ISE Integrated Systems Engineering
Release 9.0

Part 6
Utilities

1 – Introduction 6.1

1.1 About this manual 6.1
1.2 Scope of the manual 6.1
1.3 Terms and conventions 6.1

2 – DF–ISE 6.3

2.1 About DF–ISE 6.3
2.2 DF–ISE files 6.3
2.3 DF–ISE syntax 6.4
2.4 DF–ISE coordinate system 6.5
2.4.1 Reference coordinate system 6.5
2.4.2 Coordinate transformation 6.6
2.5 Location codes 6.6
2.5.1 One-dimensional geometries 6.7
2.5.2 Two-dimensional geometries 6.7
2.5.3 Three-dimensional geometries 6.7
2.6 Properties 6.7
2.7 Layout 6.8
2.8 Cell 6.9
2.9 Recursive tensor 6.11
2.10 Variable tensor 6.13
2.11 Boundary and grid 6.16
2.12 Dataset 6.20
2.13 XYPlot 6.22
2.14 Examples 6.22

3 – The file datexcodes.txt 6.25

3.1 Introduction 6.25
3.2 Material example 6.25
3.3 Variable example 6.25
3.4 Environment variable DATEX 6.26

4 – DFISETOOLS 6.27

4.1 DFISETOOLS file 6.28
4.1.1 Example: Obtaining a summary of a layout file 6.28
4.2 Cutting a 3D device with a plane 6.29
4.2.1 Example 1: Cut plane through a 3D boundary 6.29
4.2.2 Example 2: Cut plane through a 3D grid 6.30
4.3 Sampling a device with a line segment 6.31
4.3.1 Example 1: Cut line through a 3D grid 6.31
4.3.2 Example 2: Cut line through a 2D grid 6.32
4.4 Mirroring a device through a plane 6.33
4.4.1 Example 1: Mirror a layout 6.34
4.4.2 Example 2: Mirror a 1D grid 6.34
4.4.3 Example 3: Mirror a 2D boundary 6.35
4.4.4 Example 4: Mirror a 2D grid 6.36
4.4.5 Example 5: Mirror a 3D boundary 6.36
4.4.6 Example 6: Mirror a 3D grid 6.37
4.5 Extension of 2D geometry 6.37

4.6 Setting transformation on a geometry or tensor 6.38
4.7 Applying transformation on a geometry or tensor 6.39
4.8 Tessellating a geometry 6.39
4.8.1 Example 1: Tessellating a 2D boundary 6.39
4.8.2 Example 2: Tessellating a 3D boundary 6.40
4.9 Using DFISETOOLS with a Tcl script 6.40
4.10 Compressing and uncompressing 6.42

5 – Interfise 6.43

5.1 Conversion between DF–ISE and SUPREM-IV format 6.43
5.1.1 SUPREM-IV format A to DF–ISE 2D grid file converter 6.44
5.1.2 DF–ISE to SUPREM-IV format A 2D grid file converter 6.44
5.1.3 SUPREM-IV format B to DF–ISE 2D grid file converter 6.44
5.1.4 DF–ISE to SUPREM-IV format B 2D grid file converter 6.45
5.2 Old format to DF–ISE 6.45
5.3 Boundary extraction 6.45
5.4 DF–ISE boundary to OMEGA bound format 6.46
5.5 OMEGA bound format to DF–ISE boundary 6.46
5.6 Using Interfise with a Tcl script 6.46

6 – CURPOT 6.47

6.1 Overview 6.47
6.2 Using CURPOT 6.47
6.3 Example 6.48

7 – Converter for SUPREM-IV: alien2lig 6.49

7.1 Introduction 6.49
7.2 Main features 6.49
7.2.1 Input format 6.49
7.2.2 Supported commands 6.49
7.3 Specialities 6.50
7.3.1 Layout and masks 6.50
7.3.2 Etching 6.51
7.4 Running alien2lig 6.51
7.5 Output 6.51
7.6 Known problems 6.52

8 – DIOS to FLOOPS converter 6.53

8.1 Introduction 6.53
8.2 Usage 6.53
8.2.1 From LIGAMENT 6.53
8.2.2 From LIGEDIT 6.53
8.3 Supported commands 6.54
8.3.1 comment 6.54
8.3.2 deposit 6.54
8.3.3 diffusion 6.54
8.3.4 etching 6.55
8.3.5 grid 6.55
8.3.6 implantation 6.55

8.3.7 mask 6.56
8.3.8 substrate 6.56
8.3.9 title 6.56
8.4 Limitations 6.56
8.5 Customization 6.57

9 – MEASURE 6.59

9.1 Introduction 6.59
9.2 Command-line options 6.60
9.3 Matching mechanism and templates 6.62
9.4 Using MEASURE inside GENESISe 6.66
9.5 Using MEASURE inside DIOS 6.66
9.6 Limitations 6.66

Part 6 – Utilities

1 – Introduction

1.1 About this manual

This manual describes the DF–ISE format and a number of utility programs.

1.2 Scope of the manual

The main chapters are:

- Chapter 2 provides some background information about DF–ISE, and describes the organization of DF–ISE files, the cell structure, and various tensor grids.
- Chapter 3 describes the file `datexcodes.txt`.
- Chapter 4 presents the DFISETOOLS file and provides examples of how to use the DF–ISE functionality.
- Chapter 5 describes conversion between DF–ISE file formats and foreign or old file formats.
- Chapter 6 describes the postprocessing tool and provides instructions for its use.
- Chapter 7 describes the SUPREM-IV utility functionality.
- Chapter 8 describes the DIOS to FLOOPS converter.
- Chapter 9 describes the MEASURE tool, which extracts values from protocols.

1.3 Terms and conventions

Table 6.1 Standard terms

Term	Explanation
Click	Using the mouse, point to an item, press and release the left mouse button.
Double-click	Using the mouse, point to an item and in rapid succession, click the left mouse button twice.
Select	Using the mouse, point to an icon, a button, or other item and click the left mouse button.

Table 6.2 Typographic conventions

Convention	Definition or type of information
Blue	Identifies a cross-reference.
Bold	Identifies a selectable icon, button, menu, or tab, for example, the **OK** button. It also indicates the name of a field, window, dialog box, or panel.
`code`	Identifies text that is displayed on the screen, or text that the user must enter.
Italics	Used to emphasize text or identifies a component of an equation or a formula.
NOTE	Alerts the user to important information.

Part 6 – Utilities
2 – DF–ISE

2.1 About DF–ISE

A unique file format is used to store all information about the current status of a device. If new data items have to be exchanged between some of the simulators, using keywords and grouping data into data records allows the extension of the file format without changing the tools.

DF–ISE tools use a common file that contains information about materials and functions that is shared by different simulation tools. Generic properties (for example, materials insulator, semiconductor, and metal) can be used to enable some treatment of new materials or functions.

The detailed structures of the DF–ISE data blocks recognized by all IIS/ISE simulation tools are described in the following chapters. DF–ISE data blocks contain a representation of the device geometry (that is, a representation of the regions of the device or a spatial tessellation of the regions) and/or data values, defined on that tessellation.

A DF–ISE file has, at least, three main parts:

- A header that is used to identify DF–ISE files
- An `Info` block that contains general information about the data stored in this file
- A `Data` block giving a detailed description of geometry, datasets, or properties

User-defined keywords and data items can be added freely after the `Data` section. The IIS/ISE tools ignore these data records (other tools may not).

A library providing basic reading and writing has been developed. The description of the library functions with the exact calling sequences is described in the include files of this library.

2.2 DF–ISE files

DF–ISE files are organized in blocks. An overview of the top-level blocks of a DF–ISE file is presented:

`DF-ISE` This block identifies DF–ISE files. No comments or other blocks precede this identifier. The `DF-ISE` block has no arguments and no body; therefore, it does not enclose any other block.

`text | binary` This keyword follows `DF-ISE` and indicates whether the rest of the file is readable or in binary format. Only integer and floating point values can be stored in binary format.

`Info` This block provides additional information about the stored data.

`Data` This block is linked to a preceding `Info` block. Its content depends on the type entry of the information block. The following data block types are currently supported by the format:

`layout` Layout geometry used for patterning operations. `layout` file names should have the extension `.lyt`.

`cell` Cell structures, mainly used for 3D solid modeling. `cell` file names should have the extension `.cel`.

`recursive-tensor` Recursive tensor product grids used for compact storage of some devices. `recursive-tensor` – File names should have the extension `.ten`.

`boundary` The geometrical data in this block is a boundary description of a semiconductor device. `boundary` file names should have the extension `.bnd`.

