The TDF C++ Library (Draft)

Last updated 23 May 1997
Draft-time 11:40
for version 1.0

Yotam Medini (yotam_medini@tmai.com)

1 Introduction

Technology Data Format (**TDF**) is a semiconductor data interchange format for wafer data including 1D, 2D and 3D geometries, mesh, fields and other related information.

TDF is developed by '`http://www.tmai.com/`' *Technology Modeling Associates, Inc*

The primary purpose of TDF is to facilitate data interchange between the full range of semiconductor technology simulators and related applications, including visualization, mesh generation and geometry editors. The goal is to provide a unified data model with a comprehensive description of those data types which are common to all semiconductor processing, while providing enough flexibility to allow for application-specific extensions.

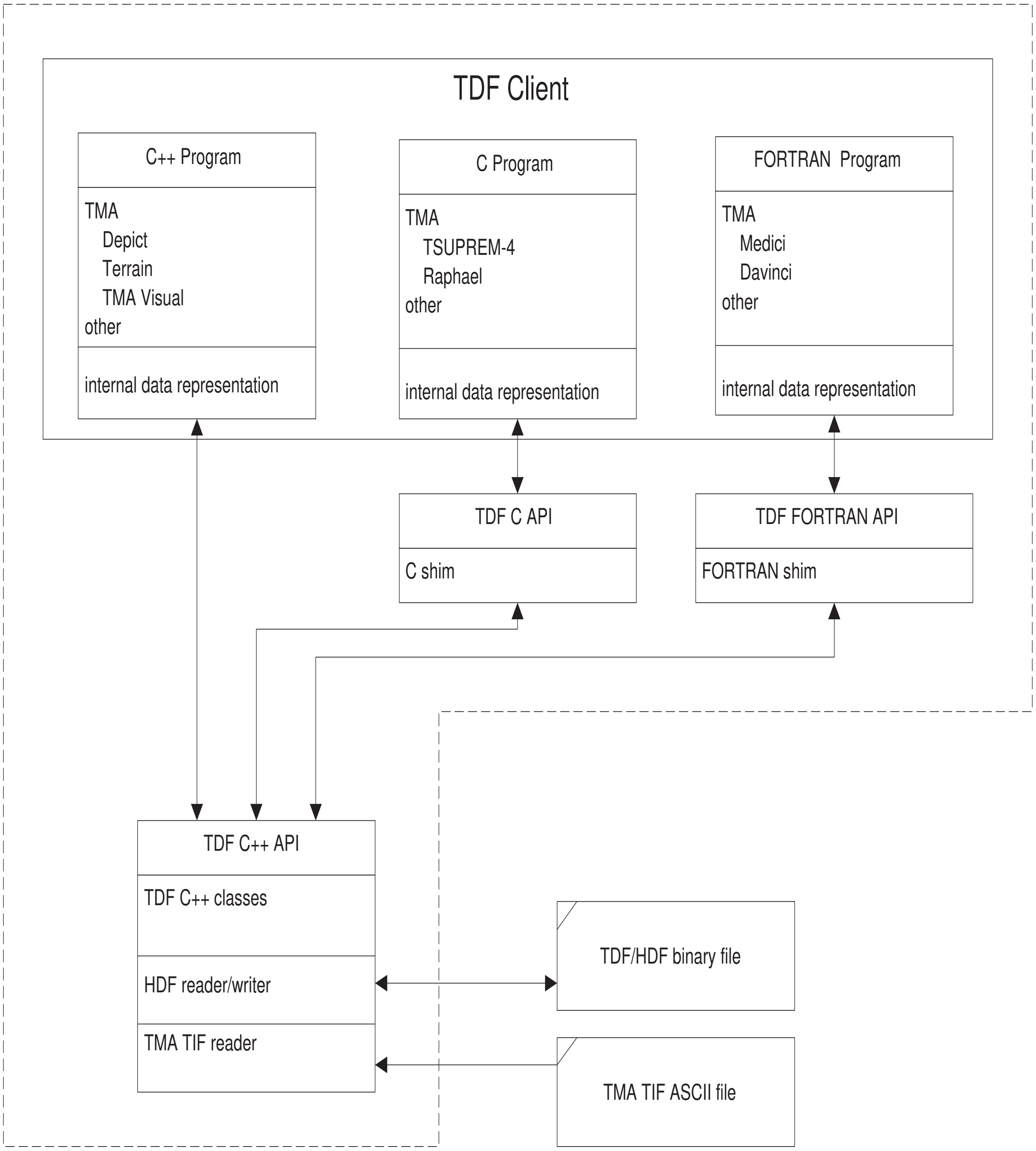
The TDF data storage model is designed for efficiency of storage and access. The data is stored in files, but unlike other file formats, applications using TDF are not forced to read, let alone understand, all of the data stored in a TDF data set; instead the application is free to access the specific portions of the data which are relevant for its own purposes. Accessing a TDF data set can be thought of as accessing a data server, rather than reading/writing a file. As a result, access times tend to be faster than with other formats.

TDF provides persistence using Hierarchical Data Format (HDF), which was developed at the University of Illinois, NCSA. This is a cross-platform, binary format, and the libraries and documentation are available in the public domain. For more information about HDF, see '`http://hdf.ncsa.uiuc.edu`'.

For the convenience of application developers, TDF includes a C++ class library. The C++ API provides a high-level view of the TDF information model, and facilitates the integration of new applications.

Note, that the API uses the C++ **Standard Template Library** (**STL**) see Section 2.8 [STL], page 41.

The following figure illustrates the integration of the TDF library with client programs written in C++, C or FORTRAN. Persistence in HDF files is handled automatically by the TDF library. The library also includes a TMA TIF ASCII file format reader for backward compatibility.



The remainder of this document describes the C++ API. Three figures showing Rumbaugh Object Model diagrams are included to give an overview of the major TDF classes. All class names shown in these diagrams are prefixed by `Tdf` for the actual C++ class names.

2 C++ Classes

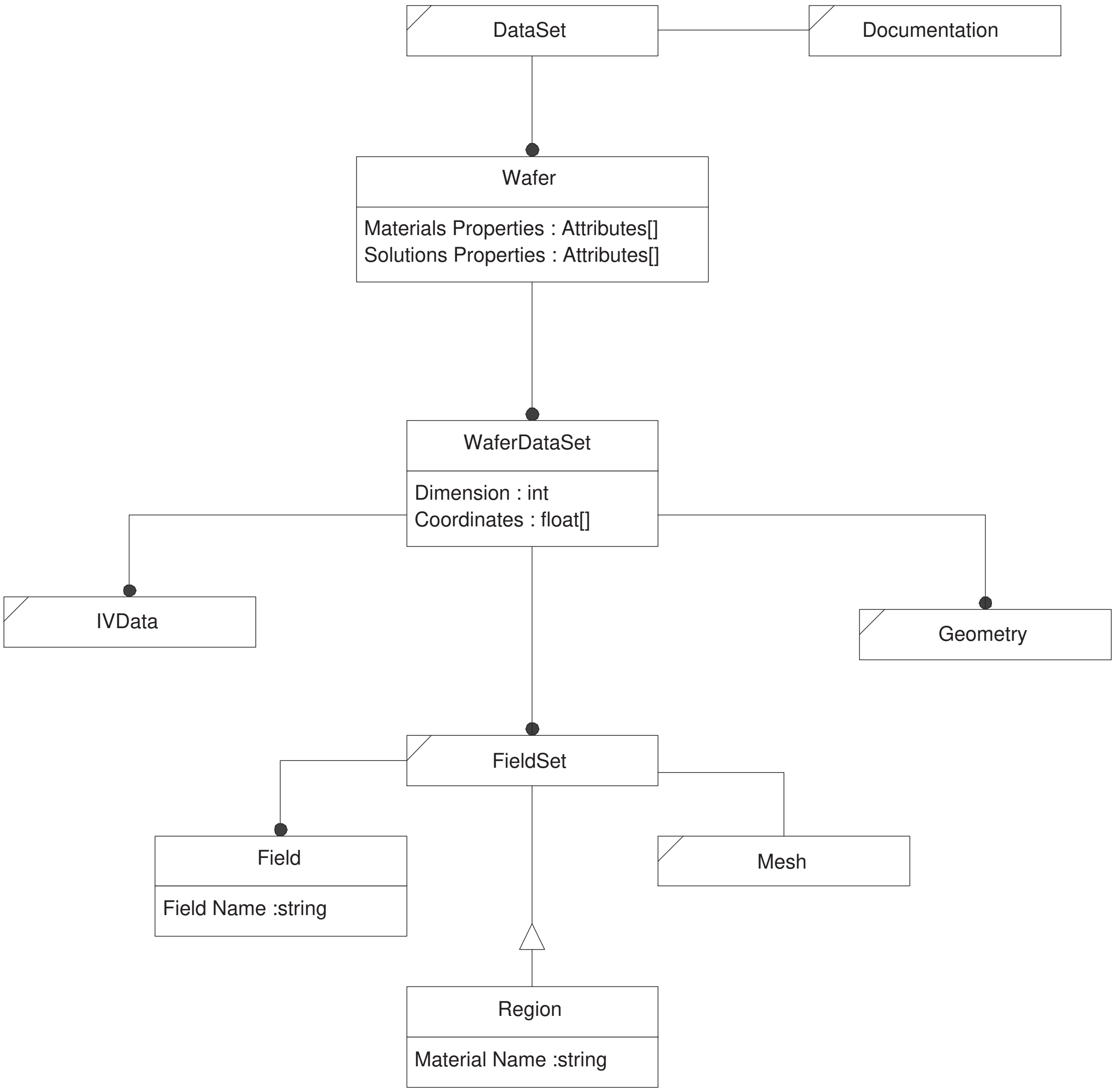
We will start our tour of TDF's **C++** classes. The classes covered are meant to include only those that are of interest for the application.

