MME Files Format

Version 2.3

November 2, 2011

Contents

Introduction

This document is a description of the MME files format. These files are not "native" files of the Pacor application. They are intended for open data import/export from/to other applications mostly written by other developers. The list of possible types of recorded objects has been constantly extending, and alterations to the record format for a specific type are possible, so that it is necessary for applications processing these files to correctly handle unknown object types and records of later versions alike.

Version History

Version 2.3

The description of transformation of coordinates (‘transform’) is supplemented in version 2.3.

Version 2.2

Version 2.2 supplements the list of admissible objects by transformation of coordinates of a device (0x3007) and recommendations on the use of that object.

Version 2.1

Version 2.1 contains a description of a MME files format for dedicated export of objects from the Pacor application. The list of admissible objects is supplemented by solutions (0x3004), girdle shape (0x3005) and coloration of limiting objects (0x3006). Version 2.1 contains error correction in a solution description (FILE_OBJECT_COMPATIBLE_DIAM, transform).

Version 1.0

The document was initially titled “Format of Model File” and contained a format structure description and five object types: diamond model shape (0x3000), pinpoint inclusions (0x3001), 3D inclusions (0x3003), sawcut layers (0x3002), and object names (0x2001).

General

The format of MME files is designed for saving models of real objects (crystals, diamonds, inclusions etc.), marking alternatives, settings for various parameters, and other. The data are stored in a binary form and admit of no unauthorized modification.

A record format description for each object type will use the following definitions.

Numbers

Unless otherwise stipulated, an integer is 4 bytes recorded in a low → high byte ordering. An unsigned integer is 4 bytes. A double-accuracy floating-point number is presented by 8 bytes (64 bits) as double per IEEE 754-1985 standard.

Transformation of Coordinates (‘transform’)

‘Transform’ is a supplementary transformation of coordinates of points in a three-dimensional space. ‘Transform’ consists of 8 floating-point numbers defining a turn (‘spin’, the first four numbers: St, Sx, Sy, Sz), shift (‘vector’, numbers from fifth to seventh: Tx, Ty, Tz) and magnification (the eighth number: Magn). The total is 64 bytes. The transformation application order is as follows: turn, stretch, shift.

If rotation in coordinate system consider as rotation on angle A around a vector X, Y, Z with length 1 passing through the beginning of coordinates, then St is a cosine of half rotation angle (cos(A/2)), and (Sx, Sy, Sz) is a sine of half rotation angle multiplied by the corresponding coordinate (X*sin(A/2), Y*sin(A/2), Z*sin(A/2)). The positive direction of turn is counter-clockwise if look from the top of vector X, Y, Z

Given below is a calculation code for transformation matrix coefficients m for a turn set by the object Spin.

```
double tt,uu,vv,ww,vw,wu,uv,ut,vt,wt;
tt=s.t*s.t; uu=s.x*s.x; vv=s.y*s.y; ww=s.z*s.z;
vw=s.y*s.z; wu=s.z*s.x; uv=s.x*s.y; ut=s.x*s.t; vt=s.y*s.t;
wt=s.z*s.t;
m.x.x=tt+uu-vv-ww; m.x.y=2*(uv-wt); m.x.z=2*(wu+vt);
m.y.y=tt+vv-ww-uu; m.y.z=2*(vw-ut); m.y.x=2*(uv+wt);
m.z.z=tt+ww-uu-vv; m.z.x=2*(wu-vt); m.z.y=2*(vw+ut);
```

Application of the turn matrix to a vector (point) looks as follows:

```
VECTOR3 operator | (const MATRIX & m, const VECTOR3 & v)
{
      return VECTOR3 (m.x|v, m.y|v, m.z|v);
}
```

where operator | for two vectors is a scalar product:

```
double operator | (const VECTOR3 &v1, const VECTOR3 &v2)
{
      return v1.x*v2.x + v1.y*v2.y + v1.z*v2.z;
}
```

Thus, total transformation of coordinates ‘transform’ set by Spin, Shift and Magn for an arbitrary point P is:

```
transform(P) = (Matr | P) * Magn + Shift
```

where Matr is the turn matrix constructed from Spin.

Operations for transformation of coordinates can be found in more detail in the vector library vector.h.

String

The standard ‘string’ is a record consisting of an integer that defines the string size and the string proper without the ending 0. For example, the string “Model” will take up 9 bytes in the record and look as follows: 0x09, 0x00, 0x00, 0x00, 0x4D, 0x6F, 0x64, 0x65, 0x6C. It will be designated as the 5 ‘Model’ in this document. An empty string will occupy four zero bytes.

Reference

A ‘reference’ to another object is an integer defining the object local number in the object allocation table (cf. Object Allocation Table section). Takes up 4 bytes.