`grid` A finite-element discretization of the device is stored in this block. The boundary representation of regions that do not need a simulation grid are not stored here, but in the `boundary data` block of the corresponding `boundary` file. `grid` file names should have the extension `.grd`.

`dataset` This section is used to store scalar or vector values on topological elements of `recursive-tensor`, `boundary`, and `grid` files. `dataset` file names should have the extension `.dat`.

`xyplot` Data that does not correspond to a simulation grid or boundary representation is stored in this section. Data can be displayed with x-y or x-y-z plot tools. `Plot` file names should have the extension`.plt`.

`property` The global property database is stored in these data blocks. IIS/ISE tools use a common database that contains information about materials and functions that need to be shared by different simulation tools. A `property` file should have the extension `.pro`.

2.3 DF–ISE syntax

A formal definition of the DF–ISE syntax follows:

`file` := DF–ISE text | binary <new_line> blocks

`blocks` := <empty> | block | blocks

`block` := keyword [‘(’ integer ‘)’] [body]

`body` := ‘{’ [blocks | values] ‘}’ | ‘=’ data

`data` := array | values

`array` := ‘[’ values ‘]’

`values` := <empty> | value | values

`value` := integer | float | keyword | string

A DF–ISE file comprises a sequence of blocks, which can contain further blocks. A block always has the following structure: `block_name (argument){body}`.

Both the argument and the body of a block are optional. The body of a block is either a sequence of blocks or a sequence of data values. Finally, a data value is an integer, a floating point value, a keyword, a string delimited by quotation marks, or an array of data values. Comments start with `#` and end at the end of line.

NOTE Comments cannot precede the `DF-ISE` block.

Blocks without body and data values are separated by space characters or new lines. The order of the blocks in a file is given by the DF–ISE format.

When storing a binary `DF-ISE` file, an application can mix readable integer and floating values with compressed values. An application that reads a binary `DF-ISE` file must be able to determine whether the next value in the file is readable or in binary format. The DF–ISE library contains all the functions needed for reading and writing values.

NOTE The precision of readable floating point values depends on the application that writes the file.

2.4 DF–ISE coordinate system

The DF–ISE coordinate system is always a right-handed Cartesian system. All coordinates are in micrometers. However, the ‘up’ direction – the vector starting from the substrate floor and pointing towards the device surface – depends on the application. The old generation of IIS/ISE tools uses the negative y-axis as the up direction. Whenever possible, the new orientation proposed below should be used.

2.4.1 Reference coordinate system

The DF–ISE reference coordinate system is right-handed as shown in Figure 6.1. The `z = 0` plane should be placed on the initial substrate surface, for example, before starting process operations.

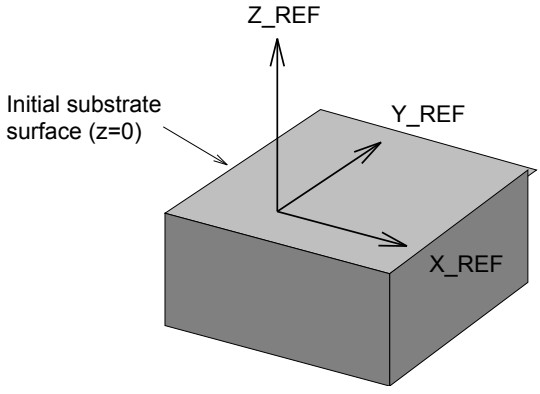


Figure 6.1 Reference coordinate system with initial substrate example

The layout data for patterning operations on the rear side of the wafer must be defined with the same coordinate system used for the front side. This means that the user who edits mask layers is always looking at the front side of the wafer (see Figure 6.2).

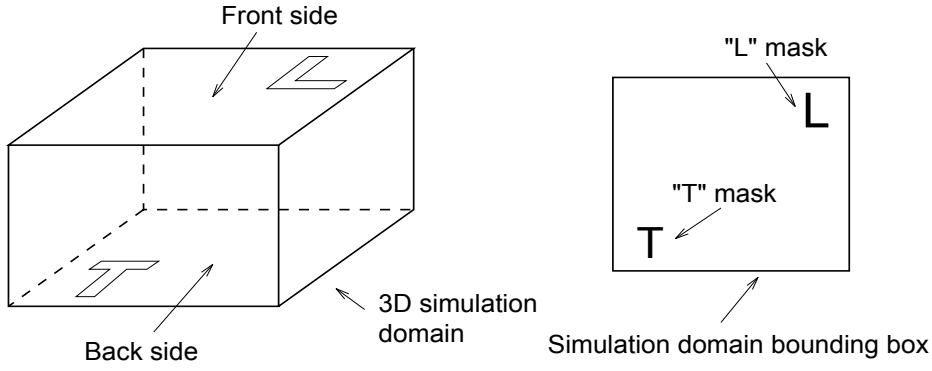


Figure 6.2 Editing layout data for front and rear sides

2.4.2 Coordinate transformation

A coordinate transformation can be specified in `recursive-tensor`, `boundary`, and `grid` files. A transformation is used to map 1D, 2D, or 3D devices to their real place in the 3D wafer. An application that reads devices only has to apply the coordinate transformation to the vertices of the device, except if the application is using the original 3D wafer coordinate system internally.

Nothing is assumed regarding the up direction of the global 3D coordinate system. However, the orientation proposed in the previous section should be used whenever possible.

The transformation from user coordinates to the global reference coordinate system is performed as follows:

$$\begin{pmatrix} x_{ref} \\ y_{ref} \\ z_{ref} \end{pmatrix} = \begin{bmatrix} xx & xy & xz \\ yx & yy & yz \\ zx & zy & zz \end{bmatrix} \times \begin{pmatrix} x_{user} \\ y_{user} \\ z_{user} \end{pmatrix} + \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix} \qquad \text{[Eq. 6.1]}$$

The translation vector and transformation matrix are given in the `CoordSystem` block in this way:

```
translate = [ <dx> <dy> <dz> ]
transform = [ <xx> <xy> <xz> <yx> <yy> <yz> <zx> <zy> <zz> ]
```

2.5 Location codes

Location codes identify the position of vertices, edges, or faces. For example, a location code tells whether a face is external (or visible), or if it is on an interface between two regions, or if the face is fully inside a 3D region. Location codes must be stored in `boundary` and `grid` files.

2.5.1 One-dimensional geometries

For one-dimensional geometries, a location code is defined for vertices. If a vertex is shared by more than one segment belonging to different regions, it obtains the location code `f` (internal interface). If the vertex is defined only in one segment of one region, it has the location code `e` (external interface), and if more than one segment of the same region share the vertex, it has the code `i` (internal). For vertices shared only by points or segments that do not belong to any region, the location code is `u` (undefined).

2.5.2 Two-dimensional geometries

For two-dimensional geometries, the location code is defined for edges. If an edge is contained only in one 2D element (triangle, rectangle, polygon) which belongs to a region, the edge is called ‘external interface.’ If it is shared by more than one 2D elements that belong to different regions, it is called ‘internal interface.’ If more than one 2D elements that belong to a region share the edge, this edge is called ‘internal,’ and if there are only elements not belonging to any region and line segments sharing the edge, the location code of the edge is ‘undefined.’

2.5.3 Three-dimensional geometries

For three-dimensional geometries, the location code is defined for faces. If a face belongs to exactly one 3D element included in a region, the face is called ‘external interface.’ If the face is shared by two 3D elements belonging to different regions, it is called ‘internal interface.’ If the face is shared by two 3D elements all belonging to the same region, the face is called ‘internal.’ Finally, if the face is shared by 3D elements that do not belong to any region or by 2D elements, the location code is ‘undefined.’

2.6 Properties

Properties identify and characterize datasets. A global file `dfise.pro` that contains a list of properties is provided in the ISE TCAD distribution. A tool that needs the information contained in this file must read it during startup. `layout`, `cell`, `recursive-tensor`, `boundary`, `grid`, `dataset`, and `plot` files reference the properties defined in the property database. Multiple properties for one region or one dataset are allowed. In this case, the material or function name is a list of property names concatenated with `&`. For example, an interface region can have the material `Silicon&Oxide`. This combined material name is identical to `Oxide&Silicon`, but should not be confused with `SiliconOxide`. The attributes of combined properties are not defined. An application can select any of the entries for setting colors, interpolation, and so on. Binary compression is not allowed for this file.

Two main property types are defined: `Material`, which is used to define region properties, and `Function`, which defines the property of any dataset. `cell` and `layout` files do not distinguish between `Material` and `Function` properties. Thus, a property name must be unique within the complete property file.

