



Advanced European Infrastructures for Detectors at Accelerators

---

# DDAlign

Alignment Support for the  
DD4hep Geometry Description  
Toolkit

M.Frank  
CERN, 1211 Geneva 23, Switzerland

## DDAlign User Manual

**Abstract**

Experimental setups in High Energy Physics are highly complex assemblies consisting of various detector devices typically called *subdetectors*. Contrary to the ideal world, where all these components are of perfect shape and at exact positions, existing devices have imperfections both in their shape and their relative and absolute positions. These are described by the alignment parameters.

To still measure the detector response from particle collisions with the highest possible precision, these imperfections are taken into account when converting measured signals to space-points in the measurement devices. This procedure is called *detector alignment*. DD4hep does not want to solve the exact problem of the detector alignment itself, but rather support firstly algorithms determining the alignment parameters and secondly support the application which apply the measured alignment parameters and apply them to the ideal geometry for further event data processing.

We will present the tools to support the detector alignment procedures using the DD4hep detector description toolkit. The DDAlign toolkit implements a modular and flexible approach to introduce and access the alignment parameters.

The design is strongly driven by easy of use; developers of detector descriptions and applications using them should provide minimal information and minimal specific code to achieve the desired result.

Document History		
Document version	Date	Author
1.0	01/04/2014	Markus Frank CERN/LHCb
1.1	30/04/2014	Markus Frank CERN/LHCb

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Generic Detector Description Model . . . . .	2
1.2	Detector Element Tree and the Geometry Hierarchy . . . . .	2
1.3	Alignment Parameters of Detector Components . . . . .	3
1.4	Iterative Application of Alignments . . . . .	5
1.5	Procedures to Determine Alignment Parameters . . . . .	6
1.6	Simulation of Non-Ideal, Real Detector Geometries . . . . .	6
<b>2</b>	<b>The DDAlign User Interface</b>	<b>7</b>
2.1	Access to Alignment Parameters from the Detector Element . . . . .	7
2.2	Manipulation of Alignment Parameters . . . . .	9
2.2.1	Manipulation of Alignment Parameters for Pedestrians using C++ . . . . .	9
2.2.2	Loading Geometrical Imperfections from XML . . . . .	11
2.2.3	Export Geometrical Imperfections to XML . . . . .	14

# 1 Introduction

This manual should introduce to the **DDAlign** framework. One goal of **DDAlign** is to easily model geometrical imperfections applied to the ideal geometry of detection devices as they are typically used in high energy physics experiments.

To avoid confusion within this document, a few terms need to be defined with respect to detector alignment:

- The *ideal geometry* describes the detector as it was designed. Such a detector is an utopic object, which can never be realized in terms of the placement of the individual components as such.
- The *actual geometry* describes the real detector in the configuration at a given time. This includes all the changes i.e. *deltas* to the *ideal geometry*. These changes are also called the *alignment parameters*. These parameters typically are only valid for a defined time interval.
- *Realignment* defines the procedure to apply a new set of temporary *misalignment parameters* to the ideal geometry. Such a procedure is applied, if a previously applied set of parameters is no longer valid with respect to the event data to be processed. In short *realignment* is necessary if the *actual geometry* of the detector is time dependent.

**DDAlign** formalizes both the access and the application of alignment parameters to the ideal geometry. The possibility to properly describe actual geometries with respect to ideal geometries is essential to understand the detector response to particle collisions and to connect response of geometrical independent areas of the experiment e.g. to one single track.

In this manual we will shortly describe the model used to describe an experiments detector description and then in more detail document the support for alignment with its programming interfaces.

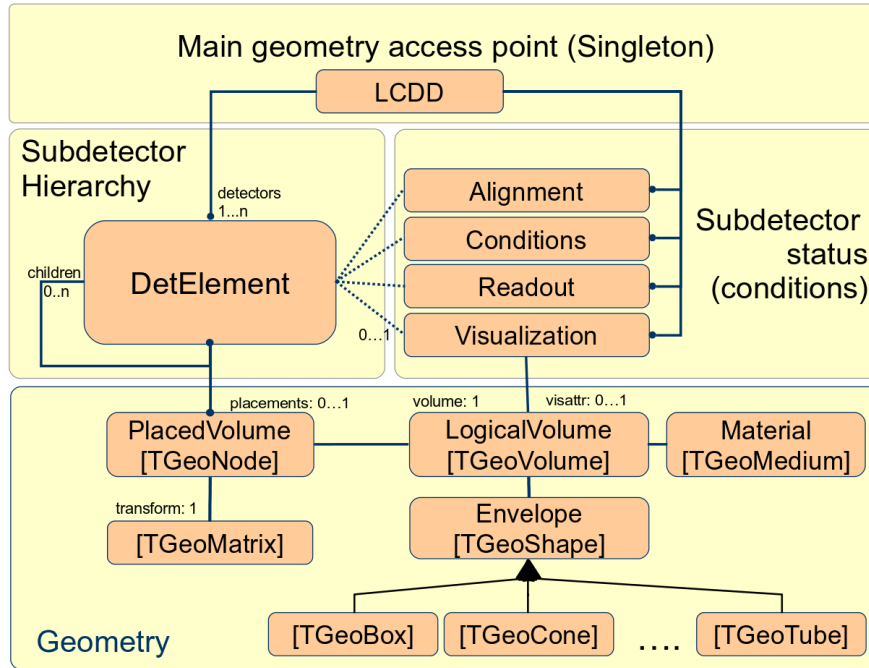


Figure 1: Class diagram with the main classes and their relations for the Generic Detector Description Model. The implementing ROOT classes are shown in brackets.