Version Control

‘Version Control’. This format was developed with account for possible new version of the same objects in the future. Forward and backward compatibility should be provided among other things. That is, new software should be able to read older files, and old software should read data files of newer versions. The following method is used to that end. Two ‘V2’ bytes (0x56, 0x32) are recorded to identify that a version control record will follow. The next two bytes represent the v number – the object version. Then n_1, n_2, ..., n_v follow – they are v integer numbers defining the data size for a specific object version. Thereafter, the data proper specific for each version follow in a reverse order: n_v bytes of version v, n_{v-1} bytes of version v-1, ..., n_1 bytes of version 1. If software encounters an object version newer than that supported by the current software, it should simply skip a corresponding number of bytes. The following notation will be used henceforth: control of v versions of an object with a description of the size and contents of each data block. It will be designated on the scheme as ”version control (v)“. The allowed number of versions is 0 to 65535. An arbitrary data block can have a version control structure of its own. Here is an example of a version control heading for two versions, the first being 328 bytes in size and the second 200 bytes:

V2 2 328 200 record_2 record_1

The text beginning with // (two slashe-symbols is C++ style comment) is the comment and is intended only for improvement of usability of the present documentation. For example, in the text often meets // version... allowing visually to separate the data of different versions. These symbols do not exist in objects records.

File Structure

Each MME file contains a heading, an object allocation table, and data of objects proper.

Heading

The heading consists of 16 bytes. The first 8 bytes are an attribute of the file in hand being a MME file. These should be characters 0x54, 0x49, 0x54, 0x4C, 0x45, 0x20, 0x30, 0x32 («TITLE 02»). This is followed by two four-byte integers, ‘NObjects’ and ‘Reserved’. The first of these defines the number of objects allocated in the file, while the second is reserved for future use and should be equal to 0. Here is a schematic example of a heading containing 14 objects:

```
‘TITLE 02’ 14 0
```

Object Allocation Table

The heading is followed by an object allocation table. The total size of table is ‘NObjects’ *16 bytes. The number of strings in that table is equal to the number of objects. Each row of the table consists of 4 integers sized 4 bytes each: ‘Offset’, ‘LocalNumber’, ‘ObjectType’ and ‘ObjectSize’. Consider these in more detail:

‘Offset’ is an object record beginning displacement in the file in hand

‘LocalNumber’ is an internal object number in the table concerned (in the file in hand). This number is used for saving references (links) between objects[1]. The value of the number cannot be 0 and should be unique for each object within the file in hand.

‘ObjectType’ is object type identifier

‘ObjectSize’ is object record size

All or only necessary objects from a file can be read using this table. Here is an example of a table for a three-object file:

```
64 1 0x3001 1200

1264 5 0x3002 81

1345 6 0x2001 23
```

Object Data

Recording particular objects is described in the following section. It is object type specific. When planning records it is strongly recommended to provide for future record extension by use of version control, even if it may seem at the moment that there is nothing else to add to the object concerned, and it will always remain the same.

Supplementary Transformation of Coordinates

Special attention must be attached to the object ‘transformation of coordinates of a device’ (0x3007) as it regulates supplementary transformation to all objects saved in a particular file.

Object Types

Type numbers from 0x3000 to 0x3FFF are reserved for MME objects. Besides, other records may also appear in the files. For instance, it is convenient enough to use a 0x2001 type object that defines object names in the file in hand. All such objects will be specified in this documentation whenever possible. The table below lists object types supplied with brief descriptions.

Object Type	Brief Description
0x2001	Object names
0x3000	Rough diamond model
0x3001	Pinpoint inclusion
0x3002	Layer (sawcut plane)
0x3003	3D inclusion

[1] Assume there are two objects A and B and there is a link between them, e.g. A knows about its “parent” B; the link should be restored as the file is read. To that end, object A requests assignment of a node number of object B (b) during save and saves it as an integer. As the file is read, object A acquires access to the newly created B object by using the node number. A drawback of this system is that the order of object creation during reading is unknown. It is advisable to follow the rule: Create all objects needed first (‘create’), then provide a reading procedure for each of them (‘read’), and then finally call the object restoration completion function for each (‘awake’). This will allow knowledge of the existence of the object at the ‘read’ stage and acquiring a reference thereto, and that reference will be guaranteed to be operable at the ‘awake’ stage.

0x3004	Solution
0x3005	Girdle shape
0x3006	Coloration of limiting objects
0x3007	Transformation of device coordinates

Object Names (0x2001)

An object is intended for recording object names in the file in hand. An object is optional, yet highly convenient to be used to get a list of names of objects available in a file, for example rough diamond models, without loading the objects as such, and leave it to the user to pick objects desired for loading.

The general form of the record would appear as follows:

```
'Nam0', n, <link₁ name₁>, <link₂ name₂>, …,<linkₙ nameₙ>
```

An object record consists of 4 bytes of the record identifier 'Nam0' (0x4E, 0x61, 0x6D, 0x30) followed by an integer n – the number of names recorded. (It may be other than the total of objects in the file.) This is followed by n pairs of the form <reference to object, name>. The name is recorded as a standard string where the number of bytes required to record the name appears in the beginning and the name itself follows. Below is an example of a record with rough diamond model names (internal object number 3 and the name "Complex" and inclusions (internal number 4 with the name "Crack").

```
'Nam0' 2
3 7 'Complex'
4 5 'Crack'
```

Rough Diamond Model (0x3000)

This object type is intended for recording a rough diamond model. The object record begins with version control to allow later extension. Two versions exist at the moment. The data of the first version contain a model name and the minimum information needed to save the polyhedron structure: coordinates of vertexes and lists of indices of vertexes in the sides. The data of the second version contain full information on the structure of polyhedron bones and coordinates of side planes. On the one hand, this information appears excessive, and on the other, different applications may calculate it differently to entail identical yet differing polyhedrons in different applications. Records of planes and bones are optional and may be absent. If a record is absent, then the corresponding value of 'nPlanes' or 'nBones' is zero. A record contains a number of elementary records of another kind: 'point', 'side', 'plane', 'bone'.