2.1 Rumbaugh Diagrams

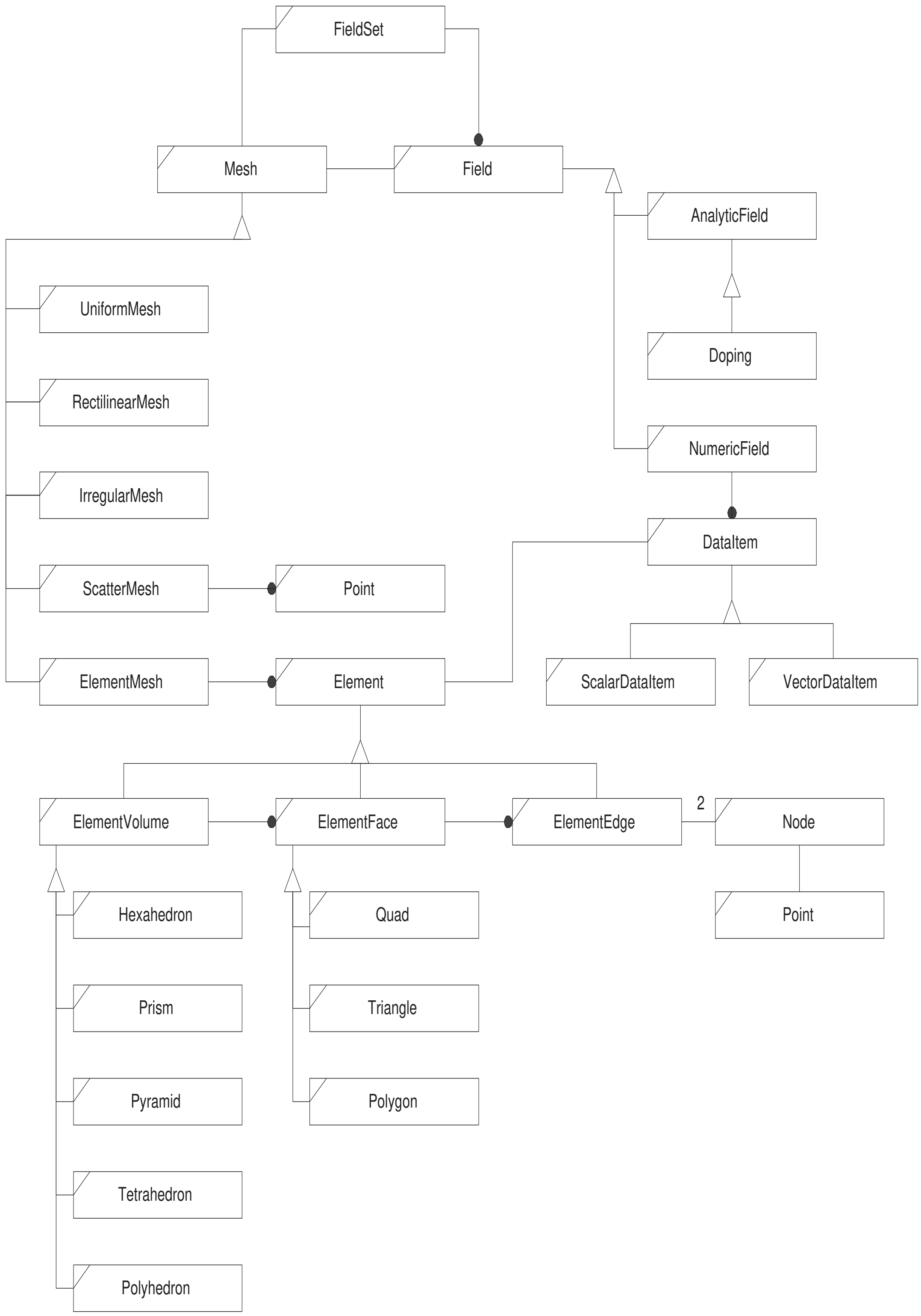
Here are four Object Model Rumbaugh diagrams showing top-level classes.

Note, that actual **C++** classes names are shown here without the `Tdf` Prefix.

2.1.1 Top Classes Diagrams

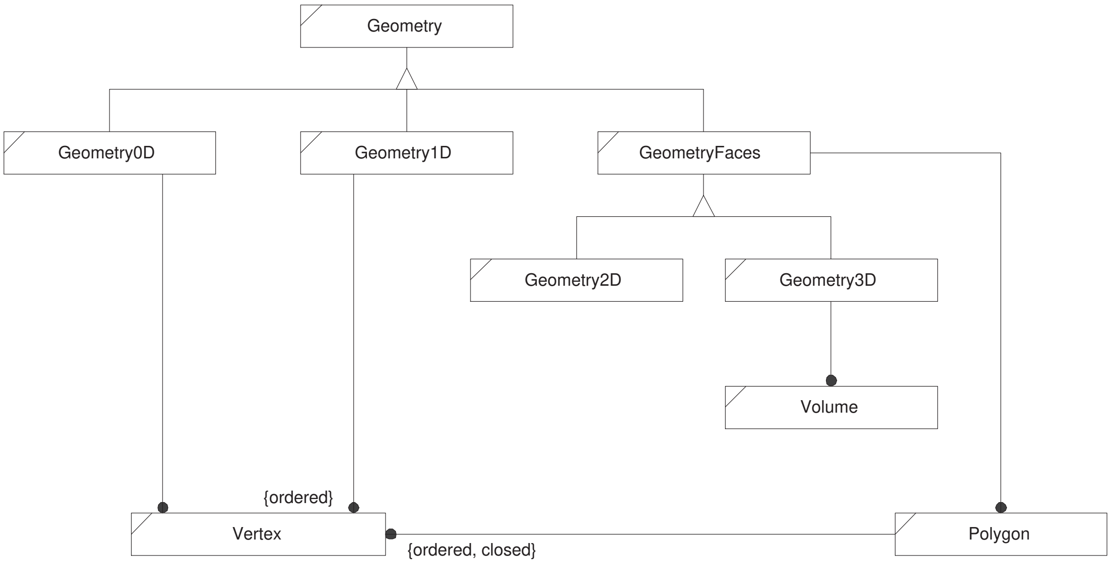


2.1.2 FieldSet – Mesh & Field Classes



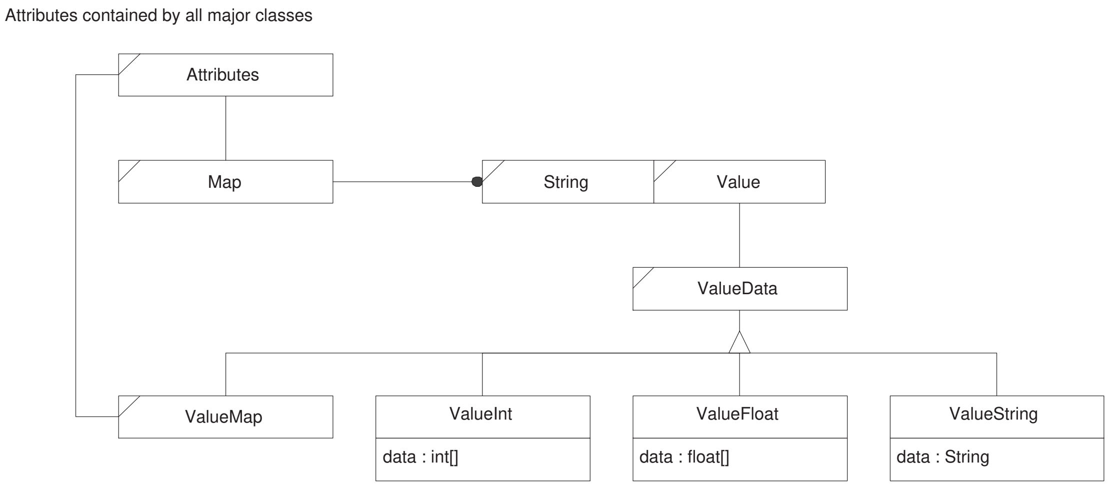
see Section 2.3.1 [FieldSet], page 16 and see Section 2.4.1 [Mesh], page 18.

2.1.3 Geometry Classes



See see Section 2.6.1 [Geometry], page 30.

2.1.4 Attribute Classes



See see Section 2.2.7 [Attributes], page 12.

2.2 Top Classes

Here are the top level C++ classes of the TDF library. TDF uses HDF which is *Hierarchal Data Format*. HDF uses a vgroup/vdata model which is analog to the UNIX directory/file system. Most of the Tdf classes correspond to HDF-vgroups.

The top classes may also be thought of vgroups that are close to the root.

2.2.1 Tdf

`#include "Tdf.hh"`

This class controls the global initialization of **TDF** library.

There are two static functions:

`static bool` **Tdf::init** () — Method

Initialize the **TDF** library. This consists of static table initialization, initial internal database opening, etc. In order to access these databases ('`sol.db`', '`mtr.db`') the environment variable `TDFLIB` has to be a UNIX-path that includes a directory where these files are located. It is safe to call more than once.

`static bool` **Tdf::close** () — Method

Close the **TDF** library. This consists of freeing memory for global variables, closing internal database opening, etc. It is safe to call more than once.

Tdf::Tdf () — Constructor

The `Tdf` class, besides being a place holder for the static `init` and `close` global functions, serves as a convenient dummy object that calls `Tdf::init()` upon construction and calls `Tdf::close()` upon destruction.

2.2.2 TdfDataSet

`#include "TdfDataSet.hh"`

This is the "top most" class to access or generate a TDF file.

`bool` **TdfDataSet::HDFread** (`const char*` *filename*) — Method

Reads and loads the whole core TDF data from a given TDF file name (*filename*) to this class. This creates other instances of TDF classes that could be accessed via this class. Returns `true` if success.

`bool` **TdfDataSet::HDFwrite** (— Method
`const char*` *filename*, `const char*` *writerName*)

Writes the Tdf core class tree that comes off this root into the given file name *filename*.

The *writerName* should be the name of the application, it is appended to the Documentation vgroup (`TdfDocumentation`) of the file. Returns `true` if success.

`const TdfDocumentation&` **TdfDataSet::documentation** () — Method
`const`

Returns a reference to the `TdfDocumentation` class member as a constant (see Section 2.2.3 [Documentation], page 7).

TdfDocumentation& **TdfDataSet::documentation** () Method

Returns a reference to the TdfDocumentation class member (see Section 2.2.3 [Documentation], page 7).

unsigned **TdfDataSet::nWafers** () const Method

Returns the number of TdfWafer contained in this TdfDataSet class instance, typically 1.

TdfWafer& **TdfDataSet::wafer** (unsigned i) Method

Returns a reference to the i'th TdfWafer — which is created if necessary — of this TdfDataSet (see Section 2.2.5 [Wafer], page 8).

const TdfWafer& **TdfDataSet::wafer** (unsigned i) const Method

Similar to the above, but returning a const reference and no creation is made.

2.2.3 TdfDocumentation

```
#include "TdfDocumentation.hh"
```

The TdfDocumentation class carries the history of writers/modifiers of the TdfDataSet it belongs to.

unsigned **TdfDocumentation::tdfVersion** () const Method

Returns the version number of TDF-LIB in which the TDF-file was last written/modified.

unsigned **TdfDocumentation::nModifiers** () const Method

Return the number of modifiers of this TdfWaferDataSet.

const TdfModifier& **TdfDocumentation::modifier** (unsigned *i*) const Method

Return the *i*-th modifier of this TdfWaferDataSet. The first modifier is the original writer.

void **TdfDocumentation::modifierAppend** (const TdfModifier& *mdf*) Method

Appends a modifier "signature" to the data (see Section 2.2.4 [Modifier], page 7).

2.2.4 TdfModifier

```
#include "TdfModifier.hh"
```

This class holds the "signature" of the writer or modifier of the data.

const char* **TdfModifier::name** () const Method

Retrieves the modifier name; typically the application's name.

`void` **TdfModifier::name** (`const char*` *nm*) `const` — Method

Sets the modifier name to *nm*; typically the application's name.

`const char*` **TdfModifier::comment** () `const` — Method

Retrieves the modifier comment; typically application specific.

`void` **TdfModifier::comment** (`const char*` *cm*) — Method

Sets the modifier comment to *cm*; typically application specific.

`long` **TdfModifier::time** () `const` — Method

Retrieves the modifier's original creation time in seconds since 01/01/1970 (Using C-library `time` function).

`ostream&` **TdfModifier::printTime** (`ostream&` *os*) `const` — Method

Prints the modifier's time and date onto *os* output stream in textual form.

2.2.5 `TdfWafer`

`#include "TdfWafer.hh"`

`TdfWafer` is a class that holds a wafer data within `TdfDataSet`. Typically there exist only one `TdfWafer` in a `TdfDataSet`. The `TdfWafer` class contains:

- Material Properties (see Section 2.2.7 [Attributes], page 12).
- Solution (Field) Properties (see Section 2.2.7 [Attributes], page 12).
- Data Sets (see Section 2.2.6 [Wafer Data Set], page 10). Typically, there's only one data-set, but it may hold several data-sets; for example representing the same device at different times.

TdfWafer — Material Properties Interface

`unsigned` **TdfWafer::nMaterialProperties** () `const` — Method

Returns the number of materials with `TdfAttributes` this `TdfWafer` holds.

`const TdfAttributes&` **TdfWafer::materialProperties** (`unsigned` *i*) `const` — Method

Returns a `const` reference to the *i*-th `material`'s attributes of this `TdfWafer`.