## 1.1 Generic Detector Description Model

This is the heart of the DD4hep detector description toolkit. Its purpose is to build in memory a model of the detector including its geometrical aspects as well as structural and functional aspects. The design reuses the elements from the ROOT geometry package and extends them in case required functionality is not available. Figure 1 illustrates the main players and their relationships [1]. Any detector is modeled as a tree of *Detector Elements*, the entity central to this design, which is represented in the implementation by the *DetElement* class [2]. It offers all applications a natural entry point to any detector part of the experiment and represents a complete sub-detector (e.g. TPC), a part of a sub-detector (e.g. TPC-Endcap), a detector module or any other convenient detector device. The main purpose is to give access to the data associated to the detector device. For example, if the user writes some TPC reconstruction code, accessing the TPC detector element from this code will provide access to the all TPC geometrical dimensions, the alignment and calibration constants and other slow varying conditions such as the gas pressure, end-plate temperatures etc. The *Detector Element* acts as a data concentrator. Applications may access the full experiment geometry and all connected data through a singleton object called *LCDD*, which provides management, bookkeeping and ownership to the model instances.

The geometry is implemented using the ROOT geometry classes, which are used directly without unnecessary interfaces to isolate the end-user from the actual ROOT based implementation. *DDAlign* allows client to access, manage and apply alignment parameters or smallish changes to the ideal geometry. The mechanism to achieve this is described in the following.

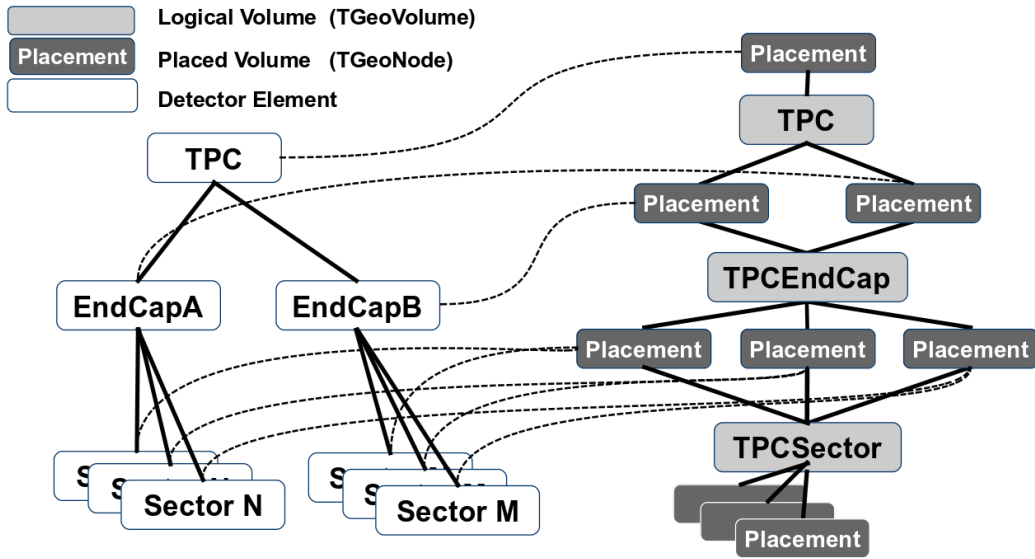


Figure 2: The object diagram of a hypothetical TPC detector showing in parallel the *Detector Element* and the *Geometry* hierarchy and the relationships between the objects.

## 1.2 Detector Element Tree and the Geometry Hierarchy

The geometry part of the detector description is delegated to the ROOT classes. *Logical Volumes* are the basic objects used in building the geometrical hierarchy. A *Logical Volume* is a shape with its dimensions and consist of a given material. They represent unpositioned objects which store all information about the placement of possibly embedded volumes. The same volume can be replicated several times in the geometry. The *Logical Volume* also represents a system of reference with respect to its containing volumes. The reuse of instances of *Logical Volumes* for different placements optimizes the memory consumption and detailed geometries for complex setups consisting of millions of volumes

may be realized with reasonable amount of memory. The difficulty is to identify a given positioned volume in space and e.g. apply alignment parameters to one of these volumes. The relationship between the Detector Element and the placements is not defined by a single reference to the placement, but the full path from the top of the detector geometry model to resolve existing ambiguities due to the reuse of *Logical Volumes*. Hence, individual volumes must be identified by their full path from mother to daughter starting from the top-level volume.

The tree structure of *Detector Elements* is a parallel structure to the geometrical hierarchy. This structure will probably not be as deep as the geometrical one since there would not need to associate detector information at very fine-grain level - it is unlikely that every little metallic screw needs associated detector information such as alignment, conditions, etc. Though this screw and many other replicas must be described in the geometry description since it may be important e.g. for its material contribution in the simulation application. Thus, the tree of Detector Elements is fully degenerate and each detector element object will be placed only once in the detector element tree as illustrated for a hypothetical Time Projection Chamber (TPC) detector in Figure 2 with an ideal geometry, where no positioning corrections are applied to neither child. It is essential to realize that the geometry tree in an ideal geometry is degenerate contrary to the tree of detector elements.