Point

Each 'point' is three double-accuracy floating-point numbers representing (x, y, z) coordinates of vertex. The total is 24 bytes. Here is an example of recording two vertexes:

```
0.5 -0.5 0
-1 0 0
```

Side

Each 'side' is a <n i_1, i_2, …, i_n> mode record of unsigned 4-byte numbers where n is the number of vertexes in a side and then n indices of vertexes i_k of which the side in hand consists. The total is 4 * (n+1) bytes. Vertexes are listed in the order specified counter-clockwise as viewed from the vertex of an outward normal. Numbering of vertex indices begins with 0 and ends with ‘nVertex-1’. An example of recording two sides of a cube:

```
4 0 1 2 3
4 0 3 7 4
```

Plane

Each ‘plane’ is four double-accuracy floating-point numbers N_x, N_y, N_z, D. The total is 32 bytes. These numbers represent the plane normal vector (N_x, N_y, N_z) and distance from the plane to the centre of coordinates. The normal vector should have a length equal to 1. In that case, the plane itself will be described by the equation $N_x*x + N_y*y + N_z*z + D = 0$. An example of recording two side planes of a cube:

```
0 0 1 0.5
1 0 0 0.5
```

Bone

Each ‘bone’ is a record consisting of four unsigned 4-bytes numbers <p1 p2 s1 s2>. The total is 16 bytes. p1, p2 are indices of vertexes that connect the bone in hand. s1, s2 are indices of sides that intersect to form the bone in hand. Numbering of indices of vertexes and sides begins with 0. An example of recording bones:

```
0 1 0 2
1 2 0 3
```

Rough Diamond Model

Revert to the description of a rough diamond model. The general record structure looks as follows:

```
version control (2)
// version 2
nPlanes
plane1, plane2, …, planenPlanes
nBones
bone1, bone2, …, bonenBones
// version 1
nVertex
point1, point2, …, pointnVertex
nSides
side1, side2, …, sidenSides
Name
```

The unsigned integer ‘nPlanes’ defines the number of side planes recorded. This value can be either 0 or coincide with the number of sides ‘nSides’. Where 0 is the case, side planes are not recorded, and the reading routine would have to subsequently calculate side planes itself by using coordinates of vertexes in the sides. Once the number of planes has been specified, the planes ‘plane_1‘ … ‘$\text{plane}_{\text{nPlanes}}$’ are recorded. A record of planes takes up a total of 4+32*nPlanes bytes.

The unsigned integer ‘nBones’ defines the number of bones recorded. It can be 0 which means no bone record. In this case, the reading routine would have to form itself the structure of bones where necessary. Records of bones ‘bone_1‘ … ‘$\text{bone}_{\text{nBones}}$’ follow indication of the number of bones recorded. Indices of vertexes and sides contained in the description of each bone are numbered from 0 through nVertex-1 and nSides-1 respectively. A record of bones takes up a total of 4+16*nBones bytes.

The unsigned integer ‘nVertex’ defines the number of vertexes in a polyhedron. Coordinates of vertexes ‘point_1‘ … ‘$\text{point}_{\text{nVertex}}$’ are recorded afterwards. A record of vertexes takes up a total of 4+24*nVertex bytes.

The unsigned integer ‘nSides’ defines the number of sides in a polyhedron. Sides ‘side_1‘ … ‘$\text{side}_{\text{nSides}}$’ are recorded afterwards. Indices of vertexes contained in the description of each side are numbered from 0 through nVertex-1.

‘Name’ is a standard string and defines the model name.

The unsigned integer ‘nVertex’ defines the number of polyhedron vertexes and is followed by coordinates of polyhedron vertexes ‘point_i’.

Model Record Example

A schematic example of recording a cube with a lateral length equal to 1 is presented below. Along with a compulsory structure of sides and vertexes, the example contains a record of bones in version 2 and no record of planes.

```
V2 2 328 200
// version 2
0
12
     0 1 0 1
     1 2 0 2
     2 3 0 3
     3 0 0 4
     0 4 1 4
     4 5 1 5
     5 1 1 2
     5 6 2 5
     6 2 2 3
     6 7 3 5
     7 3 3 4
     7 4 4 5
// version 1
8
      0.5 -0.5  0.5
      0.5  0.5  0.5
     -0.5  0.5  0.5
     -0.5 -0.5  0.5
      0.5 -0.5 -0.5
      0.5  0.5 -0.5
     -0.5  0.5 -0.5
     -0.5 -0.5 -0.5
6
     4 0 1 2 3
     4 0 4 5 1
     4 1 5 6 2
     4 2 6 7 3
     4 3 7 4 0
```

```
        4 4 7 6 5
    4 ‘Cube’
```

Pacor Application Restrictions

The Pacor application currently only handles models in which the number of vertexes, sides and bones is not in excess of 65535.