Currently, the following attributes are understood by most of the IIS/ISE tools:

`symbol`	A short quoted string that identifies this property. The symbol is for display purposes only, thus it does not have to be unique.
`color`	Any valid X11 color. It is used for displaying curves or drawing regions.
`style`	Only valid for `Function`. Line style when displaying curves (`solid`, `dashed`, `dotted`, `ldashed`, `ldotted`).

formula	Only valid for Function. It is used to compute datasets that have this property from other datasets. A mathematical expression using property names must be provided here. Variable or property names must be enclosed in angle brackets < >.
unit	Only valid for Function. This is a quoted string that defines the unit of a dataset.
interpol	Only valid for Function. Interpolation function (linear, log, arsinh).

```
DF-ISE text
Info {
  version = 1.0
  type = property
}
Data {
  Material (SiliconOxide) {
        symbol = "SiO2"
        color  = brown
  }
  ...
  Function (eDensity) {
        symbol   = "n"
        unit     = "1/cm^3"
        interpol = arsinh
        color    = red
        style    = dashed
  }
  Function (TotalDopingConcentration) {
        symbol   = "N-tot"
        unit     = "1/cm^3"
        interpol = arsinh
        color    = red
        style    = dashed
        formula  = "<BoronConcentration> + ..."
  }
  ...
}
```

2.7 Layout

The layout file type stores the starting geometry (layout files) and the photolithography data (mask files) required by all patterning operations of the process flow.

NOTE Simulation domains (a point for 1D, a line for 2D, and a polygon for 3D) can also be stored in DF-ISE layout files. They have Sim1D, Sim2D, and Sim3D as predefined property names. The property list is also used to find the color of the layers in the material blocks of the property database.

A polygon entry of the Polygons block can have one or two vertices when it is used to define a 1D or 2D simulation domain. The order of appearance of Region blocks must correspond to the regions entry list of the Info block. The file structure is:

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = layout
  nb_vertices = <int>
  nb_polygons = <int>
```

```
  nb_regions  = <int>
  regions     = [ "<region1>" "<region2>" ... ]
  materials   = [ <material1> <material2> ... ]
}

Data {

  Vertices (<nb_vertices>) {
        <x0> <y0>
        <x1> <y1>
         ...
  }

  Polygons (<nb_polygons>) {
        <nb_vertices> <vertex> <vertex> ...
        <nb_vertices> <vertex> <vertex> ...
        ...
  }

  Region ("<material1>") {
        material = <material1>
        Elements (<nb_elts>) { <elt0> <elt1> ... }
  }

  ...
}
```

2.8 Cell

The 'Cell' data structure uses a regular subdivision of the simulation domain into blocks of equal size (cubes). Each block can be refined into groups of cells by splitting it along the x-, y-, or z-axis (see Figure 6.3). The construction of general, not axis-aligned structures is achieved by defining a material density value for each cell. Storing material densities is optional – by default, the maximal density value 1.0 is assumed. An unrefined block always has a material density of 1.0.

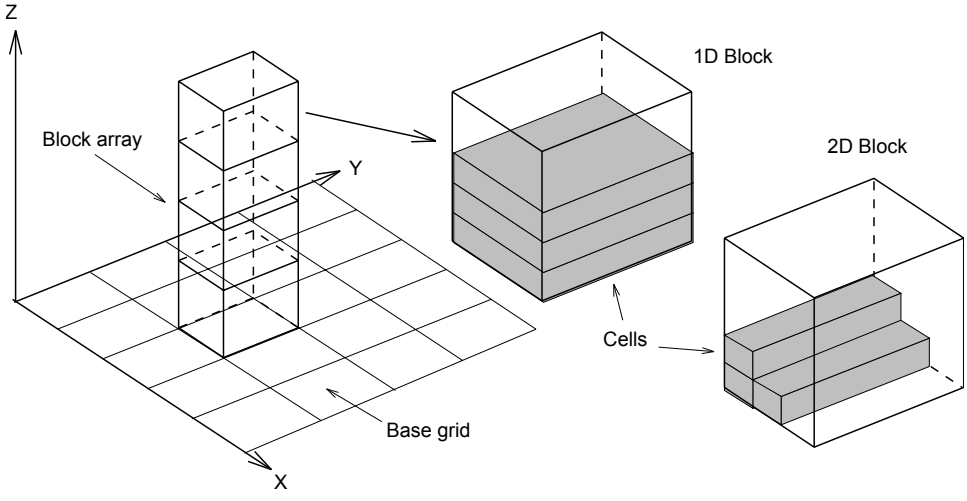


Figure 6.3 Cell data structure

Geometry and datasets are stored in the same file and use the same format. Region identifiers and material densities are considered as normal datasets. However, they have the reserved property names *regions* and *densities*.

NOTE The refinement of the different datasets does not have to be identical, except for the material density refinement, which must always match the region data. A `property` entry can either be an entry of the material list or an entry of the function list in the property database.

The `Info` block defines the simulation domain size and refinement information. `block_size` is the size in micrometers of the initial cubes building the simulation domain. The size of the simulation domain is given by multiplying `block_size` with `nb_xblocks`, `nb_yblocks`, or `nb_zblocks`. When a block (cube) is refined, `nb_xsplits`, `nb_ysplits`, and `nb_zsplits` define the number of cells for the given axis.

The numbering of blocks and cells always starts at the lowest coordinate. Indices are incremented by traversing the structure first along the x-axis, then along the y-axis, and finally along the z-axis. In practice, traversing the data structure is performed in this way:

```
for (bz = 0; bz < nb_zblocks; bz++) {
for (by = 0; by < nb_yblocks; by++) {
for (bx = 0; bx < nb_xblocks; bx++) {
   if (block[bx][by][bz] is refined)
   {
      for (cz = 0; cz < nb_zsplits; cz++) {
      for (cy = 0; cy < nb_ysplits; cy++) {
      for (cx = 0; cx < nb_xsplits; cx++) {
      ...
      }
      }
      }
   }
}
}
}
```

The order of appearance of `Dataset` blocks must correspond to the `datasets` entry list of the `Info` block. The structure of the file is:

```
DF-ISE text | binary
Info {
   version    = 1.0
   type       = cell
   dimension  = 1 | 2 | 3
   block_size = <double>
   nb_xblocks = <int>
   nb_yblocks = <int>
   nb_zblocks = <int>
   nb_xsplits = <int>
   nb_ysplits = <int>
   nb_zsplits = <int>
   nb_datasets = <int>
   datasets   = [ "<dataset1>" "<dataset2>" ... ]
   properties = [ <property1> <property2> ... ]
}
Data {

Dataset ("<dataset1>") {
   property = <property1>

   Blocks (<nb_blocks>) {
      <ref_flag> <value> ...
      <ref_flag> <value> ...
      ...
```

```
    }
}

Dataset ("<dataset2>") {
    ...
}
...