`TdfAttributes&` **TdfWafer::materialProperties** (`const char* mtrName`) — Method

Returns a `TdfAttributes` reference to the attributes of this `TdfWafer` material *mtrName* — which is created if necessary.

`int` **TdfWafer::materialPropIndex** (`const char*` *mtrName*) — Method

Returns the index of the material named *mtrName*, or -1 if it does not exist.

`TdfAttributes&` **TdfWafer::materialPropertiesOf** (`const char* mtrName`) `const` — Method

Similar to `materialProperties(mtrName)` but if such `TdfAttributes` does not exist a dummy reference is returned.

TdfWafer — Solution (Field) Properties Interface

Analog to Material Properties.

`unsigned` **TdfWafer::nSolutionProperties** () `const` — Method

Returns the number of solutions (fields) with `TdfAttributes` this `TdfWafer` holds.

`const TdfAttributes&` **TdfWafer::solutionProperties** (`unsigned` *i*) `const` — Method

Returns a `const` reference to the *i*-th `solution`'s attributes of this `TdfWafer`.

`TdfAttributes&` **TdfWafer::solutionProperties** (`const char* solName`) — Method

Returns a `TdfAttributes` reference to the attributes of this `TdfWafer` solution *solName* — which is created if necessary.

`int` **TdfWafer::solutionPropIndex** (`const char*` *solName*) — Method

Returns the index of the solution named *solName*, or −1 if it does not exist.

`TdfAttributes&` **TdfWafer::solutionPropertiesOf** (`const char* solName`) `const` — Method

Similar to `solutionProperties(solName)` but if such `TdfAttributes` does not exist a dummy reference is returned.

TdfWafer — TdfWaferDataSet Interface

`unsigned` **TdfWafer::nWaferDataSets** () `const` — Method

Returns the number of `TdfWaferDataSet`s contained in this `TdfWafer`.

`TdfWaferDataSet&` **TdfWafer::waferDataSet** (`unsigned` *i*) `const` — Method

Returns a `const` reference to the *i*-th `TdfWaferDataSet` contained in this `TdfWafer` (see Section 2.2.6 [Wafer Data Set], page 10).

`TdfWaferDataSet>&` **TdfWafer::waferDataSet** (`unsigned` *i*) `const` — Method

Returns a reference to the *i*-th `TdfWaferDataSet` — which is created if necessary — contained in this `TdfWafer` (see Section 2.2.6 [Wafer Data Set], page 10).

2.2.6 TdfWaferDataSet

`#include "TdfWaferDataSet.hh"`

The `TdfWaferDataSet` class represents a wafer or a structure. It may contain geometry, regions, mesh, element data, node data.

The most important class contained is the `TdfFieldSet` (see Section 2.3.1 [FieldSet], page 16) which is a super-class of `TdfRegion` (see Section 2.3.2 [Region], page 18).

`void` **TdfWaferDataSet::clear** () Method

Clears all data within this `TdfWaferDataSet`.

`unsigned` **TdfWaferDataSet::dimension** () `const` Method

Returns the dimension of this `TdfWaferDataSet`.

`void` **TdfWaferDataSet::dimension** (`unsigned` *d*) Method

Sets the dimension of this `TdfWaferDataSet` to d.

`unsigned` **TdfWaferDataSet::nCoordFlat** () `const` Method

Returns the number of `floats` and the flattened array of coordinates.

For example, if this wafer-data-set is of dimension d and has n nodes of unstructured mesh, then this function will return $d \times n$.

`const vector<float>&` **TdfWaferDataSet::coordinates** () `const` Method

Returns a `const` reference to array of the coordinates. Its size can also be obtained by `TdfWaferDataSet::nCoordFlat()`. For 3D the order is: $x_0, y_0, z_0, x_1, y_1, z_1, \ldots$

`vector<float>&` **TdfWaferDataSet::coordinates** () Method

Non-`const` version of the above.

`unsigned` **TdfWaferDataSet::nCoordNode** () `const` Method

Returns the number of nodes with explicit coordinates. Note that data with structured or uniform mesh does not come with explicit node coordinates.

`const float*` **TdfWaferDataSet::getCoordNode** (`unsigned` *i*, `float*` *f*) Method

Copy the coordinates of the i-th node into f which is also returned.

`const float*` **TdfWaferDataSet::coordNode** (`unsigned` *i*, `const float*` *f*) Method

Sets the d coordinates of the i-th node, where d is the dimension.

Now for some interface functions to the underlying array of field-sets. Here things may vary depending on the nature of the application. Internally, we use array of *pointers* to `TdfFieldSet` (see Section 2.3.1 [FieldSet], page 16). The pointers may actually point to a subclass of `TdfFieldSet`, mostly `TdfRegion` (see Section 2.3.2 [Region], page 18).

`unsigned` **TdfWaferDataSet::nRegions** () `const` Method

Returns the number of instances of `TdfRegions` pointed from this `TdfWaferDataSet`.

`unsigned` **TdfWaferDataSet::nFieldSets** () `const` Method

Note: This function is deprecated. Use `nRegions(`*i*`)` instead.

Returns the number of instances of `TdfFieldSets` (or a sub-class) pointed from this `TdfWaferDataSet`.

`const TdfRegion*` **TdfWaferDataSet::region** (Method
`unsigned` *i*)

Returns a `const` pointer to the *i*-the `TdfRegion` belonging to this `TdfWaferDataSet`. Null is returned if such region does not exist.

`const TdfFieldSet*` **TdfWaferDataSet::fieldSet** (Method
`unsigned` *i*) `const`

Note: This function is deprecated. Use `region(`*i*`)` instead.

Returns a `const` pointer to the *i*-the `TdfFieldSet` (or a sub-class) belonging to this `TdfWaferDataSet`.

`TdfRegion*` **TdfWaferDataSet::region** (Method
`unsigned` *i*)

Returns a pointer to the *i*-the `TdfRegion` of this `TdfWaferDataSet`. If any of the `region(`*j*`)`, where $j \leq i$, do not exist prior to the call then they are created.

`TdfRegion*` **TdfWaferDataSet::newRegion** (Method
`const char*` *rgnName*)

Allocates a new `TdfRegion` and names it *rgnName*. It is similar to:

```
TdfRegion*  rgn = region(nRegions());
rgn->objectName(rgnName);
return(rgn);
```

`TdfFieldSet*` **TdfWaferDataSet::fieldSet** (Method
`unsigned` *i*, `const char*` *subClassName* `= 0`)

Note: This function is deprecated. Use `region(`*i*`)` or `newRegion(.)` instead.

Returns a pointer to the *i*-the `TdfFieldSet` (or a sub-class) belonging to this `TdfWaferDataSet`.

If *subClassName* $\neq 0$ Creates a new appropriate subclass of `TdfFieldSet`. Recognized subclass-names are:

- `TdfFieldSet::gname` (see Section 2.3.1 [FieldSet], page 16).
- `TdfRegion::gname` (see Section 2.3.2 [Region], page 18).

All previously uninitialized `fieldSet(j)` (where $0 \leq j < i$) are created as the type of `fieldSet(`*i*`)`

Now for some interface functions to the underlying array of geometries. Here things may vary depending on the nature of the application. We use array of *pointers* to `TdfGeometry` (see Section 2.6 [Geometry classes], page 30). The pointers may actually point to a subclass of `TdfGeometry`.

`unsigned` **TdfWaferDataSet::nGeometries** () `const` — Method

Returns the number of instances of `TdfGeometry` (or a sub-class) pointed from this `TdfWaferDataSet`.

`const TdfGeometry*` **TdfWaferDataSet::geometry** (`unsigned` *i*) `const` — Method

Returns a `const` pointer to the *i*-the `TdfGeometry` (or a sub-class) belonging to this `TdfWaferDataSet` (see Section 2.6.1 [Geometry], page 30).

`TdfGeometry*` **TdfWaferDataSet::geometry** (`unsigned` *i*, `const char*` *subClassName* = `0`) — Method

Returns a pointer to a sub-class of the *i*-the `TdfGeometry` belonging to this `TdfWaferDataSet`.

If *subClassName* $\neq 0$ Creates a new appropriate subclass of `TdfGeometry`. Recognized subclass-names are:

- `TdfGeometry0D::gname` (see Section 2.6.2 [0D-Geometry], page 31).
- `TdfGeometry1D::gname` (see Section 2.6.3 [1D-Geometry], page 31).
- `TdfGeometry2D::gname` (see Section 2.6.6 [2D-Geometry], page 33).
- `TdfGeometry3D::gname` (see Section 2.6.7 [3D-Geometry], page 34).

All previously uninitialized geometry(j) (where $0 \leq j < i$) are created as the type of `geometry(`*i*`)` (see Section 2.6.1 [Geometry], page 30).

2.2.7 `TdfAttributes`

`#include "TdfAttributes.hh"`

This class maps strings to values (see Section 2.2.8 [Value], page 13). The values could be `int`s, `float`s, `char`-strings and (recursive) attributes-set — `TdfAttributes`.

TDF uses this class in every subclass of `TdfGroup` (see Section 2.7.2 [Group], page 40).

`bool` **TdfAttributes::setValue** (`const char*` *keyStr*, `const TdfValue&` *val*) — Method

Sets the mapping *keyStr* $\rightarrow$ *val* of this attributes-set. Returns `true` if no value was assigned to *keyStr* before.

`bool` **TdfAttributes::delValue** (`const char*` *keyStr*) — Method

Deletes any mapping from *keyStr* in this attributes-set. Returns `true` if indeed there was a deletion.

`unsigned` **TdfAttributes::nAttributes** () `const` — Method

Returns the number of attributes, i.e. (string,value) pairs in this attributes-set.

`const TdfValue*` **TdfAttributes::value** (Method
`const char*` *keyStr*) `const`

Returns a `const` pointer the value assigned to *keyStr* in this attributes-set. If no value is assigned, null is returned.

`const char*` **TdfAttributes::strValue** (Method
`const`

char* *keyStr*) const Convenience function that returns the string value assigned to *keyStr*. Otherwise, null is returned.

`const TdfValue&` **TdfAttributes::operator**[] (Method
`const char*` *keyStr*) `const`

Returns a `const` reference to the value assigned to *keyStr* in this attributes-set. Returns reference to dummy value if not assigned. This is a convenience function, for the case a value is *known* to be set. One may use `TdfAttributes::value(`*keyStr*`)` to check.

`TdfValue&` **TdfAttributes::operator**[] (`const char*` *keyStr*) Method

Returns a reference to the value assigned to *keyStr* in this attributes-set. If necessary, a dummy value is created and *assigned* to *keyStr*.

`TdfAttributes::KeyIter` **TdfAttributes::keyBegin** () `const` Method

Returns an iterator (see Section 2.2.9 [KeyIter], page 15) to the beginning of the keys for which mapping are defined in this attributes-set. **STL**-like.

`TdfAttributes::KeyIter` **TdfAttributes::keyEnd** () `const` Method

Returns an iterator (see Section 2.2.9 [KeyIter], page 15) to the past-end of the keys for which mapping are defined in this attributes-set. **STL**-like.

2.2.8 `TdfValue`

```
#include "TdfValue.hh"
```

`TdfValue` is a class that can hold:

- An empty value (default).
- An `int` or array of `ints` (`vector<int>`).
- A `float` or array of `floats` (`vector<float>`).
- A character string.
- An attribute set. Thus (`TdfAttributes`) (see Section 2.2.7 [Attributes], page 12) becomes a recursive class, since it can contain values of its own type.

There are several classes involved in implementing `TdfValue`, Among them is `TdfValueData`, from which applications needs to know the following `enum`eration, all of type `TdfValueData::Type`:

- `TdfValueData::Undefined`

- `TdfValueData::Int`
- `TdfValueData::Float`
- `TdfValueData::String`
- `TdfValueData::Attributes`

`TdfValueData::Type` **TdfValue::type** () `const` — Method

Returns the enumerated (see above) type of this value.