Pinpoint Inclusion (0x3001)

This object type is intended for pinpoint inclusion record. A pinpoint inclusion is a set of several points. The object record now also includes the name and clarity of the inclusion. The object record begins with version control to allow later extension. Two versions exist at the moment. The data of the first version contain the inclusion name and coordinates of points. The second version contains inclusion clarity in an arbitrary diamond grading system, such as VS2 (GIA) or 7a (Russian system). The record of coordinates of points is presented as ‘points’ described in the Rough Diamond Model section above (0x3000).

```
version control (2)
// version 2
Clarity
// version 1
nVertex
point₁, point₂, …, pointₙᵥₑᵣₜₑₓ
Name
```

The standard string ‘Clarity’ defines the clarity of the inclusion concerned in an arbitrary diamond grading system. For example SI1 in the GIA system:

```
3 ‘SI1’
```

The unsigned integer ‘nVertex’ defines the number of points in a pinpoint inclusion. Coordinates of vertexes ‘point_1‘ … ‘$\text{point}_{\text{nVertex}}$‘ are recorded afterwards. A record of vertexes takes up a total of 4+24*nVertex bytes.

‘Name’ is a standard string and defines the inclusion name.

Example of Pinpoint Inclusion Record

A schematic record example for a VS1 clarity pinpoint inclusion consisting of two points is presented below.

```
// version control (2)
V2 2 65 7
// version 2
3 ‘VS1’
// version 1
2
     0.1 0.0 0.0
     -0.2 0.12 0
9 ‘Points 43’
```

Layer (Sawcut Plane) (0x3002)

This object type is intended for recording layers (sawcut planes). A layer is two planes defining the lower and upper border of the layer through which the rough diamond is sawcut in two. The object record contains coordinates of the planes and the layer name. The object record begins with version control to allow later extension. Only one version exists as of today. The record of planes is presented as a 'plane', described in the Rough Diamond Model section above (0x3000).

```
version control (1)
// version 1
plane₁
plane₂
Name
```

'plane_1' is the upper border of a layer. The record is an equation of plane and takes up 32 bytes.

'plane_2' is the lower border of a layer. The record is an equation of plane and takes up 32 bytes.

'Name' is a standard string and defines the layer name.

Layer Record Example

Given below is a schematic record example for a layer with the name 'Layer 1', with the borders equidistanced by 0.09 mm from plane O_{XY}.

```
// version control (1)
V2 1 75
// version 1
0.0 0.0 1.0 -0.090
0.0 0.0 1.0 0.090
7 'Layer 1'
```

3D Inclusion (0x3003)

This object type is intended for recording 3D Inclusions. Three kinds of 3D inclusions are recognized as of today. These are a convex polyhedron, a nonconvex polyhedron, and a sphere. Besides, there exists a 3D inclusion of the 'Set' type that may include any set of inclusions of another kind, including 'Set' itself. Occurrences of inclusions of new types are possible. It is therefore strongly recommended for applications reading MME files to correctly skip records of inclusions of an unknown type, particularly within 'Set' inclusions. The 3D inclusion record format consists of general inclusion information and geometrical structure. The kinds of geometrical structures are given in the table below.

Type	Name	Geometrical structure description
0	Set	Set of inclusions of another kind
1	Sphere	Spherical inclusion
2	Convex	Convex polyhedron
3	Nonconvex	Nonconvex polyhedron

'Set' Inclusion

Sets of 'Set' inclusions are used when the inclusion structure is fairly complicated and cannot be presented as a single polyhedron. It is convenient to break up an inclusion into several parts, provided that it is initially presentable in the form of a doubly or multiply connected polyhedron. Also, a number of different inclusions of identical clarity can sometimes be united in a single inclusion for convenience. The number of hierarchies within a 'Set' may be chosen optionally. A record of such an inclusion contains the number of "child" inclusions as well as the size and type of each child inclusion.

```
nInclusion
     <nType_1 nSize_1 Record_1>
     <nType_2 nSize_2 Record_2>
     …
     <nType_nInclusion nSize_nInclusion Record_nInclusion>
```

The unsigned integer 'nInclusion' defines the number of child inclusions recorded within the scope of the set concerned. The type ('$nType_i$'), size ('$nSize_i$') and data record ('$Record_i$') proper for each inclusion is recorded then. This scheme permits a new inclusion kind to be safely ignored in earlier routine versions, should it appear in the specification. Here is an example of a 'Set' type inclusion where two inclusions of nonconvex polyhedron and sphere types are merged:

```
2
     3 458 Record_1
     1 32 Record_2
```

'Record1' and 'Record2' are actually records of a nonconvex polyhedron and sphere.