It should be noted, that alignment parameters may be applied to any volume of the ideal geometry. The alignment only affects the actual position of a volume it is e.g. irrelevant if the volume is sensitive or not.

### 1.3 Alignment Parameters of Detector Components

Alignment parameters never apply in the same way to *all* placements of the same volume in this hierarchy. Hence, to (re-)align a volume in the hierarchy means to lift a full branch of placements from the top volume down to the element to be (re-)aligned out of this shared hierarchy and apply a correction matrix to the last node. This procedure is illustrated in Figure 5. Re-alignment of volumes may occur at any level. In the above example of a TPC this results in the following effects:

- A realignment of the entire subdetector, i.e. the TPC as a whole, would affect consequently move all contained children with respect to the top level coordinate system. An example is shown in Figure 5 (a). A movement of the subdetector would affect all transformation between local coordinates of any part of the subdetector to the top level coordinate system. Such effects would be visible at all stages of the data processing e.g. when translating signals from particles into global coordinates.
- A realignment of parts of a subdetector affects only the partial subdetector itself and child volumes at lower levels. As in the example, where the entire subdetector is moved, here only the sectors on one side of the TPC would be affected as shown in Figure 5 (b).
- In Figure 5 (c) within one end-cap of the TPC individual sectors may not be positioned at the ideal location (Figure 5 (c) exaggerates: "flying sectors" are a rather rare case in reality). Finally also the sectors itself could be fragmented and be assemblies of other shapes, which are not ideally placed and may need correction.

The origin of the volume misplacements may be many-fold:

- Elements may be weak and assembled parts move due to weak support structures. This is a common problem e.g. for tracking detectors, where heavy and solid structures dramatically influence the measurement result. Misplaced sectors could e.g. be the consequence of a deforming end-cap frame due to the weight of the sectors.
- Environmental conditions such as the temperature may influence the position or the shape of a volume.
- Some of the measurement equipment may be moved from a parking position into a data taking position such as the two halves of the LHCb vertex detector. Whereas the position of the sensors on each half are known to a very high precision, the position of the absolute position of the two halves with respect to the full experiment may change after each movement.

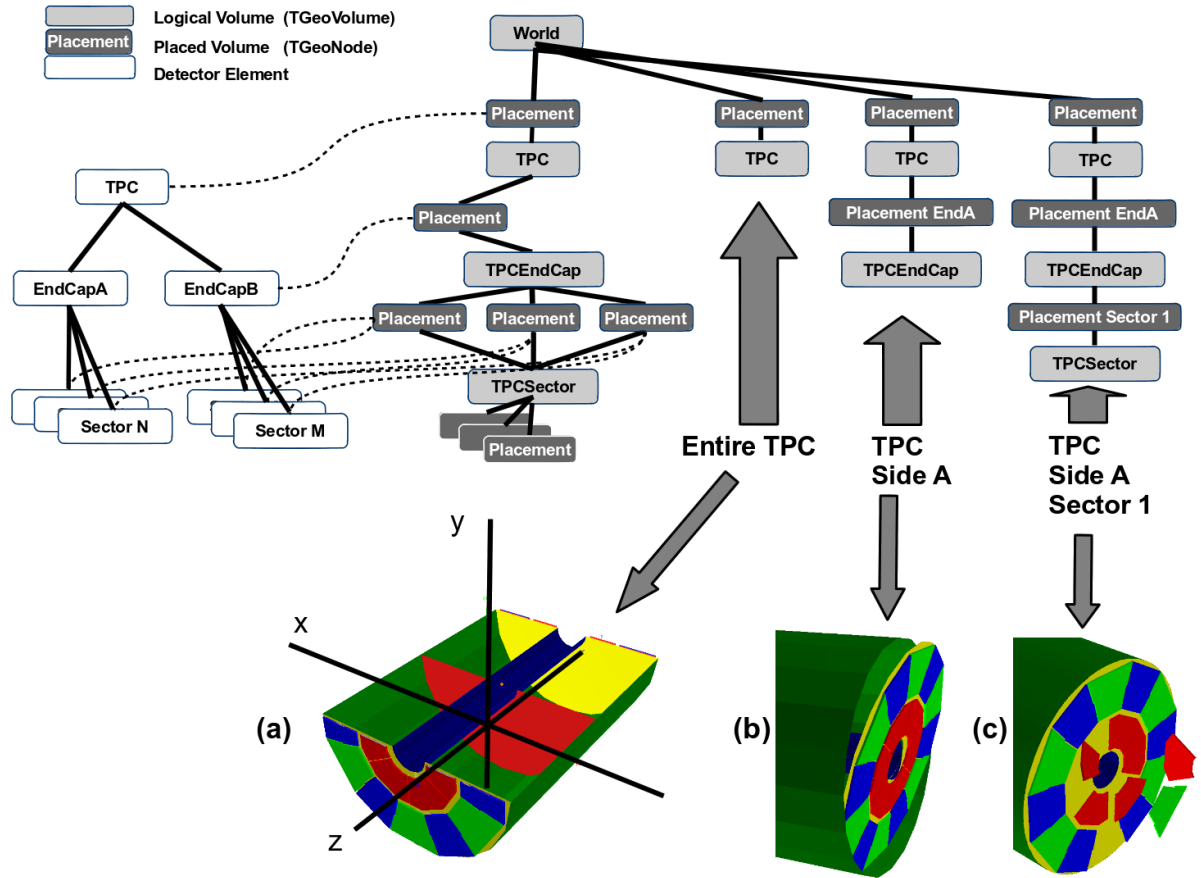


Figure 3: The object diagram of a hypothetical TPC detector showing in parallel the *Detector Element* and the *Geometry* hierarchy and examples of mispositioned detector parts: (a) mispositioned entire subdetector (translation), (b) mispositioned end-cap (tilt) and (c) mispositioned individual sectors within one endcap.

