geometry is elsewhere. Readers that take "the first SDR" get the first one the exporter happened to write, which is not always the one wanted.

7.3 The shape representation

```
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
```

Referred to above: 159 records from `block.step`; the 3 it points to directly are shown. The complete file.

```
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);
#15 = MANIFOLD_SOLID_BREP('',#16);
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#166,#167,#168)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
/* ... and 156 more: 27 CARTESIAN_POINT, 26 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
```

Three attributes, inherited unchanged from Part 43's `representation`: a `name`, the set of `items`, and the `context_of_items`. This is a **shape representation** — a representation whose items are geometry — and specifically an `ADVANCED_BREP_SHAPE_REPRESENTATION`, a subtype that promises its items are B-rep solids built from *advanced* faces (the well-behaved kind, with analytic or B-spline surfaces and fully specified edges).

The subtypes of `SHAPE_REPRESENTATION` are a vocabulary for what kind of geometry to expect:

Entity type	Items are
SHAPE_REPRESENTATION	anything — often just an axis placement
ADVANCED_BREP_SHAPE_REPRESENTATION	MANIFOLD_SOLID_BREP solids
MANIFOLD_SURFACE_SHAPE_REPRESENTATION	shells, open or closed, with no solid semantics
GEOMETRICALLY_BOUNDED_SURFACE_SHAPE_REPRESENTATION	surfaces trimmed by curves, no topology
GEOMETRICALLY_BOUNDED_WIREFRAME_SHAPE_REPRESENTATION	curves only
TESSELLATED_SHAPE_REPRESENTATION	triangle meshes (AP242)
FACETED_BREP_SHAPE_REPRESENTATION	solids with planar faces only

Each subtype has rules about what may appear in `items`; none adds attributes. For the purposes of walking the file they are all SHAPE_REPRESENTATIONs, and a reader that wants "the solid" looks for a representation whose items include a MANIFOLD_SOLID_BREP, whatever the record's type name.

7.4 The items

#10 has two items.

```
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);
#12 = CARTESIAN_POINT('',(0.,0.,0.));
#13 = DIRECTION('',(0.,0.,1.));
#14 = DIRECTION('',(1.,0.,-0.));
```

#11 is an **axis placement**: a point and two directions defining a coordinate frame — origin, Z axis, X axis. It is the part's own frame, sitting here at the world origin with the identity orientation. Nearly every shape representation carries one as its first item. It does not move the geometry; the solid's coordinates are already in this frame. It exists so that an assembly can say "put the frame of this part at *that* position," and it is what the placement transforms in Chapter 8 refer to.

```
#15 = MANIFOLD_SOLID_BREP('',#16);
#16 = CLOSED_SHELL('',(#17,#57,#97,#119,#141,#153));
```

> *Referred to above: 148 records from `block.step`; the 6 it points to directly are shown. The complete file.*
>
> ```
> #17 = ADVANCED_FACE('',(#18),#52,.F.);
> #57 = ADVANCED_FACE('',(#58),#92,.T.);
> #97 = ADVANCED_FACE('',(#98),#114,.F.);
> #119 = ADVANCED_FACE('',(#120),#136,.T.);
> #141 = ADVANCED_FACE('',(#142),#148,.F.);
> #153 = ADVANCED_FACE('',(#154),#160,.T.);
> /* ... and 142 more: 26 CARTESIAN_POINT, 24 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
> ```

#15 is the solid, and it has one attribute besides its name: the closed shell that bounds it. The shell lists six faces. From here down is Part II.

A shape representation can have more than one solid in its items — a part modelled as several disjoint bodies, or a body plus reference geometry. The clamp's assembly representation in the next chapter has four items and no solids at all: an origin and three placements for the three children.

7.5 Reading the whole path

Putting the chapter together, the complete route from "a part called BLOCK" to "a closed shell with six faces" in this file is:

```
#7  PRODUCT ('BLOCK')
#6  PRODUCT_DEFINITION_FORMATION                <- version
#5  PRODUCT_DEFINITION ('design')               <- view
#4  PRODUCT_DEFINITION_SHAPE                    <- shape property   (points UP at #5)
```

```
#3  SHAPE_DEFINITION_REPRESENTATION            <- binding  (UP at #4, DOWN at #10)
#10 ADVANCED_BREP_SHAPE_REPRESENTATION         <- container  (-> #165 for units)
#15 MANIFOLD_SOLID_BREP
#16 CLOSED_SHELL
#17 ... ADVANCED_FACE x 6
```

Nine records from the part number to the first face, four of which exist only to connect the other five. It is a lot of indirection for a single block, and the same nine records appear, once per part, in a file with a thousand parts. But every one of the joints is a place where something else can attach — a second version, a manufacturing view, a tessellated alternative, an assembly placement — and the format pays for that flexibility on every part whether it is used or not.

! In brief

`PRODUCT_DEFINITION_SHAPE` points up at the product definition and stands for its shape abstractly. `SHAPE_DEFINITION_REPRESENTATION` points up at that and down at a shape representation — one of possibly several. The shape representation's subtype names the kind of geometry; its items are the geometry plus, almost always, an axis placement that is the part's own frame. Two hops against the arrows, then forward.

8. Assemblies

> *A bracket and two pins, exported as one file of 335 records. Where the tree is, where the positions are, why they are kept apart, and the second way exporters do it.*

The clamp is the bracket of Section 11.5 with two pins standing on it: one at x = 15, one at x = 45, both on the top face (Figure 8.1). As a STEP file it has three products but only two solids' worth of geometry — the pin's geometry appears once, not twice — and about forty records of structure that say how they fit together. This chapter is about those forty.

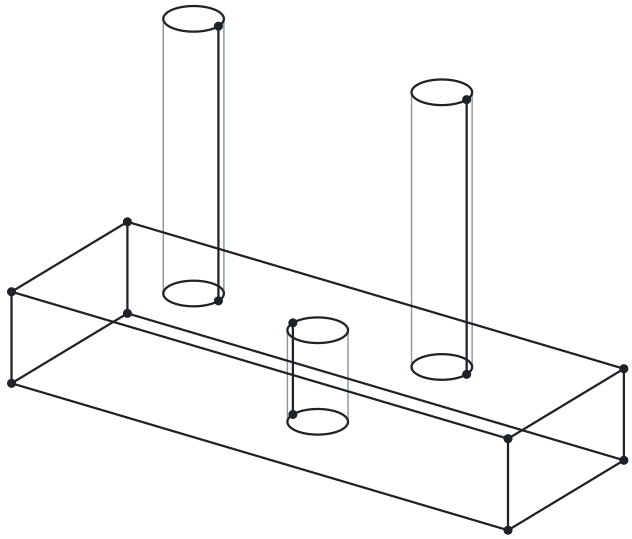


Figure 8.1: `clamp.step`, drawn by placing each component where the file's transformations put it: the bracket once, the pin twice.

8.1 Three product chains

The file opens the way every file does, with a product chain. Then it has two more:

```
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#7 = PRODUCT('CLAMP-ASSY','CLAMP-ASSY','',(#8));
    ...
#35 = PRODUCT_DEFINITION('design','',#36,#39);
#37 = PRODUCT('bracket','bracket','',(#38));
    ...
#243 = PRODUCT_DEFINITION('design','',#244,#247);
#245 = PRODUCT('pin-1','pin-1','',(#246));
```

Referred to above: 6 records from `clamp.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#38 = PRODUCT_CONTEXT('',#2,'mechanical');
#246 = PRODUCT_CONTEXT('',#2,'mechanical');
```

Each has its own formation, its own pair of contexts, its own category record. #5 is the assembly's definition, #35 the bracket's, #243 the pin's. Nothing in these records relates them to one another. The pin's product is called 'pin-1' rather than 'pin' — the exporter took the name of the first instance it met as the name of the part — which is the kind of thing you learn to expect from assembly exports and which the structure below does not depend on.

Each product also has its shape. The bracket's and pin's are ADVANCED_BREP_SHAPE_REPRESENTATIONs containing solids, exactly as in the single-part files. The assembly's is different:

```
#3 = SHAPE_DEFINITION_REPRESENTATION(#4,#10);
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#10 = SHAPE_REPRESENTATION('',(#11,#15,#19,#23),#27);
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);        /* origin */
#15 = AXIS2_PLACEMENT_3D('',#16,#17,#18);        /* (0,0,0): the bracket goes here */
#19 = AXIS2_PLACEMENT_3D('',#20,#21,#22);        /* (15,10,10): pin-1 goes here */
#23 = AXIS2_PLACEMENT_3D('',#24,#25,#26);        /* (45,10,10): pin-2 goes here */
```

Referred to above: 23 records from `clamp.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('CLAMP-ASSY','CLAMP-ASSY','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#12 = CARTESIAN_POINT('',(0.,0.,0.));
#13 = DIRECTION('',(0.,0.,1.));
#14 = DIRECTION('',(1.,0.,-0.));
#16 = CARTESIAN_POINT('',(0.,0.,0.));
#17 = DIRECTION('',(0.,0.,1.));
#18 = DIRECTION('',(1.,0.,-0.));
#20 = CARTESIAN_POINT('',(15.,10.,10.));
#21 = DIRECTION('',(0.,0.,1.));
#22 = DIRECTION('',(1.,0.,-0.));
#24 = CARTESIAN_POINT('',(45.,10.,10.));
#25 = DIRECTION('',(0.,0.,1.));
#26 = DIRECTION('',(1.,0.,-0.));
#27 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#31)) GLOBAL_UNIT_ASSIGNED_CONTEXT(
(#28,#29,#30)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
#28 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#29 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#30 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#31 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#28,
  'distance_accuracy_value','confusion accuracy');
```

The assembly has no solid. Its shape representation is a plain SHAPE_REPRESENTATION whose items are four coordinate frames: its own origin and one target position for each child. The geometry of an assembly *is* the geometry of its children, placed. This representation is the assembly's coordinate space, and the placements in it are the only numbers in the whole structure.

8.2 The tree: NEXT_ASSEMBLY_USAGE_OCCURRENCE

The parent–child relationships are three records:

```
#239 = NEXT_ASSEMBLY_USAGE_OCCURRENCE('1','bracket','',#5,#35,$);
#311 = NEXT_ASSEMBLY_USAGE_OCCURRENCE('2','pin-1','',#5,#243,$);
#317 = NEXT_ASSEMBLY_USAGE_OCCURRENCE('3','pin-2','',#5,#243,$);
```

Referred to above: 16 records from `clamp.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('CLAMP-ASSY','CLAMP-ASSY','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
```

```
#35 = PRODUCT_DEFINITION('design','',#36,#39);
#36 = PRODUCT_DEFINITION_FORMATION('','',#37);
#37 = PRODUCT('bracket','bracket','',(#38));
#38 = PRODUCT_CONTEXT('',#2,'mechanical');
#39 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#243 = PRODUCT_DEFINITION('design','',#244,#247);
#244 = PRODUCT_DEFINITION_FORMATION('','',#245);
#245 = PRODUCT('pin-1','pin-1','',(#246));
#246 = PRODUCT_CONTEXT('',#2,'mechanical');
#247 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
```

The schema, through three levels of inheritance:

```
ENTITY product_definition_relationship;
  id                          : identifier;
  name                        : label;
  description                 : OPTIONAL text;
  relating_product_definition : product_definition;     -- the parent
  related_product_definition  : product_definition;     -- the child
END_ENTITY;

ENTITY product_definition_usage
  SUBTYPE OF (product_definition_relationship);
END_ENTITY;

ENTITY assembly_component_usage
  SUBTYPE OF (product_definition_usage);
  reference_designator : OPTIONAL identifier;
END_ENTITY;

ENTITY next_assembly_usage_occurrence
  SUBTYPE OF (assembly_component_usage);
END_ENTITY;
```

`NEXT_ASSEMBLY_USAGE_OCCURRENCE` declares nothing of its own; all six parameters are inherited. Six parameters: an id, a name, a description, the **relating** (parent) definition, the **related** (child) definition, and an optional reference designator. One record is one **occurrence** of one child inside one parent. These records form a directed acyclic graph of product definitions, with a separate edge for each usage. The occurrence tree is that graph expanded from a root; a reused subassembly is expanded once per occurrence, each with its own world transform. There is no record for "the assembly structure" as a whole; there are only edges, and the tree is whatever they add up to.

Three things to notice.

`#311` and `#317` both have `#243` as their child. The pin is **defined once and used twice**. This is the distinction a bill of materials rests on: one product definition, two occurrences. The BOM line for the pin has quantity 2 because two NAUOs point at it, and there is one pin geometry in the file, not two.

The `name` attribute carries the **instance name** — `'pin-1'`, `'pin-2'` — which is how the CAD system distinguished the two occurrences in its tree. The part's own name is on the `PRODUCT` record. In this file they collide (`'pin-1'` is both the product name and the first instance name) because of the exporter's naming choice; in a SolidWorks or NX export the product would be `'PIN'` and the instances `'PIN-1'` and `'PIN-2'`.

The `reference_designator` is `$`. It is meant for the position label on an electrical schematic or a drawing balloon — `'R12'`, `'B3'` — and most mechanical exporters leave it empty.

A tree with only NAUOs is unplaced. The NAUOs say nothing about where each child sits; a viewer that draws them at the parent's origin is falling back, not reading a placement. Position is a separate structure, attached to each NAUO from the side.

8.3 The position: five records per occurrence

For the bracket:

```
#238 = PRODUCT_DEFINITION_SHAPE('Placement','Placement of an item',#239);
#235 = CONTEXT_DEPENDENT_SHAPE_REPRESENTATION(#236,#238);
```

```
#236 = ( REPRESENTATION_RELATIONSHIP('','',#40,#10)
         REPRESENTATION_RELATIONSHIP_WITH_TRANSFORMATION(#237)
         SHAPE_REPRESENTATION_RELATIONSHIP() );
#237 = ITEM_DEFINED_TRANSFORMATION('','',#11,#15);
#15 = AXIS2_PLACEMENT_3D('',#16,#17,#18);       /* in the assembly's representation */
```

Referred to above: 228 records from `clamp.step`; the 7 it points to directly are shown. The complete file.

```
#10 = SHAPE_REPRESENTATION('',(#11,#15,#19,#23),#27);
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);
#16 = CARTESIAN_POINT('',(0.,0.,0.));
#17 = DIRECTION('',(0.,0.,1.));
#18 = DIRECTION('',(1.,0.,-0.));
#40 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#41),#230);
#239 = NEXT_ASSEMBLY_USAGE_OCCURRENCE('1','bracket','',#5,#35,$);
/* ... and 221 more: 37 DIRECTION, 35 CARTESIAN_POINT, 30 ORIENTED_EDGE, 15 EDGE_CURVE, ... */
```

Reading it from the NAUO outward:

#238 is a `PRODUCT_DEFINITION_SHAPE` whose `definition` is the NAUO `#239` rather than a product definition. Chapter 7 said the `definition` attribute is a SELECT that admits either; this is the other case. It means: *the occurrence of the bracket in the clamp has a shape*. The strings `'Placement'` and `'Placement of an item'` are this exporter's habit; others write `''`.

#235 is the bridge. `CONTEXT_DEPENDENT_SHAPE_REPRESENTATION` has two attributes: a `representation_relation` and a `represented_product_relation`. It ties the occurrence-shape `#238` to a relationship between two representations.

#236 is a complex instance you can now read. It is a `REPRESENTATION_RELATIONSHIP` between `#40` (the bracket's shape representation) and `#10` (the assembly's); it is *with a transformation*, `#237`; and it is specifically a *shape representation relationship*. `REPRESENTATION_RELATIONSHIP_WITH_TRANSFORMATION` and `SHAPE_REPRESENTATION_RELATIONSHIP` are sibling subtypes of `REPRESENTATION_RELATIONSHIP`, and no single leaf names the pair. So the instance is written externally, with their shared supertype listed as well, and the blocks in alphabetical order.

#237 is the transformation, and it contains no numbers. `ITEM_DEFINED_TRANSFORMATION` has two attributes beyond name and description: `transform_item_1` and `transform_item_2`, both representation items. Here they are two axis placements: `#11`, the origin frame that sits in the bracket's own representation, and `#15`, a frame in the assembly's representation. The transformation is *whatever rigid motion carries frame `#11` onto frame `#15`*. Put the bracket's origin at `#15`′s origin, its Z along `#15`′s Z, its X along `#15`′s X.

#15 is that target frame — at (0, 0, 0) with identity orientation, because the bracket sits at the assembly origin. For pin-1 the corresponding record is `#19`, at (15, 10, 10); for pin-2 it is `#23`, at (45, 10, 10). Those three points are the only coordinates in the assembly structure.

So the NAUO says *what* goes inside *what*, and its placement chain says *where*, and the two are joined through `#238` only — and only in the inverse direction, since `#238` points at the NAUO and nothing points at `#238` except `#235`, which nothing points at either. From a NAUO, finding its position means asking "what `PRODUCT_DEFINITION_SHAPE` refers to me?" and then "what `CONTEXT_DEPENDENT_SHAPE_REPRESENTATION` refers to *that*?" — two more walks against the arrows before the references turn forward.

8.4 The frame as a transform

```
#19 = AXIS2_PLACEMENT_3D('',#20,#21,#22);
#20 = CARTESIAN_POINT('',(15.,10.,10.));
#21 = DIRECTION('',(0.,0.,1.));
#22 = DIRECTION('',(1.,0.,-0.));
```

An `AXIS2_PLACEMENT_3D` has a `location`, an optional `axis` and an optional `ref_direction`. The axis, normalised, is the frame's Z; the reference direction is projected perpendicular to Z and normalised to give X; Y is Z × X. If the axis is omitted, Z is (0, 0, 1). If the reference direction is omitted, Part 42's `first_proj_axis` projects (1, 0, 0), or (0, 1, 0) when Z is itself (±1, 0, 0): the fallback is specified, not an arbitrary perpendicular. Three references, and the result is a complete right-handed coordinate frame — which is to say a rigid transformation: a rotation plus a translation, with no scaling and no reflection.

Placing pin-1, then, means: take the pin's geometry, whose coordinates are in its own frame (`#11` in the pin's representation — the pin stands on the origin with its axis along Z), and move it so that frame lands

on #19. Translation by (15, 10, 10), no rotation. The pin's base circle, at z = 0 in its own file, is at z = 10 in the assembly: the top of the bracket.

> ⚠ In the wild: Which frame is which
>
> ITEM_DEFINED_TRANSFORMATION does not say which of its two items is the child's origin and which is the target. In AP242 the schema already ties the items to the representations: transform_item_1 must be used in rep_1 and transform_item_2 in rep_2. What no schema says, and the CAx Implementor Forum's recommended practice does, is that rep_1 is the child (component) and rep_2 the parent (assembly). This file follows both conventions: #236 has rep_1 = #40 (bracket) and rep_2 = #10 (assembly); #237 has item 1 in the bracket's representation and item 2 in the assembly's. Not every exporter does. A robust reader does not trust the order; it finds the frame that belongs to only one of the two representations — usually the target in the parent — and orients the transformation from that, because the child's origin frame is often shared by every representation in the file. When both frames are the identity, as for the bracket here, getting it backwards is invisible — which is how the bug survives.

8.5 Walking the tree

With the records understood, the procedure for reconstructing the assembly:

1. **Find the roots.** A root is a product definition that is nobody's related_product_definition. Here #5. A file may have several roots (a multi-body export, or several unrelated assemblies).
2. **Children of a definition** are the NAUOs whose relating_product_definition is it. For #5: #239, #311, #317.
3. **For each NAUO**, find the PRODUCT_DEFINITION_SHAPE that points at it, the CONTEXT_DEPENDENT_SHAPE_REPRESENTATION that points at *that*, and from there forward to the relationship, the transformation, and the two frames. If the chain is absent, try the mapped-item route below; if there is neither, readers place the child at identity, which is a convention of readers rather than a statement in the file.
4. **Geometry of a definition** is found through its own PRODUCT_DEFINITION_SHAPE and SHAPE_DEFINITION_REPRESENTATION, as in Chapter 7.
5. **World position of a child** = parent's world transform composed with the local one. Recurse into the child's own NAUOs.

Convert translations and geometry to consistent length units before composing transforms between contexts. Detect cycles along the active recursion path, but do not use a global visited set to suppress repeated definitions: that would drop legitimate occurrences and undercount a reused subassembly's children.

The **bill of materials** is steps 1, 2 and 5 without the geometry: walk the tree, group occurrences by related_product_definition, count. For the clamp: bracket × 1, pin × 2.

Two sharings in this file are worth noticing because they are typical and because they catch tools that copy records around. The origin frame #11 is an item of *all three* shape representations — #10, #40 and #248 — and is transform_item_1 in all three transformations. One record, six roles. And the three geometric contexts (#27, #230, #302) are textually identical but distinct records, one per product; each representation points at its own.

> ⚠ In the wild: The category that says 'part'
>
> This exporter writes 'part' for all three products, including the assembly (#32 points at #7, CLAMP-ASSY). That is not a defect: these are AP214 files, and AP214's list of category names has no 'assembly' in it. AP203 and AP242 have no such list, and require 'assembly' only on a product whose definition is declared an assembly definition; some exporters omit the category entirely. Either way the category is no substitute for walking the NEXT_ASSEMBLY_USAGE_OCCURRENCE records. The NAUOs are the truth: a definition with children is an assembly whatever its category says.

8.6 The other way: `MAPPED_ITEM`

Everything above is the *product-structure* way of placing components, and it is what the recommended practices call for. There is a second, older mechanism that some exporters use for assemblies and some for repeated geometry *within* a part, and a reader must know both.

```
#400 = MAPPED_ITEM('',#401,#19);
#401 = REPRESENTATION_MAP(#11,#248);
```

Referred to above: 67 records from `clamp.step`; the 3 it points to directly are shown. The complete file.