'Sphere' Inclusion

'Sphere' type inclusions are used to present spherical inclusions. An inclusion is defined by the centre of coordinates of the sphere and its radius. A special case of a sphere is a point with a zero radius. A sphere record consists of 4 floating-point numbers. The first three numbers x, y, z define coordinates of the centre of the sphere, while the fourth defines its radius. The record takes up a total of 32 bytes.

```
x y z Radius
```

Record example for a sphere with its centre at point (2.124, 4.12, -0.232) and radius 0.4:

```
2.124 4.12 -0.232 0.400
```

'Convex' and 'Nonconvex' Inclusions

Inclusions of the 'Convex' and 'Nonconvex' types are intended for presentation of polyhedrons. Their records are now absolutely identical. A record contains minimum information required to save a polyhedron. These are coordinates of vertexes and the belonging of vertexes to sides. A record of coordinates of vertexes is presented as 'point', while belonging of vertexes to sides, as 'side', both described in detail in the Rough Diamond Model section above (0x3000).

```
nVertex
point_1, point_2, …, point_nVertex
nSides
side_1, side_2, …, side_nSides
```

The unsigned integer 'nVertex' defines the number of vertexes in a polyhedron. Coordinates of vertexes '$point_1$' … '$point_{nVertex}$' are recorded afterwards. A record of vertexes takes up a total of 4+24*nVertex bytes.

The unsigned integer ‘nSides’ defines the number of sides in a polyhedron. Sides ‘$side_1$‘ … ‘$side_{nSides}$‘ are recorded afterwards. Indices of vertexes contained in the description of each side are numbered 0 through nVertex-1.
Record example for a cube:

```
8
      0.5 -0.5  0.5
      0.5  0.5  0.5
     -0.5  0.5  0.5
     -0.5 -0.5  0.5
      0.5 -0.5 -0.5
      0.5  0.5 -0.5
     -0.5  0.5 -0.5
     -0.5 -0.5 -0.5
6
     4 0 1 2 3
     4 0 4 5 1
     4 1 5 6 2
     4 2 6 7 3
     4 3 7 4 0
     4 4 7 6 5
```

3D Inclusion

Once geometrical shapes of a 3D inclusion have been sorted out, revert to recording the whole of the inclusion itself. The object record begins with version control to allow later extension. Two versions exist at the moment. The first version contains inclusion clarity, a set of geometrical shapes, and the inclusion name. The second contains Oxygen application specific data.

```
version control (2)
// version 2
nOxygenDataSize
OxygenData
// version 1
Clarity
nForm
form1, form2, …, formnForm
Name
```

The unsigned integer ‘nOxygenDataSize’ defines the size of supplementary data of the inclusion as retrieved in its construction.

‘OxygenData’ is a record of supplementary inclusion data as retrieved in the construction of the inclusion. Please refer to Oxygen application developers for more information on the format of that record. It is advisable that applications handling MME files either skip that record or save it unchanged compared to what it was like while the file was being read.

The standard string ‘Clarity’ defines clarity of the inclusion concerned in an arbitrary diamond grading system. For example SI1 in the GIA system:

```
3 ‘SI1’
```

'nForm' is the number of recorded shapes of inclusions.

'form$_1$' … 'form$_{nForm}$' are records of shapes of inclusions. Each record 'form$_i$' (nType$_i$, nSize$_i$, Record$_i$) consists of shape type 'nType$_i$', shape record size 'nSize$_i$', and shape record proper 'Record$_i$'. The type of geometrical presentation is one of the shapes 'Set', 'Sphere', 'Convex' or 'Nonconvex' described above. Actually the whole record of shapes exactly corresponds to the record of a single shape of the kind 'Set' with an arbitrary number of child records.

'Name' is a standard string and defines the inclusion name.

Example of a 3D Inclusion Record

Given below is a record example for a 3D inclusion with clarity VS1 consisting of a set of two spheres and without Oxygen data.

```
V2 2 104 4
// version 2
0
// version 1
3 'VS1'
2
     1
     32
     2.124 4.12 -0.232 0.400

     1
     32
     2.824 4.12 -0.232 0.400
9 '2-Spheres'
```

Solution (0x3004)

This object type is intended for recording a solution containing several diamonds and layers. Each diamond is recorded in the form of several records of DiamCalc (record DMC): parametrical and geometrical forms. This allows saving a set of initial cut parameters and the polyhedron shape in case the cut is inaccessible for any reason — for instance, if it is implemented in the form of an external DLL while the DLL itself is inaccessible for a particular workstation. Layers are presented in the form of two planes. Beside that, for a diamond, its weight, cut grade, color and clarity are recorded. The appraiser and price list used in obtaining the amount of the solution concerned are saved for the entire solution as well as solution weight. For more information on DMC record format, see the document DiamCalc Objects Record Format in a version not lower than 1.3 dd. August 9tht, 2011. If parametric presentation of cuts implemented as an exterior DLL has to be sorted out, applicable descriptions from exterior cut developers will be needed that are not covered herein.

Auxiliary objects 'diamond' and 'layer' are used in recording solution.

'Diamond' Record

A diamond record ('diamond') contains its own version control. The first version includes two DMC records with parametric and geometrical diamond presentations, transformation of coordinates, weight, carat weight, cut grade, color, clarity, and fluorescence.

```
version control (1)
```

```
// version 1
Cutting
nParameterType ParameterDataSize ParameterData
nGeometryType GeometryDataSize GeometryData
Transform
Weight
CaratWeight
PresenationColor
CutGrade
Color
Clarity
Fluorescence
TargetCutGrade
Name
```

‘Cutting’ is a standard string with the cut name. It serves basically for an error message if cut loading from DMC records failed for any reason.