Changes to the volume placement do not only affect sensitive material i.e. detector components with an active readout, but also passive material. The placement of any volume, passive or active, may be corrected using `DDAlign`. The determination of the alignment parameters of passive components however may be more difficult in the absence of located signals resulting e.g. from the traversal of a track.

All effects resulting from such causes obviously need to be corrected in order to fully explore the capabilities of the detection devices and to minimize measurement errors. In general any deviation from the ideal position of a volume can be described by two elementary transformations:

- a translation
- a rotation around a pivot point.

giving a full transformation matrix of the form:

$$T = L * P * R * P^{-1} \quad (1)$$

where

- $T$  is the full transformation in 3D space containing the change to the exiting placement transformation. The existing placement is the placement transformation of the volume with respect to

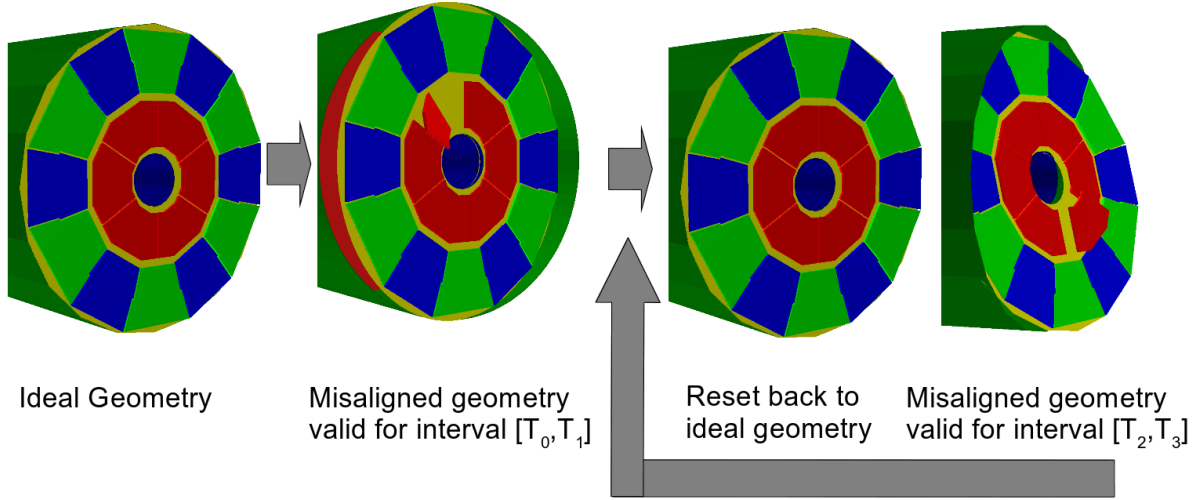


Figure 4: The iterative application of alignment parameters as described in Section 1.4. For each interval of validity ( $[T_0, T_1]$ ,  $[T_2, T_3]$ ,  $[T_4, T_5]$ , ...) a separate set of alignment constants is applied to the ideal geometry. The two steps to reset the misaligned geometry back to the ideal geometry and to re-apply a new set of alignment constants may be executed as often as necessary when processing data from particle collisions.

the mother volume.

- $L$  is a translation specifying the position change with respect to the mother volume.
- $P * R * P^{-1}$  describes a rotation around a pivot point specified in the mother volume's coordinate system.
- $P$  is the translation vector from the mother volume's origin to the pivot point. The concept of a pivot point does not introduce a new set of parameters. Pivot points only help to increase the numerical precision.

Most of the changes do not require the full set of parameters. Very often the changes only require the application of only a translation, only a rotation or both with a pivot point in the origin. These simplifications are supported in the user interface described in Section 2.

## 1.4 Iterative Application of Alignments

In the general case a given set of alignment parameters is not static and may very well change with time. For this reason it is highly important to support not only one single realignment step. Hence, the following scenario is an important use case:

1. Create the ideal detector using an ideal geometry.
2. Apply a set of alignment parameters for a given time interval corresponding to the time a set of particle collisions were collected in the experiment.
3. Process the set of collected particle collisions.
4. Reset the misaligned detector to the ideal.
5. Choose new event data input corresponding to another time interval and restart at item 2.

Graphically this use case is illustrated in Figure 4. In Section 2 the implementation to realize this use case is described.



## 1.5 Procedures to Determine Alignment Parameters

Typically the determination of alignment parameters requires a starting point which is not necessarily identical to the ideal position of a volume [3]. These volume positions are the result of a survey measurement or the result of internal position measurements of a sub-volume within a sub-detector e.g. on a measurement bench. In the following we call these parameters *survey parameters*. *Survey parameters* default to the ideal volume position if not supplied, alternatively, if set, to the provided position. *Survey parameters* are, like the alignment parameters, provided in terms of *changes* with respect to the ideal position and hence may be treated in a similar way.