```
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);
#19 = AXIS2_PLACEMENT_3D('',#20,#21,#22);
#248 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#249),#302);
/* ... and 64 more: 15 DIRECTION, 10 CARTESIAN_POINT, 6 ORIENTED_EDGE, 5 AXIS2_PLACEMENT_3D, ... */
```

A `MAPPED_ITEM` is a representation item — it sits in the *parent's* `SHAPE_REPRESENTATION.items`, next to the axis placements — that says "the representation `#248`, whose origin is `#11`, placed at frame `#19`." The `REPRESENTATION_MAP` pairs a source representation with its origin; the mapped item pairs the map with a target. Same information as `#235`–`#237`, two records instead of three, and no `PRODUCT_DEFINITION_SHAPE` on the NAUO.

When an exporter uses mapped items for an assembly, the NAUOs are still written — the product structure is unchanged — but the positions are in the parent's representation items rather than hanging off the NAUOs. A reader that only knows the `CONTEXT_DEPENDENT_SHAPE_REPRESENTATION` route will find no placements and put every child at the origin. If a NAUO has no placement chain, look in the parent's shape representation for `MAPPED_ITEM`s whose map's source is the child's representation. That finds the placements; when several occurrences share one child representation, nothing in the file says which mapped item is which NAUO, and the pairing has to be left open.

Mapped items also appear where there is no assembly at all: a repeated sub-model, or a block of annotation geometry drawn once and placed a dozen times. What they cannot do is instance a *feature*: a mapped item places a whole representation, and the faces of a bolt-hole pattern belong to the solid's one closed shell. There the mapped item is simply an instancing mechanism, and the "child" representation is not a product's shape but an anonymous representation containing the repeated geometry.

8.7 Deeper trees and other variations

The clamp is two levels. A real assembly is many, and nothing changes: each intermediate node is a product definition with NAUOs pointing at it (as a child) and NAUOs pointing from it (as a parent), and its shape representation holds frames for its own children. World transforms compose down the tree.

Variations you will meet:

- **Several shape representations per part** — a B-rep and a tessellation, say. The `REPRESENTATION_RELATIONSHIP` in the placement chain names *which* representation of the child is being placed, and a part may need a placement per representation.
- **A NAUO with no placement chain at all.** Legal: product structure may be exchanged without positions. Readers put the child at identity, which is right for files from systems that flatten everything to the world frame first and a guess otherwise.
- **`ASSEMBLY_COMPONENT_USAGE` with quantity.** All three protocols allow a `QUANTIFIED_ASSEMBLY_COMPONENT_USAGE` — one relationship carrying a count instead of one NAUO per occurrence — for parts like fasteners where individual positions are not modelled. Rare in CAD exports.
- **Promissory and specified-higher usages.** `SPECIFIED_HIGHER_USAGE_OCCURRENCE` names an occurrence deeper in the tree from a level above, carrying both the upper usage and the next usage it derives from; `PROMISSORY_USAGE_OCCURRENCE` adds nothing to a NAUO and marks a usage that is intended rather than realised. They exist for configuration management and almost never appear in geometry exchange.

! In brief

The assembly tree is a set of `NEXT_ASSEMBLY_USAGE_OCCURRENCE` edges from parent definition to child definition; one edge is one occurrence, and a bill of materials is a count of edges per child. Position is separate: a `PRODUCT_DEFINITION_SHAPE` on the NAUO, a `CONTEXT_DEPENDENT_SHAPE_REPRESENTATION`, a three-way relationship record, an `ITEM_DEFINED_TRANSFORMATION` naming two frames, and the target frame in the parent's representation. The frame is the transform. `MAPPED_ITEM` is the alternative route, and a reader must handle both.

II Geometry

9 Boundary Representation from Scratch 48

10 Reading a Solid: The Block 51

11 Curves and Surfaces 56

12 Free-Form Geometry: B-Splines 65

13 Parameter Curves, and Meshes 69

9. Boundary Representation from Scratch

> *Before reading a solid, what a solid is. Why the file keeps "what touches what" and "where it is" in separate records, what orientation means and why it is stored three times over, and the arithmetic that checks a shell.*

The 150 records of geometry in the block file describe a rectangular box. They do it in a way that seems, at first, absurdly indirect: the box's eight corners appear as sixteen records, its twelve edges as sixty, and no record anywhere says "box" or gives its dimensions. To see why it is done this way, and why the same scheme scales to a turbine blade, you need the idea the whole of Part 42 is built on.

i Engineering background: Boundary representation

There are two broad ways to describe a solid object to a computer. **Constructive** methods describe the operations that made it: a box, minus a cylinder, plus a fillet. **Boundary** methods describe its skin: the set of surfaces that separate inside from outside. STEP, like every major CAD kernel, exchanges solids as boundary representations — *B-rep* — because the skin is what two systems can agree on. A system that has no "fillet" operation can still accept a torus patch.

A B-rep has two layers. **Geometry** is the set of mathematical shapes the skin is made of: this plane, that cylinder, this line, that circle. Planes, cylinders and lines are unbounded; circles, spheres and tori are not. **Topology** is the description of how finite pieces of them connect: this *face* is the part of that plane inside this *loop*; this *edge* is the part of that line between these two *vertices*; these six faces together form a closed *shell*. Geometry says where things are; topology says what touches what.

Keeping them separate is what makes the representation robust. Two faces that share an edge share *the same edge record*, so there is no question of whether their boundaries coincide to within tolerance — they are the same object. The plane a face lies on can be replaced by a slightly different plane without changing what the face is connected to. And two adjacent faces on the same cylinder can share one cylinder record. Every kernel does this; STEP simply writes the two layers out as two families of entities.

9.1 The topological ladder

Part 42 defines the topology as a strict hierarchy. Each level is bounded by the level below it.

Level	Entity	Bounded by
Solid	`MANIFOLD_SOLID_BREP`	one closed shell (plus optional voids)
Shell	`CLOSED_SHELL`, `OPEN_SHELL`	a set of faces
Face	`ADVANCED_FACE` (a `FACE_SURFACE`)	one or more loops (`FACE_BOUND`)
Loop	`EDGE_LOOP`, `VERTEX_LOOP`	a sequence of oriented edges, or (`VERTEX_LOOP`) one vertex
Edge	`EDGE_CURVE`, via `ORIENTED_EDGE`	two vertices
Vertex	`VERTEX_POINT`	—

Each topological entity from face down carries one piece of geometry: a face has a *surface*, an edge has a *curve*, a vertex has a *point*. Shells, loops and solids have no geometry of their own; they are pure connectivity.

The block: one solid, one shell, six faces, six loops (one per face), twelve edges, eight vertices. Every face is bounded by exactly one loop because a box has no holes. The bracket of Section 11.5 has top and

bottom faces with two loops each — an outer rectangle and an inner circle. That is not the only structural difference a hole makes: it also brings a cylindrical face whose seam edge is traversed twice within one loop, and two circular edges that start and end at the same vertex. The plain count $V - E + F$ still comes to 2, because the two inner loops cancel the through-hole; it is the corrected count below that reads 0.

9.2 Sharing

The first thing to understand about the connectivity is that it is achieved by **reference to the same record**, not by coincidence of coordinates.

The block has twelve edges and each edge belongs to two faces. There are therefore twenty-four *appearances* of edges in loops, but only twelve `EDGE_CURVE` records. Each appearance is an `ORIENTED_EDGE` — a thin record that points at an `EDGE_CURVE` and adds a direction. Two faces that meet along an edge each have an `ORIENTED_EDGE` pointing at the same `EDGE_CURVE`, one saying "forwards" and the other "backwards."

Likewise, each vertex of the block is the meeting point of three edges, and all three `EDGE_CURVE`s point at the same `VERTEX_POINT`. There are eight vertex records, not twenty-four.

This is what makes a B-rep *watertight* by construction. The question "do these two faces actually meet?" is answered by the graph, not by comparing numbers. Geometry that does not quite line up — a plane that misses its edge's line by 10^{-9} — does not break connectivity; it is what the uncertainty of Chapter 6 exists to tolerate.

9.3 Orientation

The second thing to understand is that nearly everything in the topology has a direction, and the directions have to agree.

A **surface** has a normal: the plane through the origin with axis (0, 0, 1) has "up" as its positive side. A **face** must know which side of its surface is *outside the solid*. The face record carries a Boolean, `same_sense`: `.T.` if the surface normal already points outward, `.F.` if the outward normal is the reverse.

A **curve** has a direction: a line from a point along a vector, a circle counter-clockwise about its axis. An **edge** is a piece of a curve between two vertices, and `EDGE_CURVE` carries its own `same_sense`: whether the edge runs from its start vertex to its end vertex in the curve's direction or against it.

A **loop** is a cycle of edges, and the cycle has a direction: walking it with the outward normal as "up," the material of the face is on your left. For an outer bound that means counter-clockwise seen from outside; an inner bound, around a hole, runs the other way. Each `ORIENTED_EDGE` in the loop says whether the edge is to be traversed forwards or backwards to make that true. And each `FACE_BOUND` carries one more Boolean that can reverse the whole loop.

Three layers of direction on an edge alone — the curve's, the edge's, the oriented edge's — plus one on the face bound and one on the face. The redundancy is deliberate: it lets records be shared. One `EDGE_CURVE` serves two faces because each face wraps it in its own `ORIENTED_EDGE`; one `PLANE` can serve two coplanar faces whose normals oppose — a pocket's floor and an undercut's ceiling at the same height — because each face has its own `same_sense`. The cost is that a reader must compose the flags correctly, and a writer that gets one wrong produces a solid whose faces point inwards — which renders as black, meshes inside-out, and reports a negative volume. This is the most common defect in hand-written or badly converted STEP files.

9.4 Closed and manifold

A `CLOSED_SHELL` claims that its faces enclose a volume with no gaps. **Manifold** means, informally, that the surface is everywhere locally like a sheet: every edge is traversed exactly twice by the loops of the shell, once in each direction, and every vertex is surrounded by a single fan of faces. Usually the two traversals belong to two different faces; on a seam they belong to the same face twice. A `MANIFOLD_SOLID_BREP` claims both.

Neither claim is checked by the file format. But there is arithmetic that will catch a badly built one. For a solid of S shells with R inner loops and H through-holes:

$$V - E + F - R = 2(S - H) \tag{9.1}$$

This is the Euler–Poincaré characteristic, and it is the first thing a validator computes: a shell that fails it is missing an edge, has a duplicated vertex, or was never closed. Passing it proves less than failing it: a count

that comes out right says nothing about orientation, self-intersection or whether the geometry actually meets along the shared edges. The familiar short form, $V - E + F = 2$, is what it becomes for a single shell with no inner loops and no through-holes — the block: eight vertices, twelve edges, six faces, $8 - 12 + 6 = 2$. Both of those conditions are needed, and neither alone is enough. A box with a blind pocket has no through-hole, but the pocket's mouth is an inner loop, and its plain count is 3. The torus has one loop on its single face, but it has a through-hole, and its plain count is 0. The bracket, with two inner loops and one through-hole, comes back to 2 by cancellation — which is why the short form is worth running only on a shell you already know is a plain closed skin. Run the general form and you never have to ask.

9.5 What the file does not say

A B-rep in STEP is *evaluated* geometry. There is no record of how the box was made, no parameters, no features, no dimensions in the engineering sense. (AP242 does define records for construction history and for features such as holes and slots; no mainstream exporter writes them.) The 60, 20 and 10 of the block appear only as point coordinates — in vertices, in line base points, in plane origins — never as a length. If the designer had constrained the block's length to twice its width, that fact is gone. This is a limitation and a strength: it is why STEP files from any CAD system can be read by any other, and why they cannot be edited the way the original model could. Design intent lives in the CAD system's native format, or — for tolerances and dimensions specifically — in the PMI of Chapter 16.

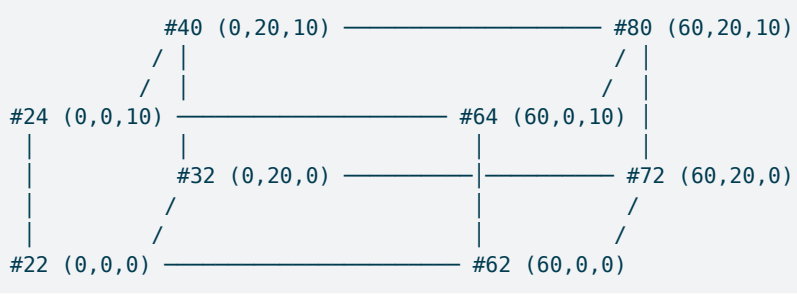
! In brief

A B-rep is a skin described in two layers: geometry (surfaces, curves, points) and topology (finite faces, edges, vertices connected to each other). Connectivity is by shared records, not shared coordinates. Everything has a direction, and the directions must compose to an outward normal with the face's material on the left of every loop. The general count is $V - E + F - R = 2(S - H)$; the familiar $V - E + F = 2$ is its special case, for one shell with no inner loops and no through-holes.

10. Reading a Solid: The Block

> *One hundred and fifty records, read from the bottom of the ladder up: points and vertices, lines and edges, loops, faces, shell, solid. Then the orientation flags, composed by hand, and the count that checks the shell.*

The block's eight corners, labelled with their vertex records:



X runs to the right (0 to 60), Y into the page (0 to 20), Z up (0 to 10). The solid is `#15`, its shell is `#16`, and the shell's six faces are `#17`, `#57`, `#97`, `#119`, `#141` and `#153`. The chapter reads it from the bottom up.

10.1 Vertices

```
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
```

A `VERTEX_POINT` is a topological vertex with one attribute beyond its name: `vertex_geometry`, a point. The `CARTESIAN_POINT` carries a list of coordinates, here three (a 2D point would have two). Two records per corner; sixteen for the block. The other seven vertices are `#24`, `#32`, `#40`, `#62`, `#64`, `#72` and `#80`, each with its own point.

Nothing in a vertex record says which edges meet there. That is in the edges.

10.2 Edges

```
#21 = EDGE_CURVE('',#22,#24,#26,.T.);
#26 = LINE('',#27,#28);
#27 = CARTESIAN_POINT('',(0.,0.,0.));
#28 = VECTOR('',#29,1.);
#29 = DIRECTION('',(0.,0.,1.));
```

Referred to above: 4 records from `block.step`. The complete file.

```
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(0.,0.,10.));
```

`EDGE_CURVE` has four attributes beyond its name: `edge_start` and `edge_end` (two vertices), `edge_geometry` (a curve), and `same_sense` (whether the edge runs along the curve's direction). `#21` runs from `#22` at the origin to `#24` at (0, 0, 10), along the line `#26`, in the line's own direction.

The line is a point and a vector. `LINE('',#27,#28)`: through `#27`, in direction `#28`. The `VECTOR` is a `DIRECTION` and a magnitude — `(0,0,1)` scaled by `1.` — which is more structure than a line needs and exactly what the schema requires. Note that `#27` duplicates the coordinates of `#23`: the exporter has written the line's base point as a separate record rather than reusing the vertex's point. It could have shared; it chose not to. Real files are full of such duplicated points, and a reader that de-duplicates by coordinate will

find the 27 `CARTESIAN_POINT`s in this file collapse to just 8 distinct locations — the eight corners, which is where every line base and every plane origin sits too.

The twelve edges of the block, from the file:

Edge	From	To	Direction	Where
#21	#22	#24	+Z	x = 0, y = 0
#31	#22	#32	+Y	x = 0, z = 0
#39	#32	#40	+Z	x = 0, y = 20
#47	#24	#40	+Y	x = 0, z = 10
#61	#62	#64	+Z	x = 60, y = 0
#71	#62	#72	+Y	x = 60, z = 0
#79	#72	#80	+Z	x = 60, y = 20
#87	#64	#80	+Y	x = 60, z = 10
#101	#22	#62	+X	y = 0, z = 0
#108	#24	#64	+X	y = 0, z = 10
#123	#32	#72	+X	y = 20, z = 0
#130	#40	#80	+X	y = 20, z = 10

Every edge's curve runs in a positive axis direction and every `same_sense` is `.T.`, because the exporter chose its lines to match its edges. Nothing requires this; an edge whose `same_sense` is `.F.` runs from its start vertex to its end vertex *against* the curve, and you will see plenty.

Each `EDGE_CURVE` costs five records: itself, a line, a point, a vector and a direction. Sixty records for twelve edges.

10.3 Loops

```
#19 = EDGE_LOOP('',(#20,#30,#38,#46));
#20 = ORIENTED_EDGE('',*,*,#21,.F.);
#30 = ORIENTED_EDGE('',*,*,#31,.T.);
#38 = ORIENTED_EDGE('',*,*,#39,.T.);
#46 = ORIENTED_EDGE('',*,*,#47,.F.);
```

Referred to above: 28 records from `block.step`. The complete file.

```
#21 = EDGE_CURVE('',#22,#24,#26,.T.);
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(0.,0.,10.));
#26 = LINE('',#27,#28);
#27 = CARTESIAN_POINT('',(0.,0.,0.));
#28 = VECTOR('',#29,1.);
#29 = DIRECTION('',(0.,0.,1.));
#31 = EDGE_CURVE('',#22,#32,#34,.T.);
#32 = VERTEX_POINT('',#33);
#33 = CARTESIAN_POINT('',(0.,20.,0.));
#34 = LINE('',#35,#36);
#35 = CARTESIAN_POINT('',(0.,0.,0.));
#36 = VECTOR('',#37,1.);
#37 = DIRECTION('',(0.,1.,0.));
#39 = EDGE_CURVE('',#32,#40,#42,.T.);
#40 = VERTEX_POINT('',#41);
#41 = CARTESIAN_POINT('',(0.,20.,10.));
#42 = LINE('',#43,#44);
#43 = CARTESIAN_POINT('',(0.,20.,0.));
#44 = VECTOR('',#45,1.);
#45 = DIRECTION('',(0.,0.,1.));
```

```
#47 = EDGE_CURVE('',#24,#40,#48,.T.);
#48 = LINE('',#49,#50);
#49 = CARTESIAN_POINT('',(0.,0.,10.));
#50 = VECTOR('',#51,1.);
#51 = DIRECTION('',(0.,1.,0.));
```

An `EDGE_LOOP` is an ordered list of oriented edges forming a closed cycle. Each `ORIENTED_EDGE` points at an `EDGE_CURVE` and adds an `orientation`: `.T.` to traverse it from start to end, `.F.` from end to start. The two `*` are derived attributes — the oriented edge's own start and end vertices, which follow from the edge and the flag — and are written as `*` because the schema says to compute them.

Follow loop `#19`:

- `#20`: edge `#21` reversed — from `#24` (0, 0, 10) to `#22` (0, 0, 0).
- `#30`: edge `#31` forwards — from `#22` to `#32` (0, 20, 0).
- `#38`: edge `#39` forwards — from `#32` to `#40` (0, 20, 10).
- `#46`: edge `#47` reversed — from `#40` to `#24`.

Each edge ends where the next begins, and the last ends where the first began. Four corners of the face at x = 0, visited in order. A well-formed loop always chains like this, and checking that it does is the second thing a validator does after counting.

The block has six loops and twenty-four oriented edges. Each of the twelve edges appears in exactly two loops. Look at `#21`: it is in loop `#19` via `#20` (orientation `.F.`) and in loop `#99` via `#106` (orientation `.T.`). Every shared edge appears once forwards and once backwards — almost. The full rule needs the next level.

10.4 Faces

```
#17 = ADVANCED_FACE('',(#18),#52,.F.);
#18 = FACE_BOUND('',#19,.F.);
#52 = PLANE('',#53);
#53 = AXIS2_PLACEMENT_3D('',#54,#55,#56);
#54 = CARTESIAN_POINT('',(0.,0.,0.));
#55 = DIRECTION('',(1.,0.,0.));
#56 = DIRECTION('',(0.,0.,1.));
```

Referred to above: 33 records from `block.step`. The complete file.

```
#19 = EDGE_LOOP('',(#20,#30,#38,#46));
#20 = ORIENTED_EDGE('',*,*,#21,.F.);
#21 = EDGE_CURVE('',#22,#24,#26,.T.);
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(0.,0.,10.));
#26 = LINE('',#27,#28);
#27 = CARTESIAN_POINT('',(0.,0.,0.));
#28 = VECTOR('',#29,1.);
#29 = DIRECTION('',(0.,0.,1.));
#30 = ORIENTED_EDGE('',*,*,#31,.T.);
#31 = EDGE_CURVE('',#22,#32,#34,.T.);
#32 = VERTEX_POINT('',#33);
#33 = CARTESIAN_POINT('',(0.,20.,0.));
#34 = LINE('',#35,#36);
#35 = CARTESIAN_POINT('',(0.,0.,0.));
#36 = VECTOR('',#37,1.);
#37 = DIRECTION('',(0.,1.,0.));
#38 = ORIENTED_EDGE('',*,*,#39,.T.);
#39 = EDGE_CURVE('',#32,#40,#42,.T.);
#40 = VERTEX_POINT('',#41);
#41 = CARTESIAN_POINT('',(0.,20.,10.));
#42 = LINE('',#43,#44);
#43 = CARTESIAN_POINT('',(0.,20.,0.));
#44 = VECTOR('',#45,1.);
#45 = DIRECTION('',(0.,0.,1.));
#46 = ORIENTED_EDGE('',*,*,#47,.F.);
#47 = EDGE_CURVE('',#24,#40,#48,.T.);
#48 = LINE('',#49,#50);
#49 = CARTESIAN_POINT('',(0.,0.,10.));
```

```
#50 = VECTOR('',#51,1.);
#51 = DIRECTION('',(0.,1.,0.));
```

An ADVANCED_FACE has, as Chapter 4 laid out, bounds (a set of face bounds), face_geometry (a surface) and same_sense. #17 has one bound, lies on plane #52, and its same_sense is .F..

The **bound** wraps the loop and adds another orientation flag: FACE_BOUND('',#19,.F.) is loop #19, reversed. A face with a hole has two bounds; the standard provides FACE_OUTER_BOUND to mark the outer one, and this exporter does not use it — both bounds are plain FACE_BOUND, and which is outer must be worked out from the geometry. Chapter 11 has the example.

The **plane** is positioned by an axis placement: origin #54 at (0, 0, 0), normal #55 along +X, and a reference direction #56 along +Z that fixes the plane's own in-plane X axis. The plane's positive side is +X.

But this face is the one at x = 0 — the left end of the block — and its outside is toward −X. Hence same_sense = .F.: the face's outward normal is the *reverse* of the plane's normal. The face on the opposite end, #57, lies on plane #92 whose normal is also +X, positioned at (60, 0, 0); there the plane normal is already outward and same_sense is .T..

The six faces:

Face	Plane origin	Plane normal	same_sense	Outward normal	Which side
#17	(0, 0, 0)	+X	.F.	−X	left end
#57	(60, 0, 0)	+X	.T.	+X	right end
#97	(0, 0, 0)	+Y	.F.	−Y	front
#119	(0, 20, 0)	+Y	.T.	+Y	back
#141	(0, 0, 0)	+Z	.F.	−Z	bottom
#153	(0, 0, 10)	+Z	.T.	+Z	top

Each face costs eight records: the face, its bound, its loop, the plane, the plane's placement, and the placement's point and two directions. Forty-eight for six faces.

10.5 Composing the orientation

Now the flags together. The rule is: after applying the edge orientations *and* the bound orientation, an outer loop runs counter-clockwise when viewed from outside — with the outward normal pointing at the viewer. An inner loop, around a hole, runs the other way.

Take #17 again. The plane's normal is +X and same_sense is .F., so the outward normal is −X. Loop #19, after its edge flags, visits (0, 0, 10) ⟶ (0, 0, 0) ⟶ (0, 20, 0) ⟶ (0, 20, 10). In the (y, z) plane that sequence is (0, 10) ⟶ (0, 0) ⟶ (20, 0) ⟶ (20, 10), which is counter-clockwise about +X — the wrong way for a face whose outside is −X. The bound #18 has orientation .F., which reverses the loop, making it clockwise about +X, which is counter-clockwise about −X. Consistent.

Now #57, the right end. Its loop #59 visits the same (y, z) sequence, counter-clockwise about +X. The face's outward normal is +X. The bound #58 has orientation .T.: no reversal needed. Consistent again.

The exporter has a habit visible in every face: FACE_BOUND.orientation equals ADVANCED_FACE.same_sense. That is a property of how OCCT builds its loops, not a rule, and other exporters will hand you a bound of .T. on a face of .F. with edge flags arranged to compensate.

The shared-edge rule, fully: for each EDGE_CURVE shared by two faces, the *effective* orientation in each face — the ORIENTED_EDGE flag combined with the FACE_BOUND flag — must be opposite. Edge #21 in face #17: oriented edge .F., bound .F., effective forwards. In face #97: oriented edge #106 is .T., bound #98 is .F., effective backwards. Opposite. This is the condition that makes the two faces' normals agree about which side is out; break it on one edge and the shell has a crease where inside becomes outside.

10.6 Shell and solid

```
#16 = CLOSED_SHELL('',(#17,#57,#97,#119,#141,#153));
#15 = MANIFOLD_SOLID_BREP('',#16);
```

Referred to above: 148 records from `block.step`; the 6 it points to directly are shown. The complete file.