The unsigned integer ‘nParameterType’ represents the identifier for a diamond record in a parametric form. If this value is 0, the next two records ‘ParameterDataSize’ and ‘ParameterData’ are absent. If it is other than 0, then ‘ParameterDataSize’ is the parametric record size, and ‘ParameterData’ is the parametric record itself. ‘nParameterType’ will normally assume the value 0x2701 (FILE_OBJECT_DIAMOND) or 0x2711 (FILE_OBJECT_GUID_DIAM_PLAIN).

‘ParameterDataSize’ is the size of the immediately following diamond record in a parametric form.

‘ParameterData’ is a diamond record in a parametric form.

The unsigned integer ‘nGeometryType’ is the identifier for a diamond record in a geometrical form. If this value is 0, the next two records ‘GeometryDataSize’ and ‘GeometryData’ are absent. If the value is other than 0, then ‘GeometryDataSize’ is the geometry record size while ‘GeometryData’ is the diamond geometry record itself. ‘nGeometryType’ is typically equal to 0x271A (FILE_OBJECT_COMPATIBLE_DIAM).

‘Transform’ is transformation of coordinates ‘transform’ that converts diamond coordinates from internal to a global coordinate system. The magnification ratio of this transformation should be 1.

‘Weight’ is a floating-point number that defines the diamond carat weight.

‘CaratWeight’ is a floating-point number that defines the diamond carat weight used in the appraisal of gem parameters and amount. This is usually rounding-off to two digits after the decimal point by a rule accepted in diamond industry where numbers up to X.xx84 are approximated downwards, while numbers X.xx85 and above, upwards. For example, 1.3284 would be approximated to 1.32, and 1.3285, to 1.33.

‘PresentationColor’ is 4 bytes defining diamond color presentation in a scene. The presentation format is “blue”, “green”, “red”, or “alpha”.

‘CutGrade’ is a string defining the diamond cut grade in parameter appraiser.

'Color' is a string defining the diamond color in the price list.

'Clarity' is a string defining diamond clarity in the price list.

'Fluorescence' is a string defining diamond fluorescence in the price list.

'TargetCutGrade' is a string defining target diamond cut grade, which string was used in the search of the solution concerned. This quality typically coincides with 'CutGrade'. Extra values are "Auto" and "Fixed".

'Name' is a standard string and defines the diamond name in the solution.

Here is a record example for Brilliant cut where parametric and geometrical presentations are absent.

```
V2 1 146
// version 1
9 'Brilliant'
0
0
1 0 0 0 0 0 0 1
1.24156895
1.24
255 0 0 0
2 'EX'
1 'H'
4 'VVS1'
4 'None'
2 'EX'
6 'Diam 1'
```

'Layer' Record

A layer record 'layer' contains its own version control. The first version includes coordinates of planes of the layer top and bottom, linkage of each plane to diamonds in the solution concerned, and the layer name.

```
version control (1)
// version 1
plane₁
plane₂
nDiamond₁
nDiamond₂
Name
```

'$plane_1$' is the upper layer border. The record is an equation of plane and takes up 32 bytes.

'$plane_2$' is the lower layer border. The record is an equation of plane and takes up 32 bytes.

'$nDiamond_1$' is an index of a diamond assigned to the upper layer border.

'$nDiamond_2$' is an index of a diamond assigned to the lower layer border.

‘Name’ is a standard string with the layer name in the solution.

Diamond indexation begins with 0. This adopted, the value -1 means that the layer part concerned is not assigned to any diamond whatsoever. Here is a record example for a layer in a solution, which layer is assigned to the first two diamonds:

```
V2 1 83
// version 1
0.0 0.0 1.0 -0.090
0.0 0.0 1.0 0.090
0
1
7 ‘Layer 1’
```

Solution

A solution record contains records of all diamonds and layers within the solution concerned, the solution identifier and solution name, as well as the entire integrated solution information: the total working weight of the solution, the amount of diamonds involved and that of the solution itself unless it was protected by dedicated encoded price lists, the appraiser name and the price list name, and the index of the active diamond in the solution. A solution record begins with version control for later extension. Only a single version exists at the moment.

```
version control (1)
// version 1
nDiamond
diamond1, diamond2,…,diamondnDiamond
nLayer
layer1, layer2, …, layernLayer
nID
CaratWeight
PresentationColor
ParameterAppraiser
PriceList
nPriceType PriceDataSize PriceData
nActiveDiamond
Name
```

‘nDiamond’ is an integer defining the number of diamonds in the solution.

‘diamond_1‘ … ‘$\text{diamond}_{\text{nDiamond}}$‘ are diamond records. Each record is a ‘diamond’ type record as described above.

‘nLayer’ is an integer defining the number of layers in the solution.

‘layer_1‘ … ‘$\text{layer}_{\text{nLayer}}$‘ are records of layers. Each record is a ‘layer’ type record as described above.

‘nID’ is an integer solution identifier.