<ref_flag> is:
    0: no refinement, only one data values follows
    1: refinement along x, nb_xsplits data values follow
    2: refinement along y, nb_ysplits data values follow
    4: refinement along z, nb_zsplits data values follow
    3: refinement along x and y, nb_xsplits * nb_ysplits values follow
    ...
    7: refinement along x, y, and z, nb_xsplits * nb_ysplits
       * nb_zsplits data values follow
}
```

2.9 Recursive tensor

A recursive grid is a generalization of a tensor grid. In addition to a base grid that is a tensor grid, it allows recursive subdivision of elements. Vertices, edges, faces, and elements are defined implicitly through the grid and its traversal rules. The tensor base grid can be:

Uniform	Rectilinear grid with constant spacing in x-, y-, and z-direction (the spacing for each direction is given separately).
Rectilinear	Rectilinear grid with nonconstant spacing between coordinate planes.
Warped	Grid with tensor mesh topology and explicit coordinates for each vertex.

The numbering of vertices, edges, faces, and elements starts with index 0 at the lowest coordinates and goes first through the x-axis, y, and then z. The entries `nb_x`, `nb_y`, and `nb_z` in the `Info` block represent the number of vertices of the base grid in each of the dimensions. Vertices, edges, faces, and elements are implicitly numbered as follows:

- There is a total of $(nb\_x)(nb\_y)(nb\_z)$ nodes. Node (i,j,k) has number $i+j(nb\_y)+k(nb\_y)(nb\_z)$, where $0<i<nb\_x$, $0<j<nb\_y$, $0<k<nb\_z$.
- There is a total of $(nb\_x-1)(nb\_y)(nb\_z)+(nb\_x)(nb\_y-1)(nb\_z)+(nb\_x)(nb\_y)(nb\_z-1)$ edges. The edges are numbered or traversed in the following way: first we go through the vertices in the order given below, and for each vertex we take first the edge pointing along the positive x-axis, then the one pointing along the positive y-axis, and finally the one pointing along the positive z-axis. At the boundary of the simulation domain, the edge index is, of course, not incremented when no edge exists in the given direction.
- There is a total of $(nb\_x)(nb\_y-1)(nb\_z-1)+(nb\_x-1)(nb\_y)(nb\_z-1)+(nb\_x-1)(nb\_y-1)(nb\_z)$ faces. The face numbering is analogous to the edge numbering. At each vertex, we take the three faces orthogonal to the x, y, and z axes.
- There is, in principle, a total of $(nb\_x-1)(nb\_y-1)(nb\_z-1)$ elements. However, additional dummy elements are added at the upper bounds of the coordinate ranges, so that total number of elements is $(nb\_x)(nb\_y)(nb\_z)$. Element (i,j,k) has, therefore, the number $i+j(nb\_y)+k(nb\_y)(nb\_z)$, where $0<i<nb\_x$, $0<j<nb\_y$, $0<k<nb\_z$.

The contents of the `BaseGrid` block depend on the tensor grid subtype as follows:

- For `uniform` grids, it contains the spacings Δx, Δy, and Δz in the directions of the coordinate planes. Vertex number 0 is implicitly at (0,0,0); the `CordSystem` block can be used for further moving.
- For `rectilinear` grids, it contains the spacings Δx_1, ..., Δx_{nb_x-1}, Δy_1, ..., Δy_{nb_y-1}, Δz_1, ..., Δz_{nb_z-1}, that is, there are *nb_x*–1 entries for the x-direction, *nb_y*–1 entries for the y-direction, and *nb_z*–1 entries for the z-direction. Vertex number 0 is implicitly at (0,0,0); the `CordSystem` block can be used for further moving.
- For `warped` grids, it contains the coordinates of the grid vertices x_0, y_0, z_0, x_1, y_1, z_1, ..., x_{nb_v}, y_{nb_v}, and z_{nb_v} where the numbering is the natural cell numbering, that is, varying fastest along the x-axis.

The `CellTree` block is used only if the mesh is recursively refined; if it is missing, the data structure is an ordinary tensor mesh. Otherwise, for each macroelement including the dummy elements, the recursive grid structure is indicated:

```
<min_lev> <max_lev> <nb_el> <sub_el0> <sub_el1> ... <sub_el_nb_el>
```

`min_lev` and `max_lev` are the minimum and maximum refinement levels in the macroelement. A value of 0 means that the macroelement is unrefined. The number of elements `nb_el` gives the number of elements in the macroelement that result in the final mesh. A value of 0 indicates that the macroelement is a dummy element or deleted from the mesh; a value of 1 means that it is unrefined.

If the minimum and maximum refinement levels of a macroelement are equal, the mesh can be constructed implicitly, and the mesh description within the macroelement is complete. If the refinement levels are not equal, they are followed by a list of the subelements `sub_el` within the macroelement. A simple numbering scheme is used to indicate the subelements. The numbering proceeds looping through each of the parent elements at a level. The following is an example of a 2D scheme:

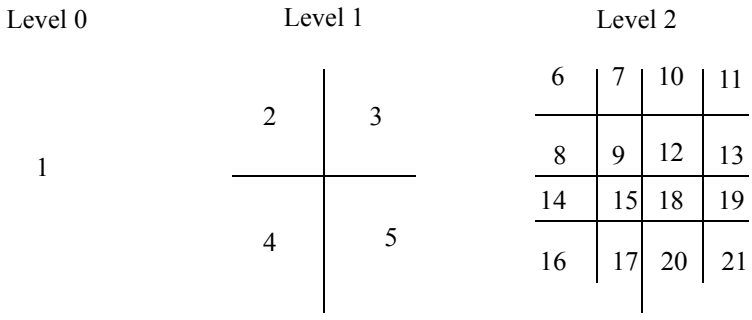


Figure 6.4 Numbering scheme for recursively subdivided cells

The coordinates of the joints (vertices) of the refined grid are currently determined uniquely by the base grid and the cell tree.

`Regions` are used to denote materials and boundary conditions. `Regions` can reference vertices, edges, faces, and elements of the base grid (not of the refined grid) through their implicitly defined number (see above). The order of appearance of `Region` blocks must correspond to the `regions` entry list of the `Info` block. The material entry can be left empty.

For data on the recursive grids, the DF–ISE data file format is used.

An example of the structure of a DF–ISE recursive tensor grid file is:

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = recursive-tensor
  subtype     = uniform | rectilinear | warped
  dimension   = 1 | 2 | 3
  extents     = <int> | <int> <int> | <int int>
  nb_regions  = <int>
  regions     = [ "<region1>" "<region2>" ... ]
  materials   = [ <material1> <material2> ... ]
}

Data {

  CoordSystem {
        translate = [ <dx> <dy> <dz> ]
        transform = [ <xx> <xy> <xz> <yx> <yy> <yz> <zx> <zy> <zz> ]
  }

  BaseGrid (<nb_entries>) {
        <e0> <e1> <e2>
  }

  CellTree (<nb_macro_elements>) {
       <min_lev> <max_lev> <nb_el> <sub_el0> <sub_el1> ... <sub_el_nb_el>
       ...
       ...
  }

Region ("<region1>") {
       material = <material1>
        Edges    (<nb_edge>) { <edge0> <edge1> ... }
        Faces    (<nb_face>) { <face0> <face1> ... }
  }

Region ("<region2") {
    material = <material2>
        Vertices (<nb_vert>) { <vert0> <vert1> ... }
  }

Region ("<region3>") {
    material = <material3>
        Faces    (<nb_fac>)  { <fac0>  <fac1>  ... }
  }
Region ("<region4 >") {
       material = <material4>
        Elements (<nb_elts>) { <elt0>  <elt1>  ... }
  }
}
```

2.10 Variable tensor

A variable tensor grid is a generalization of a recursive tensor grid. The recursive subdivisions allowed are not restricted to powers of two, and the addition of a property tree allows for property changes within a cell of the tensor grid In addition to a base grid that is a tensor grid, they allow recursive subdivision of elements.

Vertices, edges, faces, and elements are defined implicitly through the grid and its traversal rules. The tensor base grid can be:

Uniform	Rectilinear grid with constant spacing in x-, y-, and z-direction (the spacing for each direction is given separately).
Rectilinear	Rectilinear grid with nonconstant spacing between coordinate planes.
Warped	Grid with tensor mesh topology and explicit coordinates for each vertex.

The numbering of vertices, edges, faces and elements starts with index 0 at the lowest coordinates and goes first through the x-axis, y-axis, and then z-axis. The entries `nb_x`, `nb_y` and `nb_z` in the `Info` block represent the number of vertices of the base grid in each of the dimensions. Vertices, edges, faces, and elements are implicitly numbered as follows:

- There is a total of $(nb\_x)(nb\_y)(nb\_z)$ nodes. Node (i,j,k) has number $i+j(nb\_y)+k(nb\_y)(nb\_z)$, where $0<i<nb\_x$, $0<j<nb\_y$, $0<k<nb\_z$.
- There is a total of $(nb\_x-1)(nb\_y)(nb\_z)+(nb\_x)(nb\_y-1)(nb\_z)+(nb\_x)(nb\_y)(nb\_z-1)$ edges. The edges are numbered or traversed in the following way: first we go through the vertices in the order given below, and for each vertex we take first the edge pointing along the positive x-axis, then the one pointing along the positive y-axis, and finally the one pointing along the positive z-axis. At the boundary of the simulation domain, the edge index is, of course, not incremented when no edge exists in the given direction.
- There is a total of $(nb\_x)(nb\_y-1)(nb\_z-1)+(nb\_x-1)(nb\_y)(nb\_z-1)+(nb\_x-1)(nb\_y-1)(nb\_z)$ faces. The face numbering is analogous to the edge numbering. At each vertex, we take the three faces orthogonal to the x, y, and z axes.
- There is, in principle, a total of $(nb\_x-1)(nb\_y-1)(nb\_z-1)$ elements. However, additional dummy elements are added at the upper bounds of the coordinate ranges, so that total number of elements is $(nb\_x)(nb\_y)(nb\_z)$. Element (i,j,k) has, therefore, the number $i+j(nb\_y)+k(nb\_y)(nb\_z)$, where $0<i<nb\_x$, $0<j<nb\_y$, $0<k<nb\_z$.