```
#17 = ADVANCED_FACE('',(#18),#52,.F.);
#57 = ADVANCED_FACE('',(#58),#92,.T.);
#97 = ADVANCED_FACE('',(#98),#114,.F.);
#119 = ADVANCED_FACE('',(#120),#136,.T.);
#141 = ADVANCED_FACE('',(#142),#148,.F.);
#153 = ADVANCED_FACE('',(#154),#160,.T.);
/* ... and 142 more: 26 CARTESIAN_POINT, 24 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
```

The shell is a set of faces, unordered. The solid is a shell. `MANIFOLD_SOLID_BREP` has one attribute, `outer`; a subtype, `BREP_WITH_VOIDS`, adds a set of inner shells for cavities. There is nothing else at this level: no bounding box, no volume, no centroid. Those, if present, are properties attached from outside, in Chapter 15.

10.7 The count

Records	Kind	Count
`VERTEX_POINT` + `CARTESIAN_POINT`	8 vertices × 2	16
`EDGE_CURVE` + `LINE` + point + `VECTOR` + `DIRECTION`	12 edges × 5	60
`ORIENTED_EDGE`	12 edges × 2 faces	24
face + bound + loop + `PLANE` + placement + point + 2 directions	6 faces × 8	48
`CLOSED_SHELL` + `MANIFOLD_SOLID_BREP`		2
Total		**150**

And the Euler check: V − E + F = 8 − 12 + 6 = 2. The shell is closed and has no through-holes, and the count of records is exactly what the structure predicts. When a file's record counts do *not* match a pattern like this — 13 edges for a box, say — an edge or a face has been split, which is legal and common after Boolean operations; it is the connectivity, not the count, that decides whether the shell is sound.

⚠ In the wild: What varies between exporters

The structure above is universal; the habits are not. Other writers use `FACE_OUTER_BOUND` for the outer loop (the standard's intent), share `CARTESIAN_POINT`s between a vertex and the lines that start there, write `same_sense = .F.` on edges rather than choosing curve directions to avoid it, and put the plane's origin at a face corner rather than the part origin. Some CATIA exports write every face's `same_sense` as `.T.` and flip the plane's normal instead. All of these describe the same box; a reader has to compose the flags and not pattern-match the habits.

! In brief

Vertex = point. Edge = two vertices + curve + sense. Oriented edge = edge + direction. Loop = cycle of oriented edges. Bound = loop + orientation. Face = bounds + surface + sense. Shell = faces; solid = shell. Outward normal is the surface normal flipped by `same_sense`; an outer loop, after edge flags and bound flag, runs counter-clockwise around it and an inner loop clockwise; each edge is traversed twice, once in each direction. Eight, twelve, six: V − E + F = 2.

11. Curves and Surfaces

> *The pin: three faces, three edges, two vertices, and a cylinder that closes on itself. Then a hole, a sphere with no edges at all, a torus with two seams, and a cone whose straight edge is written as a spline. The analytic geometry of Part 42, read from real records.*

The block used two kinds of geometry: planes and lines. Most real parts are mostly planes and lines, with cylinders for every hole and every rounded edge, and occasionally a cone, sphere or torus. Everything else is a B-spline, which has the next chapter to itself. This chapter reads the analytic shapes, and the topology they force on a solid.

11.1 The pin

Six millimetres in diameter, thirty long, standing on the origin (Figure 11.1).



Figure 11.1: `pin.step`: three edges, which are the two circles and the seam, and two vertices. The grey lines are the cylinder's outline, which is not an edge and is not in the file.

Here is its entire solid, 53 records, with the placements' points and directions elided where they are obvious:

```
#15 = MANIFOLD_SOLID_BREP('',#16);
#16 = CLOSED_SHELL('',(#17,#50,#59));
#17 = ADVANCED_FACE('',(#18),#45,.T.);            /* the cylindrical side */
#18 = FACE_BOUND('',#19,.F.);
#19 = EDGE_LOOP('',(#20,#30,#37,#38));
#20 = ORIENTED_EDGE('',*,*,#21,.T.);
#21 = EDGE_CURVE('',#22,#24,#26,.T.);            /* the seam: (3,0,0) -> (3,0,30) */
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(3.,0.,0.));
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(3.,0.,30.));
#26 = LINE('',#27,#28);                           /* along +Z */
```

```
#30 = ORIENTED_EDGE('',*,*,#31,.T.);
#31 = EDGE_CURVE('',#24,#24,#32,.T.);            /* top circle, from #24 to #24 */
#32 = CIRCLE('',#33,3.);
#33 = AXIS2_PLACEMENT_3D('',#34,#35,#36);        /* centre (0,0,30), axis +Z */
#37 = ORIENTED_EDGE('',*,*,#21,.F.);             /* the seam again, reversed */
#38 = ORIENTED_EDGE('',*,*,#39,.F.);
#39 = EDGE_CURVE('',#22,#22,#40,.T.);            /* bottom circle */
#40 = CIRCLE('',#41,3.);
#41 = AXIS2_PLACEMENT_3D('',#42,#43,#44);        /* centre (0,0,0), axis +Z */
#45 = CYLINDRICAL_SURFACE('',#46,3.);
#46 = AXIS2_PLACEMENT_3D('',#47,#48,#49);
#47 = CARTESIAN_POINT('',(0.,0.,0.));
#48 = DIRECTION('',(-0.,-0.,-1.));
#49 = DIRECTION('',(1.,0.,0.));
#50 = ADVANCED_FACE('',(#51),#54,.F.);           /* bottom cap */
#51 = FACE_BOUND('',#52,.F.);
#52 = EDGE_LOOP('',(#53));
#53 = ORIENTED_EDGE('',*,*,#39,.T.);
#54 = PLANE('',#55);
#59 = ADVANCED_FACE('',(#60),#63,.T.);           /* top cap */
#60 = FACE_BOUND('',#61,.T.);
#61 = EDGE_LOOP('',(#62));
#62 = ORIENTED_EDGE('',*,*,#31,.T.);
#63 = PLANE('',#64);
```

Referred to above: 17 records from `pin.step`. The complete file.

```
#27 = CARTESIAN_POINT('',(3.,0.,0.));
#28 = VECTOR('',#29,1.);
#29 = DIRECTION('',(0.,0.,1.));
#34 = CARTESIAN_POINT('',(0.,0.,30.));
#35 = DIRECTION('',(0.,0.,1.));
#36 = DIRECTION('',(1.,0.,0.));
#42 = CARTESIAN_POINT('',(0.,0.,0.));
#43 = DIRECTION('',(0.,0.,1.));
#44 = DIRECTION('',(1.,0.,0.));
#55 = AXIS2_PLACEMENT_3D('',#56,#57,#58);
#56 = CARTESIAN_POINT('',(-1.709300165674E-16,-8.51133553645E-18,0.));
#57 = DIRECTION('',(0.,0.,1.));
#58 = DIRECTION('',(1.,0.,-0.));
#64 = AXIS2_PLACEMENT_3D('',#65,#66,#67);
#65 = CARTESIAN_POINT('',(-1.709300165674E-16,-8.51133553645E-18,30.));
#66 = DIRECTION('',(0.,0.,1.));
#67 = DIRECTION('',(1.,0.,-0.));
```

Three faces: a cylinder and two planar caps. Three edges: two circles and one straight line. Two vertices, both on the line. The Euler check: V − E + F = 2 − 3 + 3 = 2. Consistent with a closed shell and no holes.

The count looks wrong for a moment — how does a cylinder have only three edges? — and the answer is the most important idea in this chapter.

11.2 Closed edges and the seam

The top circle `#31` is `EDGE_CURVE('',#24,#24,#32,.T.)`: its start vertex and its end vertex are the *same* record. It is a **closed edge** — a loop of curve with a single vertex on it, where it begins and ends. The bottom circle `#39` is the same, on `#22`. A circular edge in STEP is nearly always written this way, with one vertex, rather than split into two semicircles.

The cylindrical face `#17` is bounded by one loop, `#19`, with four oriented edges: `#21` forwards, `#31`, `#21` **backwards**, `#39`. The straight edge `#21` appears twice in the same loop, once in each direction. It is a **seam**: a line up the side of the cylinder where the surface, which is periodic, has been cut so that the face can be bounded by a single loop.

Think of the cylinder's side as a rectangle of paper rolled into a tube. Its boundary is the two circles — but also the line where the two edges of the paper meet. Walking around the boundary of the flattened rectangle, you go up one side of the seam, around the top circle, *down the other side of the seam*, and

around the bottom. In 3D both sides of the seam are the same line, so the loop traverses `#21` twice. The loop is still a single closed cycle; it happens to visit one edge in both directions.

Seams are an artefact of representing a periodic surface with a face that has a boundary, and every cylindrical, conical and toroidal face in this book has at least one. A sphere can avoid them, as the one later in this chapter does, by being bounded at a pole instead. They are legal, universal, and the thing most likely to confuse code that assumes "an edge appears at most once per loop." The cylindrical face in this file is *not* two faces; it is one face with a seam.

11.3 The cylinder record

```
#45 = CYLINDRICAL_SURFACE('',#46,3.);
#46 = AXIS2_PLACEMENT_3D('',#47,#48,#49);
#47 = CARTESIAN_POINT('',(0.,0.,0.));
#48 = DIRECTION('',(-0.,-0.,-1.));
#49 = DIRECTION('',(1.,0.,0.));
```

`CYLINDRICAL_SURFACE` has a `position` and a `radius`. The position's `location` is a point on the cylinder's axis; its `axis` direction is the axis of the cylinder; its `ref_direction` fixes where the surface's angular parameter starts. Radius 3. The surface is infinite in both directions along the axis; the face `#17` is the portion of it between the two circles.

Note the axis: `(-0.,-0.,-1.)`. This cylinder's axis points *down*. The pin's circles have axes pointing up. Nothing is wrong: the exporter's kernel built the cylinder one way and its edges another, and the face's `same_sense` (`.T.`) and the bound's orientation (`.F.`) compose to make the outward normal correct. The surface normal of a cylinder points *away from* the axis regardless of which way the axis runs; the axis direction affects only the direction of the angular parameter, and so the sense in which loops must run.

11.4 Circles

```
#32 = CIRCLE('',#33,3.);
#33 = AXIS2_PLACEMENT_3D('',#34,#35,#36);
#34 = CARTESIAN_POINT('',(0.,0.,30.));
#35 = DIRECTION('',(0.,0.,1.));
#36 = DIRECTION('',(1.,0.,0.));
```

A `CIRCLE` is a `position` and a `radius`. The circle lies in the plane through the position's `location`, perpendicular to its `axis`; it starts at the `ref_direction` and runs counter-clockwise about the axis. So `#32` is a circle of radius 3 centred at (0, 0, 30), in the plane z = 30, starting at (3, 0, 30) — which is exactly where vertex `#24` sits — and running anticlockwise seen from above. An edge that is the *whole* circle starts and ends at that vertex. An edge that is *part* of a circle — an arc — has two different vertices and the arc is the piece of the circle from one to the other in the circle's direction (or against it, if `same_sense` is `.F.`).

`ELLIPSE` is the same with two radii. `HYPERBOLA` and `PARABOLA` exist and are almost never seen.

11.5 The hole in the bracket

The bracket is the block with a 6 mm hole down through its centre (Figure 11.2).

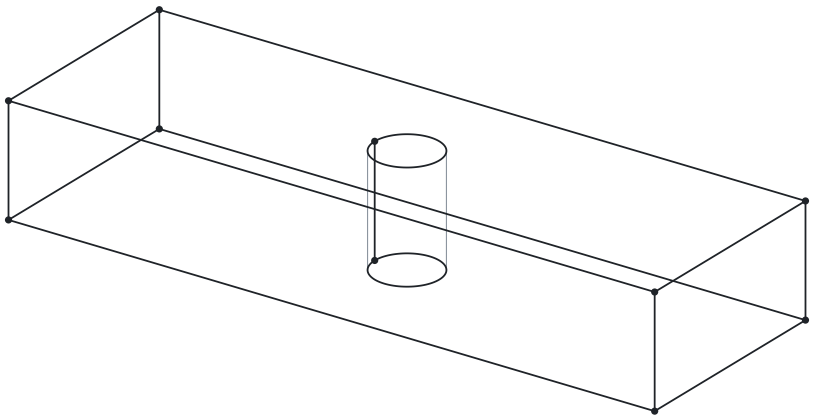


Figure 11.2: `bracket.step`: the block's twelve edges, and the hole's two circles and seam.

Two faces change and one is added. The top face:

```
#88 = ADVANCED_FACE('',(#89,#107),#118,.T.);
#89 = FACE_BOUND('',#90,.T.);                       /* outer rectangle */
#90 = EDGE_LOOP('',(#91,#92,#93,#101));
#107 = FACE_BOUND('',#108,.T.);                     /* inner circle */
#108 = EDGE_LOOP('',(#109));
#109 = ORIENTED_EDGE('',*,*,#110,.T.);
#110 = EDGE_CURVE('',#111,#111,#113,.T.);
#111 = VERTEX_POINT('',#112);
#112 = CARTESIAN_POINT('',(27.,10.,10.));
#113 = CIRCLE('',#114,3.);
#114 = AXIS2_PLACEMENT_3D('',#115,#116,#117);
#115 = CARTESIAN_POINT('',(30.,10.,10.));
#116 = DIRECTION('',(0.,0.,-1.));
#117 = DIRECTION('',(-1.,0.,0.));
```

Referred to above: 37 records from `bracket.step`. The complete file.

```
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(0.,0.,10.));
#40 = VERTEX_POINT('',#41);
#41 = CARTESIAN_POINT('',(0.,20.,10.));
#47 = EDGE_CURVE('',#24,#40,#48,.T.);
#48 = LINE('',#49,#50);
#49 = CARTESIAN_POINT('',(0.,0.,10.));
#50 = VECTOR('',#51,1.);
#51 = DIRECTION('',(0.,1.,0.));
#70 = EDGE_CURVE('',#24,#71,#73,.T.);
#71 = VERTEX_POINT('',#72);
#72 = CARTESIAN_POINT('',(60.,0.,10.));
#73 = LINE('',#74,#75);
#74 = CARTESIAN_POINT('',(0.,0.,10.));
#75 = VECTOR('',#76,1.);
#76 = DIRECTION('',(1.,0.,0.));
#91 = ORIENTED_EDGE('',*,*,#47,.F.);
#92 = ORIENTED_EDGE('',*,*,#70,.T.);
#93 = ORIENTED_EDGE('',*,*,#94,.T.);
#94 = EDGE_CURVE('',#71,#95,#97,.T.);
#95 = VERTEX_POINT('',#96);
#96 = CARTESIAN_POINT('',(60.,20.,10.));
#97 = LINE('',#98,#99);
#98 = CARTESIAN_POINT('',(60.,0.,10.));
#99 = VECTOR('',#100,1.);
#100 = DIRECTION('',(0.,1.,0.));
#101 = ORIENTED_EDGE('',*,*,#102,.F.);
#102 = EDGE_CURVE('',#40,#95,#103,.T.);
#103 = LINE('',#104,#105);
```

```
#104 = CARTESIAN_POINT('',(0.,20.,10.));
#105 = VECTOR('',#106,1.);
#106 = DIRECTION('',(1.,0.,0.));
#118 = PLANE('',#119);
#119 = AXIS2_PLACEMENT_3D('',#120,#121,#122);
#120 = CARTESIAN_POINT('',(0.,0.,10.));
#121 = DIRECTION('',(0.,0.,1.));
#122 = DIRECTION('',(1.,0.,0.));
```

Two bounds. `#89` is the rectangle, as in the block. `#107` is a loop of one closed edge — the circle of the hole, radius 3, centred at (30, 10, 10). A face with a hole is a face with two loops; the inner loop runs the *opposite* way to the outer one (clockwise seen from outside), so that the material is still on the left as you walk it. Nothing on the record says which bound is the outer one. The standard offers `FACE_OUTER_BOUND` for that purpose, this exporter writes plain `FACE_BOUND` for both, and a reader must decide from the geometry — typically, the bound whose loop encloses the others in the surface's parameter space.

The hole's wall:

```
#187 = ADVANCED_FACE('',(#188),#199,.F.);
#188 = FACE_BOUND('',#189,.F.);
#189 = EDGE_LOOP('',(#190,#191,#197,#198));
#190 = ORIENTED_EDGE('',*,*,#162,.F.);            /* bottom circle of the hole */
#191 = ORIENTED_EDGE('',*,*,#192,.F.);            /* seam, down */
#192 = EDGE_CURVE('',#111,#163,#193,.T.);
#197 = ORIENTED_EDGE('',*,*,#110,.T.);            /* top circle of the hole */
#198 = ORIENTED_EDGE('',*,*,#192,.T.);            /* seam, up */
#199 = CYLINDRICAL_SURFACE('',#200,3.);
```

Referred to above: 24 records from `bracket.step`. The complete file.

```
#110 = EDGE_CURVE('',#111,#111,#113,.T.);
#111 = VERTEX_POINT('',#112);
#112 = CARTESIAN_POINT('',(27.,10.,10.));
#113 = CIRCLE('',#114,3.);
#114 = AXIS2_PLACEMENT_3D('',#115,#116,#117);
#115 = CARTESIAN_POINT('',(30.,10.,10.));
#116 = DIRECTION('',(0.,0.,-1.));
#117 = DIRECTION('',(-1.,0.,0.));
#162 = EDGE_CURVE('',#163,#163,#165,.T.);
#163 = VERTEX_POINT('',#164);
#164 = CARTESIAN_POINT('',(27.,10.,0.));
#165 = CIRCLE('',#166,3.);
#166 = AXIS2_PLACEMENT_3D('',#167,#168,#169);
#167 = CARTESIAN_POINT('',(30.,10.,0.));
#168 = DIRECTION('',(0.,0.,-1.));
#169 = DIRECTION('',(-1.,0.,0.));
#193 = LINE('',#194,#195);
#194 = CARTESIAN_POINT('',(27.,10.,10.));
#195 = VECTOR('',#196,1.);
#196 = DIRECTION('',(0.,0.,-1.));
#200 = AXIS2_PLACEMENT_3D('',#201,#202,#203);
#201 = CARTESIAN_POINT('',(30.,10.,10.));
#202 = DIRECTION('',(0.,0.,-1.));
#203 = DIRECTION('',(-1.,0.,0.));
```

The same cylinder-with-seam structure as the pin, turned inside out: this face's `same_sense` is `.F.`, because the surface normal of a cylinder points away from its axis and the *outside* of the bracket, at the hole wall, is *toward* the axis. The top circle `#110` is shared: it bounds the top face's inner loop and the hole wall's loop, traversed forwards in one and — after the bound flags — backwards in the other, as every shared edge must be.

The bracket has 8 + 2 = 10 vertices (the hole adds one on each circle), 12 + 2 + 1 = 15 edges (two circles and a seam), and 6 + 1 = 7 faces. V − E + F = 10 − 15 + 7 = 2. That is *not* the 0 you might expect for a solid with a through-hole, and the reason is instructive. The simple formula assumes every face is a disc — one outer loop, nothing inside it. The bracket's top and bottom faces are *rings*: each has an inner loop. The general form of the check subtracts them:

$$V - E + F - R = 2(S - H) \tag{11.1}$$

where R is the number of inner loops, S the number of shells and H the number of through-holes (the genus). For the bracket: $10 - 15 + 7 - 2 = 0 = 2 \times (1 - 1)$. For the block: $8 - 12 + 6 - 0 = 2 = 2 \times (1 - 0)$. For the pin: $2 - 3 + 3 - 0 = 2$. Seams, it turns out, do not disturb the count — a seam adds an edge but the face it cuts is still one face with one loop — and this is the form validators actually compute.

11.6 The sphere: a face with no edges

```
#16 = CLOSED_SHELL('',(#17));
#17 = ADVANCED_FACE('',(#18),#22,.T.);
#18 = FACE_BOUND('',#19,.T.);
#19 = VERTEX_LOOP('',#20);
#20 = VERTEX_POINT('',#21);
#21 = CARTESIAN_POINT('',(3.061616997868E-16,-7.498798913309E-32,-5.));
#22 = SPHERICAL_SURFACE('',#23,5.);
```

Referred to above: 4 records from `sphere.step`. *The complete file.*

```
#23 = AXIS2_PLACEMENT_3D('',#24,#25,#26);
#24 = CARTESIAN_POINT('',(0.,0.,0.));
#25 = DIRECTION('',(0.,0.,1.));
#26 = DIRECTION('',(1.,0.,0.));
```

A sphere of radius 5 as OCCT writes it with the parameter curves suppressed: one face, one bound, and the bound's loop is a **vertex loop** — a loop consisting of a single vertex and no edges. The spherical surface is closed in every direction; there is nothing to trim. But the schema requires a face to have at least one bound, so the face is "bounded" by a degenerate loop at the south pole (the coordinates are (0, 0, −5) with floating-point noise), the only thing in Figure 11.3 that the file records. V − E + F = 1 − 0 + 1 = 2.

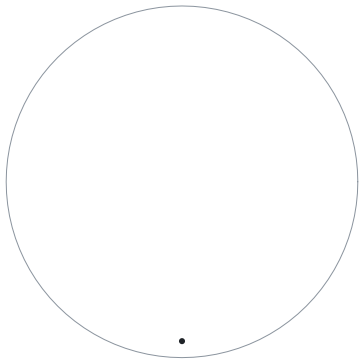


Figure 11.3: `sphere.step`: no edges, and one vertex at the south pole. The circle is the sphere's outline, which the file does not record.

Vertex loops also appear at the apex of a cone and at the poles of a sphere that *has* been cut by a seam: the degenerate edge where a meridian meets the pole collapses to a point. A `VERTEX_LOOP` is the standard's way of saying "this boundary has zero length."

11.7 The torus: two seams

```
#17 = ADVANCED_FACE('',(#18),#38,.T.);
#19 = EDGE_LOOP('',(#20,#29,#36,#37));
#20 = ORIENTED_EDGE('',*,*,#21,.F.);
#21 = EDGE_CURVE('',#22,#22,#24,.T.);              /* radius 12: the outer equator */
#29 = ORIENTED_EDGE('',*,*,#30,.F.);
#30 = EDGE_CURVE('',#22,#22,#31,.T.);              /* radius 2: a meridian */
#36 = ORIENTED_EDGE('',*,*,#21,.T.);
```

```
#37 = ORIENTED_EDGE('',*,*,#30,.T.);
#38 = TOROIDAL_SURFACE('',#39,10.,2.);
```

Referred to above: 17 records from `torus.step`. The complete file.

```
#18 = FACE_BOUND('',#19,.T.);
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(12.,-2.939152317954E-15,-4.898587196589E-16));
#24 = CIRCLE('',#25,12.);
#25 = AXIS2_PLACEMENT_3D('',#26,#27,#28);
#26 = CARTESIAN_POINT('',(0.,0.,-4.898587196589E-16));
#27 = DIRECTION('',(0.,0.,1.));
#28 = DIRECTION('',(1.,0.,-0.));
#31 = CIRCLE('',#32,2.);
#32 = AXIS2_PLACEMENT_3D('',#33,#34,#35);
#33 = CARTESIAN_POINT('',(10.,-2.449293598295E-15,0.));
#34 = DIRECTION('',(-2.449293598295E-16,-1.,0.));
#35 = DIRECTION('',(1.,-2.449293598295E-16,0.));
#39 = AXIS2_PLACEMENT_3D('',#40,#41,#42);
#40 = CARTESIAN_POINT('',(0.,0.,0.));
#41 = DIRECTION('',(0.,0.,1.));
#42 = DIRECTION('',(1.,0.,-0.));
```

A torus is periodic in both directions, so a single face on it needs two seams: one around (the equator, `#21`, radius 12 = 10 + 2) and one across (a meridian, `#30`, radius 2). Both are closed edges on the single vertex `#22` where they cross, and both appear twice in the loop (Figure 11.4).

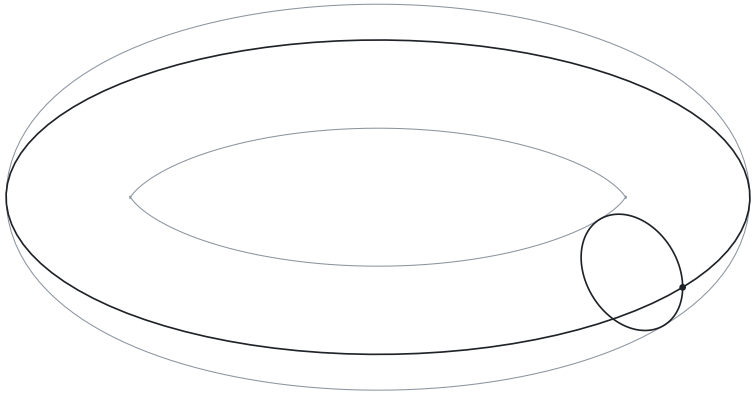


Figure 11.4: `torus.step`: the equator and a meridian, the two seams, crossing at its only vertex. The grey curves are the torus's outline.

One face, two edges, one vertex: V − E + F = 1 − 2 + 1 = 0, which for once is the textbook genus-1 answer, because here every edge is a seam and they cancel.

`TOROIDAL_SURFACE` has a `position`, a `major_radius` (centre to tube centre, 10) and a `minor_radius` (tube, 2).

11.8 The cone

```
#33 = B_SPLINE_CURVE_WITH_KNOTS('',1,(#34,#35),.UNSPECIFIED.,.F.,.F.,
        (2,2),(0.,1.),.PIECEWISE_BEZIER_KNOTS.);
#34 = CARTESIAN_POINT('',(5.,0.,0.));
#35 = CARTESIAN_POINT('',(2.,0.,10.));
#44 = CONICAL_SURFACE('',#45,5.,0.291456794478);
#45 = AXIS2_PLACEMENT_3D('',#46,#47,#48);
#46 = CARTESIAN_POINT('',(0.,0.,0.));
#47 = DIRECTION('',(-0.,-0.,-1.));
```

Referred to above: 1 record from `cone.step`. *The complete file.*