The survey parameters are - like alignment parameters - accessible to users through the interface offered by the *DetElement* objects.

## 1.6 Simulation of Non-Ideal, Real Detector Geometries

It is a standard procedure in high energy physics to at least verify the measured detector response of a given physics process in particle collisions with the expected simulated detector response. For most purposes the simulation of an ideal detector is certainly sufficient - though not describing the full truth. Sometimes however, the detector geometry must be simulated with a geometry as close to the known geometry as possible.

The simulation of such a geometry with applied alignment parameters can rather easily be realized using the *DD4hep*, *DDAlign* and the *DDG4* frameworks:

- The ideal geometry is constructed using the standard procedures of *DD4hep* [1].
- Then the alignment parameters are applied and finally
- the corrected geometry is translated to *Geant4* [6] using the *DDG4* [4] package. All particle collisions simulated with this translated geometry correspond to the modified geometry including the geometry modifications.

There is a caveat though: The application of alignment parameters can easily create volume overlaps, which are highly disliked by the *Geant4* runtime. If the above described procedure is applied, it is highly advised to check the resulting geometry for overlaps. Both, *ROOT* [5] and *Geant4* [6] offer tools to perform such tests.

## 2 The DDAlign User Interface

DDAlign implements a machinery to apply and access the alignment parameters describing the difference between an ideal detector given by an ideal geometry and the geometry of the actually built assembly in real life. To ease its usage for the clients and to shield clients from the internals when actually dealing with realigned geometries, a set of helper classes was designed. The access to the alignment parameters in read-only mode was separated from the import or export thereof.

As a basic concept within DD4hep any *sizable* detector component can be realigned. *Sizable* as a rule of thumb is anything, which is manufactured as an individual piece and which you may "hold in your hands". Such objects are also described by a *detector element* of type `DetElement`. An example is e.g. a single silicon wafer of a tracking device or the entire tracking detector itself. The access to the alignment parameters is possible from each `DetElement` instance as described in Section 2.1. The interface assumes "planar" alignment parameters i.e. the shape of a given volume does not change <sup>1</sup>. Please be aware that the extensive use of misalignments is highly memory consuming.

### 2.1 Access to Alignment Parameters from the Detector Element

The `DetElement` class as shown in Figure 1 gives the user access to the alignment structure of type `Alignment` as illustrated in the following example:

```

1  DetElement wafer = ... // Valid handle to a detector element
2  Alignment wafer_alignment = wafer.alignment();
3  if ( wafer_alignment.isValid() ) {
4      // This wafer's placement differs from the ideal geometry when
5      // alignment parameters are present.
6
7      // Access the misalignment transformation with respect to the parent volume:
8      Transform3D tr = wafer_alignment.toMotherDelta();
9  }
```

The access to details of an invalid alignment object results in a runtime exception. The following calls allow clients to access alignment information from the `DetElement` structure:

```

1  /// Access to the actual alignment information
2  Alignment alignment() const;
3
4  /// Access to the survey alignment information
5  Alignment surveyAlignment() const;
```

The call to `alignment()` return the parameters *applied* to the the existing ideal geometry. The call `surveyAlignment()` returns optional constants used to perform numerical calculations as described in section 1.5.

All functionality of the `DetElement`, which depends on applied alignment parameters are automatically updated in the event of changes. These are typically the geometry transformations with respect to the mother- and the world volume:

```

1  /// Create cached matrix to transform to world coordinates
2  const TGeoHMatrix& worldTransformation() const;
3
4  /// Create cached matrix to transform to parent coordinates
5  const TGeoHMatrix& parentTransformation() const;
6
7  /// Transformation from local coordinates of the placed volume to the world system
8  bool localToWorld(const Position& local, Position& global) const;
```

<sup>1</sup>This is a restriction to the possibilities provided by the ROOT implementation [5] based on experience [3]. If at a later time the need arises the provided alignment interface may be extended to support shape changes.

```

9
10    /// Transformation from local coordinates of the placed volume to the parent system
11    bool localToParent(const Position& local, Position& parent) const;
12
13    /// Transformation from world coordinates of the local placed volume coordinates
14    bool worldToLocal(const Position& global, Position& local) const;
15
16    /// Transformation from world coordinates of the local placed volume coordinates
17    bool parentToLocal(const Position& parent, Position& local) const;

```

it is worth noting that the update of cached information is performed by the *DetElement* objects, other user defined cached information is **not** updated. To update user caches it is mandatory to provide a user defined update callback to the *DetElement*:

```

1    template <typename Q, typename T>
2    void callAtUpdate(unsigned int type, Q* pointer,
3                      void (T::*pmf)(unsigned long typ, DetElement& det, void* opt_par)) const;