‘CaratWeight’ is the working weight of the solution which is equal to the total of working weights of diamonds in the solution. The differences of the working weight from the weight for a diamond are described in the diamond record.

‘PresentationColor’ is 4 bytes defining color presentation of a solution in a list of solutions. The presentation format is “blue”, “green”, “red”, or “alpha”.

‘ParameterAppraiser’ is a string meaning the name of the parameter appraiser with which the solution concerned was created.

‘PriceList’ is a string meaning the name of the price list with which the solution concerned was created.

‘nPriceType’ is an integer and means the form in which the record of diamond amounts is present. 0 value means that an amount record is entirely absent. In this case the next two records ‘PriceDataSize’ and ‘PriceData’ are absent. If the value is other than 0, then the size of the amount record and the record itself follow immediately afterwards.

‘PriceDataSize’ is the size of a record of the amount of the solution and that of diamonds in the same.

‘PriceData’ is a solution amount record. Two kinds of amount records exist currently. They correspond to values 1, 2 of the variable ‘nPriceListType’.

Record Type	Record Format
0	Record is absent
1	The amount record is present in an open form. First comes the amount of the entire solution, then the number of diamonds and the amount of each diamond in the solution. That totals to 8 + 4 + nDiamond * 8 bytes
2	A relative record of the solution amount against a previous solution. This amount saving method imposes a certain borderline (say $200) at excess of which only information is given that the borderline is exceeded. If the borderline is not exceeded, the margin of the actual solution amount against a previous solution is given. If the borderline is exceeded, the borderline value is given. Such information is recorded in the 9th byte. The first byte is the type of the margin against a previous solution, bytes 2 through 9 are the margin value. The first byte may assume the following values: 0 is the basic solution. The margin given is 0 1: The amount of the solution differs from that of a previous solution by less than the borderline value. The margin value is given 2: The amount of the solution differs from that of a previous solution by more than the borderline value. A signed borderline value is given — thus, if the borderline is 200, the amount of the previous solution is 800 and that of the active solution is 350, 2 and -200 will be recorded.

‘nActiveDiamond’ is the active diamond in the solution in hand. Numbering of diamond indices begins with 0. The value -1 is also admissible and means that none of the diamonds in the solution is active.

‘Name’ is a standard string and defines the solution name.

Example of Solution Record

Shown below is a record example for a solution with two diamonds, one layer, and an open method for amount recording.

```
V2 1 559
```

```
// version 1
2
     // First diamond
     V2 1 146
     // version 1
     9 ‘Brilliant’
     0
     0
     1 0 0 0 0 0 0 1
     1.24156895
     1.24
     255 0 0 0
     2 ‘EX’
     1 ‘H’
     4 ‘VVS1’
     4 ‘None’
     2 ‘EX’
     6 ‘Diam 1’
     // Second diamond
     V2 1 141
     // version 1
     4 ‘Pear’
     0
     0
     1 0 0 0 0 0 0 1
     0.7489
     0.75
     0 255 0 0
     2 ‘EX’
     1 ‘H’
     4 ‘VVS1’
     4 ‘None’
     2 ‘EX’
     6 ‘Diam 2’
1
     // Layer
     V2 1 83
     // version 1
     0.0 0.0 1.0 -0.090
     0.0 0.0 1.0 0.090
     7 ‘Layer 1’

39

1.99

0 255 0 0

16 ‘Lexus 2011.06.01’

16 ‘Lexus 2011.03.11’

1

     9391.20 2 7291.20 2100.00

-1
```

```
61 ‘39) BP:1.99 (9391.20) Brilliant: 1.24(VVS1), Pear:
0.75(VVS1)’
```

Girdle Shape (0x3005)

This object type is intended for diamond girdle shape and transfer of the same to an executing device for girdle bruting. The record contains the diamond girdle shape set in the form of a set of polygonal strings, diamond projections onto the girdle plane, and the shape name. The object record begins with version control to allow later extension. Only one version exists as of today. The girdle shape record and rough diamond projections are presented in the form of a set of polygon vertexes ‘2D-form’.

‘2D-form’ Record

A polygon record ‘2D-form’ contains a set of vertexes forming a polygon in a three-dimensional space. The record contains its own version control now consisting of a single version.

```
version control (1)
// version 1
nVertex
point1, point2, …, pointnVertex
```

‘nVertex’ is an unsigned integer designating the number of vertexes in a polyhedron

‘point_1‘ … ‘$\text{point}_{\text{nVertex}}$‘ are coordinates of vertexes. The listing order of vertexes should be such that adjacent points form polyhedron bones.

‘2D-form’ record example for a rectangle:

```
V2 1 100
// version 1
4
     1.0 2.0 0.0
     1.0 -2.0 0.0
     -1.0 -2.0 0.0
     -1.0 2.0 0.0
```

Girdle Shape Record

The first girdle shape record version directly contains the girdle shape and accuracy to which that shape was produced, transformation of coordinates from the diamond internal system to a global coordinate system, and the object name. A rough diamond model projection onto the polished diamond girdle plane can be recorded as an option. The sequence order of vertexes in ‘2D-forms’ is undefined. A critical requirement is that adjacent as well as the first and last points should form polyhedron bones.

```
version control (1)
// version 1
Girdle
dblAccuracy
Transform
bHasProjection Projection
Name
```

‘Girdle’ is a ‘2D-form’ girdle shape record in the diamond coordinate system.