```
#48 = DIRECTION('',(1.,0.,0.));
```

`CONICAL_SURFACE` has a `position`, a `radius` at the position's location, and a `semi_angle` in the context's plane angle unit, here radians. This cone has radius 5 at z = 0 and half-angle 0.2915 rad (16.7°): over 10 mm of height it narrows to radius 2, and the axis, again, points down (Figure 11.5).

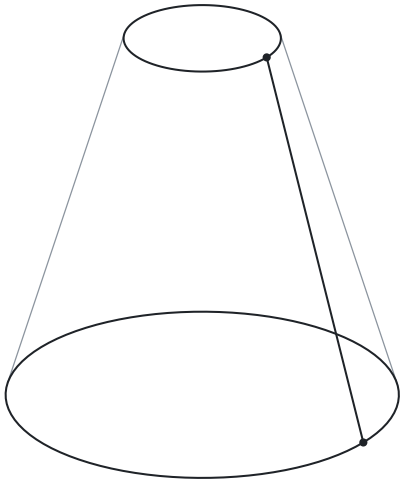


Figure 11.5: `cone.step`: two circles, and a seam written as a degree-1 B-spline, which draws as the straight line it is. The grey lines are the cone's outline.

The seam of this cone is a straight line from (5, 0, 0) to (2, 0, 10) — and the exporter has written it not as a `LINE` but as a **degree-1 B-spline with two control points**. It is a line in everything but name. This happens when a kernel has represented a curve internally as a spline and does not bother to recognise that it is analytic on the way out. A reader that special-cases `LINE` and `CIRCLE` and treats everything else as "free-form" will call this a free-form edge. It is not wrong to do so; it is just less informative. The next chapter takes the B-spline record apart.

11.9 The analytic vocabulary

Most analytic curves and surfaces in Part 42 are positioned by an `AXIS2_PLACEMENT_3D` — a point, an axis (the local Z) and a reference direction (the local X) — and then given one or two numbers. The swept surfaces at the foot of the table are the exception: they carry a curve and a direction instead.

Surfaces

Entity	Extra attributes	Position means
`PLANE`	—	a point on the plane; axis = normal
`CYLINDRICAL_SURFACE`	`radius`	a point on the axis; axis = cylinder axis
`CONICAL_SURFACE`	`radius`, `semi_angle`	a point on the axis where the radius applies
`SPHERICAL_SURFACE`	`radius`	the centre
`TOROIDAL_SURFACE`	`major_radius`, `minor_radius`	the centre; axis = the ring's axis
`SURFACE_OF_REVOLUTION`	`swept_curve`, `axis_position`	a curve rotated about an axis (an `AXIS1_PLACEMENT`)

Entity	Extra attributes	Position means
SURFACE_OF_LINEAR_EXTRUSION	swept_curve, extrusion_axis	a curve swept along a vector

Curves

Entity	Attributes	Notes
LINE	pnt, dir (a VECTOR)	infinite; the point at parameter t is pnt + t × dir, so the parameter is a distance only when the vector's magnitude is 1
CIRCLE	position, radius	in the position's XY plane, CCW about its axis
ELLIPSE	position, semi_axis_1, semi_axis_2	
TRIMMED_CURVE	basis_curve, trim_1, trim_2, sense_agreement, master_representation	a piece of another curve, trimmed by points or parameters
COMPOSITE_CURVE	segments, self_intersect	several curves end to end
POLYLINE	points	straight segments through a list of points

TRIMMED_CURVE is the record where the typed wrapper of Chapter 4 appears most: trim_1 and trim_2 are each a set of one or two values, each a SELECT of a CARTESIAN_POINT or a PARAMETER_VALUE — a writer may give the point, the parameter, or both, which is why they are parenthesised — so a curve trimmed by parameter looks like TRIMMED_CURVE('',#30,(PARAMETER_VALUE(0.)),(PARAMETER_VALUE(1.5708)),.T.,.PARAMETER.). Inside a B-rep, trimming is done by the topology — the edge's vertices — and TRIMMED_CURVE is rare; it appears in wireframes, in sketches, and in PMI annotation geometry.

⚠ In the wild: Floating-point noise

The sphere's pole is at (3.061616997868E-16, -7.498798913309E-32, -5.). The torus's vertex is at (12., -2.939152317954E-15, -4.898587196589E-16). These are zeros, written as the kernel computed them — the residue of a cosine of 90° that came out as 6×10^{-17} rather than 0. They are harmless, they are everywhere, and they are why comparing coordinates for equality is never the right operation on STEP geometry; comparing within a tolerance is, and the file's uncertainty is the natural starting value for one.

! In brief

Analytic surfaces are positioned by an axis placement plus one or two radii or an angle; their normals point away from the axis (cylinder, cone), outward from the centre (sphere), away from the tube's centre circle (torus, so inward on the inside of the ring), or along the axis (plane), and same_sense reconciles that with the solid. A periodic surface that has been cut is bounded by a single loop that traverses each **seam** edge twice, once in each direction. A full-circle edge is **closed**: one vertex, start = end; an arc has two. A surface with nothing to trim gets a **vertex loop**. Lines sometimes arrive as degree-1 B-splines.

12. Free-Form Geometry: B-Splines

> *The one non-analytic shape STEP needs, because every CAD kernel converges on it. What the eight parameters after the name in a B-spline record are, why a circle written as a spline has seven control points and weights of one-half, and how to tell a real free-form surface from an analytic one in disguise.*

A plane, a cylinder and a sphere can each be written down with a handful of numbers because each is a closed-form equation. A car body panel, a turbine blade or a filleted corner blended between three faces cannot. Every CAD system represents those as **B-spline** curves and surfaces — specifically, non-uniform rational B-splines, **NURBS** — and so does STEP. It is the only free-form representation in Part 42 that matters, and it can also represent every analytic shape exactly, which is why exporters sometimes write circles and cylinders as splines.

i Engineering background: NURBS in one page

A B-spline curve is a smooth path controlled by a sequence of **control points**. It does not, in general, pass through them; it is pulled toward them, the way a flexible ruler is bent by pegs. Three things fix its shape:

Degree. The polynomial degree of the pieces. Degree 1 gives straight segments between control points; degree 2 gives parabolic arcs; degree 3, cubic, is the usual choice for free-form work because it is the lowest degree that can be smooth in curvature. The number of polynomial pieces is the number of distinct knot intervals inside the curve's parameter range. For a clamped curve whose interior knots are all distinct, a degree-p curve with n control points has $n - p$ of them; repeated interior knots reduce the count.

Knots. A non-decreasing list of parameter values that says where one piece ends and the next begins. The knot vector has $n + p + 1$ entries. Repeating a knot value p times forces the curve *through* a control point (and lets it have a corner there); repeating the first and last knots $p + 1$ times **clamps** the curve so that it starts and ends exactly at the first and last control points. Almost every curve in a CAD file is clamped. Knots that are evenly spaced give a *uniform* B-spline; STEP's are *non-uniform* in general.

Weights. A **rational** B-spline attaches a positive weight to each control point; a heavier point pulls harder. Weights are what let a spline represent a circle *exactly*, which a polynomial spline cannot. A curve whose weights are all 1 is *non-rational*, and the weights are omitted.

A B-spline **surface** is the same construction in two parameters, u and v: a rectangular *grid* of control points, a degree in each direction, a knot vector in each direction, and optionally a grid of weights. Every point on the surface is a weighted blend of a $(p+1) \times (q+1)$ window of control points.

None of this is needed to *read* a STEP file — the records are just lists of numbers — but it is needed to know which list is which, and to recognise the special cases.

12.1 A B-spline curve record

From the cone of Chapter 11, its straight seam written as a spline:

```
#33 = B_SPLINE_CURVE_WITH_KNOTS('',1,(#34,#35),.UNSPECIFIED.,.F.,.F.,
        (2,2),(0.,1.),.PIECEWISE_BEZIER_KNOTS.);
#34 = CARTESIAN_POINT('',(5.,0.,0.));
#35 = CARTESIAN_POINT('',(2.,0.,10.));
```

The attributes, from the inheritance chain `representation_item` → `b_spline_curve` → `b_spline_curve_with_knots`:

Parameter	Attribute	Declared in
`''`	`name`	`representation_item`
`1`	`degree`	`b_spline_curve`
`(#34,#35)`	`control_points_list`	`b_spline_curve`
`.UNSPECIFIED.`	`curve_form`	`b_spline_curve`
`.F.`	`closed_curve`	`b_spline_curve`
`.F.`	`self_intersect`	`b_spline_curve`
`(2,2)`	`knot_multiplicities`	`b_spline_curve_with_knots`
`(0.,1.)`	`knots`	`b_spline_curve_with_knots`
`.PIECEWISE_BEZIER_KNOTS.`	`knot_spec`	`b_spline_curve_with_knots`

Degree 1, two control points: a straight segment from (5, 0, 0) to (2, 0, 10). The knots are stored compressed — *distinct values* with *multiplicities* rather than the full vector — so `(2,2)` and `(0.,1.)` expand to the knot vector (0, 0, 1, 1): $n + p + 1 = 2 + 1 + 1 = 4$ entries, with the ends repeated $p + 1 = 2$ times. Clamped, as promised.

`curve_form` is a hint: `.POLYLINE_FORM.`, `.CIRCULAR_ARC.`, `.ELLIPTIC_ARC.`, `.PARABOLIC_ARC.`, `.HYPERBOLIC_ARC.` or `.UNSPECIFIED.`. Writers almost always put `.UNSPECIFIED.` even when the curve is a circle, so the hint is rarely useful. `closed_curve` and `self_intersect` are `LOGICAL` — `.T.`, `.F.` or `.U.` for unknown — and are what they say. `knot_spec` describes the knot pattern: `.UNIFORM_KNOTS.`, `.QUASI_UNIFORM_KNOTS.`, `.PIECEWISE_BEZIER_KNOTS.` (every interior knot has multiplicity p, so each piece is an independent Bézier segment) or `.UNSPECIFIED.`. It, too, is a hint; the multiplicities are the truth.

12.2 A B-spline surface record

From a small part whose top edge is a free-form spline extruded downward:

```
#56 = B_SPLINE_SURFACE_WITH_KNOTS('',3,1,
        ((#57,#58),(#59,#60),(#61,#62),(#63,#64),(#65,#66),(#67,#68)),
        .UNSPECIFIED.,.F.,.F.,.F.,
        (4,1,1,4),(2,2),
        (0.,10.770329614269,22.43223340396,33.612573291459),(-5.,0.),
        .UNSPECIFIED.);
#57 = CARTESIAN_POINT('',(0.,0.,5.));
#58 = CARTESIAN_POINT('',(0.,0.,0.));
#59 = CARTESIAN_POINT('',(3.579071051472,4.965892889794,5.));
#60 = CARTESIAN_POINT('',(3.579071051472,4.965892889794,0.));
    /* ... three more pairs ... */
#67 = CARTESIAN_POINT('',(30.,3.,5.));
#68 = CARTESIAN_POINT('',(30.,3.,0.));
```

Referred to above: 6 records from `spline-nurbs.step`. The complete file.

```
#61 = CARTESIAN_POINT('',(10.398290711036,5.637385477547,5.));
#62 = CARTESIAN_POINT('',(10.398290711036,5.637385477547,0.));
#63 = CARTESIAN_POINT('',(19.789053380421,-3.826046807322,5.));
#64 = CARTESIAN_POINT('',(19.789053380421,-3.826046807322,0.));
#65 = CARTESIAN_POINT('',(26.463909496082,-2.731306369852,5.));
#66 = CARTESIAN_POINT('',(26.463909496082,-2.731306369852,-0.));
```

Parameter	Attribute	Meaning
`3, 1`	`u_degree`, `v_degree`	cubic along u, linear along v
`((#57,#58),...,(#67,#68))`	`control_points_list`	a list of 6 rows, each of 2 points: a 6 × 2 grid

Parameter	Attribute	Meaning
`.UNSPECIFIED.`	`surface_form`	hint (`.PLANE_SURF.`, `.CYLINDRICAL_SURF.`, …, `.UNSPECIFIED.`)
`.F., .F., .F.`	`u_closed`, `v_closed`, `self_intersect`	logicals
`(4,1,1,4), (2,2)`	`u_multiplicities`, `v_multiplicities`	
`(0.,10.77,22.43,33.61), (-5.,0.)`	`u_knots`, `v_knots`	distinct knot values
`.UNSPECIFIED.`	`knot_spec`	

Read it as two curves at right angles. Along u: degree 3, six control points, knots expanding to (0, 0, 0, 0, 10.77, 22.43, 33.61, 33.61, 33.61, 33.61) — ten entries, $6 + 3 + 1$, clamped at both ends, with two interior knots dividing the curve into three cubic pieces. The interior knot values are the cumulative chord lengths of the points the original spline was fitted through, which is how the exporter's kernel parametrised it. A reader never has to know that; the knots only have to be non-decreasing. Along v: degree 1, two control points, knots (−5, −5, 0, 0). A straight line.

Each row of the control grid is a pair of points with the same x and y and z = 5 or z = 0. The surface is the u-curve — the free-form spline along the top of the part — swept straight down five millimetres. It is, in other words, a `SURFACE_OF_LINEAR_EXTRUSION`, and the exporter that wrote this file was asked to convert everything to NURBS, so it wrote it as one. Nothing in the record says "extrusion"; you can only tell from the degree-1 v direction and the parallel rows.

12.3 Rational splines: the circle as a spline

Convert the pin of Chapter 11 to NURBS and its circles come out like this:

```
#31 = ( BOUNDED_CURVE()
        B_SPLINE_CURVE(2,(#32,#33,#34,#35,#36,#37,#38),.UNSPECIFIED.,.T.,.F.)
        B_SPLINE_CURVE_WITH_KNOTS((1,2,2,2,2,1),
          (-2.094395102393,0.,2.094395102393,4.188790204786,
           6.28318530718,8.377580409573),
          .UNSPECIFIED.)
        CURVE()
        GEOMETRIC_REPRESENTATION_ITEM()
        RATIONAL_B_SPLINE_CURVE((1.,0.5,1.,0.5,1.,0.5,1.))
        REPRESENTATION_ITEM('') );
```

Referred to above: 7 records from `pin-nurbs.step`. The complete file.

```
#32 = CARTESIAN_POINT('',(3.,0.,30.));
#33 = CARTESIAN_POINT('',(3.,5.196152422707,30.));
#34 = CARTESIAN_POINT('',(-1.5,2.598076211353,30.));
#35 = CARTESIAN_POINT('',(-6.,7.347880794884E-16,30.));
#36 = CARTESIAN_POINT('',(-1.5,-2.598076211353,30.));
#37 = CARTESIAN_POINT('',(3.,-5.196152422707,30.));
#38 = CARTESIAN_POINT('',(3.,0.,30.));
```

A complex instance, and Chapter 4 says why: `RATIONAL_B_SPLINE_CURVE` and `B_SPLINE_CURVE_WITH_KNOTS` are *sibling* subtypes of `b_spline_curve` — one adds weights, the other adds knots — and a curve that needs both has no single leaf name. So the record is written externally: every type in the chain, alphabetically, each with its own attributes. `BOUNDED_CURVE`, `CURVE`, `GEOMETRIC_REPRESENTATION_ITEM` and `REPRESENTATION_ITEM` are the supertypes contributing nothing but the name.

The content is the textbook NURBS circle. Degree 2. Seven control points. Weights alternating 1, ½, 1, ½, 1, ½, 1. Knots expanding to (−2.094, 0, 0, 2.094, 2.094, 4.189, 4.189, 6.283, 6.283, 8.378) — ten entries, $7 + 2 + 1$, with every interior knot doubled. Three pieces, each a quadratic rational arc spanning 120° (2.094 radians), joined at the knots of multiplicity 2, meeting to form a full circle. The half-weight points are the corners of the circumscribing triangle; the unit-weight points lie on the circle. `closed_curve` is `.T.`.

The knot values are angles in radians because that is how the kernel parametrised the original circle. This curve is *not* clamped: its end multiplicities are 1, not $p + 1$, and the outer knots at −2.094 and 8.378 lie outside the curve's actual range of 0 to 2π. That is how a closed periodic curve is written, and it is the common exception to the rule that CAD curves are clamped. `closed_curve` itself only says that the ends meet; it is the periodic knot layout that makes them meet smoothly.

The equivalent surface — the pin's cylindrical side as a rational B-spline — is the same construction in u with a degree-1 straight line in v, and its weights are a 7 × 2 grid.

12.4 Recognising analytic geometry in spline clothing

A file can legitimately contain a cylinder written as `CYLINDRICAL_SURFACE` or as a rational B-spline surface, and a reader may need to know that they are the same shape. Some exporters convert everything to NURBS on the way out (older CATIA translators did this for anything not planar). The signs, none of which is proof on its own:

- A degree-1 curve with two control points is a **line**.
- A degree-2 rational curve with 7 control points, weights (1, ½)*, and knots of multiplicity 2 at 120° intervals is a **full circle**. Quarter-circles use 3 control points and weights (1, $\sqrt{2}/2$, 1). Other arc splits exist.
- A surface with degree 1 in one direction and control rows that are translates of each other is an **extrusion**.
- A surface with degree 2 rational in one direction with the circle pattern, and degree 1 in the other, is a **cylinder**.
- A surface whose control net is planar is a **plane** (a bilinear 2 × 2 patch, as `#92` in the same file: degree 1 in both directions, four points).

Each sign is necessary, not sufficient: stretch the circle's seven control points and the same degree, knots and weights describe an ellipse, and a cone frustum has the cylinder's degree pattern, so the control points have to be checked as well.

None of this is required. Any reader that can evaluate a NURBS can evaluate these. But a tool that wants to report "this part has 14 cylindrical holes," or to compare two files for equivalence, has to look through the representation to the shape.

12.5 What the geometry chapters leave out

Part 42 defines several more B-spline forms — `BEZIER_CURVE`, `QUASI_UNIFORM_CURVE`, `UNIFORM_CURVE` and their surface equivalents — which are B-splines with knot vectors implied by the name instead of stored. They appear rarely; `..._WITH_KNOTS` is what every mainstream exporter writes. It also defines `OFFSET_CURVE_3D`, `OFFSET_SURFACE`, `CURVE_REPLICA` and `SURFACE_REPLICA`, `RECTANGULAR_TRIMMED_SURFACE` (a surface cut to a parameter rectangle), and `CURVE_BOUNDED_SURFACE` (a surface cut by curves rather than by topology). Each is what its name suggests, each is in the AP242 schema, and a reader that has understood the records in this chapter can look any of them up.

> ! In brief
>
> A B-spline record is degree, control points, three flags, and a knot vector stored as distinct values plus multiplicities; the full vector has $n + p + 1$ entries and is almost always clamped. A surface is the same in two directions with a grid of control points. Rational splines add a weight per control point and are written as complex instances because the rational and with-knots subtypes are siblings. Circles are seven points with weights 1, ½ alternating; lines are degree 1 with two points; extrusions are degree 1 in one direction — patterns that suggest a shape, which the control points then confirm.

13. Parameter Curves, and Meshes

> *Two things a geometry chapter has to mention because they are in the files: the second, two-dimensional copy of every edge that doubles the size of a B-rep, and the AP242 records that carry a triangle mesh instead of — or alongside — a B-rep.*

13.1 The edge that is written twice

The block of Chapter 10 was exported with a setting that suppresses *parameter curves*. Without it, the same block is 350 records instead of 170, and edge `#21` looks like this:

```
#21 = EDGE_CURVE('',#22,#24,#26,.T.);
#26 = SURFACE_CURVE('',#27,(#31,#43),.PCURVE_S1.);
#27 = LINE('',#28,#29);                          /* the 3D line, as before */
#31 = PCURVE('',#32,#37);
#32 = PLANE('',#33);                             /* the x=0 face's own plane, shared */
#37 = DEFINITIONAL_REPRESENTATION('',(#38),#42);
#38 = LINE('',#39,#40);                          /* a 2D line in parameter space */
#39 = CARTESIAN_POINT('',(0.,0.));
#40 = VECTOR('',#41,1.);
#41 = DIRECTION('',(1.,0.));
#42 = ( GEOMETRIC_REPRESENTATION_CONTEXT(2)
        PARAMETRIC_REPRESENTATION_CONTEXT()
        REPRESENTATION_CONTEXT('2D SPACE','') );
#43 = PCURVE('',#44,#49);
#44 = PLANE('',#45);                             /* the y=0 face's own plane, shared */
#49 = DEFINITIONAL_REPRESENTATION('',(#50),#54);
    /* ... another 2D line, another 2D context ... */
```

Referred to above: 20 records from `block-with-pcurves.step`. The complete file.

```
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(0.,0.,10.));
#28 = CARTESIAN_POINT('',(0.,0.,0.));
#29 = VECTOR('',#30,1.);
#30 = DIRECTION('',(0.,0.,1.));
#33 = AXIS2_PLACEMENT_3D('',#34,#35,#36);
#34 = CARTESIAN_POINT('',(0.,0.,0.));
#35 = DIRECTION('',(1.,0.,0.));
#36 = DIRECTION('',(0.,0.,1.));
#45 = AXIS2_PLACEMENT_3D('',#46,#47,#48);
#46 = CARTESIAN_POINT('',(0.,0.,0.));
#47 = DIRECTION('',(0.,1.,0.));
#48 = DIRECTION('',(0.,0.,1.));
#50 = LINE('',#51,#52);
#51 = CARTESIAN_POINT('',(0.,0.));
#52 = VECTOR('',#53,1.);
#53 = DIRECTION('',(1.,0.));
#54 = ( GEOMETRIC_REPRESENTATION_CONTEXT(2)
PARAMETRIC_REPRESENTATION_CONTEXT() REPRESENTATION_CONTEXT('2D SPACE',''
  ) );
```

The edge's geometry is no longer a `LINE` but a `SURFACE_CURVE`: a 3D curve (`#27`, the line) together with a list of one or two entries, each normally a `PCURVE`, describing the *same* edge as a 2D curve in the parameter space of one of the faces it borders. Edge `#21` lies on two faces, so it has two pcurves.

A `PCURVE` has a `basis_surface` and a `reference_to_curve`. The reference is a `DEFINITIONAL_REPRESENTATION` — a representation whose single item is a 2D curve and whose context is a 2D *parametric*

context (dimension 2, no units, because parameter space is unitless). Here the 2D curve is a line from (0, 0) in direction (1, 0): the edge runs along the plane's local X parameter.

The last attribute of the surface curve, `master_representation`, says which of the three descriptions is authoritative when they disagree: `.CURVE_3D.`, `.PCURVE_S1.` or `.PCURVE_S2.`. This exporter says the first pcurve is the master.

> **i** Engineering background: Why a kernel keeps parameter curves
>
> A face is a piece of a surface bounded by a loop of edges. To *use* the face — to mesh it, to find which side of an edge is inside — the kernel needs the boundary in the surface's own coordinates: the (u, v) parameters, not the (x, y, z) positions. It can compute that by projecting each edge's 3D curve onto the surface, but projection is slow, numerically delicate, and ambiguous on periodic surfaces (which side of the seam?) and at degenerate points (a cone's apex maps to a whole line in parameter space). So kernels store the 2D curve directly, for every edge, on every face it bounds. Those are the pcurves. Writing them into the exchange file spares the importing kernel the projection and, more importantly, removes the ambiguities. It also roughly doubles the number of records: each edge gains a `SURFACE_CURVE` above the 3D line it already had, two `PCURVE`s, two representations, two 2D contexts, and a 2D line, point, vector and direction for each — fifteen extra records per edge in this file.

> **⚠** In the wild: Shared or duplicated surfaces
>
> The pcurve `#31` points at `#32`, a `PLANE` — and that is the very record the face `#17` uses as its own surface. This exporter shares one plane between a face and every pcurve that lies on it, so the file still holds only six `PLANE` records for six faces. Other exporters write a fresh copy of the surface for each pcurve, and the same part then carries thirty planes instead of six. Neither is wrong, and this is one reason two files of the same part from two systems can differ by a factor of two in size while describing identical geometry.

Three forms of surface curve turn up in files. Plain `SURFACE_CURVE` is the ordinary case. `SEAM_CURVE` is the seam of Chapter 11: a surface curve whose *two* pcurves lie on the *same* surface, one on each side of the cut. The pin of Chapter 11 was exported without pcurves. Exported again with them, as `pin-with-pcurves.step`, its seam is:

```
#21 = EDGE_CURVE('',#22,#24,#26,.T.);
#26 = SEAM_CURVE('',#27,(#31,#43),.PCURVE_S1.);
#31 = PCURVE('',#32,#37);
#32 = CYLINDRICAL_SURFACE('',#33,3.);
#37 = DEFINITIONAL_REPRESENTATION('',(#38),#42);
#38 = LINE('',#39,#40);
#39 = CARTESIAN_POINT('',(-0.,0.));            /* u = 0 */
#43 = PCURVE('',#32,#44);
#44 = DEFINITIONAL_REPRESENTATION('',(#45),#49);
#45 = LINE('',#46,#47);
#46 = CARTESIAN_POINT('',(-6.28318530718,0.)); /* u = -2*pi */
```

Referred to above: 18 records from `pin-with-pcurves.step`. The complete file.