```

The interface of the *Alignment* structure to access detector alignment parameters is as follows (see also the corresponding header file DD4hep/Alignment.h):

```

1    /// Number of nodes in this branch (=depth of the placement hierarchy from the top level volume)
2    int numNodes() const;
3
4    /// Access the placement of this node
5    PlacedVolume placement() const;
6
7    /// Access the placement of the mother of this node
8    PlacedVolume motherPlacement(int level_up = 1) const;
9
10   /// Access the placement of a node in the chain of placements for this branch
11   PlacedVolume nodePlacement(int level=-1) const;
12
13   /// Access the currently applied alignment/placement matrix with respect to the world
14   Transform3D toGlobal(int level=-1) const;
15
16   /// Transform a point from local coordinates of a given level to global coordinates
17   Position toGlobal(const Position& localPoint, int level=-1) const;
18
19   /// Transform a point from global coordinates to local coordinates of a given level
20   Position globalToLocal(const Position& globalPoint, int level=-1) const;
21
22   /// Access the currently applied alignment/placement matrix with respect to mother volume
23   Transform3D toMother(int level=-1) const;
24
25   /// Access the currently applied alignment/placement matrix (mother to daughter)
26   Transform3D nominal() const;
27
28   /// Access the currently applied correction matrix (delta) (mother to daughter)
29   Transform3D delta() const;
30
31   /// Access the inverse of the currently applied correction matrix (delta) (mother to daughter)
32   Transform3D invDelta() const;

```

- The calls in line 3-8 allow access to the relative position of the *nth.* element in the alignment stack with respect to its next level parent. Element *numNodes()* - 1 denotes the lowest level and element 0 is the world volume. The default argument (-1) addresses the lowest placement in the hierarchy.

- Calls in line 9-12 allow to access/execute transformations from a given level in the placement hierarchy to coordinates in the top level volume (world).
- The call in line 14 allows to transform a global coordinate to the local coordinate system in a given level of the hierarchy.
- The call *toMother* in line 16 returns the local transformation of the node at a given level to the mother's coordinate system.
- The calls in line 17-20 give access to the nominal placement matrix of the realigned node with respect to the parent volume and the changes thereof.

Besides these convenience calls the full interface to the class `TGeoPhysicalNode`, which implements in the ROOT geometry package alignment changes, is accessible from the *Alignment* handle using the overloaded *operator* `->()`. Further documentation is available directly from the ROOT site .

## 2.2 Manipulation of Alignment Parameters

There are multiple possibilities to apply alignment parameters:

- The pedestrian way "by hand" using C++ as described in Subsection 2.2.1
- Loading a whole set of misalignment constants from XML, the "poor man's" database. This mechanism is described in Subsection 2.2.2
- Loading a whole set of misalignment constants from a database. This possibility depends heavily on the database and its schema used. A typical use case is to load misalignment constants depending on the experiment conditions at the time the event data were collected. `DDAlign` does not provide an implementation. This possibility here is only mentioned for completeness and will be subject to further developments to support conditions in `DD4hep` .

### 2.2.1 Manipulation of Alignment Parameters for Pedestrians using C++

In this section we describe how to apply geometry imperfections to an existing detector geometry in memory using C++. To apply misalignment to an existing geometry two classes are collaborating, the `AlignmentCache` attached to the geometry container `LCDD` and a temporary structure the `AlignmentStack`. The `AlignmentCache` allows to access all existing alignment entries based on their subdetector. The `AlignmentStack` may exist in exactly one instance and is used to insert a consistent set of alignment entries. Consistency is important because changes may occur at any hierarchical level and internal transformation caches of the ROOT geometry package must be revalidated for all branches containing a higher level node. **For this reason it is highly advisable to apply re-alignment constants for a complete subdetector.** Note that this restriction is not imposed, in principle a consistent set of misalignments may be applied at any level of the geometry hierarchy.

Though the application of alignment is much simpler using XML files, the following description should give an insight on the mechanisms used behind the scene and to understand the concept.

Any manipulations are transaction based must be embraced by the following two calls opening and closing a transaction:

```

1 // Required include file(s)
2 #include "DDAlign/AlignmentCache.h"
3
4 LCDD& lcdd = ....;
5 AlignmentCache* cache = lcdd.extension<Geometry::AlignmentCache>();
6
7 // First things first: open the transaction.
8 cache->openTransaction();
9
10 // Prepare the entry containing the alignment data
11 AlignmentStack::StackEntry* entry = .....;

```

```

12 //.... and add the element to the AlignmentStack .....
13 AlignmentStack::insert(entry);
14
15 // Finally close the transaction. At this moment the changes are applied.
16 cache->closeTransaction();

```

In the following we describe the mechanism to create and prepare the **StackEntry** instances of the above code snippet. The calls to open and close the alignment transaction do not have to be in the same code fragment where also the alignment entries are prepared. However, all changes are only applied when the transaction is closed. The alignment entries do not necessarily have to be prepared in the sequence of the hierarchy they should be applied, internally the entries are re-ordered and follow the geometry hierarchy top to bottom i.e. mother volumes are always re-aligned **before** the daughters are re-aligned.

The **StackEntry** instances carry all information to apply the re-alignment of a given volume. This information contains:

- The transformation matrix describing the positional change of the volume with respect to its mother volume.
- The placement path of the volume to be realigned.
- A flag to reset the volume to its ideal position **before** the change is applied.
- A flag to also reset all daughter volumes to their ideal position **before** the change is applied.
- A flag to check for overlaps after the application of the change and
- the actual precision used to perform this check.