The contents of the `BaseGrid` block depend on the tensor grid subtype as follows:

- For `uniform` grids, it contains the spacings, Δ_x, Δ_y, and Δ_z in the directions of the coordinate planes. Vertex number 0 is implicitly at (0,0,0); the `CordSystem` block can be used for further moving.
- For `rectilinear` grids, it contains the spacings $\Delta x_1, ..., \Delta x_{nb\_x-1}, \Delta y_1, ..., \Delta y_{nb\_y-1}, \Delta z_1, ..., \Delta z_{nb\_z-1}$, that is, there are nb_x-1 entries for the x-direction, nb_y-1 entries for the y-direction, and nb_z-1 entries for the z-direction. Vertex number 0 is implicitly at (0,0,0); the `CordSystem` block can be used for further moving.
- For `warped` grids, it contains the coordinates of the grid vertices $x_0, y_0, z_0, x_1, y_1, z_1, ..., x_{nb\_v}, y_{nb\_v}$, and z_{nb_v} where the numbering is the natural cell numbering, that is, varying fastest along the x-axis.

The `CellTree` block is used only if the mesh is recursively refined; if it is missing, the data structure is an ordinary tensor mesh. Only those macro cells that are refined will have an entry in the `CellTree`. Such entries are defined as follows:

```
<index> <x-refinement> <y-refinement> <z-refinement> <nb_el> <sub_el0> <sub_el1> ... <sub_el_nb_el>
```

`index` refers to the cell index of the refined macro cell. `x-refinement`, `y-refinement`, and `z-refinement`, give the number of refinement levels in the respective direction. The number of elements `nb_el` gives the number of elements in the macroelement that result in the final mesh. This is followed by a list of the subelements `sub_el`

within the macroelement. A simple numbering scheme is used to indicate the subelements. The numbering proceeds looping through each of the parent elements at a level. This example shows a 2D scheme:

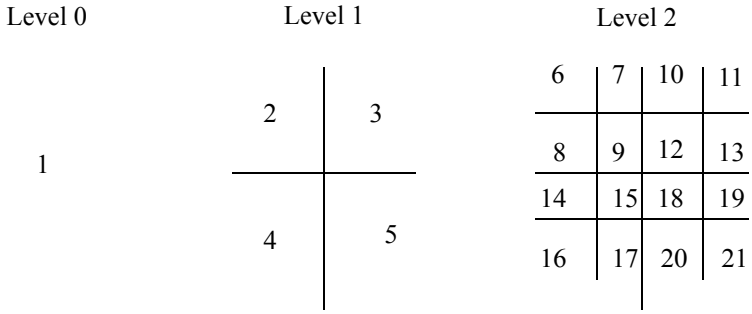


Figure 6.5 Numbering scheme for recursively subdivided cells

The coordinates of the joints (vertices) of the refined grid is currently determined uniquely by the base grid and the cell tree. The numbering scheme and calculation of coordinates are performed analogously to the case of the recursive tensor grid.

The *Properties* can be used to assign different properties, for example, materials, to the subcells of a refined macro cell:

```
<index> <nb_el> <prop_sub_el0> <prop_sub_el1> ... <prop_sub_el_nb_el>
```

`index` refers to the cell index of the refined macro cell; it should also appear in the `CellTree`. The refinements are taken from `CellTree`, and the properties of the subcells have to refer one-to-one to the subcells of the respective entry in `CellTree`. Although each entry of *Properties* should have an entry in `CellTree`, the reverse is not true. In general, single entries or the entire `Properties` block can be missing, in which case all subcells are assumed to have the same property.

Regions are used to denote materials and boundary conditions. Regions can reference vertices, edges, faces, and elements of the base grid (not of the refined grid) through their implicitly defined number (see above). The order of appearance of `Region` blocks must correspond to the `regions` entry list of the `Info` block. The `material` entry can be left empty.

For data on the variable tensor grids, the DF–ISE data file format is used.

An example of the structure of a DF–ISE variable tensor grid file is:

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = variable-tensor
  subtype     = uniform | rectilinear | warped
  dimension   = 1 | 2 | 3
  extents     = <int> | <int> <int> | <int int>
  nb_regions  = <int>
  regions     = [ "<region1>" "<region2>" ... ]
  materials   = [ <material1> <material2> ... ]
}

Data {

  CoordSystem {
        translate = [ <dx> <dy> <dz> ]
        transform = [ <xx> <xy> <xz> <yx> <yy> <yz> <zx> <zy> <zz> ]
```

```
  }

BaseGrid (<nb_entries>) {
        <e0> <e1> <e2>
  }

  CellTree (<nb_macro_elements>) {
    <index> <x-refinement> <y-refinement> <z-refinement> <nb_el> <sub_el0> <sub_el1> ...
    <sub_el_nb_el>
    ...
    ...
  }


Properties(<nb_macro_elements>) {
    <index> <nb_el> <prop_sub_el0> <prop_sub_el1> ... <prop_sub_el_nb_el>
    ...
    ...
  }

Region ("<region1>") {
    material = <material1>
        Edges    (<nb_edge>) { <edge0> <edge1> ... }
        Faces    (<nb_face>) { <face0> <face1> ... }
  }
Region ("<region2") {
    material = <material2>
        Vertices (<nb_vert>) { <vert0> <vert1> ... }
  }
Region ("<region3>") {
    material = <material3>
        Faces    (<nb_fac>)  { <fac0>  <fac1>  ... }
  }
Region ("<region4 >") {
    material = <material4>
        Elements (<nb_elts>) { <elt0>  <elt1>  ... }
  }

}
```

2.11 Boundary and grid

Boundary and grid files have a similar format. However, the interpretation of the contents is different. Each region must have one material name. The numbering of vertices, edges, and faces is given in Figure 6.6 on page 6.19 and Figure 6.7 on page 6.20. The second picture is only required when data must be stored on vertices or edges locally for each element.

An index to a face is signed. When negative, the real face index is computed by: *face_index* = –*index*–1. The negative sign means that the edge order of the face as well as the edge orientation must be inverted. Face orientation is important to obtain a consistent orientation of the object (counterclockwise orientation of the edges when looking from outside of the solid including this face). An edge index is signed in the same way as a face index. A negative edge sign means that the first vertex and not the second is connected to the next edge of a loop.

NOTE The DF–ISE library provides a function to fix the orientation of the faces of a 3D boundary.

Elements have a shape code that defines the element geometry and the data following this code. Elements of different shape can be given in any order. The following elements are predefined:

Point	`0 vertex`
Segment	`1 vertex0 vertex1`
Triangle	`2 edge0 edge1 edge2`
Rectangle	`3 edge0 edge1 edge2 edge3`
Polygon	`4 nb_edges edge0 edge1...`
Tetrahedron	`5 face0... face3`
Pyramid	`6 face0... face4`
Prism	`7 face0... face4`
Brick	`8 face0... face5`
Tetrabrick	`9 face0... face6`
Polyhedron	`10 nb_faces face0 face1...`

The edge and face lists of polygons and polyhedra do not have to be connected. They can be a set of loops and shells that define the outer boundary and internal holes of an element.

An edge orientation for each element is also predefined. The edge orientation is used when storing data on edges locally for each element. The vertex order for each edge of the elements are given below:

Segment:
e0: v0 v1

Triangle:
e0: v0 v1
e1: v1 v2
e2: v2 v0

Rectangle:
e0: v0 v1
e1: v1 v2
e2: v2 v3
e3: v3 v0

Polygon:
e0: v0 v1
e1: v1 v2
…

Prism:
e0: v0 v1
e1: v1 v2
e2: v2 v0
e3: v0 v3
e4: v1 v4
e5: v2 v5
e6: v3 v4
e7: v4 v5
e8: v5 v3

Tetrahedron:
e0: v0 v1
e1: v1 v2
e2: v2 v0
e3: v0 v3
e4: v1 v3
e5: v2 v3

Pyramid:
e0: v0 v1
e1: v1 v2
e2: v2 v3
e3: v3 v0
e4: v0 v4
e5: v1 v4
e6: v2 v4
e7: v3 v4

Brick:
e0: v0 v1
e1: v1 v2
e2: v2 v3
e3: v3 v0
e4: v0 v4
e5: v1 v5
e6: v2 v6
e7: v3 v7
e8: v4 v5
e9: v5 v6
e10: v6 v7
e11: v7 v4

Tetrabrick:

e0: v0 v1	e6: v2 v5
e1: v1 v2	e7: v2 v6
e2: v2 v3	e8: v3 v6
e3: v3 v0	e9: v4 v5
e4: v0 v4	e10: v5 v6
e5: v1 v5	e11: v6 v4

The order of appearance of `Region` blocks must correspond to the `regions` entry list of the `Info` block. Several regions with the same name are allowed.

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = boundary | grid
  dimension   = 1 | 2 | 3
  nb_vertices = <int>
  nb_edges    = <int>
  nb_faces    = <int>
  nb_elements = <int>
  nb_regions  = <int>
  regions     = [ "<region1>" "<region2>" ... ]
  materials   = [ <material1> <material2> ... ]
}
Data {

    CoordSystem {
         translate = [ <dx> <dy> <dz> ]
         transform = [ <xx> <xy> <xz> <yx> <yy> <yz> <zx> <zy> <zz> ]
  }
    for 1d, 2d, and 3d
    Vertices (<nb_vertices>) {
         <x0> <y0> <z0>
         <x1> <y1> <z1>
          ...
  }
    for 2d and 3d
    Edges (<nb_edges>) {
         <vertex0> <vertex1>
         <vertex0> <vertex1>
         ...
  }
    for 3d
    Faces (<nb_faces>) {
         <nb_edges> <edge0> <edge1> ... <edge_n>
         <nb_edges> <edge> <edge> ...
         ...
  }
    nb_vert. for 1d, nb_edges for 2d, nb_faces for 3d
    Locations (<nb_vertices> | <nb_edges> | <nb_faces>) {
         i (internal) | f (internal interface) |
         e (external interface) | u (undefined)
         ...
  }
Elements (<nb_elements>) {
         0 <vertex>
         1 <vertex0 <vertex1>
         2 <edge0 <edge1> <edge2>
         3 <edge0 <edge1> <edge2> <edge3>
         4 <nb_edges> <edge0> <edge1> ... <edge_n>
```

```
        5 <face0 ... <face3>
        6 <face0 ... <face4>
        7 <face0 ... <face4>
        8 <face0 ... <face5>
        9 <face0 ... <face6>
        10 <nb_faces> <face0> <face1> ... <face_n>
       ...
  }
Region ("<region1>") {
        material = <material1>
        Elements (<nb_elts>) { <elt0> <elt1> ... }
  }
Region ("<region2>") {
        material = <material2>
        Elements (<nb_elts>) { ... }
  }
}
```