```
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(3.,0.,0.));
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(3.,0.,30.));
#27 = LINE('',#28,#29);
#28 = CARTESIAN_POINT('',(3.,0.,0.));
#29 = VECTOR('',#30,1.);
#30 = DIRECTION('',(0.,0.,1.));
#33 = AXIS2_PLACEMENT_3D('',#34,#35,#36);
#34 = CARTESIAN_POINT('',(0.,0.,0.));
#35 = DIRECTION('',(-0.,-0.,-1.));
#36 = DIRECTION('',(1.,0.,-0.));
#40 = VECTOR('',#41,1.);
#41 = DIRECTION('',(-0.,-1.));
#42 = ( GEOMETRIC_REPRESENTATION_CONTEXT(2)
PARAMETRIC_REPRESENTATION_CONTEXT() REPRESENTATION_CONTEXT('2D SPACE',''
```

```
  ) );
#47 = VECTOR('',#48,1.);
#48 = DIRECTION('',(-0.,-1.));
#49 = ( GEOMETRIC_REPRESENTATION_CONTEXT(2)
PARAMETRIC_REPRESENTATION_CONTEXT() REPRESENTATION_CONTEXT('2D SPACE',''
  ) );
```

Both pcurves lie on the one cylinder #32, and both are the same line of constant u in its parameter space, a whole turn apart: $u = 0$ for the first, $u = -2\pi$ for the second, the two sides of the cut. Both run in direction $(0, -1)$, decreasing v; this cylinder's axis points down, so decreasing v is up the pin, the way the 3D line #27 runs.

`INTERSECTION_CURVE` marks an edge where two surfaces meet, which is nearly all of them; some exporters write it, most write plain `SURFACE_CURVE`.

Are pcurves needed? For a reader that only walks structure, no: an edge's 3D curve is `surface_curve.curve_3d`, one more hop than before, and everything else is unchanged. For a reader that rebuilds solids in a geometric kernel, they are a strong hint and occasionally the only way to trim a face correctly. Most CAD exporters write them. OCCT and a few others make it optional. A file without them is smaller, not wrong.

13.2 Meshes: tessellated geometry

Everything so far has been exact geometry. AP242 added a second kind: **tessellated** geometry, in which a shape is a set of triangles. It exists because the two things most often done with a CAD model downstream — viewing it, and comparing it — do not need surfaces and are far cheaper on a mesh; because lightweight visualisation formats (JT above all) had been carrying meshes for years and STEP could not; and because some geometry (scanned surfaces, simulation results) never had an exact form. A tessellated shape lives in a `TESSELLATED_SHAPE_REPRESENTATION` and is built from these records:

```
#500 = TESSELLATED_SHAPE_REPRESENTATION('',(#501),#165);
#501 = TESSELLATED_SOLID('',(#503),#15);
#503 = TRIANGULATED_FACE('',#504,4,((0.,0.,-1.)),#141,(1,2,3,4),((1,3,2),(1,4,3)));
#504 = COORDINATES_LIST('',4,((0.,0.,0.),(60.,0.,0.),(60.,20.,0.),(0.,20.,0.)));
```

Referred to above: 155 records from `block-ap242.step`; the 3 it points to directly are shown. The complete file.

```
#15 = MANIFOLD_SOLID_BREP('',#16);
#141 = ADVANCED_FACE('',(#142),#148,.F.);
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#166,#167,#168)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
/* ... and 152 more: 26 CARTESIAN_POINT, 24 DIRECTION, 24 ORIENTED_EDGE, 12 EDGE_CURVE, ... */
```

(These four records are hand-written to the AP242 schema — the bottom face of the block as two triangles — rather than exported; the exporter used in this book does not write tessellation.)

`COORDINATES_LIST` is the vertex buffer: a count and a list of 3D points, as bare lists of reals rather than `CARTESIAN_POINT` records — the first place in the format where geometry is written *without* a record per point, precisely because a mesh may have a million of them.

`TRIANGULATED_FACE` is the index buffer. Its attributes: the coordinates list; `pnmax`, the number of points used; `normals`, a list of either one normal (flat shading, as here) or one per point; an optional `geometric_link` back to the exact `ADVANCED_FACE` it approximates (#141, the bottom face); `pnindex`, a list of 1-based indices into the coordinates list selecting the points this face uses; and `triangles`, a list of triples of 1-based indices *into* `pnindex`. So `(1,3,2)` means the first, third and second entries of `pnindex`, which are points 1, 3 and 2 of the coordinates list. That order winds the triangle clockwise seen from $+Z$, which is counter-clockwise about this face's own $-Z$ normal: that is the agreement graphics pipelines assume, though neither AP242 nor the tessellation recommended practice states it, and a mesh wound the other way renders inside out. If `pnindex` is empty, the triangles index the coordinates directly.

`TESSELLATED_SHELL` and `TESSELLATED_SOLID` are alternatives rather than levels, and each holds its tessellated faces directly: a shell cannot be an item of a solid, because the schema admits only tessellated faces, edges and vertices there. A shell carries an optional `topological_link` to the exact `CONNECTED_FACE_SET`, closed or open; a solid carries an optional `geometric_link` to the exact `MANIFOLD_SOLID_BREP`. Here the solid's `geometric_link` points at the exact `MANIFOLD_SOLID_BREP` #15, and the face's own at the exact `ADVANCED_FACE` #141: this is how a **hybrid** file carries the exact B-rep *and* a mesh of it, and says

which is which. A `COMPLEX_TRIANGULATED_FACE` is an alternative to `TRIANGULATED_FACE` rather than an extension of it: it carries triangle strips and fans instead of a plain triangle list, and must have at least one of them. `TRIANGULATED_SURFACE_SET` and `TESSELLATED_CURVE_SET` cover meshes and polylines with no solid behind them; `TESSELLATED_POINT_SET` covers scanned points.

Tessellated PMI — annotations drawn as triangles rather than curves — uses the same records and is in Chapter 16.

Two things to know when meeting a tessellated file. First, the mesh is *approximate*, and in most files it carries no statement of how approximate. AP242 does provide `TESSELLATED_SHAPE_REPRESENTATION_WITH_ACCURACY_PARAMETERS`, which can state a chordal deviation, an angular deviation or a maximum edge length, but writers seldom use it, and a plain `TESSELLATED_SHAPE_REPRESENTATION` records nothing. Second, in a hybrid file the mesh and the B-rep are independent representations of the same `PRODUCT_DEFINITION_SHAPE`; nothing requires them to agree, and a reader that wants to trust the mesh should check it against the exact geometry or against the validation properties of Chapter 15.

! In brief

A `SURFACE_CURVE` is a 3D curve plus its images in the parameter spaces of the faces it bounds (`PCURVE`s, each a 2D curve in a unitless 2D context); a `SEAM_CURVE` has both images on one surface. They help importing kernels trim faces and roughly double a file. AP242 tessellation stores a mesh as a `COORDINATES_LIST` and `TRIANGULATED_FACE`s with 1-based indices, optionally linked to the exact topology; it is approximate, and in practice states no tolerance, though AP242 has records for one.

III Everything Else in the File

14 Appearance: Colours and Layers . . . 74

15 Properties and Validation 77

16 Product Manufacturing Information . 81

17 Application Protocols and the Real World 89

18 Beyond Edition 2 92

14. Appearance: Colours and Layers

> *Nine records to make a pin orange. Why the colour is attached to the geometry from the outside, how a colour reaches a single face, and where the layers went.*

The clamp assembly was exported with the bracket grey and the pins orange. Both colours are in the file, and neither is anywhere near the geometry it colours. Here is the pin's:

```
#318 = MECHANICAL_DESIGN_GEOMETRIC_PRESENTATION_REPRESENTATION('',(#319),#302);
#319 = STYLED_ITEM('color',(#320),#249);
#320 = PRESENTATION_STYLE_ASSIGNMENT((#321));
#321 = SURFACE_STYLE_USAGE(.BOTH.,#322);
#322 = SURFACE_SIDE_STYLE('',(#323));
#323 = SURFACE_STYLE_FILL_AREA(#324);
#324 = FILL_AREA_STYLE('',(#325));
#325 = FILL_AREA_STYLE_COLOUR('',#326);
#326 = COLOUR_RGB('',0.800000010877,0.500000003429,0.199999998682);
```

Referred to above: 58 records from `clamp.step`; the 2 it points to directly are shown. The complete file.

```
#249 = MANIFOLD_SOLID_BREP('',#250);
#302 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#306)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#303,#304,#305)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
/* ... and 56 more: 11 DIRECTION, 8 CARTESIAN_POINT, 6 ORIENTED_EDGE, 5 AXIS2_PLACEMENT_3D, ... */
```

Nine records, of which one carries the colour and one names the thing being coloured. The rest is Part 46 — *visual presentation* — doing what the integrated resources always do: expressing a simple idea in terms general enough to cover every kind of presentation anyone might want.

14.1 Reading the chain

From the bottom:

- `#326 COLOUR_RGB` — red 0.8, green 0.5, blue 0.2. Components are reals from 0 to 1. The name is empty. (The values are not quite 0.8, 0.5 and 0.2 because the exporter converts colours to a linear working space and back in single precision; deviations of this size carry no meaning.)
- `#325 FILL_AREA_STYLE_COLOUR` — a colour used for filling areas. The alternative is a hatch pattern or a tile.
- `#324 FILL_AREA_STYLE` — a set of fill styles (here, one). Fill styles are what a *face* gets; curve styles are what an *edge* gets.
- `#323 SURFACE_STYLE_FILL_AREA` — the fill style applied as a surface style element. Other elements exist for boundaries, silhouettes, parameter lines, rendering.
- `#322 SURFACE_SIDE_STYLE` — a set of surface style elements for one side of a surface.
- `#321 SURFACE_STYLE_USAGE` — that side style applied to `.BOTH.` sides of the surface (or `.POSITIVE.` or `.NEGATIVE.`).
- `#320 PRESENTATION_STYLE_ASSIGNMENT` — a set of styles (here, one surface style usage) to be assigned together.
- `#319 STYLED_ITEM` — the assignment. `name` is `'color'`, `styles` is `(#320)`, and `item` is `#249`: the pin's `MANIFOLD_SOLID_BREP`. This is the record that connects appearance to geometry, and it does so by pointing *at* the solid — the solid does not know it has a colour.
- `#318 MECHANICAL_DESIGN_GEOMETRIC_PRESENTATION_REPRESENTATION` — a representation whose items are styled items, in the same geometric context as the geometry it decorates (`#302`). Its job is to be a container a reader can look for. There are two such containers in practice, this one and `DRAUGHTING_MODEL`, and a file with PMI in it will use the latter: *find every representation of either type, and you have found every colour in the file.*

Every *surface* colour in every AP214 or AP242 file follows this chain. The middle six records are almost always exactly these, differing only in instance numbers, and once you have read one you can skim the rest. What varies is the two ends: which colour, and which item.

14.2 What gets coloured

`STYLED_ITEM.item` can point at any representation item. In practice:

A solid. As here. The whole body takes the colour. This is what most exporters write for a part with a single body colour, and it is what a viewer applies by default to every face of the solid.

A face. `STYLED_ITEM('',(#s),#88)` with `#88` an `ADVANCED_FACE` colours one face. Files with per-face colours have one styled item per face, and they can number in the thousands. Style is inherited downwards, so a face takes the body's colour unless something says otherwise, and the thing that says otherwise is `OVER_RIDING_STYLED_ITEM` — a Part 46 subtype of `STYLED_ITEM` with an extra attribute pointing at the styled item it overrides. The recommended practice requires it for a per-face colour; a bare second styled item on the face is not enough.

An edge or a curve. Via a `CURVE_STYLE` instead of a surface style: `CURVE_STYLE('',#font,POSITIVE_LENGTH_MEASURE(0.1),#colour)` — a line font, a width (a wrapped measure, as Chapter 4 predicted), a colour. Edge colours appear in wireframe exports and in PMI annotation geometry.

A mapped item or a whole representation, so that an assembly component can be coloured as a unit without touching the child's geometry. Less common.

The pin's colour is on the pin's *solid*, `#249`, which belongs to the pin's product. Both occurrences of the pin are therefore orange, because both are instances of the same product and there is one solid. Had the CAD system coloured pin-2 differently, the exporter would have written a `CONTEXT_DEPENDENT_OVER_RIDING_STYLED_ITEM`, whose `style_context` is the top-down list of representation relationships naming the one occurrence. It is in all three schemas, and "instance colour" is still one of the features most often lost in exchange, because many exporters and readers ignore it.

14.3 Named colours

`COLOUR_RGB` is one of two colour entities. The other:

```
#340 = DRAUGHTING_PRE_DEFINED_COLOUR('red');
```

A `DRAUGHTING_PRE_DEFINED_COLOUR` has only a name, from a fixed list: `'red'`, `'green'`, `'blue'`, `'yellow'`, `'magenta'`, `'cyan'`, `'black'`, `'white'`. It stands in for the RGB triple wherever one is expected. It is inherited from the drawing-exchange parts of the standard, where line colours were symbolic, and is what a few exporters write for their default palette. The schema gives the eight names no RGB values, but the CAx-IF recommended practice fixes them as the pure corners of the colour cube, plus black and white.

14.4 Layers

CAD systems group entities onto named layers, and Part 46 carries them:

```
#350 = PRESENTATION_LAYER_ASSIGNMENT('Pins','locating pins',(#249));
```

> *Referred to above: 53 records from `clamp.step`; the 1 it points to directly are shown. The complete file.*
>
> ```
> #249 = MANIFOLD_SOLID_BREP('',#250);
> /* ... and 52 more: 11 DIRECTION, 8 CARTESIAN_POINT, 6 ORIENTED_EDGE, 5 AXIS2_PLACEMENT_3D, ... */
> ```

A name, a description, and a set of the representation items on the layer: here only `#249`, the pin's solid, and not the bracket's solid `#41`. As always, the layer points at its members; a face does not know its layer. A file's layers are found by collecting every `PRESENTATION_LAYER_ASSIGNMENT` and inverting. Layer assignments are far less common in STEP than in native CAD files: many exporters do not write them, and when they do the members are typically solids rather than faces. AP203 first edition has no layer entity at all.

14.5 Visibility

A layer can be switched off, or an item hidden, with `INVISIBILITY`:

```
#360 = INVISIBILITY((#319));
```

Referred to above: 61 records from `clamp.step`; the 1 it points to directly are shown. The complete file.

```
#319 = STYLED_ITEM('color',(#320),#249);
/* ... and 60 more: 11 DIRECTION, 8 CARTESIAN_POINT, 6 ORIENTED_EDGE, 5 AXIS2_PLACEMENT_3D, ... */
```

— a set of styled items (or layer assignments, draughting callouts, or whole representations) that are not to be displayed. `INVISIBILITY` itself carries only a list of items; to hide something in one view rather than everywhere, the subtype `CONTEXT_DEPENDENT_INVISIBILITY` adds a context — a presentation representation or a presentation set, and in AP242 a draughting model too, and it is rare in mechanical exchange except inside saved views, which belong to Chapter 16.

14.6 Names on geometry

Not appearance, but often asked about in the same breath: *where is the name of a face?* Every representation item has a `name` attribute, and it is almost always `''`. Some systems — NX and Creo among them, when asked — write the user's face or body name there: `ADVANCED_FACE('Mounting surface',(#18),#52,.F.)`. The only other place is from outside: a `SHAPE_ASPECT` whose name is the label and whose `GEOMETRIC_ITEM_SPECIFIC_USAGE` points at the face, the mechanism Chapter 16 uses for datum features; few exporters use it for plain names. Feature names go the same way or not at all: the B-rep knows nothing about holes, fillets or pockets, only about faces, and although AP242 defines feature records, mainstream exporters do not write them.

⚠ In the wild: What AP203 files cannot say

`STYLED_ITEM`, `COLOUR_RGB` and `PRESENTATION_LAYER_ASSIGNMENT` are not in AP203 first edition (`CONFIG_CONTROL_DESIGN`). A file with that schema is grey and layerless by definition. Files exist that declare `CONFIG_CONTROL_DESIGN` and contain colour records anyway; strict readers drop the colours and lenient ones keep them. AP203 second edition added presentation; AP242 carries everything here.

! In brief

Colour reaches geometry from the outside: a `STYLED_ITEM` points at a solid, face or curve and at a `PRESENTATION_STYLE_ASSIGNMENT`, which descends through five fixed records to a `COLOUR_RGB` or a named colour. Styled items are collected in a `MECHANICAL_DESIGN_GEOMETRIC_PRESENTATION_REPRESENTATION` or, in files with PMI, a `DRAUGHTING_MODEL`. A face colour overrides a body colour only through an `OVER_RIDING_STYLED_ITEM`. Layers are `PRESENTATION_LAYER_ASSIGNMENT`s listing their members. Face names live in `representation_item.name`, when they live anywhere.

15. Properties and Validation

> *How a file carries "material: 6061-T6" and "mass: 0.324 kg" — and how it carries its own checksum, so that a receiving system can tell whether the geometry survived the trip.*

Chapter 7 introduced `PRODUCT_DEFINITION_SHAPE` as a *property definition* — the shape property of a part — and promised the rest of the family. The same three-record pattern that attaches a shape attaches anything else, and it is how both user attributes and the standard's built-in validation data travel.

15.1 The property pattern

```
#420 = PROPERTY_DEFINITION('Material','',#5);
#421 = PROPERTY_DEFINITION_REPRESENTATION(#420,#422);
#422 = REPRESENTATION('',(#423),#165);
#423 = DESCRIPTIVE_REPRESENTATION_ITEM('Material','6061-T6');
```

Referred to above: 11 records from `block.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#166,#167,#168)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#167 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#168 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
```

(The examples in this chapter are written to the pattern in the CAx Implementor Forum's recommended practices; the exporter used for the running example does not write properties.)

- **`PROPERTY_DEFINITION`** names the property and says what it is a property *of*. `name` is `'Material'`; `definition` is `#5`, the block's product definition. The `definition` attribute is the `characterized_definition` SELECT that `PRODUCT_DEFINITION_SHAPE` inherits, though a shape property is narrowed by a rule to product definitions and occurrences. A plain `PROPERTY_DEFINITION` may use the whole of it: a `PRODUCT_DEFINITION_SHAPE` (a property of the geometry), a `SHAPE_ASPECT` (a property of one feature), or a `NEXT_ASSEMBLY_USAGE_OCCURRENCE` (a property of one occurrence).
- **`PROPERTY_DEFINITION_REPRESENTATION`** binds the property to a representation, exactly as `SHAPE_DEFINITION_REPRESENTATION` binds a shape to one.
- **`REPRESENTATION`** is the container, with a context. For text properties the context is irrelevant but required; writers reuse the geometric context.
- **`DESCRIPTIVE_REPRESENTATION_ITEM`** carries a string value: `name 'Material'`, `description '6061-T6'`.

For a numeric property the item is a **`MEASURE_REPRESENTATION_ITEM`**, and the unit comes back:

```
#430 = PROPERTY_DEFINITION('Mass','',#5);
#431 = PROPERTY_DEFINITION_REPRESENTATION(#430,#432);
#432 = REPRESENTATION('',(#433),#165);
#433 = MEASURE_REPRESENTATION_ITEM('Mass',MASS_MEASURE(0.324),#434);
#434 = ( MASS_UNIT() NAMED_UNIT(*) SI_UNIT(.KILO.,.GRAM.) );
```

Referred to above: 11 records from `block.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#166,#167,#168)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#167 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#168 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
```

`MEASURE_REPRESENTATION_ITEM` is a complex-instance-free way of saying "a representation item that is a measure with unit": a name, a wrapped value, a unit. The kilogram is `SI_UNIT(.KILO.,.GRAM.)` — the SI base name is `gram`, and the kilogram takes a prefix. The CAx-IF practice adds two more records to every user attribute: a `GENERAL_PROPERTY` carrying the attribute's name, and a `GENERAL_PROPERTY_ASSOCIATION` tying it to the `PROPERTY_DEFINITION`. It is the presence of that `GENERAL_PROPERTY` that tells a reader this is a user attribute rather than a validation property. It is not an AP242 addition: AP203 second edition and AP214 have it too.

This is the mechanism by which a CAD system's custom properties — part number, material, finish, supplier, cost centre — reach a STEP file, when the exporter is configured to send them. Most are not by default. When they are, expect one `PROPERTY_DEFINITION` per attribute per product, each with its own representation, and expect the `name` strings to be whatever the CAD system calls the attribute.

15.2 Validation properties

The most important use of the property pattern is one the standard's implementors invented and later formalised: **geometric validation properties**. The writer computes a few numbers that summarise the geometry — volume, surface area, centroid — and puts them in the file. The reader, after importing, computes the same numbers from what it built and compares. If they match to within tolerance, the geometry survived; if the volume is off by a face's worth, a face was lost; if the centroid moved, a component is misplaced. A match is evidence rather than proof — different shapes can share a volume, an area and a centroid — but a mismatch is a certain sign that something went wrong.

```
#400 = PROPERTY_DEFINITION('geometric validation property',
  'volume of part',#4);
#401 = PROPERTY_DEFINITION_REPRESENTATION(#400,#402);
#402 = REPRESENTATION('volume',(#403),#165);
#403 = MEASURE_REPRESENTATION_ITEM('volume measure',
  VOLUME_MEASURE(12000.),#404);
#404 = DERIVED_UNIT((#405));
#405 = DERIVED_UNIT_ELEMENT(#166,3.);

#406 = PROPERTY_DEFINITION('geometric validation property',
  'surface area of part',#4);
#407 = PROPERTY_DEFINITION_REPRESENTATION(#406,#408);
#408 = REPRESENTATION('surface area',(#409),#165);
#409 = MEASURE_REPRESENTATION_ITEM('surface area measure',
  AREA_MEASURE(4000.),#410);
#410 = DERIVED_UNIT((#411));
#411 = DERIVED_UNIT_ELEMENT(#166,2.);

#412 = PROPERTY_DEFINITION('geometric validation property',
  'centroid of part',#4);
#413 = PROPERTY_DEFINITION_REPRESENTATION(#412,#414);
#414 = REPRESENTATION('centroid',(#415),#165);
#415 = CARTESIAN_POINT('centre point',(30.,10.,5.));
```

Referred to above: 12 records from `block.step`. The complete file.

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#166,#167,#168)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#167 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#168 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
```

The block: $60 \times 20 \times 10 = 12\,000\,\text{mm}^3$; $2 \times (1200 + 600 + 200) = 4000\,\text{mm}^2$; centre at $(30, 10, 5)$. Three things make these records findable by a reader that knows nothing else about the file:

- `PROPERTY_DEFINITION.name` is the fixed string `'geometric validation property'`.
- `REPRESENTATION.name` is `'volume'`, `'surface area'` or `'centroid'`.
- The item's name is `'volume measure'`, `'surface area measure'` or `'centre point'`.

Those strings are prescribed by the recommended practice, not by the schema, and every major exporter follows them. The `definition` points at the `PRODUCT_DEFINITION_SHAPE` (`#4`) — these are properties of the *shape* — and the units are derived: cubic millimetres for volume, square millimetres for area, built from the length unit as Chapter 6 described. The centroid is an ordinary `CARTESIAN_POINT` in the geometric context, so it is in millimetres automatically.

Validation properties extend well beyond solids, as separate families rather than as extensions of this one. `'assembly validation property'` with `'number of children'` (an `INTEGER_REPRESENTATION_ITEM` in AP242, a `COUNT_MEASURE` in AP214) records how many components an assembly should have, and `'notional solids centroid'` checks that placements composed correctly. The same pattern with a different `PROPERTY_DEFINITION` name gives `'tessellated validation property'`, `'attribute validation property'`, `'composite validation property'` and `'pmi validation property'`, the last with items named `'number of annotations'`, `'number of views'` and `'affected area'`, so that a receiver can confirm that not just the shape but the annotations arrived.

> **i** Engineering background: Why an exchange format needs a checksum
>
> Every step of a geometry exchange can lose information silently. A face whose surface the importer cannot rebuild is dropped; a solid whose shell no longer closes is quietly converted to a surface body; a component whose placement was read with the frames reversed lands in the wrong place; an inch file read as millimetres is 25.4 times too small. None of these produce an error. The file is valid, the import succeeds, and the model is wrong.
>
> Validation properties are the industry's answer. They are cheap to compute, they are sensitive to exactly the failures that matter — a missing face changes the area; a lost solid changes the volume; a misplaced component changes the centroid; a unit error changes everything — and they are independent of the representation, so a B-rep in one system and a B-rep in another can be compared without either understanding the other's kernel. Most CAD importers check them automatically and report a discrepancy as a warning. They are also, for anyone writing tools that transform STEP files, the easiest correctness test there is.

15.3 Where names hide

The chapter on identity said the part's name is in `PRODUCT.name`. That is the *product's* name. Other names in a file:

- **Occurrence names** — `'pin-1'`, `'pin-2'` — are in `NEXT_ASSEMBLY_USAGE_OCCURRENCE.name`.
- **Body names**, in multi-body parts, are in `MANIFOLD_SOLID_BREP.name` when the exporter writes them, and in `SHAPE_REPRESENTATION.name` for some others.

- **Face names**, rarely, in `ADVANCED_FACE.name`.
- **Feature names** exist only where a `SHAPE_ASPECT` names a feature, as the PMI of Chapter 16 does for a datum feature. There is no feature tree.
- **Descriptions** of anything are in the entity's `description` attribute, which is `''` or `$` nearly everywhere.
- **File-level metadata** — author, organisation, originating system — is in the header. People, organisations and approvals attached to the product itself are data-section records, which AP203 first edition required and the later protocols make optional.

A reader that wants "the name of this thing" has to know which thing and look in the right slot; there is no universal name table.

15.4 What is not here

Properties are strings and numbers hung on the product graph. They are not *parametric*: a mass property does not update when the geometry changes, and a material property does not tell a kernel what density to use unless the reader chooses to interpret it. They are not *behavioural*: no constraints, no formulas, no design intent. STEP has parts for some of that — Part 108 for parameterisation and constraints, AP209 for analysis — and almost no CAD exporter writes them. What travels is the evaluated model and a set of labels on it.