`unsigned` **TdfValue::nVal** () `const` — Method

Returns the number of elements of this value. It could be > 1 only for arrays, which may exist only for types `TdfValueData::Int` or `TdfValueData::Float`.

Integer Values

`void` **TdfValue::setInt** (`int` *v*) — Method

Sets this value to *v* of `TdfValueData::Int` type.

`int` **TdfValue::getInt** (`int` *v*) `const` — Method

If this hold `TdfValueData::Int` type value returns the value, otherwise return 0.

`void` **TdfValue::setIntArray** (— Method
`const int*` *begin*, `const int*` *end*)

Sets this to `TdfValueData::Int` value-type, with the array int values of the **STL**-like range [*begin*, *end*).

`const vector<int>*` **TdfValue::getIntArray** () `const` — Method

If this holds `TdfValueData::Int` type value then return a `const` pointer to the vector of `ints`. otherwise return null-pointer.

Float Values

`void` **TdfValue::setFloat** (`float` *v*) — Method

Sets this value to *v* of `TdfValueData::Float` type.

`float` **TdfValue::getFloat** (`float` *v*) `const` — Method

If this hold `TdfValueData::Float` type value returns the value, otherwise return 0.0.

`void` **TdfValue::setFloatArray** (— Method
`const float*` *begin*, `const float*` *end*)

Sets this to `TdfValueData::Float` value-type, with the array of `float` values of the **STL**-like range [*begin*, *end*).

`const vector<float>*` **TdfValue::getFloatArray** () `const` — Method

If this holds `TdfValueData::Float` type value then return a `const` pointer to the vector of `floats`. otherwise return null-pointer.

String Values

`void` **TdfValue::setString** (`const char*` *cs*) Method

Sets this value to *cs* of `TdfValueData::String` type.

`void` **TdfValue::getString** (`const string&` *st*) `const` Method

Sets this value to *st* of `TdfValueData::String` type.

`const char*` **TdfValue::setString** (`int` *v*) `const` Method

If this hold `TdfValueData::String` type value returns a `const` pointer to the character-string array. otherwise return empty string `""`.

Attributes Values

Again, this is a recursion. A value of `TdfAttribute` type can map string to values.

`void` **TdfValue::setMap** (`const TdfAttributes*` *v*) Method

Sets this value to the *v* attribute-set. Note that this actually copies *v*. if *v* is null then this value is cleared.

`const TdfAttributes*` **TdfValue::getMap** () `const` Method

If this holds `TdfValueData::Attributes` type value then return a `const` pointer to the value, otherwise return null-pointer.

`TdfAttributes*` **TdfValue::getMap** () Method

If this holds `TdfValueData::Attributes` type value then return a pointer to the value, otherwise return null-pointer.

2.2.9 `TdfAttributes::KeyIter`

`#include "TdfAttributes.hh"`

`TdfAttributes::KeyIter` is an iterator nested class of `TdfAttributes` (see Section 2.2.7 [Attributes], page 12). It can be used to iterate over the keys for which attributes are defined.

`const` **char*** `TdfAttributes::keyIter::operator*` () `const` Method

Returns a `const char*` string which is the key of the (string,value) pair this attribute-iterator points to. This string may be invalidated by future operations on the attribute-set.

`TdfAttributes::keyIter` Method

TdfAttributes::keyIter::operator++ ()

Advances this iterator, and returns its *new* value.

`TdfAttributes::keyIter` Method

TdfAttributes::keyIter::operator++ (`int`)

Advances this iterator, and returns its *original* value.

2.3 FieldSet Classes

The FieldSet classes are classes that describes the mesh and the fields and the relation between them.

Note: In some previous versions of TDF-API there was also `TdfInterface` sub-class which was removed. Also `TdfFieldSet` is considered obsolete as an instance, but is maintained for backward compatibility as a base class for `TdfRegion`.

2.3.1 `TdfFieldSet`

```
#include "TdfFieldSet.hh"
```

Note: It is recommedned that application will the subclass instantiate `TdfRegion` (see Section 2.3.2 [Region], page 18) rather than `TdfFieldSet`.
Still, the member functions of `TdfFieldSet` are valid for `TdfRegion`.

The `TdfFieldSet` class holds

- mesh
- field node data
- field element data

The fields are closely related to the mesh.

The `TdfFieldSet` has a unique mesh. An instance *fs* of `TdfFieldSet` always belongs to a `TdfWaferDataSet` *w* (see Section 2.2.6 [Wafer Data Set], page 10). In the case of unstructured mesh, the coordinates of the mesh nodes are owned by *w*. In such case, the nodes coordinates are shared by all of *w*'s instances of `TdfFieldSets`. If a node *n* is common to more than a single `TdfFieldSet` – say fs_0, fs_1 – of the same `TdfWaferDataSet`, then it must be on the boundaries of the `TdfFieldSets` and it may have specific node-data for each fs_i

`void` **TdfFieldSet::clear** () — Method

Clears all data within this `TdfFieldSet`.

`const TdfMesh*` **TdfFieldSet::mesh** () `const` — Method

Returns a `const` pointer to this `TdfFieldSet`'s mesh (see Section 2.4.1 [Mesh], page 18).

`TdfMesh*` **TdfFieldSet::mesh** () — Method

Returns a pointer to this `TdfFieldSet`'s mesh (see Section 2.4.1 [Mesh], page 18).

`TdfMesh*` **TdfFieldSet::setMesh** (— Method
`const char*` *meshClassName*)

If current mesh exists, clear it, and initialize new mesh subclass according to the specified *meshClassName*. Supported class-names are:

- `TdfMeshElement::gname`
- `TdfMeshRectilinear::gname`
- `TdfMeshUniform::gname`

Since node-data is used more often than element data, we generally omit the term 'node' from the member names. So `TdfFieldSet::nField` and `TdfFieldSet::field` refer to node-data fields.

`unsigned` **TdfFieldSet::nField** () `const` — Method

Returns the number of node-data fields defined for this `TdfFieldSet`.

`const TdfField *` **TdfFieldSet::field** (`unsigned` *i*) `const` — Method

Returns a `const` pointer of the *i*-th node-data `TdfField` of this `TdfFieldSet` (see Section 2.5.2 [Field], page 27).

`TdfField*` **TdfFieldSet::field** (— Method
`unsigned` *i*, `const char*` *subClassName* `= 0`)

Returns a pointer to a sub-class of the *i*-the `TdfFieldSet` belonging to this `TdfFieldSet`.

If *subClassName* $\neq 0$ then a new appropriate subclass of `TdfFieldSet` is created. Recognized subclass-names are:

- `TdfFieldNumeric::gname` (see Section 2.5.3 [Field Numeric], page 28).
- `TdfFieldAnalytic::gname` (see Section 2.5.4 [Field Analytic], page 29).

All previously uninitialized `field(j)` (where $0 \leq j < i$) are created as the type of `field(`*i*`)` (see Section 2.5.2 [Field], page 27.

And now for the element data fields.

`void` **TdfFieldSet::nElementData** () `const` — Method

Returns the number of element-data fields defined for this `TdfFieldSet`.

`const TdfFieldNumeric&` **TdfFieldSet::elementData** (— Method
`unsigned` *i*) `const`

Returns a `const` reference to the *i*-th elementData-data `TdfFieldNumeric&` that belong to this `TdfFieldSet` (see Section 2.5.3 [Field Numeric], page 28).

`TdfFieldNumeric&` **TdfFieldSet::elementData** (— Method
`unsigned` *i*)

Returns a reference to the *i*-th elementData-data `TdfFieldNumeric&` that belong to this `TdfFieldSet` (see Section 2.5.3 [Field Numeric], page 28).

`const TdfAttributes*` **TdfFieldSet::materialProperties** () — Method
`const`

virtual function that returns a pointer to the `TdfAttributes` of the material of this `TdfFieldSet`. For instances of `TdfFieldSet` it returns 0. This method is actually used for the `TdfRegion` subclass (see Section 2.3.2 [Region], page 18).

`const TdfWafer void` **TdfFieldSet::pWafer** () `const` — Method

Returns a `const` pointer to the `TdfWafer` *w* which this `TdfFieldSet` *fs* belong to. Note, there must be a `TdfWaferDataSet` *wds* which belongs to *w* and contains this *fs* (see Section 2.1.1 [Top Classes Diagrams], page 3).

2.3.2 TdfRegion

`#include "TdfRegion.hh"`

The `TdfRegion` is a subclass of `TdfFieldSet` (see Section 2.3.1 [FieldSet], page 16). The only addition here is the material that the region has.

`const char*` **TdfRegion::materialName** () `const` Method

Returns the material name of this `TdfRegion`.

`void` **TdfRegion::materialName** (`const char*` *mn*) Method

Sets the material name of this `TdfRegion` to *mn*.

2.4 Mesh Classes

The mesh classes define the partition of the geometrical structure to finite elements thus also introducing sets of nodes. These elements and nodes are used for assigning or calculating data.

There are several types of meshes (see Section 2.1.2 [FieldSet Classes Diagrams], page 4).

The mesh instances are owned by `TdfFieldSet` (see Section 2.3.1 [FieldSet], page 16), That in turn has a `TdfWaferDataSet` parent (see Section 2.2.6 [Wafer Data Set], page 10). Hence, every instance of a class or sub-class of `TdfMesh` has a well-defined ancestor instance of `TdfWaferDataSet` (grandparent)..

Mesh - Cast Down Functions

There are `const` and non-`const` safe cast down functions for the specific mesh sub-classes. They can also be used to determine the meh-type. If the cast down is not appropriate for the object the null pointer is returned

- `const` TdfMeshElement* TdfMesh::thisMeshElement() `const`
- TdfMeshElement* TdfMesh::thisMeshElement()
- `const` TdfMeshRectilinear* TdfMesh::thisMeshRectilinear() `const`
- TdfMeshRectilinear* TdfMesh::thisMeshRectilinear()
- `const` TdfMeshUniform* TdfMesh::thisMeshUniform() `const`
- TdfMeshUniform* TdfMesh::thisMeshUniform()

2.4.1 TdfMesh

`#include "TdfMesh.hh"`

Here is the enumeration `TdfMesh::Type` of the mesh-types:

`TdfMesh::Uniform`

`TdfMesh::Rectilinear`

`TdfMesh::Irregular`

`TdfMesh::Scatter`

`TdfMesh::Element`, Also referred to as "Unstructured Mesh".

`void` **TdfMesh::clear** () Method

Clears all data within this `TdfMesh`.

`TdfMesh::Type` **TdfMesh::type** () `const` Method

Returns the type of the mesh of this `TdfMesh`.

`unsigned` **TdfMesh::dimension** () `const` Method

Returns the dimensionality of this mesh. If it is not explicitly (see `TdfMesh::dimension(int)`) set, then the parent `TdfWaferDataSet`'s dimension is returned. (see Section 2.4.1.3 [MeshElement], page 22)

`void` **TdfMesh::setDimension** (`int d`) Method

Explicitly define the dimensionality of the mesh. It may be less than the dimensionality of the parent `TdfWaferDataSet`.

`void` **TdfMesh::nNodes** () Method

Returns the number of nodes this mesh defines (pure virtual). Note, for structured mesh this number has no relation with the `TdfWaferDataSet::nCoordNode`.

`int` **TdfMesh::coordIndex** (`unsigned` *nodeIndex*) `const` Method

Maps from mesh-indexing to the entry in the global coordinate arrays. This, of course, is applicable only to meshes that uses such array (see Section 2.4.1.3 [MeshElement], page 22).