The **ROOT::Math** library provides several ways to construct the required 3D transformation as described in Section 1.3:

```

1 // Required include file(s)
2 #include "DD4hep/Objects.h"
3
4 Position      trans(x_translation, y_translation, z_translation);
5 RotationZYX   rot (z_angle, y_angle, x_angle);
6 Translation3D pivot(x_pivot, y_pivot, z_pivot);
7
8 Transform3D trafo;
9 /// Construct a 3D transformation for a translation and a rotation around a pivot point:
10 trafo = Transform3D(Translation3D(trans)*pivot*rot*(pivot.Inverse()));
11
12 /// Construct a 3D transformation for a translation and a rotation around the origin
13 trafo = Transform3D(rot,pos);
14
15 /// Construct a 3D transformation for a rotation around a pivot point
16 trafo = Transform3D(piv*rot*(piv.Inverse()));
17
18 /// Construct a 3D transformation for a rotation around the origin
19 trafo = Transform3D(rot);
20
21 /// Construct a 3D transformation for simple translation
22 trafo = Transform3D(pos);
23

```

The following code snippet shows how to extract this information from the **DetElement** and prepare such a **StackEntry** instance:

```

1 // Required include file(s)
2 #include "DDAlign/AlignmentStack.h"
3

```

```

4 // Prepare the entry containing the alignment data
5 typedef AlignmentStack::StackEntry Entry;
6 /// Detector element to be realigned
7 DetElement element = ...;
8 /// The transformation describing the relative change with respect to the mother volume
9 Transform3D trafo = ...;
10 /// Instantiate a new alignment entry
11 Entry* entry = new Entry(element);
12 entry->setTransformation(trafo) // Apply the transformation matrix
13     .applyReset(/* argument default: true */) // Set the reset flag
14     .applyResetChildren(/* argument default: true */) // Set the daughter reset flag
15     .checkOverlaps(/* argument default: true */) // Set flag to check overlaps
16     .overlapPrecision(0.001/mm); // With this precision in mm
17
18 /// Now add the entry to the alignment stack:
19 AlignmentStack::insert(entry);

```

The constructor will automatically determine the volumes placement path from the `DetElement`. Then the transformation is applied and the flags to reset the volume, its children and to trigger the overlap checks with the given precision.

When passing the entry to the `AlignmentStack` the `AlignmentStack` takes ownership and subsequently the entry is deleted after being applied to the geometry. For further shortcuts in the calling sequence please consult the `AlignmentStack` header file.

### 2.2.2 Loading Geometrical Imperfections from XML

In this section we describe how to load geometry imperfections and to apply them to an existing geometry. Loading the XML file is done automatically using the standard XML loader plugin provided by `DD4hep`. This plugin is interfaced to the `LCDD` instance and invoked from code as follows:

```

1 LCDD& lcdd = ...;
2 lcdd.fromXML("file:AlepTPC_alignment.xml");

```

To fully exploit the capabilities it is important to understand the interpreted structure of the XML file being processed. At the top level of the primary input file (i.e. the file given to the XML processor) the following structure is expected:

```

1 <alignment>
2 <!-- Open the alignment transaction -->
3 <open_transaction/>
4 <subdetectors> <!-- Container with the list of subdetectors to be processed. -->
5   <detelement path="TPC" reset="true" reset_children="true">
6     <!-- Move the entire TPC in the world volume -->
7     <position x="30" y="30" z="80"/>
8
9     <!-- Now add daughter detector elements -->
10
11     <!-- Twist a bit the entire endcap by rotating it around the x and the y axis -->
12     <detelement path="/world/TPC/TPC_SideA" check_overlaps="false">
13       <position x="0" y="0" z="0"/>
14       <rotation x="-0.2" y="-0.2" z="0"/>
15       <!-- Apply corrections of type Translation*Rotation to a single sector
16       <detelement path="TPC_SideA_sector02" check_overlaps="true">
17         <position x="0" y="0" z="0"/>
18         <rotation x="0.5" y="0.1" z="0.2"/>
19       </detelement>
20     </detelement>

```

```

21
22      <!-- And the full shooting match of transformations for this sector      -->
23      <detelement path="TPC_SideA/TPC_SideA_sector03" check_overlaps="true">
24          <position x="0" y="0"      z="290.0*mm"/>
25          <rotation x="0" y="pi/2" z="0"/>
26          <pivot    x="0" y="0"      z="100"/>
27      </detelement>
28
29      ....
30
31      <!-- Include alignment files to be processed in the context of the "TPC" DetElement
32      <include ref="file-name"/>
33
34  </detElement>
35 </subdetectors>
36
37 <!-- Include alignment files to be processed at the top level context      -->
38 <include ref="file-name"/>
39
40 <!-- Close the alignment transaction -->
41 <close_transaction/>
42 </alignment>

```

The structure of the alignment file explained quickly:

Line	
1	The root tag for the primary alignment file is <code>&lt;alignment/&gt;</code> . The primary tag name is mandatory and actually is used to invoke the correct interpreter.
2,41	Trigger the alignment transaction by specifying the transaction tags in the main XML file.
4	Defintion of the set of <code>subdetectors</code> to be processed. A valid alias for this directove is <code>detelements</code> .
5	The first subdetector: TPC. The subdetector tag is <code>detelement</code> Each <code>detelement</code> may recursively contain other <code>detelement</code> tags. as they were defined in the <code>DetElement</code> hierarchy. Internal <code>detelement</code> elements are processed in the context of the outer element i.e. pathes may be specified relative to the parent or as absolute pathes with respect to the world (starting with a <code>'/'</code> ).
7	Global movement of the TPC
12-20	Realignment entry for the TPC endcap A named <code>TPC.SideA</code>
16-19	Realignment entry for sector named <code>TPC.SideA_sector02</code> of the TPC endcap A. Here the sector is specified directly as a daughter of the endcap. The name of the <code>DetElement</code> is relative to the parent.
23-27	Realignment entry for sector named <code>TPC.SideA_sector03</code> of the TPC endcap A containing a full transformation: $Translation * Pivot * Rotation * Pivot^{-1}$
32	Optionally <code>detelement</code> elements may include other alignment files specifying lower volume levels. These files are interpreted in the context of the calling detector element.
38	Optionally the subdetector alignment constants may be fragmented into several files, which can be loaded using the <code>include</code> directive. Each file could for example describe one single detector.