! In brief

A property is `PROPERTY_DEFINITION` (name, and what it is a property of) → `PROPERTY_DEFINITION_REPRESENTATION` → `REPRESENTATION` → an item: `DESCRIPTIVE_REPRESENTATION_ITEM` for text, `MEASURE_REPRESENTATION_ITEM` with a wrapped value and a unit for numbers. Validation properties use this pattern with prescribed names — `'geometric validation property'`, `'volume'`, `'surface area'`, `'centroid'` — to carry a checksum of the geometry that the receiving system recomputes and compares; it catches most transfer errors without proving equivalence.

16. Product Manufacturing Information

> *Tolerances, datums and dimensions — the information that makes a model manufacturable — and the two entirely different ways a STEP file can carry them: as pictures of symbols, or as data. Why AP242 exists, in one chapter.*

A B-rep says what a part's nominal shape is. It does not say how far a real part may deviate from that shape and still be acceptable, which surfaces must be flat and how flat, which holes must line up with which faces and how closely. That information is **product manufacturing information**, PMI: dimensions, geometric tolerances, notes, surface texture and the other manufacturing requirements, of which geometric dimensioning and tolerancing, GD&T, is the largest part. It lived for a century on drawings, and moving it into the 3D model is the whole project of *model-based definition*. AP242 was created, more than for any other reason, to carry it.

i Engineering background: Geometric dimensioning and tolerancing

A manufactured part never matches its model exactly. GD&T is the language for saying how much mismatch is allowed, in a way that a machinist, an inspector and a designer all read the same way. Its core ideas:

Features are the physical surfaces of the part — a face, a hole, a slot — as opposed to the model's ideal geometry.

Datums are theoretically exact references derived from real features. A **datum feature** is the actual surface (the bottom face of the bracket, say), and the **datum** is the perfect plane that surface is taken to represent. Datums are labelled A, B, C.

A **datum reference frame** is an ordered set of datums — primary, secondary, tertiary, as many as the tolerance needs — that locates the part for that tolerance. *A|B|C* means: sit the part on A, slide it against B, then against C. Order matters because it decides which contact takes precedence.

A **feature control frame** is the rectangular box on a drawing that states one tolerance: a **characteristic symbol** (position, flatness, perpendicularity, and eleven others, each with its own glyph), a **tolerance zone** (a width or a diameter within which the feature's real surface or axis must lie), optional **modifiers** (the circled M, "maximum material condition," lets the zone grow as the feature departs from the size at which it holds the most material — largest for a shaft, smallest for a hole), and the **datum references** the tolerance is measured against.

Dimensions give sizes and locations, each with a plus/minus tolerance or a fit class.

Two standards say how to write all this: ASME Y14.5 in North America and the ISO GPS family (ISO 1101 and its neighbours) elsewhere. They agree on most symbols and differ in some default rules, which is why a file with semantic PMI names the standard it follows, in an `APPLIED_DOCUMENT_REFERENCE` attached to the part's product definition.

16.1 Two kinds of PMI

STEP can carry a tolerance in two ways, and the difference is the single most important thing in this chapter.

Graphical (presentation) PMI is the feature control frame as a *drawing*: the lines that make up the box, the symbol and the text, positioned in 3D space near the feature. A viewer can display it. Nothing can read it: the "0.2" is a set of line segments in the shape of digits. Every CAD system can export this, and every viewer can show it, and it carries exactly as much meaning as a photograph of a drawing.

Semantic (representation) PMI is the tolerance as *data*: a `POSITION_TOLERANCE` record with a magnitude of 0.2 millimetres, referring to a `SHAPE_ASPECT` that is linked to a specific face, with a datum system whose datums are linked to other faces. A program can read it, check it, feed it to an inspection planner. Only some systems export it and only some readers consume it. AP214 and AP203 second edition already carry the basic tolerance records; what AP242 adds is the datum system — `DATUM_SYSTEM` and

its compartments — and the tolerance modifiers; the record that links semantic data to a drawn callout, DRAUGHTING_MODEL_ITEM_ASSOCIATION, is already in all three.

A file may have either, both, or — best — both with a link from each graphical callout to the semantic data it depicts. The CAx Implementor Forum's PMI recommended practice specifies how to write all three, and the records below follow it. *(They are written to that pattern rather than exported, since the exporter used for the running example does not write PMI. They tolerance the hole in the bracket of Chapter 11, and all of them are in bracket-pmi.step, after the bracket's geometry exported as AP242; Open CASCADE's PMI reader recovers from that file every datum, tolerance and dimension described below.)*

16.2 Graphical PMI

A callout is drawn with curves collected under an annotation occurrence:

```
#600 = ANNOTATION_CURVE_OCCURRENCE('position callout',(#601),#605);
#601 = PRESENTATION_STYLE_ASSIGNMENT((#602));
#602 = CURVE_STYLE('',#603,POSITIVE_LENGTH_MEASURE(0.13),#604);
#603 = DRAUGHTING_PRE_DEFINED_CURVE_FONT('continuous');
#604 = COLOUR_RGB('',0.,0.,0.);
#605 = GEOMETRIC_CURVE_SET('position',(#606,#607));
#606 = POLYLINE('',(#620,#621,#622,#623,#620));      /* the frame's box */
#607 = POLYLINE('',(#624,#625));                     /* its first divider */
```

Referred to above: 6 records from bracket-pmi.step. The complete file.

```
#620 = CARTESIAN_POINT('',(35.,7.,10.));
#621 = CARTESIAN_POINT('',(55.,7.,10.));
#622 = CARTESIAN_POINT('',(55.,13.,10.));
#623 = CARTESIAN_POINT('',(35.,13.,10.));
#624 = CARTESIAN_POINT('',(41.,7.,10.));
#625 = CARTESIAN_POINT('',(41.,13.,10.));
```

ANNOTATION_CURVE_OCCURRENCE is a STYLED_ITEM — the same entity that carried colour in Chapter 14 — whose item is a GEOMETRIC_CURVE_SET of polylines and arcs, and whose style says how to draw them. The set's name, 'position', comes from a list in the recommended practice that says what kind of PMI is drawn; it is a label for people, not data. This frame stops at its box and the divider after the symbol. A real one carries on in the same set with the strokes of the symbol, the digits "0.2" and the datum letters, each a polyline shaped like the character. Arrowheads are filled triangles under an ANNOTATION_FILL_AREA_OCCURRENCE. In AP242 the curves may instead be a mesh under a TESSELLATED_ANNOTATION_OCCURRENCE, using the tessellation records of Chapter 13.

Several occurrences making one callout are grouped:

```
#610 = DRAUGHTING_CALLOUT('position callout',(#600,#611,#612));
```

Referred to above: 28 records from bracket-pmi.step. The complete file.

```
#600 = ANNOTATION_CURVE_OCCURRENCE('position callout',(#601),#605);
#601 = PRESENTATION_STYLE_ASSIGNMENT((#602));
#602 = CURVE_STYLE('',#603,POSITIVE_LENGTH_MEASURE(0.13),#604);
#603 = DRAUGHTING_PRE_DEFINED_CURVE_FONT('continuous');
#604 = COLOUR_RGB('',0.,0.,0.);
#605 = GEOMETRIC_CURVE_SET('position',(#606,#607));
#606 = POLYLINE('',(#620,#621,#622,#623,#620));
#607 = POLYLINE('',(#624,#625));
#608 = PRESENTATION_STYLE_ASSIGNMENT((#602,#609));
#609 = FILL_AREA_STYLE('',(#619));
#611 = ANNOTATION_CURVE_OCCURRENCE('position callout',(#601),#613);
#612 = ANNOTATION_FILL_AREA_OCCURRENCE('position callout',(#608),#632,#631);
#613 = GEOMETRIC_CURVE_SET('position',(#614));
#614 = POLYLINE('',(#626,#627));
#619 = FILL_AREA_STYLE_COLOUR('',#604);
#620 = CARTESIAN_POINT('',(35.,7.,10.));
#621 = CARTESIAN_POINT('',(55.,7.,10.));
#622 = CARTESIAN_POINT('',(55.,13.,10.));
#623 = CARTESIAN_POINT('',(35.,13.,10.));
#624 = CARTESIAN_POINT('',(41.,7.,10.));
#625 = CARTESIAN_POINT('',(41.,13.,10.));
#626 = CARTESIAN_POINT('',(35.,10.,10.));
```

```
#627 = CARTESIAN_POINT('',(33.,10.,10.));
#628 = CARTESIAN_POINT('',(34.,9.65,10.));
#629 = CARTESIAN_POINT('',(34.,10.35,10.));
#631 = CARTESIAN_POINT('',(33.5,10.,10.));
#632 = ANNOTATION_FILL_AREA('position',(#633));
#633 = POLYLINE('',(#627,#628,#629,#627));
```

— the frame, its leader line and its arrowhead, each an annotation occurrence of its own, all carrying the callout's name as the recommended practice asks. The callout lies on an annotation plane, which fixes the plane in 3D that the 2D-looking symbol is drawn in, here the plane of the top face. The practice orients it for reading: its x axis runs along the text, its y axis up the characters, and its z axis toward the reader. The plane carries a style of its own, in practice a null style meaning *draw this however you like*. A rule in the schema asks for a fill-area style with a `PLANE` and a curve style with a `PLANAR_BOX`; the null style the recommended practice prescribes satisfies neither, so files that follow the practice fail that rule:

```
#615 = ANNOTATION_PLANE('',(#618),#616,(#610));
#616 = PLANE('',#617);
#618 = PRESENTATION_STYLE_ASSIGNMENT((NULL_STYLE(.NULL.)));
```

Referred to above: 33 records from `bracket-pmi.step`. The complete file.

```
#600 = ANNOTATION_CURVE_OCCURRENCE('position callout',(#601),#605);
#601 = PRESENTATION_STYLE_ASSIGNMENT((#602));
#602 = CURVE_STYLE('',#603,POSITIVE_LENGTH_MEASURE(0.13),#604);
#603 = DRAUGHTING_PRE_DEFINED_CURVE_FONT('continuous');
#604 = COLOUR_RGB('',0.,0.,0.);
#605 = GEOMETRIC_CURVE_SET('position',(#606,#607));
#606 = POLYLINE('',(#620,#621,#622,#623,#620));
#607 = POLYLINE('',(#624,#625));
#608 = PRESENTATION_STYLE_ASSIGNMENT((#602,#609));
#609 = FILL_AREA_STYLE('',(#619));
#610 = DRAUGHTING_CALLOUT('position callout',(#600,#611,#612));
#611 = ANNOTATION_CURVE_OCCURRENCE('position callout',(#601),#613);
#612 = ANNOTATION_FILL_AREA_OCCURRENCE('position callout',(#608),#632,#631);
#613 = GEOMETRIC_CURVE_SET('position',(#614));
#614 = POLYLINE('',(#626,#627));
#617 = AXIS2_PLACEMENT_3D('',#634,#635,#636);
#619 = FILL_AREA_STYLE_COLOUR('',#604);
#620 = CARTESIAN_POINT('',(35.,7.,10.));
#621 = CARTESIAN_POINT('',(55.,7.,10.));
#622 = CARTESIAN_POINT('',(55.,13.,10.));
#623 = CARTESIAN_POINT('',(35.,13.,10.));
#624 = CARTESIAN_POINT('',(41.,7.,10.));
#625 = CARTESIAN_POINT('',(41.,13.,10.));
#626 = CARTESIAN_POINT('',(35.,10.,10.));
#627 = CARTESIAN_POINT('',(33.,10.,10.));
#628 = CARTESIAN_POINT('',(34.,9.65,10.));
#629 = CARTESIAN_POINT('',(34.,10.35,10.));
#631 = CARTESIAN_POINT('',(33.5,10.,10.));
#632 = ANNOTATION_FILL_AREA('position',(#633));
#633 = POLYLINE('',(#627,#628,#629,#627));
#634 = CARTESIAN_POINT('',(35.,7.,10.));
#635 = DIRECTION('',(0.,0.,1.));
#636 = DIRECTION('',(1.,0.,0.));
```

And every annotation plane, and through its elements every callout, is collected in one **`DRAUGHTING_MODEL`**, a representation in the same geometric context as the part:

```
#630 = DRAUGHTING_MODEL('',(#615),#204);
```

Referred to above: 41 records from `bracket-pmi.step`; the 2 it points to directly are shown. The complete file.

```
#204 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#208)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#205,#206,#207)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
#615 = ANNOTATION_PLANE('',(#618),#616,(#610));
/*  ...  and  39  more:  12  CARTESIAN_POINT,  4  POLYLINE,  3  PRESENTATION_STYLE_ASSIGNMENT,  2
ANNOTATION_CURVE_OCCURRENCE, ... */
```

The recommended practice calls this the *global* draughting model: it has an empty name and lists every annotation plane in the file, and its context, `#204`, is the bracket's own geometric context. A reader looking for "all the PMI drawn on this part" finds the draughting models that share the part's geometric context and walks their items.

That is graphical PMI: geometry that happens to look like symbols, organised so a viewer knows to draw it and where.

16.3 Semantic PMI: features

Semantic PMI is built on **shape aspects** — named regions of a part's shape that a tolerance can refer to:

```
#700 = SHAPE_ASPECT('top face','',#4,.T.);
#701 = GEOMETRIC_ITEM_SPECIFIC_USAGE('','',#700,#10,#88);
#703 = SHAPE_ASPECT('hole wall','',#4,.T.);
#705 = GEOMETRIC_ITEM_SPECIFIC_USAGE('','',#703,#10,#187);
#702 = DERIVED_SHAPE_ASPECT('hole axis','',#4,.F.);  /* its centre line */
#704 = SHAPE_ASPECT_DERIVING_RELATIONSHIP('','',#702,#703);
```

Referred to above: 206 records from `bracket-pmi.step`; the 4 it points to directly are shown. The complete file.

```
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#204);
#88 = ADVANCED_FACE('',(#89,#107),#118,.T.);
#187 = ADVANCED_FACE('',(#188),#199,.F.);
/* ... and 202 more: 33 CARTESIAN_POINT, 33 DIRECTION, 30 ORIENTED_EDGE, 15 EDGE_CURVE, ... */
```

`SHAPE_ASPECT` has a `name`, a `description`, `of_shape` (the part's `PRODUCT_DEFINITION_SHAPE` — the abstract shape from Chapter 7, now earning its keep) and a `product_definitional` flag. It says "there is a feature of this part's shape called *top face*." It does not, by itself, say which geometry that is.

`GEOMETRIC_ITEM_SPECIFIC_USAGE` does: its `definition` is the shape aspect, its `used_representation` is the shape representation, and its `identified_item` is face `#88`, the top. `#705` does the same for the hole's wall, the cylindrical face `#187`. One usage record names one item. A feature spanning several faces is written either as one shape aspect per face, or with the supertype `ITEM_IDENTIFIED_REPRESENTATION_USAGE` and a typed `SET_REPRESENTATION_ITEM(...)` holding them all. This is the record that connects a tolerance's *target* to a specific face in the B-rep, and it is where semantic PMI most often goes wrong in practice — a tolerance whose shape aspect has no `GEOMETRIC_ITEM_SPECIFIC_USAGE`, or one pointing at a face that was re-numbered, is a tolerance on nothing.

A feature made of several faces (the two halves of a slot, a pattern of holes) is a `COMPOSITE_SHAPE_ASPECT` or a `COMPOSITE_GROUP_SHAPE_ASPECT` with `SHAPE_ASPECT_RELATIONSHIP`s to its members. A hole's axis, as opposed to its wall, is a `DERIVED_SHAPE_ASPECT` with `product_definitional .F.`, tied to the wall's shape aspect by a `SHAPE_ASPECT_DERIVING_RELATIONSHIP` whose *relating* aspect is the axis. That is `#702` and `#703` above, and the schema requires at least one such relationship for every derived shape aspect. The tolerances later in this chapter refer to both. Particular derivations have their own subtypes — `APEX`, `CENTRE_OF_SYMMETRY`, `TANGENT` and others — and the recommended practices ask for a meaningful `name` string without fixing one. The vocabulary is the standard's usual: general entities, fixed strings.

16.4 Semantic PMI: datums

```
#710 = DATUM_FEATURE('','',#4,.T.);
#711 = GEOMETRIC_ITEM_SPECIFIC_USAGE('','',#710,#10,#147);   /* the bottom face */
#712 = DATUM('','',#4,.F.,'A');
#713 = SHAPE_ASPECT_RELATIONSHIP('','',#710,#712);
```

Referred to above: 206 records from `bracket-pmi.step`; the 3 it points to directly are shown. The complete file.

```
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#204);
#147 = ADVANCED_FACE('',(#148,#159),#170,.F.);
/* ... and 203 more: 33 CARTESIAN_POINT, 33 DIRECTION, 30 ORIENTED_EDGE, 15 EDGE_CURVE, ... */
```

Both `DATUM_FEATURE` and `DATUM` are subtypes of `SHAPE_ASPECT`. The datum feature is the real surface — linked to face `#147` — and the datum is the theoretical plane it establishes, carrying the letter `'A'` in an

extra `identification` attribute. The relationship says the feature establishes the datum. Datums B and C follow the same pattern, on the front face `#57` and the left face `#17`.

A datum reference frame collects them in order:

```
#720 = DATUM_SYSTEM('','',#4,.F.,(#721,#722,#723));
#721 = DATUM_REFERENCE_COMPARTMENT('',$,#4,.F.,#712,$);        /* A */
#722 = DATUM_REFERENCE_COMPARTMENT('',$,#4,.F.,#714,$);        /* B */
#723 = DATUM_REFERENCE_COMPARTMENT('',$,#4,.F.,#724,$);        /* C */
```

Referred to above: 10 records from `bracket-pmi.step`. The complete file.

```
#2 = APPLICATION_CONTEXT('Managed model based 3d engineering');
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BRACKET','bracket with 6mm hole','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#712 = DATUM('','',#4,.F.,'A');
#714 = DATUM('','',#4,.F.,'B');
#724 = DATUM('','',#4,.F.,'C');
```

Each compartment refers to one datum (or a common datum, A–B) and carries optional modifiers; the system lists its compartments in precedence order — primary first — in its `constituents` attribute. This is one of the few places in the format where the order of a list carries meaning.

16.5 Semantic PMI: geometric tolerances

A flatness tolerance on the top face:

```
#740 = FLATNESS_TOLERANCE('','',#741,#700);
#741 = LENGTH_MEASURE_WITH_UNIT(LENGTH_MEASURE(0.05),#205);
```

Referred to above: 9 records from `bracket-pmi.step`. The complete file.

```
#2 = APPLICATION_CONTEXT('Managed model based 3d engineering');
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BRACKET','bracket with 6mm hole','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#205 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#700 = SHAPE_ASPECT('top face','',#4,.T.);
```

`GEOMETRIC_TOLERANCE`, the supertype, has `name`, `description`, `magnitude` (a measure with unit: the zone width) and `toleranced_shape_aspect`. `FLATNESS_TOLERANCE` adds nothing; it is the *type* that says which characteristic this is. There are fifteen subtypes, roughly one per symbol: the schema keeps coaxiality and concentricity apart where ISO 1101 gives them a single symbol and ASME Y14.5 had only concentricity until the 2018 edition dropped it.

A position tolerance on the hole, referred to datums, with a modifier and a cylindrical zone, is where the complex instance returns:

```
#750 = ( GEOMETRIC_TOLERANCE('','',#751,#702)
         GEOMETRIC_TOLERANCE_WITH_DATUM_REFERENCE((#720))
         GEOMETRIC_TOLERANCE_WITH_MODIFIERS((.MAXIMUM_MATERIAL_REQUIREMENT.))
         POSITION_TOLERANCE() );
#751 = LENGTH_MEASURE_WITH_UNIT(LENGTH_MEASURE(0.2),#205);
#752 = TOLERANCE_ZONE('','',#4,.F.,(#750),#753);
#753 = TOLERANCE_ZONE_FORM('cylindrical or circular');
```

Referred to above: 16 records from `bracket-pmi.step`. The complete file.

```
#2 = APPLICATION_CONTEXT('Managed model based 3d engineering');
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BRACKET','bracket with 6mm hole','',(#8));
```

```
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#205 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#702 = DERIVED_SHAPE_ASPECT('hole axis','',#4,.F.);
#712 = DATUM('','',#4,.F.,'A');
#714 = DATUM('','',#4,.F.,'B');
#720 = DATUM_SYSTEM('','',#4,.F.,(#721,#722,#723));
#721 = DATUM_REFERENCE_COMPARTMENT('',$,#4,.F.,#712,$);
#722 = DATUM_REFERENCE_COMPARTMENT('',$,#4,.F.,#714,$);
#723 = DATUM_REFERENCE_COMPARTMENT('',$,#4,.F.,#724,$);
#724 = DATUM('','',#4,.F.,'C');
```

`GEOMETRIC_TOLERANCE_WITH_DATUM_REFERENCE` adds the datum system; `GEOMETRIC_TOLERANCE_WITH_MODIFIERS` adds the circled-M modifier; `POSITION_TOLERANCE` names the characteristic. Three siblings under `geometric_tolerance`, no leaf name for the combination, external mapping, alphabetical — exactly the mechanism of Chapter 4. The tolerance zone is a separate shape aspect (`TOLERANCE_ZONE`) whose form record's name comes from a fixed list that the schema enforces with a global rule: `'cylindrical or circular'` is the Ø in front of the 0.2, a flatness zone is `'between two equidistant complex surfaces or two parallel planes'`, and `'unknown'` is what a writer uses when it cannot tell.

The `toleranced_shape_aspect`, `#702`, is the hole's axis shape aspect — position is a tolerance on where the axis is, not on the wall.

16.6 Semantic PMI: dimensions

A diameter with a plus/minus tolerance:

```
#760 = DIMENSIONAL_SIZE(#703,'diameter');
#761 = DIMENSIONAL_CHARACTERISTIC_REPRESENTATION(#760,#762);
#762 = SHAPE_DIMENSION_REPRESENTATION('',(#763),#204);
#763 = ( LENGTH_MEASURE_WITH_UNIT() MEASURE_REPRESENTATION_ITEM()
         MEASURE_WITH_UNIT(LENGTH_MEASURE(6.),#205)
         QUALIFIED_REPRESENTATION_ITEM((#764))
         REPRESENTATION_ITEM('nominal value') );
#764 = VALUE_FORMAT_TYPE_QUALIFIER('NR2 1.2');
#765 = PLUS_MINUS_TOLERANCE(#766,#760);
#766 = TOLERANCE_VALUE(#767,#768);
#767 = LENGTH_MEASURE_WITH_UNIT(LENGTH_MEASURE(-0.02),#205);
#768 = LENGTH_MEASURE_WITH_UNIT(LENGTH_MEASURE(0.05),#205);
```

Referred to above: 13 records from `bracket-pmi.step`. The complete file.

```
#2 = APPLICATION_CONTEXT('Managed model based 3d engineering');
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BRACKET','bracket with 6mm hole','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#204 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#208)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#205,#206,#207)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
#205 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#206 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#207 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#208 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#205,
  'distance_accuracy_value','confusion accuracy');
#703 = SHAPE_ASPECT('hole wall','',#4,.T.);
```

`DIMENSIONAL_SIZE` says the hole's wall (`#703`) has a size of kind `'diameter'`. Its value is a representation — the property pattern of Chapter 15 again — whose item is a nominal length of 6 mm, qualified with a format string (`'NR2 1.2'`: one digit before the decimal point, two after) that says how to display it. `PLUS_MINUS_TOLERANCE` attaches a lower and upper deviation. A distance between two features is a `DIMENSIONAL_LOCATION` with two shape aspects instead of one; a fit class is a `LIMITS_AND_FITS` record with grade and deviation strings.

16.7 Linking the two

A graphical callout can point at the semantic data it depicts:

```
#780 = DRAUGHTING_MODEL_ITEM_ASSOCIATION('PMI representation to presentation link',
         '',#750,#630,#610);
```

Referred to above: 59 records from `bracket-pmi.step`; the 3 it points to directly are shown. The complete file.

```
#610 = DRAUGHTING_CALLOUT('position callout',(#600,#611,#612));
#630 = DRAUGHTING_MODEL('',(#615),#204);
#750 = ( GEOMETRIC_TOLERANCE('','',#751,#702)
  GEOMETRIC_TOLERANCE_WITH_DATUM_REFERENCE((#720))
  GEOMETRIC_TOLERANCE_WITH_MODIFIERS((.MAXIMUM_MATERIAL_REQUIREMENT.))
  POSITION_TOLERANCE() );
/* ... and 56 more: 12 CARTESIAN_POINT, 4 POLYLINE, 3 DATUM, 3 DATUM_REFERENCE_COMPARTMENT, ... */
```

— *the callout `#610`, in draughting model `#630`, presents tolerance `#750`*. The draughting model named here is always the global one. A reader can then show the picture and know what it means, or check that every semantic tolerance has a picture. The recommended practice prescribes the name string. Without this record the two halves of the PMI are unrelated, and a file can perfectly well have a semantic flatness tolerance and a graphical callout that happen to describe the same thing with no connection between them.