`float*` **TdfMesh::nodeCoords** (`float*` *coords*, `unsigned` *ni*) Method
`const`

Returns the coordintates of the *ni*-th node of this mesh into the supplied *coords* array. The dimensionality is taken from `TdfWaferDataSet` ancestor. Returns *coords* for convenience .

`unsigned` **TdfMesh::nElements** () `const` Method

Returns the number of elements of this mesh (pure virtual).

`void` **TdfMesh::elementVertices** (Method
`vector<unsigned>&` *vi*, `unsigned` *ei*) `const`

Returns thru *vi*, the node indices of the *ei*-th element of this mesh.

`const TdfMeshElement*` **TdfMesh::meshElement** () `const` Method

Returns 0 unless this instance of `TdfMesh` is of a subclass `TdfMeshElement` in which case it returns `this`.

2.4.1.1 TdfMeshUniform

`#include "TdfMeshUniform.hh"`

The `TdfMeshUniform` class is the simplest sub-class of `TdfMesh` (see Section 2.4.1 [Mesh], page 18). It divides a rectangle or box (2D or 3D respectively) into equally sized sub rectangles or sub boxes.

Thus, the mesh is determined by:

- Its main diagonal
- The numbers: $n_x, n_y(, n_z)$ that determine how each dimension of the rectangle or box is divided.

Getting Functions

`const float*` **TdfMeshUniform::minXYZ** `() const` Method

Returns a 3-`float` pointer to this uniform mesh *min* vertex of its major diagonal. **Note**: For 2D only the first two numbers are meaningful.

`const float*` **TdfMeshUniform::maxXYZ** `() const` Method

Returns a 3-`float` pointer to this uniform mesh *max* vertex of its major diagonal. **Note**: For 2D only the first two numbers are meaningful.

`const unsigned*` **TdfMeshUniform::nXYZ** `() const` Method

Returns a pointer to a 3-`unsigned` array representing the numbers by which dimension is divided in the uniform mesh. **Note**: For 2D only the first two numbers are meaningful.

Setting Functions

`void` **TdfMeshUniform::minXYZ** `(const float* v3)` Method

Sets the *min* vertex of this uniform mesh's major diagonal to the supplied 3-`float` array pointer *v3*. **Note**: Must supply 3-sized arrays even in 2D case.

`void` **TdfMeshUniform::min** `(unsigned i, float v)` Method

Sets the *i*-th ($i = 0, 1, 2$) component of the *min* vertex of this uniform mesh's major diagonal to *v*.

`void` **TdfMeshUniform::maxXYZ** `(const float* v3)` Method

Sets the *max* vertex of this uniform mesh's major diagonal to the supplied 3-`float` array pointer *v3*.
Note: Must supply 3-sized arrays even in 2D case.

`void` **TdfMeshUniform::max** `(unsigned i, float v)` Method

Sets the *i*-th ($i = 0, 1, 2$) component of the *min* vertex of this uniform mesh's major diagonal to *v*.

void **TdfMeshUniform::nXYZ** (`const unsigned* v3`) Method

Sets the numbers by which this uniform mesh divides the rectangle or box in each dimension to n_x, n_y, n_z. **Note**: Must supply 3-sized arrays even in 2D case, in which, n_z will be ignored.

void **TdfMeshUniform::n** (`unsigned i, float v`) Method

Sets the number by which this uniform mesh divides the rectangle or box in each dimension. **Note**: Must supply 3-sized arrays even in 2D case.

2.4.1.2 `TdfMeshRectilinear`

```
#include "TdfMeshRectilinear.hh"
```

The `TdfMeshRectilinear` class is a sub-class of `TdfMesh` (see Section 2.4.1 [Mesh], page 18). It divides a rectangle or box (2D or 3D respectively) into sub rectangles or sub boxes. This is done by dividing each dimesnion of the rectangle or boxes independently and then taking cartesian products.

We have,

$$x_0 \leq x_1 \leq ...x_{n_X-1}$$

$$y_0 \leq y_1 \leq ...y_{n_Y-1}$$

$$z_0 \leq z_1 \leq ...z_{n_Z-1}$$

We refer to the W_i ($W = x, y.z$) as the *ticks* on the W-dimension.

In the 2D case we should have $\forall i, z_i = 0.$ and $(n_X - 1)(n_Y - 1)$ sub rectangles.

In the 3D case we have $(n_X - 1)(n_Y - 1)(n_Z - 1)$ boxes.

void **TdfMeshRectilinear::nTicks** (`unsigned axis`) `const` Method

Sets the number of ticks this rectilinear mesh has in its *axis*-th dimension.

void **TdfMeshRectilinear::getTicks** (`unsigned axis, float* ticks`) `const` Method

Gets the ticks this rectilinear mesh has in its *axis*-th dimension thru the supplied `float` array *ticks* of sufficient size.

void **TdfMeshRectilinear::setTicks** (`unsigned axis, unsigned nt, const float* ticks`) `const` Method

Sets this rectilinear mesh ticks on in its *axis*-th dimension. It will be assigned *nt* ticks with values given in the `float` array *ticks*.

2.4.1.3 TdfMeshElement

`#include "TdfMeshElement.hh"`

The `TdfMeshElement` class is the most complicated sub-class of `TdfMesh` (see Section 2.4.1 [Mesh], page 18). The main source of complexity comes from the fact that the element of the mesh, that this class carries, may vary in types. Still, for efficiency we would like to be able to access nodes and elements fast.

`TdfMesh::Type` **TdfMeshElement::type** () `const` — Method

Returns `TdfMesh::Element`.

The elements of the mesh defined by an instance of `TdfMeshElement` can be of any of the `enum`eration defined by `TdfElement::Type` (see Section 2.4.1.4 [Element], page 26). We distinguish between element of *fixed*-type topology and general polygons and general polyhedrons. The reason for this as the fact that the fixed type are very common, and treating them separately simplifies the random access of a fixed-type element.

Each element has node indices, pointing to the common coordinate array owned by the `TdfWaferDataSet` ancestor (`TdfGroup::pWDS` see Section 2.7.2 [Group], page 40). These indices are concatenated into a *connection list*. Naturally, connection list will have repetitions of indices, since same vertices appear in different elements.

The `TdfMeshElement` has a separate connection-list for each of the 8 *fixed* elements types; i.e.:
Node, Edge, Triangle, Quad, Tetrahedron, Pyramid, Prism, Hexahedron.
Note that every such fixed type has a constant number of nodes (`TdfElement::nElementVertices` see Section 2.4.1.4 [Element], page 26).

There are member functions accessing these fixed-type connection-list. Rather then specifying them for the each of the 8 fixed types we'll describe their common *pattern* using **foo** and **Foo**. The actual members names are obtained by replacing:

foo with one of:
singleNode, *edge*, *triangle*, *quad*, *tetrahedron*, *pyramid*, *prism*, *hexahedron*

Foo with one of:
SingleNode, *Edge*, *Triangle*, *Quad*, *Tetrahedron*, *Pyramid*, *Prism*, *Hexahedron*

`unsigned` **TdfMeshElement::nFoos** () `const` — Method

Returns the number of *Foo* elements in this `TdfMeshElement` (e.g: `TdfMeshElement:nTriangles()`.

`const vector<unsigned>&` **TdfMeshElement::foo** () `const` — Method

Returns a `const` reference to the connection-list of the *foo* in this `TdfMeshElement` (e.g: `TdfMeshElement:triangle()`).

There is a non-const variant of this function, but it should be used with special care, since it is not safe in term of consistency of arrays lengths, and the synchronizing with the array given by `TdfMeshElement::allNodes()`.

void **TdfMeshElement::fooAppend** (const unsigned* *pu*) const — Method

Appends a copy of *nv* node-indices to the connection-list of the *foo* (fixed-type) element of this `TdfMeshElement`, where *nv* is well defined by *foo*.
For example,

```
TdfMeshElement::triangleAppend(const unsigned *pu)
```

with *pu* points to *nv*=3 indices.

void **TdfMeshElement::foosReshape** (const unsigned *nel*) const — Method

Reshapes the connection-list array of the *foo* (fixed-type) element of this `TdfMeshElement`, to hold *nel* *foo* elements. For example:

```
TdfMeshElement::triangleAppend(unsigned 5)
```

will make room for 5 triangles, i.e. 15 node indices.

In Addition to the "patterned" members there are some interface to the fixed-type element connection lists, where the actual type is given as a parameter.

const vector<unsigned>& **TdfMeshElement::connection** (TdfElement::Type *elemType*) const — Method

Returns a `const` reference to the connection-list of the *elemType* fixed-type element in this `TdfMeshElement`.

vector<unsigned>& **TdfMeshElement::connection** (TdfElement::Type *elemType*) — Method

Returns a reference to the connection-list of the *elemType* fixed-type element in this `TdfMeshElement`.

unsigned **TdfMeshElement::nElements** (TdfElement::Type *elemType*) — Method

Returns the number of elements of the fixed type *elemType* in this `TdfMeshElement`.

TdfElement::Type **TdfMeshElement::elementType** (unsigned *ei*) — Method

Returns the element type of the *ei*-th element. in this `TdfMeshElement`.

const vector<unsigned>& **TdfMeshElement::allNodes** () const — Method

Returns an array of all vertices indices that belongs to the mesh in this `TdfMeshElement`. This array is uniquely sorted and cached. It is recalculated on demand if any of the connection list has been updated.

Now, for the non-fixed element types.

unsigned **TdfMeshElement::nPolygon** () const — Method

Returns the number of arbitrary-number nodes polygons in this `TdfMeshElement`. Note, the triangle or quad specific connection lists are maintained separately.

`unsigned` **TdfMeshElement::nPolyhedron** () `const` Method

Returns the number of arbitrary polyhedrons. in this `TdfMeshElement`.Note: the connection lists of: *tetrahedron*, *pyramid*, *prism*, *hexahedron* are maintained separately.

From now on in this `TdfMeshElement`–section, whenever we say polygon, or polyhedron, we do not refer to the fixed-type elements.

The polygons are maintained internally by two arrays. One that holds the sizes (number of vertices), and the other a simple concatenation of all the vertices indices of all polygons.

`const vector<unsigned>&` **TdfMeshElement::polygonNNodes** () `const` Method

Returns a `const` reference to the array of polygons sizes.

`vector<unsigned>&` **TdfMeshElement::polygonNNodes** () Method

Returns a reference to the array of polygons sizes.

`const vector<unsigned>&` **TdfMeshElement::polygonConnectList** () `const` Method

Returns a `const` reference to the connection list of all polygons.

Thus, if there are n polygons, the p-th polygon P $(0 \leq p < n)$ has

$$N_p = \texttt{polygonNNodes}()[p]$$

vertices.

Let

$$i_p = \sum_{k=0}^{p-1} N_k \qquad (i_0 = 0),$$

then the nodes indices of P are given by:

$$\texttt{polygonConnectList}()[j], \qquad (i_p \leq \quad j \quad < i_p + N_p)$$

`void` **TdfMeshElement::polygonConnectReshape** (`unsigned` l) Method

Reshapes the connection list array of the polygons to hold l node indices.

`void` **TdfMeshElement::polygonConnectAppend** (`const unsigned*` *ua*, `unsigned` n) Method

Appends a polygon with n vertices whose indices are copied from the array pointed by *ua*.

A polyhedron are defined by its boundary which consists of polygons. The polygons are maintained internally by two arrays. One that holds the sizes (number of boundary polygons), and the other a simple concatenation of all the polygon indices of all polyhedrons.

const vector<unsigned>& **TdfMeshElement::polyhedronNBdyPolygons** () const — Method

Returns a `const` reference to the array of polygons sizes.

vector<unsigned>& **TdfMeshElement::polyhedronNBdyPolygons** () — Method

Returns a reference to the array of polyhedrons sizes.