The specification of any transformation element is optional:

- The absence of a translation implies the origin (0,0,0)
- The absence of a pivot point implies the origin (0,0,0)
- The absence of a rotation implies the identity rotation. Any supplied pivot point in this case is ignored.

The absence of a transformation element is absolutely legal and does not issue any warning or error.

All transformations describe the change of placement with respect to the coordinate system of the closest mother-volume in the volume hierarchy, i.e. translations, rotations and pivot points are local to the mother coordinate system.



Included files may directly start with the `root` tags `subdetectors`, `detelements` or `detelement` and may recursively include other files. Except for the top level these files are processed in the calling context.

And finally, the result:

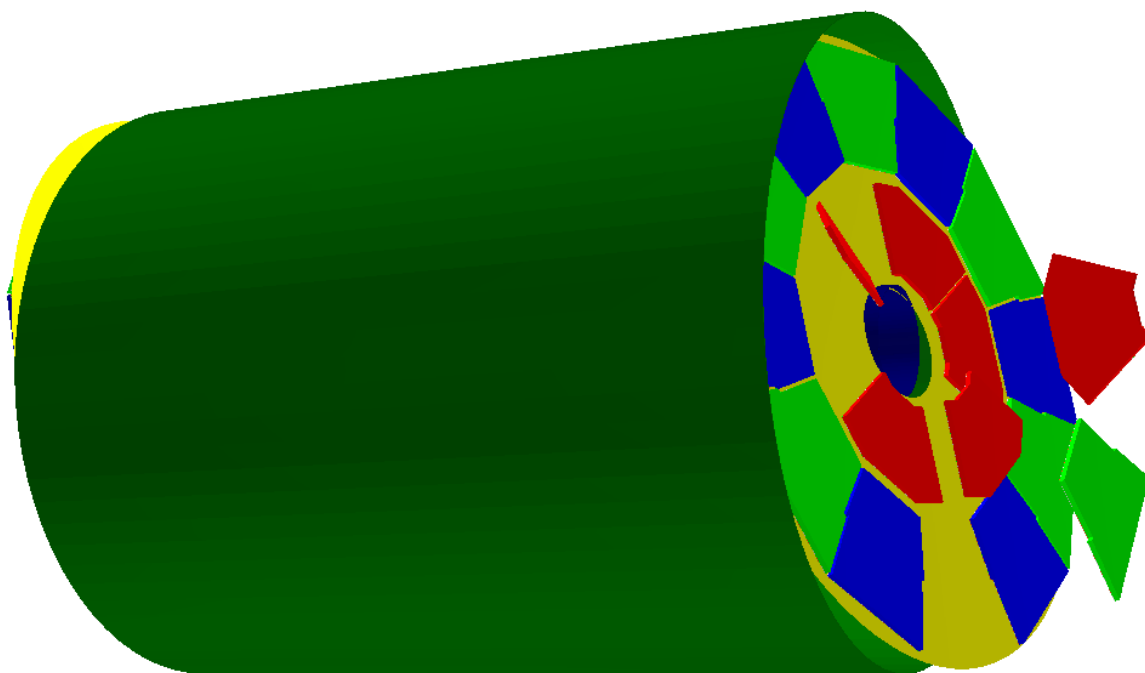


Figure 5: The ALEPH TPC after the import of the alignment file.

### 2.2.3 Export Geometrical Imperfections to XML

In this section we describe how to export geometry imperfections to an XML file. A small helper class `AlignmentWriter` achieves this task as shown in the snippet:

```
1  LCDD& lcdd = ....;
2  DetElement top = ....;
3  if ( top.isValid() ) {
4      AlignmentWriter wr(lcdd);
5      return wr.write(top,output,enable_transactions);
6  }
```

This code will dump all alignment constants contained in the `DetElement` hierarchy of `top` to the output file `output`. The optional argument `enable_transactions` (default: `true`) will add the tags `<open_transaction/>` and `<close_transaction/>` to the output file. The output file conforms to the specifications described in Section 2.2.2 and may later be imported by another process.

## References

- [1] M. Frank et al, "DD4hep: A Detector Description Toolkit for High Energy Physics Experiments", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2013), Amsterdam, Netherlands, 2013, proceedings.
- [2] S. Ponce et al., "Detector Description Framework in LHCb", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2003), La Jolla, CA, 2003, proceedings.
- [3] C. Parkes, private communications.
- [4] M.Frank, "DDG4 - A Simulation Toolkit for High Energy Physics Experiments using Geant4 and the DD4hep Geometry Description".
- [5] R.Brun, A.Gheata, M.Gheata, "The ROOT geometry package", Nuclear Instruments and Methods **A** 502 (2003) 676-680.
- [6] S. Agostinelli et al., "Geant4 - A Simulation Toolkit", Nuclear Instruments and Methods **A** 506 (2003) 250-303.