16.8 Saved views

Drawings have views; models with PMI have *saved views*: a camera position and the set of annotations visible from it, so that a receiving system can show the PMI the way the author arranged it rather than all at once.

```
#800 = DRAUGHTING_MODEL('view content 1',(#801,#610,#615),#204);
#801 = CAMERA_MODEL_D3('Top',#802,#803);
#802 = AXIS2_PLACEMENT_3D('',#804,#805,#806);  /* the eye; it looks along its z axis */
#803 = VIEW_VOLUME(.PARALLEL.,#807,100.,0.,.F.,0.,.F.,.F.,#808);
#809 = MODEL_GEOMETRIC_VIEW('','',#801,#800);
#814 = MECHANICAL_DESIGN_AND_DRAUGHTING_RELATIONSHIP('','',#800,#630);
```

Referred to above: 51 records from `bracket-pmi.step`; the 9 it points to directly are shown. The complete file.

```
#204 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#208)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#205,#206,#207)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
#610 = DRAUGHTING_CALLOUT('position callout',(#600,#611,#612));
#615 = ANNOTATION_PLANE('',(#618),#616,(#610));
#630 = DRAUGHTING_MODEL('',(#615),#204);
#804 = CARTESIAN_POINT('',(30.,10.,110.));
#805 = DIRECTION('',(0.,0.,-1.));
#806 = DIRECTION('',(-1.,0.,0.));
#807 = CARTESIAN_POINT('',(0.,0.,0.));
#808 = PLANAR_BOX('',80.,40.,#810);
/* ... and 42 more: 13 CARTESIAN_POINT, 4 DIRECTION, 4 POLYLINE, 3 PRESENTATION_STYLE_ASSIGNMENT, ... */
```

A saved view is a `DRAUGHTING_MODEL` paired with a `CAMERA_MODEL_D3` among its items: the model says what is shown, the camera says from where. The recommended practice puts the view's *name* on the camera, `'Top'`, and uses the draughting model's name for the set of things shown. AP242 also has `MODEL_GEOMETRIC_VIEW`, which ties the camera to its draughting model explicitly; the NIST test files that write it leave its name empty, so a reader should look for the view's name on the camera. The camera's `view_reference_system` is the eye: its location is where the camera stands, here 100 mm above the bracket, and its z axis is the direction of view, here straight down. The `VIEW_VOLUME` gives the rest, and the recommended practice fixes most of it: a parallel projection, the projection point `#807` at the camera's own origin, no front or back clipping, and a view-plane distance equal to the distance from the eye to what it looks at, 100 mm down to the top face. In a parallel projection that distance does not set the zoom; the window does. Its window is a `PLANAR_BOX`, 80 mm by 40, centred on the line of sight. The part's geometry would appear in the view through a `MAPPED_ITEM` of its shape representation among the model's items; this view shows only the callout. The same callout may appear in several views. `#814` relates the view to the

global draughting model, with the view in `rep_1` and the global model in `rep_2`: the direction Part 43 gives the attributes, and the one the recommended practice settled on in its version 4.0, after earlier versions had them the other way round. `MECHANICAL_DESIGN_AND_DRAUGHTING_RELATIONSHIP` does not exist in AP214, where a plain `REPRESENTATION_RELATIONSHIP` does the job. This is why a viewer sometimes shows six annotations out of ninety: it has opened the first view.

⚠ In the wild: What actually arrives

Every mainstream CAD system exports graphical PMI when asked, and most produce it as polylines; the tessellated form is less common. Semantic PMI is exported by NX, Creo, CATIA and SolidWorks in their current versions, with coverage that varies by tolerance type and is documented, system by system, in the CAx Implementor Forum's implementation coverage matrix. NIST's MBE PMI test models — a set of parts with known PMI and published answer keys — are the standard way to find out what a particular exporter gets right. It is normal for a file to have complete graphical PMI and partial semantic PMI: the graphical side is drawn from the CAD system's display and is complete unless the export drops annotations, the semantic side requires the system to have modelled the tolerance as data in the first place, and many drawings-first workflows never did.

! In brief

Graphical PMI is symbols drawn as curves: `ANNOTATION_CURVE_OCCURRENCE`s grouped into `DRAUGHTING_CALLOUT`s on `ANNOTATION_PLANE`s, collected in a `DRAUGHTING_MODEL`. Semantic PMI is data: `SHAPE_ASPECT`s linked to faces by `GEOMETRIC_ITEM_SPECIFIC_USAGE`; `DATUM_FEATURE` → `DATUM` → `DATUM_SYSTEM`; `GEOMETRIC_TOLERANCE` subtypes (complex instances when they carry datums or modifiers) with a magnitude and a target; `DIMENSIONAL_SIZE/LOCATION` with a nominal value and a `PLUS_MINUS_TOLERANCE`. `DRAUGHTING_MODEL_ITEM_ASSOCIATION` links a picture to its meaning; saved views are draughting models with a camera.

17. Application Protocols and the Real World

> *What differs between an AP203, an AP214 and an AP242 file at the level of records; what "conformance class" means and why vendors care; and how to read a file's origin from its habits.*

Most of this book has been written against records that are identical across the three mechanical application protocols; presentation, PMI and tessellation were the first places where the protocol mattered. The block's geometry is the same in all of them; so is its product chain, its units, its assembly structure. This chapter is about where they differ, and about the gap between what a protocol permits and what any given exporter writes.

17.1 Three protocols, one lineage

	AP203 ed. 1 (1994)	AP214 (2001, 2003, 2010)	AP242 (2014, 2020, 2022, 2025)
Schema name	`CONFIG_CONTROL_DESIGN`	`AUTOMOTIVE_DESIGN`	`AP242_MANAGED_MODEL_BASED_3D_ENGINEERING_MIM_LF`
Product structure, B-rep	yes	yes	yes
Colours, layers	**no**	yes	yes
Graphical PMI	no	yes (drawing-oriented)	yes
Semantic PMI	no	limited	**yes**
Tessellation	no	no	**yes**
Validation properties	by convention	by convention	yes, incl. PMI and tessellation
Kinematics, composites	no	kinematics	both, extended
PDM (XML)	no	no	business object model

AP203′s second edition (2011) adopted the modular architecture and added presentation, closing most of the gap to AP214; its schema name is `AP203_CONFIGURATION_CONTROLLED_3D_DESIGN_OF_MECHANICAL_PARTS_AND_ASSEMBLIES_MIM_LF`, and it is comparatively rare in the wild. AP242 was designed as the successor to both and is a superset of each for everything a CAD system exports.

The practical consequences for a reader:

- A file declaring **AP203 first edition** has geometry, product structure and configuration-management data — people, approvals, security classification — and no presentation. If it contains colour records, the writer violated its own declaration.
- A file declaring **AP214** may have everything in Parts I and II of this book plus colours, layers, graphical PMI and user properties. Its semantic PMI, if any, uses an older and less complete set of entities.
- A file declaring **AP242** may have all of that plus tessellation, full semantic PMI, saved views, and the AP242-specific validation properties.

And for a writer: if colours or PMI matter, write AP242. Writing AP214 for colour is still common and still works; writing AP203 first edition for anything beyond geometry and product structure is a mistake that many exporters' default settings make.

17.2 Conformance classes

An AP is large, and no implementation supports all of it. The standard therefore defines **conformance classes**: named subsets an implementation can claim. AP203′s original classes were six shape levels — configuration-controlled design without shape, then wireframe, wireframe with topology, manifold surfaces, faceted B-rep and advanced B-rep — each in two variants, twelve in all. AP214 has twenty. AP242′s are organised by capability — geometry, assembly, PMI, tessellation — and a vendor's export dialog often exposes them as options.

For a reader of files the classes matter in one way: they tell you what the *writer* claims to support. A CAD vendor who says their exporter conforms to AP242 "with semantic PMI" is claiming that scope, and the CAx-IF test rounds are where such claims are checked; the claim alone does not mean every file will carry the records of Chapter 16, which need the tolerances to have been modelled and the export option switched on. A vendor who does not make that claim may still write them, or may write some of them, or may write them wrongly.

17.3 Recommended practices

The standard says what is legal. It does not say, for most things, what is *good*. Two exporters can write the same assembly placement with `rep_1` and `rep_2` swapped, both legally, and a reader has to handle both. The gap is filled by the **recommended practices** of the CAx Implementor Forum (CAx-IF), a joint body of CAD vendors and users that has run interoperability test rounds since the 1990s and publishes, freely, documents that pin down how to write each capability so that it will actually be read: assembly structure, colours and layers, user-defined attributes, validation properties, PMI (graphical and semantic), tessellation, external references.

Several things in this book are recommended practice rather than standard: the string `'geometric validation property'`, the names `'volume'`, `'surface area'` and `'centroid'`, the orientation of `rep_1` and `rep_2`, the `'PMI representation to presentation link'` association name. The tolerance zone form names are not among them: AP242 fixes that list with a global rule, as Chapter 16 says. A reader that follows the recommended practices reads every mainstream exporter's files; a reader that follows only the standard reads none of them reliably. The documents are at the forum's site (now part of the MBx Interoperability Forum), and for anyone working with real files they are more useful than the standard itself.

17.4 Reading a file's origin from its habits

With the header's `originating_system` as confirmation, each exporter has a signature. What follows is characteristic rather than guaranteed; versions change.

Open CASCADE (and FreeCAD, CadQuery, and many small tools): `FACE_BOUND` rather than `FACE_OUTER_BOUND` for every bound; duplicated `CARTESIAN_POINT`s for line origins; one surface record shared between a face and the pcurves on it; product names defaulting to `Open CASCADE STEP translator N`; one geometric context per product; `-0.` in directions; category `'part'` for assemblies too.

SolidWorks: product `id` and `name` from the filename; `FACE_OUTER_BOUND` used; colours per face when face colours exist, otherwise per body; a `SHAPE_REPRESENTATION` with only an axis placement for each part alongside the B-rep representation.

NX (via ST-Developer): very clean, spec-conformant structure; `FACE_OUTER_BOUND`; shared points; product number in `id`; comprehensive validation properties; the best semantic PMI coverage of the mainstream systems.

CATIA V5: `PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE` with `.NOT_KNOWN.`; older versions converting many surfaces to B-splines; per-face colours common; `MAPPED_ITEM` used in some assembly exports.

Creo: historically prone to writing many small surfaces where one would do; product names from the part name; assembly structure via `CONTEXT_DEPENDENT_SHAPE_REPRESENTATION` as in this book.

None of this changes what the records *mean*. It changes what a reader will find, in what quantity, and where.

17.5 When files are wrong

Three kinds of error, in increasing order of difficulty.

Syntactic. A string not closed, a record not terminated, a keyword in lower case, a real without a decimal point. Rare from CAD exporters, common from scripts and hand edits. Most readers reject these; some repair the obvious ones.

Schema. An entity that does not exist in the declared schema; too many or too few parameters; a string where a reference belongs. These come from exporters that mislabel the AP, from software that edits files without understanding them, and from bugs. Readers vary from strict rejection to silent acceptance of anything they recognise.

Semantic. Everything is legal and the model is wrong: a shell that is not closed, a face whose loop does not chain, a placement with the frames reversed, a validation property that does not match the geometry, a tolerance on a shape aspect that points at nothing. These are the errors that matter and the only ones the format cannot catch by inspection. They are why validation properties exist, why validators such as NIST's STEP File Analyzer exist, and why "opens fine in my CAD system" is not the same as "correct."

⚠ In the wild: Test files

Two public resources are worth knowing. The **CAx-IF** publishes its test suite definitions and, per release, an implementation coverage matrix saying which capabilities each participating CAD system supports; the test files themselves stay inside the Implementor Group. **NIST's MBE PMI validation models** are a set of parts designed to exercise semantic PMI, with published drawings, expected results, and exports of each from several CAD systems, and they are free to download. Between them they answer most "does system X write Y correctly?" questions without guesswork.

! In brief

AP203, AP214 and AP242 share geometry and structure; they differ in presentation, PMI and tessellation, with AP242 a superset of both older protocols. Conformance classes name what an implementation claims to support; the CAx-IF test rounds are where the claims are checked. The recommended practices of the CAx Implementor Forum, not the standard, decide how real files are written, and a reader must follow them. Exporters have recognisable habits; none change the meaning of the records.

18. Beyond Edition 2

> *What the 2016 edition of the file format added — anchors, external references, signatures, and a container — and the other ways the same data can be serialised.*

Every file in this book declares `'2;1'`, which says it uses nothing beyond what edition 1 of Part 21 defined. That is true of essentially all the STEP files in existence, whatever edition the writer worked from. The third edition, published in 2016, kept everything and added UTF-8 encoding and a set of optional features, all aimed at the same problem: a Part 21 file is an island. Its instance names mean nothing outside it, it cannot refer to anything in another file, and it cannot be split.

18.1 What edition 3 adds

UTF-8. Edition 2 strings were nominally 8-bit ISO 8859 with escape sequences for anything else; edition 3 makes UTF-8 the native encoding. Most writers had already switched. A file declaring `'4;1'` may contain `'Ångström'` directly; a file declaring `'2;1'` or `'3;1'` must escape every character above ASCII, as `'\X2\00C5\X0\ngstr\X2\00F6\X0\m'`, and usually does not. Writers that emit UTF-8 while still declaring `'2;1'` are common, and are strictly wrong.

Anchors. A new optional section between `HEADER` and `DATA` gives stable names to instances:

```
ANCHOR;
<clamp> = #5;
<bracket> = #35;
ENDSEC;
```

An anchor is a name in angle brackets bound to an instance. Instance names are file-local and change whenever a file is rewritten; anchors are the file's public interface, and a URI of the form `clamp.stp#bracket` means "the instance that file calls *bracket*" regardless of what number it has today.

References. The mirror of anchors: a section that declares instance names whose definitions are *elsewhere*.

```
REFERENCE;
#35 = <bracket.stp#bracket>;
#243 = <pin.stp#pin>;
ENDSEC;
```

Inside this file, `#35` and `#243` may be used like any other instance — a `NEXT_ASSEMBLY_USAGE_OCCURRENCE` can name `#35` as its child — but they are not defined here. They resolve to anchors in other files. This is how an assembly can be one file per part with a master file holding only the structure, which is how every PDM system wants to store it and what edition 2 could not express without the workaround described below.

Signatures. An optional trailing section carrying a digital signature over the file's contents, for provenance.

A container. Edition 3 also defines a ZIP archive holding the exchange structure together with the files it references, so that a multi-file assembly can travel as one object. The standard requires the archive's root file to be named `ISO-10303.p21`. In practice the CAx-IF conventions are what implementations follow: `.stpZ` for a single compressed Part 21 file and `.stpA` for a multi-file archive, whose root files are named in an `stpa.ini` at the top of the archive.

A file using any of these declares `'4;1'`, `'4;2'` or `'4;3'` in `FILE_DESCRIPTION` — class 2 if it has a reference section or a multi-file ZIP archive, class 3 if it defines value instances, references EXPRESS constants or carries ECMAScript — and the header gains an optional `SCHEMA_POPULATION` record that lists the external files whose data completes this one.

18.2 How much of this is in use

Less than one might expect, a decade on. Anchors and references are supported by some PDM-oriented exporters and by the major toolkits, but the CAx Implementor Forum's recommended practice for external references is still the edition 2 mechanism described below, and it has not been revised since 2014. But the overwhelming majority of files exchanged between CAD systems are still single, self-contained files that use nothing edition 3 added, because that is what every reader accepts and because the AP242 XML described below took over much of the multi-file use case. A reader that does not implement edition 3 will meet files it cannot read; a reader that implements it will use the capability rarely.

18.3 The edition 2 workaround for external files

Before anchors, a master file referred to a part in another file through the document entities of Part 41 and the AP's own file-reference extensions to them:

```
#900 = DOCUMENT_FILE('bracket','',$,#901,'',$);
#901 = DOCUMENT_TYPE('geometry');
#902 = DOCUMENT_REPRESENTATION_TYPE('digital',#900);
#903 = APPLIED_DOCUMENT_REFERENCE(#900,'',(#35));
#904 = IDENTIFICATION_ROLE('external document id and location',$);
#905 = APPLIED_EXTERNAL_IDENTIFICATION_ASSIGNMENT('bracket.stp',#904,#906,(#900));
#906 = EXTERNAL_SOURCE(IDENTIFIER('./parts/'));
```

Referred to above: 6 records from `clamp.step`. *The complete file.*

```
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#35 = PRODUCT_DEFINITION('design','',#36,#39);
#36 = PRODUCT_DEFINITION_FORMATION('','',#37);
#37 = PRODUCT('bracket','bracket','',(#38));
#38 = PRODUCT_CONTEXT('',#2,'mechanical');
#39 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
```

A `DOCUMENT_FILE` stands for the external file (six parameters, because it is both a `document` and a `characterized_object`, flattened), and an `APPLIED_DOCUMENT_REFERENCE` attaches it to the product definition whose geometry it contains. The `DOCUMENT_REPRESENTATION_TYPE` is not decoration: a rule on `document_file` requires exactly one, naming `'digital'` or `'physical'`. The document type is free text; `'geometry'` is what the recommended practice uses for an external shape file. There is a `DOCUMENT_PRODUCT_EQUIVALENCE` as well, but only when the document itself is managed as a product, which CAD exporters do not do. The master file carries the product chains and NAUOs for the whole assembly and no geometry; each part file carries one product and its shape. The filename travels on an `APPLIED_EXTERNAL_IDENTIFICATION_ASSIGNMENT` attached to the `DOCUMENT_FILE`, and the directory on the `EXTERNAL_SOURCE` it names. The `IDENTIFICATION_ROLE` is what says the assignment carries a filename and location rather than some other identifier, and the recommended practice fixes its name as `'external document id and location'`. The document's own id often repeats the filename — here it is only `'bracket'` — and a reader falls back to it when the assignment is absent. There is no anchor, so the reader must then find the right `PRODUCT_DEFINITION` in the external file by matching `product.id`, `product_definition_formation.id` and `product_definition.id`. This is fragile and it is what AP214 multi-file exports actually did.

18.4 Other encodings

Part 21 is one serialisation of an EXPRESS population. The standard defines others, and two are in use.

Part 28: XML. Any EXPRESS schema maps to an XML schema and any Part 21 file to an XML document, mechanically: each instance becomes an element named for its entity, attributes become child elements or attributes, references become ID/IDREF. The result is several times larger than Part 21 and no more readable. It is not used for CAD geometry.

The AP242 XML business object model. AP242 defines, alongside its EXPRESS schema, a separate XML schema for the *product data management* half of its scope: product structure, versions, effectivity, documents, properties, external geometry references. An AP242 XML file describes an assembly's structure and points to Part 21 files for the parts' shapes. This is the form PDM system integrations use, and it is why anchors and references are less needed than they might have been: the multi-file case moved to XML.

Part 26: binary. A mapping of EXPRESS populations to HDF5, for very large datasets. Defined, implemented in a few toolkits, essentially unused in mechanical exchange.

JSON encodings have been proposed and prototyped by several groups; none is part of the standard as this is written.

18.5 The shape of things

A reader coming from other formats often expects STEP to evolve toward something lighter — binary geometry, a compact mesh format, a modern schema language. It has not, and the reasons are instructive. The verbosity of Part 21 is the price of being self-describing and hand-readable, which is what has kept thirty-year-old files usable. The generality of the integrated resources is the price of one standard serving shipbuilding and aerospace and electronics, which is what has kept the vendors at the table. What *has* changed is at the edges: AP242 added tessellation so that STEP could carry the lightweight form other formats specialise in; edition 3 added the multi-file machinery; the XML business model gave PDM systems the interface they wanted. The core — the Part 21 file, the EXPRESS schema, the B-rep in Part 42 — is the same as it was in 1994, and a reader that understands the block file in Appendix A will understand the files being written in 2040.

! In brief

Edition 3 (2016) adds UTF-8, an `ANCHOR` section naming instances for external use, a `REFERENCE` section importing instances from other files by URI, a trailing signature section, and a ZIP container; a file using them declares `'4;1'` or higher. Edition 2 files refer to external parts by filename, carried on an identification assignment attached to a `DOCUMENT_FILE`. The AP242 XML business object model carries product structure for PDM with geometry left in Part 21; Part 28 XML and Part 26 binary exist and are little used.