The polyhedron interface is not yet completed.

Boundary

Although the boundary may be calculated from the mesh (See `TdfMeshElement::generateBoundary`), it is more efficient and consistent to store and load it.

The boundary consists of an array of geometries where each is a boundary component. For 2D mesh the boundary is of `TdgGeometry1D` components. For 3D mesh the boundary is of `TdgGeometry2D` components. When the mesh-dimension is the same as the space-dimension (i.e. not the case of 2D-mesh in 3D-space) the first boundary component is external the others are holes.

Note: do not confuse the mesh-boundary discussed here, with the other function that refers to boundary of single elements.

unsigned **TdfMeshElement::nBoundary** () const — Method

Returns the number of boundary components assigned with this `TdfMeshElement`.

TdfGeometry* **TdfMeshElement::boundary** (unsigned i) const — Method

Retrieves a `const` pointer to the *i*-th component of the boundary which is a subclass of `TdfGeometry` (see Section 2.6.1 [Geometry], page 30). Note that it could be calculated via `TdfMeshElement::generateBoundary`.

bool **TdfMeshElement::boundaryAppendComponent** (const TdfGeometry& g i) const — Method

Appends *g* as a boundary component for this `TdfMeshElement`.

void **TdfMeshElement::boundary** (const TdfGeometry& *bdy*) — Method

Sets this mesh's boundary to *bdy*. **Note**: there is no consistency check for this boundary *bdy* to be consistent with the underlying mesh.

bool **TdfMeshElement::generateBoundary** (vector<TdfVGDBase*>& geoms, float coplanarCosine = 0.999) const — Method

Generate the boundary of this element mesh. The boundary is returned thru *geoms*. The caller is responsible for the deletions of geoms pointers.

This calculation is rather complicated. The algorithm basically traverses all the elements' boundaries. These elements' boundaries are edges for 2D, and polygons for 3D. Every element-boundary may be a boundary of one or two elements - but not more! Every internal element-boundary is traversed exactly *twice*. The resulted boundary comprises the union of all the elements' boundaries that were traversed exactly *once*. In the case of 2D-mesh in 3D-space (see later) there is further complexity, that double edges are not "canceled out" if they belong to faces which are coplanar (via *coplanarCosine*). F_1 and F_2 are said to be co-planar if

$$\cos(\angle(\text{normal}(F_1)), \angle(\text{normal}(F_2))) \geq \mathit{coplanarCosine}.$$

- 2D-Mesh
  Boundary is a list of 1D-Geometries explicitly closed polylines. (see Section 2.6.3 [1D-Geometry], page 31
  If it is in 2D-space, the first is the external and the rest are holes.
  If it is in 3D-space, all polylines are oriented as the mesh. Polylines are co-planar. The threshold for being coplanar is by the *coplanarCosine*.
- 3D-Mesh
  Boundary is list of 2D-Geometries (see Section 2.6.6 [2D-Geometry], page 33. The first is the external and the rest are holes.

Notes:

- *geoms* should logically be of the type: `vector<TdfGeometry*>` but we want to save on templates, so we use a super class as a pointer. Safe `Tdfgeometry` cast down functions are available (see Section 2.7.1 [VGDBase], page 40).
- The geometries are dynamically allocated via `new` of some subclass of `TdfGeometry`.

`unsigned` **TdfMeshElement::dimension** `()` `const` Method

Returns the dimensionality of this mesh. Overrides `TdfMeshElement::dimension()` (see Section 2.4.1 [Mesh], page 18). If it is not explicitly given and the mesh is not empty, return the highest element dimension.

`bool` **TdfMeshElement::generateSelfBoundary** `(` Method
`float coplanarCosine = 0.999)`

This is a convenience function that calls `generateBoundary` and sets the resulting boundary.

2.4.1.4 TdfElement

`#include "TdfElement.hh"`

The `TdfElement` provides further interface that may be desired to instantiate elements defined via `TdfMeshElement` (see Section 2.4.1.3 [MeshElement], page 22).

There are some useful static definitions for this class.

`TdfElement` has `Type enum`eration that defines the element geometric type for *nFixedType* basic fixed topology types and for general polygon or polyhedron.

`TdfElement::Node`

`TdfElement::Edge`

`TdfElement::Triangle`

`TdfElement::Quad`

`TdfElement::Tetrahedron`

`TdfElement::Pyramid`

`TdfElement::Prism`

`TdfElement::Hexahedron`

`TdfElement::nFixedType`
: Number of fixed-type topology

`TdfElement::Polygon`

`TdfElement::Polyhedron`

`TdfElement::nType`
: Equals `nFixedType + 2`.

2.5 Field Classes

2.5.1 `TdfFieldDef`

```
#include "TdfFieldDef.hh"
```

2.5.2 `TdfField`

```
#include "TdfField.hh"
```

Fields are used for assigning values to nodes or elements. They could be scalar or multi-valued. The values can be given by numerical data, or calculated by analytic formula.

The `TdfField` is an abstract base class from which other field classes are defined and could be instantiated (see Section 2.1.2 [FieldSet Classes Diagrams], page 4).

`void` **TdfField::clear** () Method
: Clears all data within this `TdfField`.

`TdfFieldDef::Type` **TdfField::type** () Method
: Returns the enumerated type `TdfFieldDef::Type` of the field (see Section 2.5.1 [FieldDef], page 27). Currently, only `TdfFieldDef::Float` – that corresponds to **C**'s `float` – is supported.

`unsigned` **TdfField::order** () `const` Method
: Returns the order of the field. For scalar valued field it is 1, for vector field it may be more. Frequently, the terms *columns* or *number of columns* are used instead.

`unsigned` **TdfField::nRows** () `const` — Method

Returns the number of nodes or elements this field applies to.

`float*` **TdfField::getData** (`unsigned` *i*, `float*` *fa*) `const` — Method

Get the value of this field in its *i*-th row (node or element), *i* `<` `order()`. The data is returned thru the supplied array *fa* that must be of the `order()` size. For convenience, *fa* is returned.

`float*` **TdfField::getDataArray** (`unsigned` *row0*, `unsigned` *nRows*, `float*` *fa*) `const` — Method

Get the values of this field for *nRows* starting from its *row0*-th row (nodes or elements), $\text{row0} + \text{nRows} < \text{order()}$ must hold.
The data is returned thru the supplied array *fa* that must be of the `order()*nRows` size. For convenience, *fa* is returned.

2.5.3 TdfFieldNumeric

```
#include "TdfFieldNumeric.hh"
```

`TdfFieldNumeric` is a subclass of `TdfField` (see Section 2.5.2 [Field], page 27). It holds numerical data, both for scalar or vector field. Internally the data is always flattened to a linear array. Still we have the notion of number of rows and number of columns which is also referred to as *order*.

`void` **TdfField::reserve** (`unsigned` *nRows*) `const` — Method

Hint to this `TdfFieldNumeric` to reserve space for *nRows* rows.

`float*` **TdfFieldNumeric::setData** (`unsigned` *i*, `float*` *fa*) `const` — Method

Copies from *fa* *n* floats to set the *i*-th row numerical data of this `TdfFieldNumeric`, where *n* is the order of this field.

`void` **TdfFieldNumeric::setDataArray** (`unsigned` *row0*, `unsigned` *nRows*, `const float*` *fa*) — Method

Sets $\text{nRows} \times \text{order}$ float values onto *nRows* beginning with *row0*. More efficient than calling `setData` *nRows* times:
`for (r=row0; r<row0+nr; setData(r, fa + r*order()), r++)`

`float*` **TdfFieldNumeric::getDataArray** (`unsigned` *row0*, `unsigned` *nRows*, `float*` *fa*) `const` — Method

Implements the virtual `TdfFieldNumeric::getDataArray(.,.,.)` to be done more efficiently (see Section 2.5.2 [Field], page 27).

`const vector<float>&` **TdfFieldNumeric::data** () `const` Method

Returns a `const` reference to the (flattened) array of numerical data.

`vector<float>&` **TdfFieldNumeric::data** () Method

Returns a reference to the (flattened) array of numerical data.

2.5.4 TdfFieldAnalytic

`#include "TdfFieldAnalytic.hh"`

`TdfFieldAnalytic` is a subclass of `TdfFieldNumeric` (see Section 2.5.3 [Field Numeric], page 28).

2.5.5 TdfAnalyticDoping

`#include "TdfAnalyticDoping.hh"`

`TdfFieldNumeric` is a subclass of `TdfFieldAnalytic` (see Section 2.5.4 [Field Analytic], page 29). It hold 10 numerical constants that define a formula described below.

formula methods not yet documented

Given a box B main diagonal:

$$V_i = (x_i, y_i, z_i) \qquad (i = 1, 2)$$

and scalar values: $\text{peak}, c_x, c_y, c_z$ — the *Analytic Doping* is defined by:

$$\textbf{AnalyticDoping}_{V_1,V_2,\text{peak},c_x,c_y,c_z}(x,y,z) = \text{peak} \cdot e^{-((\frac{x-x_m}{c_x})^2+(\frac{y-y_m}{c_y})^2+(\frac{z-z_m}{c_z})^2)}$$

where (x_m, y_m, z_m) is some point in the box:

$$B = [x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$$

achieving the distance d with (x, y, z):

$$d = d((x,y,z), B) = ||(x,y,z) - (x_m, y_m, z_m)|| = \min_{(x',y',z') \in R} ||(x,y,z) - (x',y',z')||.$$

Note that if $(x, y, z) \in B$ then $(x, y, z) = (x_m, y_m, z_m)$, the term over the exponent is zero and thus

$$\textbf{AnalyticDoping} = \text{peak}.$$

For the 2D case, we consider all z values to be zero.

2.6 Geometry

2.6.1 TdfGeometry

`#include "TdfGeometry.hh"`

The `TdfGeometry` is an abstract class. For each of the $d = 0, 1, 2, 3$ dimensions there corresponds a class `TdfGeometry`dD.

Geometry - Cast Down Functions

There are `const` and non-`const` safe cast down functions for the dimension-specific subclasses. They also can be used to determine dimensionality.

For $d = 1, 2, 3$ the virtual functions — `TdfGeometry::geometry`dD()
return `this` if this is class `TdfGeometry`dD and `0` otherwise.

`const TdfGeometry0D*` **TdfGeometry::thisGeometry0D** () `const` — Method

`TdfGeometry0D*` **TdfGeometry::thisGeometry0D** () — Method

`const TdfGeometry1D*` **TdfGeometry::thisGeometry1D** () `const` — Method

`TdfGeometry1D*` **TdfGeometry::thisGeometry1D** () — Method

`const TdfGeometry2D*` **TdfGeometry::thisGeometry2D** () `const` — Method

`TdfGeometry2D*` **TdfGeometry::thisGeometry2D** () — Method

`const TdfGeometry3D*` **TdfGeometry::thisGeometry3D** () `const` — Method

`TdfGeometry3D*` **TdfGeometry::thisGeometry3D** () — Method

`float*` **TdfGeometry::point** (`float*` *coord*, `unsigned` *i*) `const` — Method

Writes the coordinates value of the *i*-th vertex to the *coord* supplied array of the dimensionality size. For convenience, returns *coord*. A call to this function is valid if and only if this `TdfGeometry` instance belongs to an instance of `TdfWaferDataSet` using its coordinates (see Section 2.2.6 [Wafer Data Set], page 10).

`virtual unsigned` **TdfGeometry::nVertices** () `const` — Method

Returns the number of vertices this `TdfGeometry` has.