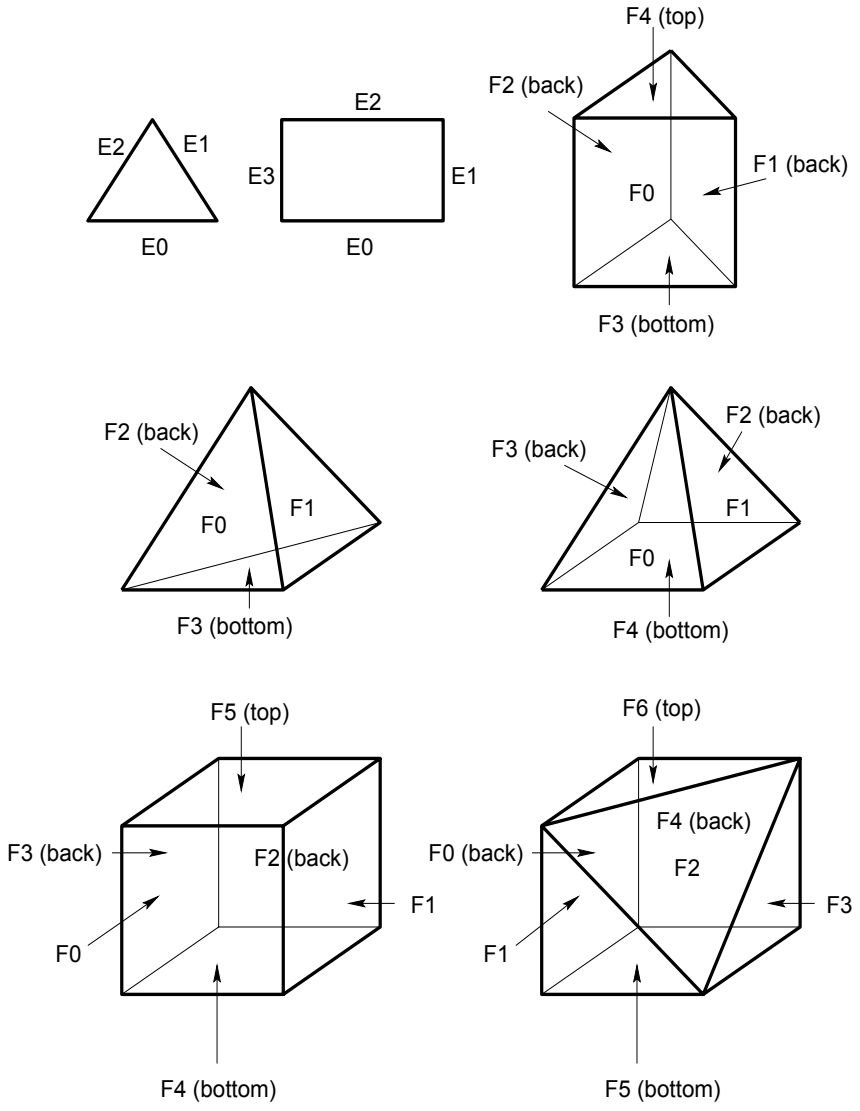


Figure 6.6 Edge and face numbering

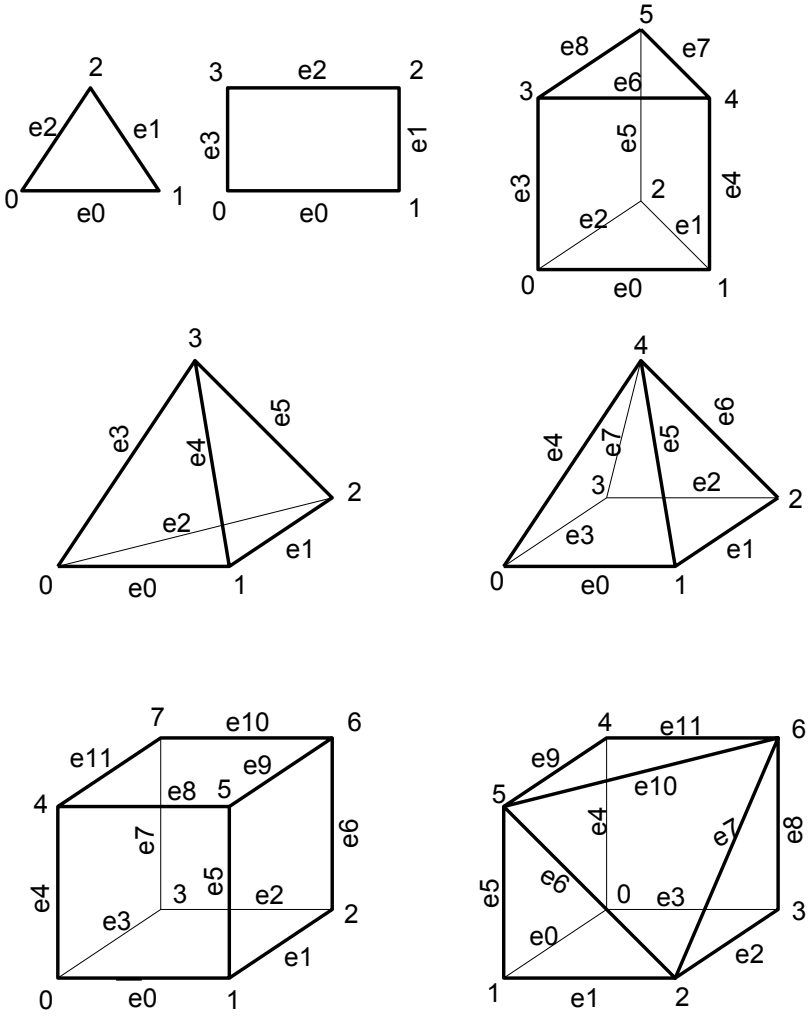


Figure 6.7 Edge and vertex numbering

2.12 Dataset

The `dataset` files add scalar or vector values to the nodes, edges, faces, elements, or regions of a `grid` file. The data *location* is currently limited to `vertex`, `edge`, `face`, `element`, or `region`. Two additional location codes are used to store Voronoï data: `vertex_element` and `edge_element`. These codes are only understood by DESSIS.

`dimension` gives the set number of data values per location (1 for scalars, > 1 for vectors). The `dimension` entry has a different meaning than the geometry dimension given in the `Info` block. The type of a dataset is currently either a `scalar` or `vector`. More types (`matrix`, `array`) will be added later.