Appendices

A. The Block File, Complete

The running example of Part I and Chapter 10, exactly as written by Open CASCADE Technology 7.9 with parameter curves suppressed, apart from the product id and name, and the file name, author and organisation fields in the header. One hundred and seventy records. Open CASCADE wraps long records at commas to keep lines within 72 columns, except in the header, where the `FILE_NAME` strings leave it no break point.

```
ISO-10303-21;
HEADER;
FILE_DESCRIPTION(('Open CASCADE Model'),'2;1');
FILE_NAME('block.step','2026-09-17T16:14:23',('J.       Chultarsky'),('Physna'),'Open
CASCADE STEP processor 7.9','Open CASCADE 7.9'
  ,'Unknown');
FILE_SCHEMA(('AUTOMOTIVE_DESIGN { 1 0 10303 214 1 1 1 1 }'));
ENDSEC;
DATA;
#1 = APPLICATION_PROTOCOL_DEFINITION('international standard',
  'automotive_design',2000,#2);
#2 = APPLICATION_CONTEXT(
  'core data for automotive mechanical design processes');
#3 = SHAPE_DEFINITION_REPRESENTATION(#4,#10);
#4 = PRODUCT_DEFINITION_SHAPE('','',#5);
#5 = PRODUCT_DEFINITION('design','',#6,#9);
#6 = PRODUCT_DEFINITION_FORMATION('','',#7);
#7 = PRODUCT('BLOCK','block 60x20x10','',(#8));
#8 = PRODUCT_CONTEXT('',#2,'mechanical');
#9 = PRODUCT_DEFINITION_CONTEXT('part definition',#2,'design');
#10 = ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#15),#165);
#11 = AXIS2_PLACEMENT_3D('',#12,#13,#14);
#12 = CARTESIAN_POINT('',(0.,0.,0.));
#13 = DIRECTION('',(0.,0.,1.));
#14 = DIRECTION('',(1.,0.,-0.));
#15 = MANIFOLD_SOLID_BREP('',#16);
#16 = CLOSED_SHELL('',(#17,#57,#97,#119,#141,#153));
#17 = ADVANCED_FACE('',(#18),#52,.F.);
#18 = FACE_BOUND('',#19,.F.);
#19 = EDGE_LOOP('',(#20,#30,#38,#46));
#20 = ORIENTED_EDGE('',*,*,#21,.F.);
#21 = EDGE_CURVE('',#22,#24,#26,.T.);
#22 = VERTEX_POINT('',#23);
#23 = CARTESIAN_POINT('',(0.,0.,0.));
#24 = VERTEX_POINT('',#25);
#25 = CARTESIAN_POINT('',(0.,0.,10.));
#26 = LINE('',#27,#28);
#27 = CARTESIAN_POINT('',(0.,0.,0.));
#28 = VECTOR('',#29,1.);
#29 = DIRECTION('',(0.,0.,1.));
#30 = ORIENTED_EDGE('',*,*,#31,.T.);
#31 = EDGE_CURVE('',#22,#32,#34,.T.);
#32 = VERTEX_POINT('',#33);
#33 = CARTESIAN_POINT('',(0.,20.,0.));
#34 = LINE('',#35,#36);
#35 = CARTESIAN_POINT('',(0.,0.,0.));
#36 = VECTOR('',#37,1.);
```

```
#37 = DIRECTION('',(0.,1.,0.));
#38 = ORIENTED_EDGE('',*,*,#39,.T.);
#39 = EDGE_CURVE('',#32,#40,#42,.T.);
#40 = VERTEX_POINT('',#41);
#41 = CARTESIAN_POINT('',(0.,20.,10.));
#42 = LINE('',#43,#44);
#43 = CARTESIAN_POINT('',(0.,20.,0.));
#44 = VECTOR('',#45,1.);
#45 = DIRECTION('',(0.,0.,1.));
#46 = ORIENTED_EDGE('',*,*,#47,.F.);
#47 = EDGE_CURVE('',#24,#40,#48,.T.);
#48 = LINE('',#49,#50);
#49 = CARTESIAN_POINT('',(0.,0.,10.));
#50 = VECTOR('',#51,1.);
#51 = DIRECTION('',(0.,1.,0.));
#52 = PLANE('',#53);
#53 = AXIS2_PLACEMENT_3D('',#54,#55,#56);
#54 = CARTESIAN_POINT('',(0.,0.,0.));
#55 = DIRECTION('',(1.,0.,0.));
#56 = DIRECTION('',(0.,0.,1.));
#57 = ADVANCED_FACE('',(#58),#92,.T.);
#58 = FACE_BOUND('',#59,.T.);
#59 = EDGE_LOOP('',(#60,#70,#78,#86));
#60 = ORIENTED_EDGE('',*,*,#61,.F.);
#61 = EDGE_CURVE('',#62,#64,#66,.T.);
#62 = VERTEX_POINT('',#63);
#63 = CARTESIAN_POINT('',(60.,0.,0.));
#64 = VERTEX_POINT('',#65);
#65 = CARTESIAN_POINT('',(60.,0.,10.));
#66 = LINE('',#67,#68);
#67 = CARTESIAN_POINT('',(60.,0.,0.));
#68 = VECTOR('',#69,1.);
#69 = DIRECTION('',(0.,0.,1.));
#70 = ORIENTED_EDGE('',*,*,#71,.T.);
#71 = EDGE_CURVE('',#62,#72,#74,.T.);
#72 = VERTEX_POINT('',#73);
#73 = CARTESIAN_POINT('',(60.,20.,0.));
#74 = LINE('',#75,#76);
#75 = CARTESIAN_POINT('',(60.,0.,0.));
#76 = VECTOR('',#77,1.);
#77 = DIRECTION('',(0.,1.,0.));
#78 = ORIENTED_EDGE('',*,*,#79,.T.);
#79 = EDGE_CURVE('',#72,#80,#82,.T.);
#80 = VERTEX_POINT('',#81);
#81 = CARTESIAN_POINT('',(60.,20.,10.));
#82 = LINE('',#83,#84);
#83 = CARTESIAN_POINT('',(60.,20.,0.));
#84 = VECTOR('',#85,1.);
#85 = DIRECTION('',(0.,0.,1.));
#86 = ORIENTED_EDGE('',*,*,#87,.F.);
#87 = EDGE_CURVE('',#64,#80,#88,.T.);
#88 = LINE('',#89,#90);
#89 = CARTESIAN_POINT('',(60.,0.,10.));
#90 = VECTOR('',#91,1.);
#91 = DIRECTION('',(0.,1.,0.));
#92 = PLANE('',#93);
#93 = AXIS2_PLACEMENT_3D('',#94,#95,#96);
#94 = CARTESIAN_POINT('',(60.,0.,0.));
#95 = DIRECTION('',(1.,0.,0.));
#96 = DIRECTION('',(0.,0.,1.));
```

```
#97 = ADVANCED_FACE('',(#98),#114,.F.);
#98 = FACE_BOUND('',#99,.F.);
#99 = EDGE_LOOP('',(#100,#106,#107,#113));
#100 = ORIENTED_EDGE('',*,*,#101,.F.);
#101 = EDGE_CURVE('',#22,#62,#102,.T.);
#102 = LINE('',#103,#104);
#103 = CARTESIAN_POINT('',(0.,0.,0.));
#104 = VECTOR('',#105,1.);
#105 = DIRECTION('',(1.,0.,0.));
#106 = ORIENTED_EDGE('',*,*,#21,.T.);
#107 = ORIENTED_EDGE('',*,*,#108,.T.);
#108 = EDGE_CURVE('',#24,#64,#109,.T.);
#109 = LINE('',#110,#111);
#110 = CARTESIAN_POINT('',(0.,0.,10.));
#111 = VECTOR('',#112,1.);
#112 = DIRECTION('',(1.,0.,0.));
#113 = ORIENTED_EDGE('',*,*,#61,.F.);
#114 = PLANE('',#115);
#115 = AXIS2_PLACEMENT_3D('',#116,#117,#118);
#116 = CARTESIAN_POINT('',(0.,0.,0.));
#117 = DIRECTION('',(0.,1.,0.));
#118 = DIRECTION('',(0.,0.,1.));
#119 = ADVANCED_FACE('',(#120),#136,.T.);
#120 = FACE_BOUND('',#121,.T.);
#121 = EDGE_LOOP('',(#122,#128,#129,#135));
#122 = ORIENTED_EDGE('',*,*,#123,.F.);
#123 = EDGE_CURVE('',#32,#72,#124,.T.);
#124 = LINE('',#125,#126);
#125 = CARTESIAN_POINT('',(0.,20.,0.));
#126 = VECTOR('',#127,1.);
#127 = DIRECTION('',(1.,0.,0.));
#128 = ORIENTED_EDGE('',*,*,#39,.T.);
#129 = ORIENTED_EDGE('',*,*,#130,.T.);
#130 = EDGE_CURVE('',#40,#80,#131,.T.);
#131 = LINE('',#132,#133);
#132 = CARTESIAN_POINT('',(0.,20.,10.));
#133 = VECTOR('',#134,1.);
#134 = DIRECTION('',(1.,0.,0.));
#135 = ORIENTED_EDGE('',*,*,#79,.F.);
#136 = PLANE('',#137);
#137 = AXIS2_PLACEMENT_3D('',#138,#139,#140);
#138 = CARTESIAN_POINT('',(0.,20.,0.));
#139 = DIRECTION('',(0.,1.,0.));
#140 = DIRECTION('',(0.,0.,1.));
#141 = ADVANCED_FACE('',(#142),#148,.F.);
#142 = FACE_BOUND('',#143,.F.);
#143 = EDGE_LOOP('',(#144,#145,#146,#147));
#144 = ORIENTED_EDGE('',*,*,#31,.F.);
#145 = ORIENTED_EDGE('',*,*,#101,.T.);
#146 = ORIENTED_EDGE('',*,*,#71,.T.);
#147 = ORIENTED_EDGE('',*,*,#123,.F.);
#148 = PLANE('',#149);
#149 = AXIS2_PLACEMENT_3D('',#150,#151,#152);
#150 = CARTESIAN_POINT('',(0.,0.,0.));
#151 = DIRECTION('',(0.,0.,1.));
#152 = DIRECTION('',(1.,0.,0.));
#153 = ADVANCED_FACE('',(#154),#160,.T.);
#154 = FACE_BOUND('',#155,.T.);
#155 = EDGE_LOOP('',(#156,#157,#158,#159));
#156 = ORIENTED_EDGE('',*,*,#47,.F.);
```

```
#157 = ORIENTED_EDGE('',*,*,#108,.T.);
#158 = ORIENTED_EDGE('',*,*,#87,.T.);
#159 = ORIENTED_EDGE('',*,*,#130,.F.);
#160 = PLANE('',#161);
#161 = AXIS2_PLACEMENT_3D('',#162,#163,#164);
#162 = CARTESIAN_POINT('',(0.,0.,10.));
#163 = DIRECTION('',(0.,0.,1.));
#164 = DIRECTION('',(1.,0.,0.));
#165 = ( GEOMETRIC_REPRESENTATION_CONTEXT(3)
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#169)) GLOBAL_UNIT_ASSIGNED_CONTEXT
((#166,#167,#168)) REPRESENTATION_CONTEXT('Context #1',
  '3D Context with UNIT and UNCERTAINTY') );
#166 = ( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#167 = ( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#168 = ( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
#169 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-07),#166,
  'distance_accuracy_value','confusion accuracy');
#170 = PRODUCT_RELATED_PRODUCT_CATEGORY('part',$,(#7));
ENDSEC;
END-ISO-10303-21;
```

B. Entity Quick Reference

The entities that appear in typical mechanical files, with their attributes in file order (inherited attributes included, derived ones marked *) and the chapter where each is discussed. Types in italics are shown with only the attributes their own block contributes to a complex instance, not with inherited attributes flattened in.

B.1 Header

Entity	Attributes, in order	Where
FILE_DESCRIPTION	description (list), implementation_level	Chapter 3
FILE_NAME	name, time_stamp, author (list), organization (list), preprocessor_version, originating_system, authorization	Chapter 3
FILE_SCHEMA	schema_identifiers (list)	Chapter 3

B.2 Context and identity

Entity	Attributes, in order	Where
APPLICATION_CONTEXT	application	Chapter 3, Chapter 5
APPLICATION_PROTOCOL_DEFINITION	status, application_interpreted_model_schema_name, application_protocol_year, application	Chapter 3
PRODUCT_CONTEXT	name, frame_of_reference, discipline_type	Chapter 5
PRODUCT_DEFINITION_CONTEXT	name, frame_of_reference, life_cycle_stage	Chapter 5
PRODUCT	id, name, description, frame_of_reference (set)	Chapter 5
PRODUCT_DEFINITION_FORMATION	id, description, of_product	Chapter 5
PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE	id, description, of_product, make_or_buy	Chapter 5
PRODUCT_DEFINITION	id, description, formation, frame_of_reference	Chapter 5
PRODUCT_RELATED_PRODUCT_CATEGORY	name, description, products (set)	Chapter 5

B.3 Shape binding

Entity	Attributes, in order	Where
PRODUCT_DEFINITION_SHAPE	name, description, definition	Chapter 7
SHAPE_DEFINITION_REPRESENTATION	definition, used_representation	Chapter 7
SHAPE_REPRESENTATION and subtypes	name, items (set), context_of_items	Chapter 7
AXIS2_PLACEMENT_3D	name, location, axis, ref_direction	Chapter 7, Chapter 8

B.4 Assembly

Entity	Attributes, in order	Where
NEXT_ASSEMBLY_USAGE_OCCURRENCE	id, name, description, relating_product_definition, related_product_definition, reference_designator	Chapter 8
CONTEXT_DEPENDENT_SHAPE_REPRESENTATION	representation_relation, represented_product_relation	Chapter 8
REPRESENTATION_RELATIONSHIP	name, description, rep_1, rep_2	Chapter 8
REPRESENTATION_RELATIONSHIP_WITH_TRANSFORMATION	transformation_operator	Chapter 8
SHAPE_REPRESENTATION_RELATIONSHIP	—	Chapter 8
ITEM_DEFINED_TRANSFORMATION	name, description, transform_item_1, transform_item_2	Chapter 8
MAPPED_ITEM	name, mapping_source, mapping_target	Chapter 8
REPRESENTATION_MAP	mapping_origin, mapped_representation	Chapter 8

B.5 Units and context

Entity	Attributes, in order	Where
REPRESENTATION_CONTEXT	context_identifier, context_type	Chapter 6
GEOMETRIC_REPRESENTATION_CONTEXT	coordinate_space_dimension	Chapter 6
GLOBAL_UNIT_ASSIGNED_CONTEXT	units (set)	Chapter 6
GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT	uncertainty (set)	Chapter 6
NAMED_UNIT	dimensions * or explicit	Chapter 6

Entity	Attributes, in order	Where
SI_UNIT	prefix, name	Chapter 6
CONVERSION_BASED_UNIT	name, conversion_factor	Chapter 6
LENGTH_UNIT, *PLANE_ANGLE_UNIT*, *SOLID_ANGLE_UNIT*, *MASS_UNIT*	—	Chapter 6
DIMENSIONAL_EXPONENTS	length_exponent, mass_exponent, time_exponent, electric_current_exponent, thermodynamic_temperature_exponent, amount_of_substance_exponent, luminous_intensity_exponent	Chapter 6
UNCERTAINTY_MEASURE_WITH_UNIT	value_component, unit_component, name, description	Chapter 6
LENGTH_MEASURE_WITH_UNIT etc.	value_component, unit_component	Chapter 6
DERIVED_UNIT	elements (set)	Chapter 6
DERIVED_UNIT_ELEMENT	unit, exponent	Chapter 6

B.6 Topology

Entity	Attributes, in order	Where
MANIFOLD_SOLID_BREP	name, outer	Chapter 10
BREP_WITH_VOIDS	name, outer, voids (set)	Chapter 10
CLOSED_SHELL, OPEN_SHELL	name, cfs_faces (set)	Chapter 10
ADVANCED_FACE	name, bounds (set), face_geometry, same_sense	Chapter 10
FACE_BOUND, FACE_OUTER_BOUND	name, bound, orientation	Chapter 10
EDGE_LOOP	name, edge_list (list)	Chapter 10
VERTEX_LOOP	name, loop_vertex	Chapter 11
ORIENTED_EDGE	name, edge_start *, edge_end *, edge_element, orientation	Chapter 10
EDGE_CURVE	name, edge_start, edge_end, edge_geometry, same_sense	Chapter 10
VERTEX_POINT	name, vertex_geometry	Chapter 10

B.7 Geometry

Entity	Attributes, in order	Where
CARTESIAN_POINT	name, coordinates (list)	Chapter 10
DIRECTION	name, direction_ratios (list)	Chapter 10

Entity	Attributes, in order	Where
VECTOR	name, orientation, magnitude	Chapter 10
LINE	name, pnt, dir	Chapter 10
CIRCLE	name, position, radius	Chapter 11
ELLIPSE	name, position, semi_axis_1, semi_axis_2	Chapter 11
TRIMMED_CURVE	name, basis_curve, trim_1 (set), trim_2 (set), sense_agreement, master_representation	Chapter 11
POLYLINE	name, points (list)	Chapter 11, Chapter 16
PLANE	name, position	Chapter 10
CYLINDRICAL_SURFACE	name, position, radius	Chapter 11
CONICAL_SURFACE	name, position, radius, semi_angle	Chapter 11
SPHERICAL_SURFACE	name, position, radius	Chapter 11
TOROIDAL_SURFACE	name, position, major_radius, minor_radius	Chapter 11
SURFACE_OF_LINEAR_EXTRUSION	name, swept_curve, extrusion_axis	Chapter 11
SURFACE_OF_REVOLUTION	name, swept_curve, axis_position	Chapter 11
B_SPLINE_CURVE_WITH_KNOTS	name, degree, control_points_list, curve_form, closed_curve, self_intersect, knot_multiplicities, knots, knot_spec	Chapter 12
B_SPLINE_SURFACE_WITH_KNOTS	name, u_degree, v_degree, control_points_list (list of lists), surface_form, u_closed, v_closed, self_intersect, u_multiplicities, v_multiplicities, u_knots, v_knots, knot_spec	Chapter 12
RATIONAL_B_SPLINE_CURVE	weights_data (list)	Chapter 12
RATIONAL_B_SPLINE_SURFACE	weights_data (list of lists)	Chapter 12
SURFACE_CURVE, SEAM_CURVE	name, curve_3d, associated_geometry (list), master_representation	Chapter 13
PCURVE	name, basis_surface, reference_to_curve	Chapter 13
DEFINITIONAL_REPRESENTATION	name, items (set), context_of_items	Chapter 13

B.8 Tessellation (AP242)

Entity	Attributes, in order	Where
COORDINATES_LIST	name, npoints, position_coords (list of lists)	Chapter 13
TRIANGULATED_FACE	name, coordinates, pnmax, normals, geometric_link, pnindex, triangles	Chapter 13
TESSELLATED_SHELL	name, items (set), topological_link	Chapter 13
TESSELLATED_SOLID	name, items (set), geometric_link	Chapter 13

B.9 Appearance

Entity	Attributes, in order	Where
STYLED_ITEM	name, styles (set), item	Chapter 14
PRESENTATION_STYLE_ASSIGNMENT	styles (set)	Chapter 14
SURFACE_STYLE_USAGE	side, style	Chapter 14
SURFACE_SIDE_STYLE	name, styles (set)	Chapter 14
SURFACE_STYLE_FILL_AREA	fill_area	Chapter 14
FILL_AREA_STYLE	name, fill_styles (set)	Chapter 14
FILL_AREA_STYLE_COLOUR	name, fill_colour	Chapter 14
COLOUR_RGB	name, red, green, blue	Chapter 14
DRAUGHTING_PRE_DEFINED_COLOUR	name	Chapter 14
CURVE_STYLE	name, curve_font, curve_width, curve_colour	Chapter 14
PRESENTATION_LAYER_ASSIGNMENT	name, description, assigned_items (set)	Chapter 14
MECHANICAL_DESIGN_GEOMETRIC_PRESENTATION_REPRESENTATION	name, items (set), context_of_items	Chapter 14

B.10 Properties

Entity	Attributes, in order	Where
PROPERTY_DEFINITION	name, description, definition	Chapter 15
PROPERTY_DEFINITION_REPRESENTATION	definition, used_representation	Chapter 15
REPRESENTATION	name, items (set), context_of_items	Chapter 15

Entity	Attributes, in order	Where
DESCRIPTIVE_REPRESENTATION_ITEM	name, description	Chapter 15
MEASURE_REPRESENTATION_ITEM	name, value_component, unit_component	Chapter 15

B.11 PMI

Entity	Attributes, in order	Where
SHAPE_ASPECT	name, description, of_shape, product_definitional	Chapter 16
SHAPE_ASPECT_RELATIONSHIP	name, description, relating_shape_aspect, related_shape_aspect	Chapter 16
GEOMETRIC_ITEM_SPECIFIC_USAGE	name, description, definition, used_representation, identified_item	Chapter 16
DATUM	name, description, of_shape, product_definitional, identification	Chapter 16
DATUM_FEATURE	name, description, of_shape, product_definitional	Chapter 16
DATUM_SYSTEM	name, description, of_shape, product_definitional, constituents (list, in precedence order)	Chapter 16
DATUM_REFERENCE_COMPARTMENT	name, description, of_shape, product_definitional, base, modifiers	Chapter 16
GEOMETRIC_TOLERANCE	name, description, magnitude, toleranced_shape_aspect	Chapter 16
GEOMETRIC_TOLERANCE_WITH_DATUM_REFERENCE	datum_system (set)	Chapter 16
GEOMETRIC_TOLERANCE_WITH_MODIFIERS	modifiers (set)	Chapter 16
FLATNESS_TOLERANCE, POSITION_TOLERANCE, ...	as GEOMETRIC_TOLERANCE	Chapter 16
TOLERANCE_ZONE	name, description, of_shape, product_definitional, defining_tolerance (set), form	Chapter 16
DIMENSIONAL_SIZE	applies_to, name	Chapter 16
DIMENSIONAL_LOCATION	name, description, relating_shape_aspect, related_shape_aspect	Chapter 16
DIMENSIONAL_CHARACTERISTIC_REPRESENTATION	dimension, representation	Chapter 16

Entity	Attributes, in order	Where
PLUS_MINUS_TOLERANCE	range, toleranced_dimension	Chapter 16
TOLERANCE_VALUE	lower_bound, upper_bound	Chapter 16
ANNOTATION_CURVE_ OCCURRENCE	name, styles (set), item	Chapter 16
DRAUGHTING_CALLOUT	name, contents (set)	Chapter 16
ANNOTATION_PLANE	name, styles (set), item, elements (set)	Chapter 16
DRAUGHTING_MODEL	name, items (set), context_of_ items	Chapter 16
DRAUGHTING_MODEL_ITEM_ ASSOCIATION	name, description, definition, used_representation, identified_item	Chapter 16
CAMERA_MODEL_D3	name, view_reference_system, perspective_of_volume	Chapter 16

C. Sources and Further Reading

The standard. ISO 10303-21:2016 (Part 21, the file format) is the one document worth buying if you work with the files directly; the final edition-3 text is also hosted freely by STEP Tools. ISO 10303-11 (EXPRESS) is useful once, for the grammar of the schema language. ISO 10303-42 (geometry and topology) is the reference for Part II and is readable as prose. ISO 10303-242:2025 is the current AP242 publication and is consulted rather than read.

The schemas. STEP Tools hosts a merged EXPRESS schema covering AP203, AP214, AP242 and others at https://steptools.com/stds/step/, with an entity-by-entity browser. The declarations quoted in this book were checked against the published MIM long forms, which the book's source repository vendors from the STEPcode project so the checks can be re-run.

Recommended practices. The CAx Implementor Forum's documents — now under the MBx Interoperability Forum at https://mbx-if.org — cover assembly structure, colours and layers, user-defined attributes, geometric and assembly validation properties, PMI representation and presentation, tessellated geometry, external references and more. They are free, and they describe how real files are written. The PMI records in Chapter 16 follow the *Recommended Practices for the Representation and Presentation of Product Manufacturing Information (AP242)*, version 4.0, and its saved-view records were checked against NIST's PMI test models.

Test data. NIST's MBE PMI Validation and Conformance Testing project publishes CAD models with fully specified PMI, exports from several systems, and the expected results; its STEP File Analyzer and Viewer (free) reports on a file's conformance, validation properties and PMI. The CAx-IF publishes its test suite definitions and an implementation coverage matrix for the participating systems.

Reading files. Open CASCADE Technology (https://github.com/Open-Cascade-SAS/OCCT) is the open-source kernel used to produce every example in this book, apart from the hand-written PMI in `bracket-pmi.step`, and reads and writes AP203, AP214 and AP242; FreeCAD and CadQuery expose it. The NIST analyzer above is the best tool for inspecting a file's structure without a kernel.

Checking interpretations. The frame defaults in Chapter 8 follow the functions `build_axes` and `first_proj_axis` in Part 42's EXPRESS, and the tolerance zone names in Chapter 16 are the list in the AP242 rule `tolerance_zone_form_name_constraint`; both are in the schema text itself. The OCCT STEP translator guide documents the read and write precision modes discussed in Chapter 6.

D. Glossary

Aggregate
a collection-valued type: `SET` (unordered, no duplicates), `BAG` (unordered, duplicates allowed), `LIST` (ordered) or `ARRAY` (ordered, fixed bounds). The file writes all four as a parenthesised list.

AIM
Application Interpreted Model: an AP's data model expressed in integrated-resource entities. What the file contains.

Anchor
(edition 3) a stable public name for an instance or value, for reference from other files.

AP
Application Protocol: a scoped data model for a domain. AP203, AP214, AP242.

ARM
Application Reference Model: an AP's data model in domain vocabulary. What the committee designed.

Attribute
one named slot of an entity, declared in the schema. A record supplies one parameter per *explicit* attribute, in declaration order; the file writes the values and never the names. Attributes an entity derives itself, and inverse attributes, take no parameter at all.

B-rep
boundary representation: a solid described by its bounding faces, edges and vertices.

Complex entity instance
a record with several entity types (external mapping), for an object whose combination of subtypes has no single name.

Conformance class
a named subset of an AP that an implementation can claim to support. Part 21 uses the term separately, for the three syntactical conformance classes of the file format itself.

Datum
a theoretically exact reference plane, axis or point derived from a real feature.

Defined type
a name given to a base type so the schema can say what a value is *for*, as in `TYPE label = STRING;`. The file does not distinguish them, except where a defined type is the alternative chosen for a `SELECT`, when its name is written around the value.

Derived attribute
one whose value is computed from others rather than stored. An attribute a subtype redeclares as derived keeps its slot and is written `*`; one the entity declares itself takes no slot and is not written at all.

Entity
a type declared in the schema, with an ordered list of attributes. One record in the data section is one instance, of one entity or — in the complex form — of several at once.

Enumeration
a type whose values are a fixed set of names, written in the file between full stops: `.MILLI.`, `.METRE.`. `BOOLEAN` and `LOGICAL` are EXPRESS base types, but Part 21 encodes their values as enumeration values too.

EXPRESS
the schema language of ISO 10303 (Part 11).

External mapping
the parenthesised multi-type form of a record. See complex entity instance.

Feature control frame
the boxed statement of one geometric tolerance on a drawing or model.

GD&T
geometric dimensioning and tolerancing; the language of tolerances and datums.

Instance name
the `#n` identifier of a record; file-local.

Integrated resources
the shared library of entity definitions (Parts 41–99 and 101–199) from which every AP is built.

Internal mapping
the ordinary one-type record form with all inherited attributes flattened.

Knot vector
the parameter breakpoints of a B-spline; stored as distinct values plus multiplicities.

Long form
an AP schema flattened into a single file (`_MIM_LF`).

NAUO
`NEXT_ASSEMBLY_USAGE_OCCURRENCE`: one occurrence of a child in a parent.

NURBS
non-uniform rational B-spline; the free-form curve and surface type.

Part 21
ISO 10303-21, the clear-text file format. “A STEP file.”

Pcurve
a curve in the parameter space of a surface; the 2D image of an edge on a face.

PMI
product manufacturing information: tolerances, datums, dimensions, notes.

Representation
a set of representation items in a context; the Part 43 container for geometry and properties.

Schema
a collection of EXPRESS declarations naming every entity a conforming file may use, and the constraints on them. A file names the schema it claims to follow in `FILE_SCHEMA`.

Seam
an edge along which a periodic surface is cut, traversed twice by one loop.

SELECT
a type that admits any one of several alternatives. A value is wrapped in the name of the alternative chosen when that alternative is not an entity, as in `LENGTH_MEASURE(1.E-07)`.

Semantic PMI
PMI as data; contrast graphical PMI, PMI as drawn symbols.

Shape aspect
a named region of a shape that a tolerance or property can refer to.

Subtype, supertype
an entity that extends another, and the one it extends. A record carries the supertypes’ attributes first, from the root of the chain down; see internal mapping.

Uncertainty
the writing kernel’s coincidence tolerance, carried in the representation context: two points closer than it are treated as the same point. It is a geometric comparison, not a topological one; vertices are shared by reference, never merged by proximity.

Validation property
volume, area, centroid or count written by the exporter so the importer can check the transfer.