2.6.2 TdfGeometry0D

`#include "TdfGeometry0D.hh"`

The `TdfGeometry0D` class is a subclass of `TdfGeometry` (see Section 2.6.1 [Geometry], page 30). It holds discrete vertices. Note that this may not be connected as the other `TdfGeometry`*d*`D` for $1 \leq j \leq 3$

unsigned **TdfGeometry0D::nVertices** () const — Method

Returns the number of vertices this `TdfGeometry0D` has.

unsigned **TdfGeometry0D::vertex** (unsigned *i*) const — Method

Returns the coordinates-index of the `i`-th vertex of this `TdfGeometry0D`.

unsigned **TdfGeometry0D::vertexAppend** (unsigned *i*) const — Method

Appends the coordinates-index of the `i`-th vertex of this `TdfGeometry0D`.

float* **TdfGeometry0D::vertexPoint** (float* *coord*, unsigned *i*) const — Method

Write to *coords* the coordinates of this `TdfGeometry0D`'s *i*-th vertex. Return *coord*.

bool **TdfGeometry0D::hasVertex** (unsigned *ci*) const — Method

Check if this `TdfGeometry0D`'s contains a vertex whose coordinates-index is `ci`.

2.6.3 TdfGeometry1D

`#include "TdfGeometry1D.hh"`

The `TdfGeometry1D` class is a subclass of `TdfGeometry` (see Section 2.6.1 [Geometry], page 30). It contains a connected polyline. Closed polylnes are implemented by having an equal coordinate-index for their first and last vertices.

A polyline $P_1, P_2, P_3, ...P_n$ may be defined as a sequence of edges where vertices are repeated: $[P_1, P_2], [P_2, P_3]...[P_{n-1}, P_n]$

A vertex is represented as an index. An edge is represented as a pair of two indices. The indices are actually used with coordinates arrays of `TdfWaferDataSet` (see Section 2.2.6 [Wafer Data Set], page 10).

TdfGeometry1D — Vertices

unsigned **TdfGeometry1D::vertex** (unsigned *i*) const — Method

Returns the coordinates index of the *i*-th vertex of this `TdfGeometry1D`.

void **TdfGeometry1D::vertexAppend** (unsigned *ci*) const — Method

Appends a vertex coordinates-index *ci* to this `TdfGeometry1D`.

`float*` **TdfGeometry1D::vertexPoint** (`float*` *coord*, `unsigned` *i*) `const` — Method

Returns the coordinates of the *i*-th vertex, thru *coord* - (See `TdfGeometry::point(...)` see Section 2.6.1 [Geometry], page 30).

`void` **TdfGeometry1D::polyLineAppend** (`const unsigned*` *via*, `unsigned` *nv*) `const` — Method

Appends *nv* vertices coordinate indices given in *via* to this `TdfGeometry1D`.

TdfGeometry1D — Edges

`unsigned` **TdfGeometry1D::nEdges** () `const` — Method

Returns the number of edges this `TdfGeometry1D` has (see Section 2.6.4 [Edge], page 32).

`TdfGeometry1D::Edge` **TdfGeometry1D::edge** (*ei*) `const` — Method

Returns the *ei*-th edge of this `TdfGeometry1D` (see Section 2.6.4 [Edge], page 32).

`bool` **TdfGeometry1D::edgeAppend** (`const TdfGeometry1D::Edge` *e*) — Method

Appends the edge *e* to this `TdfGeometry1D` if and only if it is the first edge or its first end-point equals the last end-point of the current poly-line (see Section 2.6.4 [Edge], page 32). The restriction is for ensuring a connected poly-line.

`float*` **TdfGeometry1D::hasEdge** (`unsigned` *v0*, `unsigned` *v1*) `const` — Method

Returns `true` if this `TdfGeometry1D` has the edge [*v0*, *v1*], `false` otherwise.

2.6.4 `TdfGeometry1D::Edge`

```
#include "TdfGeometry1D.hh"
```

The `TdfGeometry1D::Edge` is a trivial nested class of `TdfGeometry1D`.

TdfGeometry1D::Edge::Edge (`unsigned` *v0*, `unsigned` *v1*) — Constructor

Construct an instance of `TdfGeometry1D::Edge` with vertices indices *v0* and *v1*.

`void` **TdfGeometry1D::Edge::v** (`unsigned` *i*, `unsigned` *vi*) — Method

Sets the *i*-th endpoint vertex index of this `TdfGeometry1D::Edge` to *vi*.

`unsigned` **TdfGeometry1D::Edge::v** (`unsigned` *i*) `const` — Method

Returns the *i*-th endpoint vertex index of this `TdfGeometry1D::Edge`.

2.6.5 TdfGeometryFaces

`#include "TdfGeometryFaces.hh"`

This class is a base class for `TdfGeometry2D` (see Section 2.6.6 [2D-Geometry], page 33) and `TdfGeometry3D` (see Section 2.6.6 [2D-Geometry], page 33. It is a container of faces (polygons).

We `typedef TdfGeometryFaces::Face` to be simply,
`vector<unsigned> Face`.

`unsigned` **TdfGeometryFaces::nFaces::v** () `const` Method

Return the number of faces of this `TdfGeometryFaces`

`void` **TdfGeometryFaces::faceAppend** (`const TdfGeometryFaces::Face` *f*) `const` Method

Appends a face to the list of faces of this `TdfGeometryFaces`.

`const` **TdfGeometryFaces::Face** `TdfGeometryFaces::face` (`unsigned` *fi*) `const` Method

Returns a reference for the *fi*-th face of this `TdfGeometryFaces`.

Some convenient functions:

`const` **unsigned** `TdfGeometryFaces::faceNEdges` (`unsigned` *fi*) `const` Method

Returns the number of edges of the *fi*-th face of this `TdfGeometryFaces`.

`const` **unsigned** `TdfGeometryFaces::faceNEdges` (`unsigned` *fi*, `unsigned` *vi*) `const` Method

Returns the coordinate index of the *vi*-th vertex of the *fi*-th face of this `TdfGeometryFaces`.

`const` **unsigned** `TdfGeometryFaces::nFacesVertices` () `const` Method

Returns the sum of all faces number of vertices of this `TdfGeometryFaces`. Repeated vertices are counted. So cube-faces will give 24=6*4 (not 8).

2.6.6 TdfGeometry2D

`#include "TdfGeometry2D.hh"`

The `TdfGeometry2D` class represents a 2-Dimensional linear Geometry. It is a subclass of `TdfGeometryFaces` (see Section 2.6.5 [Faces-Geometry], page 33).

There are two distinctive cases:

- This Geometry belongs to a 2D world. In this case the first face represents the outer boundary; the remaining faces (if exist) are holes. The convention is that the outer boundary polygon is anti-clockwise (mathematically postive) and the other are clockwise. In any case the *region* is always on the left side.
- This Geometry belongs to a 3D world. In this case the geometry represents a surface made of polygons with no specific order.

Functions applicable to a 2D-world case.

const TdfGeometryFaces::face& **TdfGeometry2D::outerBoundary** () const — Method

Returns the 1st face of this `TdfGeometry2D`.

unsigned **TdfGeometry2D::nHoles** () const — Method

Returns the number of holes of this `TdfGeometry2D`.

const TdfGeometryFaces::face& **TdfGeometry2D::innerBoundary** (unsigned *bi*) const — Method

Returns the face representing the *bi*-th hole of this `TdfGeometry2D`.

2.6.7 TdfGeometry3D

```
#include "TdfGeometry3D.hh"
```

`TdfGeometry3D` is a class representing a connected polyhedra that may have holes. It is a subclass of `TdfGeometryFaces` (see Section 2.6.6 [2D-Geometry], page 33).

With some analogy with `TdfGeometry2D`, the geometry of `TdfGeometry3D` is assumed to be connected, but not necessarily simply connected; i.e. it may have holes. The first volume (`TdfGeometry3D::volume(0)`) is the outside boundary. If `TdfGeometry2D` :: `nVolumes`() > 1 then the rest of the volumes are the boundaries of holes. Note, that we do not allow a 3D-geometry composed of volumes within 3D-holes. Such a geometry should be divided into several (connected) `TdfGeometry3D`s.

The `TdfGeometry3D` class contains volumes, each of which is an array of face indices.

We `typedef TdfGeometry3D::Volume` to be simply,
`vector<unsigned> Face`.

unsigned **TdfGeometry3D::nVolumes** () const — Method

Returns the number of volumes this `TdfGeometry3D` has.

unsigned **TdfGeometry3D::nHoles** () const — Method

Returns the number of holes this `TdfGeometry3D` has.

void **TdfGeometry3D::volumeAppend** (const Volume& *v*) — Method

Appends the volume *v* to the array of volumes of this `TdfGeometry3D`.

const TdfGeometry3D::Volume& **TdfGeometry3D::volume** (unsigned *vli*) — Method

Returns a `const` reference to the *vli*-th volume of this `TdfGeometry3D`.

Note: that for (*vli* > 0) it means the (*vli*-1)-th hole.

const TdfGeometry3D::Volume& **TdfGeometry3D::outerBoundary** (unsigned *vli*) — Method

Returns a `const` reference to outer boundary which is the 1st volume of this `TdfGeometry3D`. Equivalent to `volume(0)`.

const TdfGeometry3D::Volume& **TdfGeometry3D::innerBoundary** (unsigned *bi*) Method

Returns a `const` reference to the *bi*-th hole's volume of this `TdfGeometry3D`. Equivalent to `volume(`*bi*`-1)`.

unsigned **TdfGeometry3D::volumeNFaces** (unsigned *vli*) Method

Returns The number of boundary faces of the *vli*-th volume of this `TdfGeometry3D`. See note about *vli* in `TdfGeometry3D::volume(`*vli*`)`).

unsigned **TdfGeometry3D::volumeFaceNVertices** (unsigned *vli*, unsigned *fi*) Method

Returns The number of vertices in the *fi*-th face of the *vli*-th volume of this `TdfGeometry3D`. See note about *vli* in `TdfGeometry3D::volume(`*vli*`)`).

unsigned **TdfGeometry3D::volumeFaceVertex** (unsigned *vli*, unsigned *fi*, unsigned *vi*) Method

Returns The coordinate index of the *vi*-th vertex in the *fi*-th face of the *vli*-th volume of this `TdfGeometry3D`. See note about *vli* in `TdfGeometry3D::volume(`*vli*`)`).

2.6.8 Specific Geometries

The motivation for specific geometries comes mainly from the **Raphael** '`http://www.tmai.com/PRODUCT/raphael.html`' application.

2.6.9 TdfXform

```
#include "TdfXform.hh"
```

This class defines a 3D transformation.

It defines two `typedefs`

- `typedef float Mat3x3[3][3]` — for rotation.
- `typedef float XMat[4][3]` — for rotation and translation.

bool **TdfXform::isIdentityXform** () const Method

Returns whether this `TdfXform` represents identity transformation, which is the default, and actually does not carry any `float` data.

void **TdfXform::identityXform** () Method

Declare this `TdfXform` to represents identity transformation. Actually it will delete internal `float` data if it exists from previous settings.

const TdfXform::XMat **TdfXform::matrix** () const Method

Returns a `const` reference to the transformation matrix. If it is an identity, the reference is to a static matrix.

`void` **TdfXform::matrix** (`const TdfXform::XMat&` *x*); Method
Sets this transformation to the value of given *x* matrix.

`void` **TdfXform::translate** (`const float*` *v3*) Method
Translate this transformation by the given *v3* vector.

`void` **TdfXform::rotate3D** (`const TdfXform::Mat3x3&` *mat*) Method
Rotate this 3D-transformation by the given *mat* matrix.

`static float` **TdfXform::determinant** (`const TdfXform::Mat3x3&` *mat*) Method
Rotate this transformation by the given *mat* matrix.