‘dblAccuracy’ is a floating-point number that defines the accuracy to which the girdle shape in hand was produced. Accuracy is set in the same units as all dimensional quantities: in millimeters.

‘Transform’ is transformation of coordinates from the diamond coordinate system to a global coordinate system. This transformation contains a unitary ratio of enlargement.

‘bHasProjection’ is 1 byte. A logical quantity assumes the value 1 if followed by a record of a rough diamond model projection onto the polished diamond girdle plane. The value assumed is 0 if such record is absent.

‘Projection’ is a record of a ‘2D-form’ rough diamond model projection onto the polished diamond girdle plane. Coordinates of this projection are set in the same coordinate system as a girdle shape projection. ‘Transform’ transformation is needed to convert it to a global system. The ‘Projection’ record is optional and is only present if the quantity bHasProjection is equal to 1.

‘Name’ is a standard string with the name of the girdle shape object.

Example of Girdle Shape Record

Here is a girdle shape record example for a rectangular cut with an accuracy of 5 µm and without rough diamond projection:

```
V2 1 201
// version 1
// Girdle 2D-form
     V2 1 100
     // version 1
     4
          1.0 2.0 0.0
          1.0 -2.0 0.0
          -1.0 -2.0 0.0
          -1.0 2.0 0.0
0.005
1 0 0 0 0 0 0 1
0
16 ‘Only girdle form’
```

Coloration of Limiting Objects (0x3006)

This object type is intended for saving coloration of limiting objects, in particular inclusions and restrictions in the form of planes. Inclusions are painted red, yellow or green and, depending on this, are used or disused at different stages of algorithms. Thus, red inclusions are always removed, green ones are only used in appraising resulting diamonds, while both alternatives are checked for the yellow, i.e. the algorithm both considers withdrawal of these inclusions outside of the diamond and leaves room for retaining these inclusions within the diamond. The object record begins with version control to allow later extension. Only one version exists as of today. It contains coloration of objects.

Coloration of Objects

The first record version for limiting objects coloration contains a set of pairs consisting of a reference to the record of an object and its color status.

```
version control (1)
// version 1
nRecords
Object₁ Status₁
Object₂ Status₂
…
ObjectnRecords StatusnRecords
```

‘nRecords’ is an integer defining the number of records of the condition of limiting objects.

‘Object_1‘ is a reference. Local index of the limiting object for which a color status follows next.

‘Status_1‘ is 1 byte. Limiting object color status. The status may assume any of the values: 1 (red), 2 (green) and 0 (yellow).

Further on similarly, ‘Object_2,‘ …, ‘$\text{Status}_{\text{nRecords}}$‘ define references and statuses of the remaining restrictions. A coloration record for ‘nRecords’ objects takes up a total of 12 + nRecords * 5 bytes.

Example of Limiting Objects Coloration Record

Here is a record example for coloration for two inclusions colored yellow and red with local indexes 6 and 8:

```
V2 1 14
// version 1
2
6 0
8 1
```

Transformation of Device Coordinates (0x3007)

This type of objects is intended for export of data of transformation of coordinates of a physical device to be saved in a file. This is necessary where a laboratory or enterprise disposes of more than one physical device for gem processing, for instance involving rough diamond scanning, removal of inclusions, sawing up, bruting etc. Availability of this information allows operation in a unified system of coordinates on these devices. This transformation is regarded as transformation of coordinates for an object within a device to some global system of coordinates. Each device has such a transformation, and it is necessary to first apply transformation of the other device and then inverse transformation of the current device when importing data. Transformation to a global system of coordinates may have already been applied in file saving, but anyway information about it is stated.

If a file lacks an object of the type concerned, transformation of coordinates of the device is assumed to have already been applied when saving the objects, and all objects have been recorded in global coordinates.

Transformation of Coordinates of a Device

The object record begins with version control to allow later extension. Only one version exists at the moment. The first version of recording transformation of coordinates is a flag telling whether a transformation was applied when saving the file, and the name of the application or device which the transformation concerned relates to.

```
version control (1)
// version 1
Transform
Flag
Name
```

‘Transform’ is transformation of coordinates ‘transform’ from the system of coordinates of the device to a global system of coordinates. This transformation contains a unitary ratio of enlargement.

‘Flag’ is 1 byte. The logical quantity assumes the value 1 if transformation of coordinates of the device was applied in the file in hand when recording other objects. Or, the value assumed will be 0 if such transformation was not applied and the objects are recorded in the device’s internal coordinates.

‘Name’ is a standard string with the name of the device or application the transformation concerned relates to.

Example of Device Coordinates Transformation Record

Here is an example of a record of transformation of coordinates for the Helium Rough device that has a 100 µm shift in height for data conversion to a global system. This transformation was already applied in saving the file for other objects:

```
V2 1 14
// version 1
1 0 0 0 0 0 0.1 1
1
11 ‘HR 1:4D 232’
```