`float` **TdfXform::rotate2D** () `const` Method
Returns the rotation angle of this 2D-transformation.

`void` **TdfXform::rotate2D** (`float` *ang*) Method
Rotate this 2D-transformation by the given *ang* angle.

`float*` **TdfXform::xformVec** (`float*` *r3*, `const float*` *v3*)) Method
Transform the given *v3* vector into the given buffer *r3*. Return *r3* for convenience.

2.6.10 `TdfGeomXform`

```
#include "TdfGeomXform.hh"
```

This class serves as a base class for other specific geometries that carry a transformation matrix with them.

`const TdfXform&` **TdfGeomXform::xform** () `const` Method
Returns a `const` reference to this `TdfGeomXform`'s transform (see Section 2.6.10 [GeomXform], page 36).

`TdfXform&` **TdfGeomXform::xform** () Method
Returns a reference to this `TdfGeomXform`'s transform (see Section 2.6.10 [GeomXform], page 36).

`const TdfXform::XMat&` **TdfGeomXform::xform** () `const` Method
Returns a `const` reference to this `TdfGeomXform`'s transform matrix (see Section 2.6.10 [GeomXform], page 36).

2.6.11 TdfGeomArcCircle

`#include "TdfGeomyArcCircle.hh"`

`TdfGeomArcCircle` is a class representing a circle or a arc-section of a circle. It is a subclass of `TdfGeomXform` (see see Section 2.6.10 [GeomXform], page 36).

`void` **TdfGeomArcCircle::setValues** (Method
  `float` *cx*, `float` *cy*,
  `float` *r*,
  `float` *ang1*`=0.`, `float` *ang2*`=0.`)

Sets this circle-arc to be centered at (*cx*,*cy*) with radius *r*, with *ang1* as starting angle and *ang2* as ending angle.

`void` **TdfGeomArcCircle::getValues** (Method
  `floa&t` *cx*, `float&` *cy*,
  `float&` *r*,
  `float&` *ang1*, `float&` *ang2*) `const`

Returns this circle-arc center thru (*cx*,*cy*), radius thru *r*, and starting/ending angles thru *ang1* and *ang2*.

There are also queries for specific circle parameters:

`float` **TdfGeomArcCircle::cx** () `const` Method

Returns this circle's center **x**-component.

`float` **TdfGeomArcCircle::cy** () `const` Method

Returns this circle's center **y**-component.

`float` **TdfGeomArcCircle::r** () `const` Method

Returns this circle's radius.

`float` **TdfGeomArcCircle::ang1** () `const` Method

Returns this circle-arc's starting angle.

`float` **TdfGeomArcCircle::ang2** () `const` Method

Returns this circle-arc's ending angle.

2.6.12 TdfGeomCylinder

`#include "TdfGeomCylinder.hh"`

`TdfGeomCylinder` is a class representing a 3-dimensional cylinder. It is a subclass of `TdfGeomXform` (see see Section 2.6.10 [GeomXform], page 36).

`bool` **TdfGeomCylinder::setValues** (Method
  `const float*` *center*, `const float*` *direction*,
  `float` *rad*, `float` *h*)

Sets this cylinder to be centered at (*center*[0],*center*[1],*center*[2]) with radius *rad* and height *h*.

The direction is given by a non zero vector — (*direction*[0],*direction*[1],*direction*[2]).

The function returns `true` if successfully set.

`void` **TdfGeomCylinder::getValues** (Method
`float*` *center*, `float*` *direction*,
`float&` *rad*, `float&` *h*)

Returns this cylinder's: center, direction, radius and height thru: *center*, *direction*, *rad* and height *h* respectively.

There are also queries for specific cylinder parameters:

`const float*` **TdfGeomCylinder::center** () `const` Method

Returns this cylinder's center.

`float*` **TdfGeomCylinder:direction** (`float*` *dir3*) `const` Method

Calculates and returns this cylinder's: direction thru the supplied 3-`float` array *dir3*.

`float` **TdfGeomCylinder::r** () `const` Method

Returns this cylinder's radius.

`float` **TdfGeomCylinder::h** () `const` Method

Returns this cylinder's height.

2.6.13 TdfGeomRect

```
#include "TdfGeomRect.hh"
```

`TdfGeomRect` holds a 2-Dimensional rotate-able rectangle. It is a subclass of `TdfGeomXform` (see see Section 2.6.10 [GeomXform], page 36).

`void` **TdfGeomRect::setValues** (Method
`float` *cx*, `float` *cy*,
`float` *w*, `float` *h*,
`float` *ang*)

Sets this rectangle to be centered at (*cx*,*cy*) with sizes (*w*,*h*) rotated by angle *ang*.

`void` **TdfGeomRect::getValues** (Method
`float&` *cx*, `float&` *cy*,
`float&` *w*, `float&` *h*,
`float&` *ang*) `const`

Returns this rectangle's values. The center thru (*cx*,*cy*), the sizes thru (*w*,*h*) and the rotation angle thru *ang*.

There are also queries for specific rectangle parameters:

`float` **TdfGeomRect::cx** () `const` Method

Returns this rectangle's center `x`-component.

float **TdfGeomRect::cy** () const Method

Returns this rectangle's center **y**-component.

float **TdfGeomRect::ang** () const Method

Returns this rectangle's rotation angle.

float **TdfGeomRect::w** () const Method

Returns this cylinder's width.

float **TdfGeomRect::h** () const Method

Returns this cylinder's height.

2.6.14 TdfGeomSphere

```
#include "TdfGeomSphere.hh"
```

TdfGeomSphere is a class representing a 3-dimensional sphere. It is a subclass of TdfGeomXform (see see Section 2.6.10 [GeomXform], page 36).

void **TdfGeomSphere::setValues** (Method
const float* *center*, float *r*)

Sets this sphere to be centered at (*center*[0],*center*[1],*center*[2]) with radius *r*.

setValues **TdfGeomSphere::getValues** (Method
float* *center*, float& *r*) const

Returns this sphere paremeters. The center thru the supplied 3-float array *center*, the radius thru *r*.

float **TdfGeomSphere::cx** () const Method

Returns this sphere's center **x**-component.

float **TdfGeomSphere::cy** () const Method

Returns this sphere's center **y**-component.

float **TdfGeomSphere::cz** () const Method

Returns this sphere's center **z**-component.

float **TdfGeomSphere::r** () const Method

Returns this sphere's radius.

2.7 Miscellaneous Classes

2.7.1 TdfVGDBase

`#include "TdfVGDBase.hh"`

Actually, the `TdfVGDBase` should not be of the application's concern. It is a base class for both

- `TdfGroup` (see Section 2.7.2 [Group], page 40) that represents **HDF**-*vgroup*.
- `TdfLeaf` (Not documented) that represents **HDF**-*vdata*.

For the sake of reducing the number of internal template code, the **TDF** library uses arrays (STL-`vectors`) of pointers `TdfVGDBase*` to represent arrays of subclasses of `TdfGroup` (HDF-vgroups) or subclasses of `TdfLeaf` (HDF-vdata).

The drawback is some loss of type safety. This is somewhat compensated by a rich collection of safe cast down functions for most of `TdfVGDBase` subclasses.

We have the following family (*Foo*) of safe cast down function (See the **C** preprocessor smart macros `TdfThisCastDown0` and `TdfThisCastDown` in '`TdfVGDBase.hh`').
Some possible *Foo*'s: `Group`, `Attributes`, `Field`, `FieldNumeric`, `FieldSet`, `Geometry`, `Mesh`, `Modifier`, `Wafer` and `WaferDataSet`.

`const Tdf`*Foo*`*` **TdfVGDBase::this***Foo* () `const` Method

If this `TdfVGDBase` is `Tdf`*Foo* or a subclass of `Tdf`*Foo*, cast `this` to `Tdf`*Foo*`*`.

`Tdf`*Foo*`*` **TdfVGDBase::this***Foo* () Method

Non `const` version of the above.

Thus, we can use

```
#include "TdfGeometry1D.hh"
int  nEdgesIfGeom1D(const TdfVGDBase* p)
{
    int  ne = -1; // undefined for non 1D-geometry
    const TdfGeometry*  pGeo = p->thisGeometry();
    if (pGeo)
    { // is geometry! now see if it is 1D
       const TdfGeometry1D  pGeo1D = pGeo->thisGeemetry1D();
       if (pGeo1D)  ne = TdfGeometry1D->nEdges();
    }
    return ne;
}
```

2.7.2 TdfGroup

`#include "TdfGroup.hh"`

The `TdfGroup` class is an abstract base class from which all other **TDF** classes that represents HDF-Vgroups are derived (see Chapter 1 [Introduction], page 1).

Some members that are documented here, are actually defined in classes that `TdfGroup` is derived from, but hopefully they are not of direct concern to the application.

`const char*` **TdfGroup::objectName** () `const` — Method

Returns the name of the HDF-Vgroup that is represented by this `TdfGroup`.

`const char*` **TdfGroup::className** () `const` — Method

Returns the class of the HDF-Vgroup that is represented by this `TdfGroup`.

`Tdf::Id` **TdfGroup::Id** () `const` — Method

Returns the HDF-Vgroup id that is represented by this `TdfGroup`. If this `TdfGroup` has not been loaded from or saved to a TDF/HDF file the id returned is -1.

`Tdf:Id` is `typedef`'ed to `long int` and must agree with HDF's `int32` definition.

Every `TdfGroup` may have attributes which are (string, value) pairs (see Section 2.2.7 [Attributes], page 12). Following are the related interface member functions.

`const TdfAttributes*` **TdfGroup::attributes** () `const` — Method

Returns a `const` pointer to this class's (vgroup's) attributes. It may be null.

`TdfAttributes*` **TdfGroup::attributes** () — Method

Returns a pointer to this class's (vgroup's) attributes. It may be null.

`TdfAttributes*` **TdfGroup::attributesOpen** () — Method

Return a pointer to this class's (vgroup) attributes set. Initalize attributes set for this class (vgroup) if it is not yet initialized.

`TdfAttributes*` **TdfGroup::attributesClear** () — Method

Clear this class's (vgroup's) attributes.

2.8 STL

The **TDF** library uses **STL** (*Standard Template Library*). Most used STL template class is `vector`.

Some recommeded STL references:

- David R. Musser's STL page '`http://www.cs.rpi.edu/~musser/stl.html`'.
- **Silicon Graphics**'s STL implemenation & Documentation '`http://www.sgi.com/Technology/STL/`'.

  Note that it extends the standard with hash variants of the associative containers '`http://www.sgi.com/Technology/STL/HashedAssociativeContainer.html`'.
- The Dinkum C/C++ Library Reference '`http://www.dinkumware.com/htm_cpl/index.html#STL`'.
- ISO/ANSI C++ Draft '`http://www.cygnus.com/misc/wp/dec96pub/`'.
- STL implemenatation from **ObjectSpace** '`http://www.objectspace.com/Products/cpp_libraries/stltk/`'.
- STL Quick-Reference Card '`http://www.blueneptune.com:/~yotam/stl.html`'.

2.9 Things you may ignore

Many of the classes in the **TDF** API support **HDF** I/O operations. These operation are supported by base classes or static member functions and data.

The application should not need to use these. The reason for them not always being **C++-`private`** is that it would have required too many `friend`ship relationships.

The following are repeated base-classes, and class members that the Tdf-API user is encouraged to ignore:

- `TdfClassDefVGroup`
- `TdfClassDefVData`
- `sGroupDef`
- `groupDef`
- `subNode`
- `subNodeArrayItem`
- `subNodeArrayLength`
- `subNodeArrayReset`

Index

O

P

Q

R

S

T

U

V

W

X

Table of Contents