`validity` gives the list of regions in which the dataset is defined. For consistency, data values should only be defined for the entries belonging to these validity regions.

NOTE The DF–ISE library provides some functions to access the values of a dataset for a given validity region. For the moment, this can only be used with datasets whose location is `vertex`.

Discontinuous datasets are a matter of interpretation of the data file format. Several datasets with the *same* quantity can be present in a single data file. Each instance of the dataset should be defined for a different set of regions (including the boundary points of a region). The entire dataset can, therefore, be discontinuous across boundaries. Because of the interpretational nature of this definition of discontinuous datasets, different tools may handle data files with such datasets differently. Refer to specific manuals for detailed descriptions of individual tools.

Data values are given in the order predefined by the `location` entry. When vectors are stored, all components are given per item (`vertex`, `edge`, ...) in the `Values` block.

The order of appearance of `Dataset` blocks must correspond to the `datasets` entry list of the `Info` block. Functions are references to the `function` blocks stored in the property file. Several datasets with the same name are allowed (with different validity regions, this may be used to define data discontinuities).

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = dataset
  dimension   = 1 | 2 | 3
  nb_vertices = <int>
  nb_edges    = <int>
  nb_faces    = <int>
  nb_elements = <int>
  nb_regions  = <int>
  datasets    = [ "<dataset1>" "<dataset2>" ... ]
  functions   = [ <function1> <function2> ... ]
}

Data {

  Dataset ("<dataset1>") {
        function  = <function1>
        type      = scalar | vector
        dimension = 1 (1 for scalar, > 1 for vectors or arrays)
        location  = vertex | edge | face | element | region
        validity  = [ "<region0>" "<region1>" ... ]
        Values (<nb_values>) { <value0> <value1> ... }
  }

  Dataset ("<dataset2>") {
        function  = <function2>
  ...
}
```

2.13 XYPlot

The `xyplot` files are used to store curves. The property list gives additional information for each data column of the file. The x-axis is not always in the first column, since the user can freely select the dataset mapped on the x-axis. For undefined values (discontinuities in the curve), the character ‘u’ is used as a placeholder:

```
DF-ISE text | binary

Info {
  version    = 1.0
  type       = xyplot
  datasets   = [ "<dataset1>" "<dataset2>" ... ]
  functions  = [ <function1> <function2> ... ]
}

Data {
  <v0_d1> <v0_d2> ... (one value for each data set)
  <v1_d1> <v1_d2> ... (one value for each data set)
  ...
}
```

2.14 Examples

This section provides examples of the DF–ISE file format.

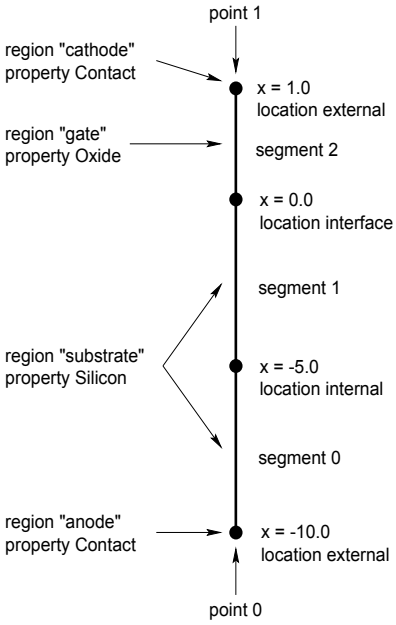


Figure 6.8 One-dimensional grid

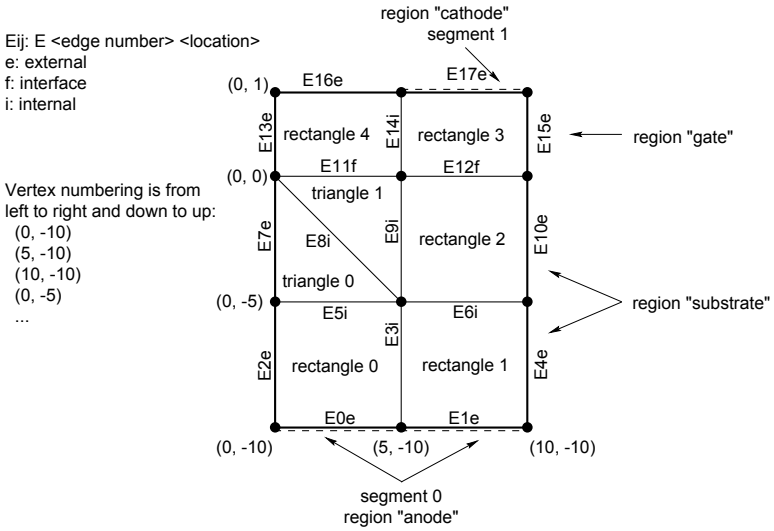


Figure 6.9 Two-dimensional grid

.

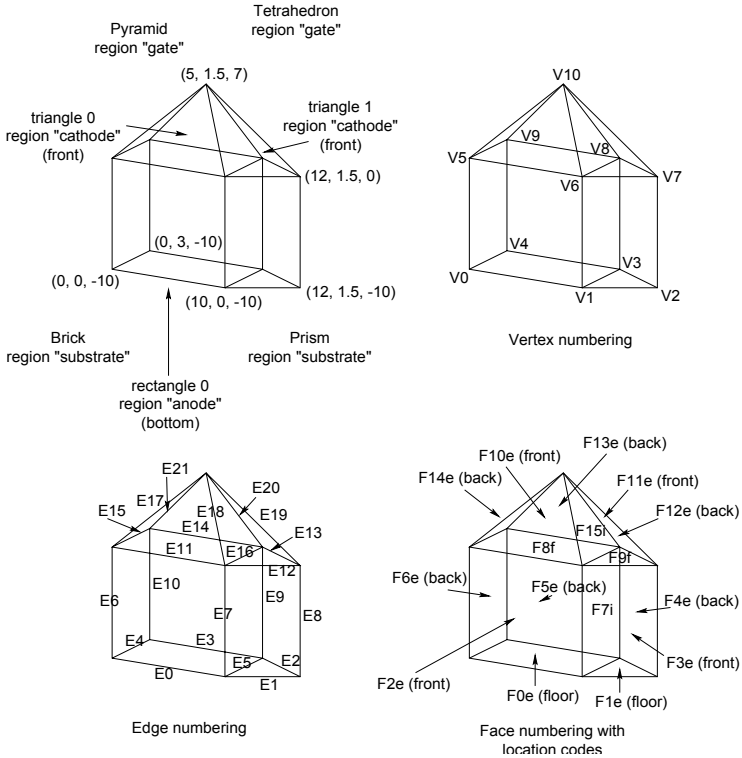


Figure 6.10 Three-dimensional grid

Part 6 – Utilities

3 – The file datexcodes.txt

3.1 Introduction

The file `datexcodes.txt` is the ISE configuration database for materials, doping species, and other quantities that are used in the semiconductor process and device simulation. The database is referred to by the various ISE TCAD software for different purposes. It does not contain physical properties of materials or quantities, but configuration properties such as names, colors, and labels.

The file is divided into two main sections, namely, Materials and Variables. Each material in the Materials section and each quantity in the Variables section are described by several properties that are explained in the following sections.

3.2 Material example

```
Titanium {
    code    = 22
    label   = "Ti"
    group   = Conductor
    color   = #e0eeee
    alter1  = Ti
    alter2  = 14
}
```

`Titanium`	`DATEX` material name.
`code`	Number which is used internally by tools.
`label`	Name used for display purposes.
`group`	Material classification (`Semiconductor`, `Conductor`, `Insulator`, or `All`).
`color`	Color used for display.
`alter1`	Name used for translation from and to SUPREM-4a.
`alter2`	Name used for translation from and to SUPREM-4b.

3.3 Variable example

```
ElectricField {
    code      = 18
    label     = "electric field"
    symbol    = "E"
    unit      = "V/cm"
    factor    = 1.0e+04
    precision = 4
    interpol  = linear
    material  = All
    alter1    = em
    alter2    = 103
}
```

ElectricField	DATEX variable name.
code	Number which is used internally by tools.
label	Name used for display purposes.
symbol	Symbol used for display.
unit	Unit used for display and data exchange.
factor	Scaling factor used for arsinh interpolation mode.
precision	Used for display of values in graphics tools.
interpol	Default interpolation mode (linear, log, or arsinh).
material	Specifies the validity of this quantity in material groups.
alter1	Name used for translation from and to SUPREM-4a.
alter2	Name used for translation from and to SUPREM-4b.

3.4 Environment variable DATEX

All ISE tools that use the datexcodes.txt file use the environment variable DATEX to locate the file. By default, the environment variable DATEX is set by the ISE generic startup script to:

```
$ISEROOT/tcad/$ISERELEASE/lib/datexcodes.txt
```

To use a customized version of the datexcodes.txt file:

1. Copy the datexcodes.txt file from the default location.
2. Edit the file as required.
3. Set the environment variable DATEX to the corresponding path.

Part 6 – Utilities
4 – DFISETOOLS

For DF–ISE `layout`, `boundary` and `grid` files, the following utilities are available:

- Print general information about the file
- Cut a 3D device with a plane
- Sample a device with a line segment
- Mirror a device through a plane
- Extend a 2D geometry in 3D
- Set or apply a transformation to a geometry
- Tessellate a 2D or 3D geometry
- Compress and decompress dataset files

Each operation corresponds to a `dfisetools` option:

`-ai`	Add interfaces to a `bnd` or `grd` file. This option creates a copy of the original file, which contains interfaces.
`-aid`	Add interfaces to a `bnd` or `grd` file, and the corresponding `dat` file. This option creates a copy of the original files, which contain interfaces.
`-compress`	Compress dataset files.
`-cut`	Cut.
`-expand`	Expand compressed dataset files.
`-extend`	Extend.
`-plot`	Plot.
`-r`	Resume.
`-reflect`	Mirror.
`-ri`	Remove interfaces from a `bnd` or `grd` file. Files created by FLOOPS contain interfaces between neighboring regions. Files that contain interfaces cannot be handled by all ISE TCAD tools. This option creates a copy of the original file without interfaces.
`-rid`	Remove interfaces from a `bnd` or `grd`, and a corresponding `dat` file. Files created by FLOOPS contain interfaces between neighboring regions. Files that contain interfaces cannot be handled by all ISE TCAD tools. This option creates a copy of the original files without interfaces.
`-set_trans`	Set transformation.
`-tessellate`	Tessellate.
`-transform`	Apply transformation.

DFISETOOLS can be used interactively or through a Tcl script file. DFISETOOLS performs all the operations specified by the options in the interactive mode on the given DF-ISE files. The produced files obtain an operation-dependent suffix *_cut* for cut plane results or *_plot* for plot results.

Usage `dfisetools` *option_list filename_list*

When used through a Tcl script, DFISETOOLS executes each command for the specified DF–ISE files, with the possibility to specify the suffix of the produced files (see Section 4.9 on page 6.40).

Usage `dfisetools` *script_name*

Examples on how to use DFISETOOLS can be found in the `DF-ISE/dfisetools` directory in the GENESISe example library.

4.1 DFISETOOLS file

Usage `dfisetools -r` *filename*

Validity DF–ISE layout files; 1D, 2D, 3D DF–ISE boundary files; 1D, 2D, 3D DF–ISE grid files

This function prints out:

- Type (layout, boundary, grid)
- Dimension (1D, 2D, or 3D)
- Number of vertices
- Number of edges
- Number of faces
- Number of elements
- Number of regions
- Bounding box of the object

4.1.1 Example: Obtaining a summary of a layout file

```
dfisetools -r DF-ISE/dfisetools/layout/layout.lyt

**************
3dgrid.grd
**************

grid file
dimension = 3
number of points = 1680
number of edges = 4822
number of faces = 4620
number of elements = 1567
number of regions = 7

Bounding box:
min = 0 0 1
max = 2 6.4 9.4
```

4.2 Cutting a 3D device with a plane

Usage `dfisetools -cut` *filename*

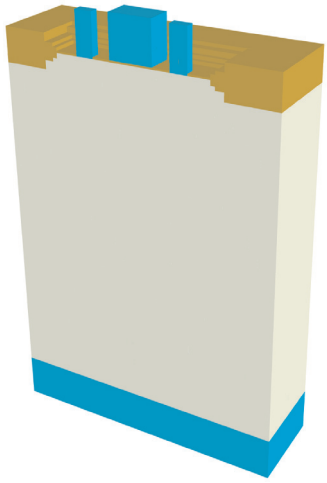
For a cut in a 3D grid, both `.grd` and `.dat` files are required.
Several `.dat` files can be associated to a same grid file.

Validity 3D DF–ISE boundary files, 3D DF–ISE grid files

For a 3D boundary, the result is a DF–ISE 2D boundary (*filename*`_cut.bnd`) that can be displayed with MDRAW. For a 3D grid, the result is a DF–ISE 2D grid (*filename*`_cut.grd and` *filename*`_cut.dat)` that can be examined with Tecplot-ISE. The user defines the cut plane by typing its origin and normal vector.

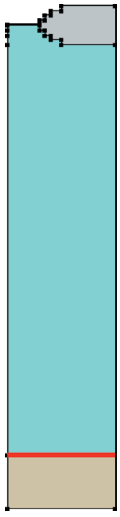
4.2.1 Example 1: Cut plane through a 3D boundary

Initial object



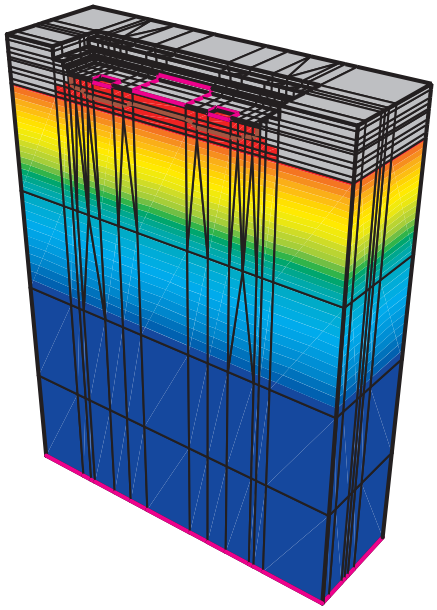
```
dfisetools -cut DF-ISE/dfisetools/3d/3dbound.bnd
```

Cut plane input: [origin = (0, 3.8, 1) - normal = (0, 1, 0)]



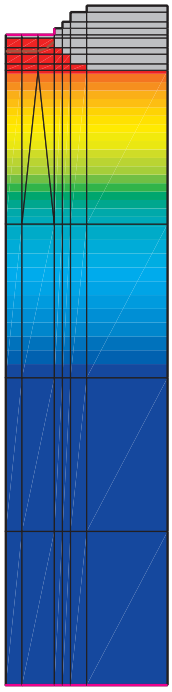
4.2.2 Example 2: Cut plane through a 3D grid

Initial object



```
dfisetools -cut DF-ISE/dfisetools/3d/3dgrid.grd DF-ISE/dfisetools/3d/3dgrid.dat
```

Cut plane input: [origin = (0, 3.8, 1) - normal = (0, 1, 0)]



4.3 Sampling a device with a line segment

Usage `dfisetools -plot` *filename*

For a plot in a 1D, 2D, or 3D grid; both `.grd` and `.dat` files are required.
Several `.dat` files can be associated to a same grid file.

Validity 1D, 2D, 3D DF–ISE grid files

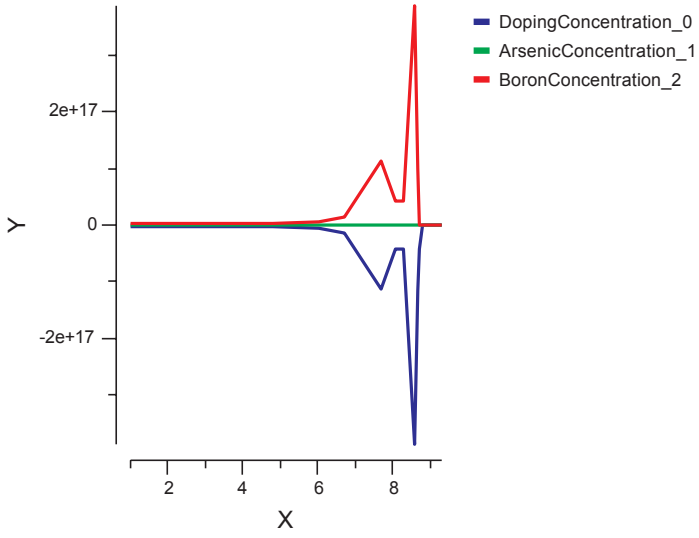
The result is a `DF-ISE plot` file (*filename*`_plot.plt`) that can be examined with INSPECT. The user defines the cut line by typing its two end-points.

4.3.1 Example 1: Cut line through a 3D grid

Initial object (see Section 4.2.2 on page 6.30)

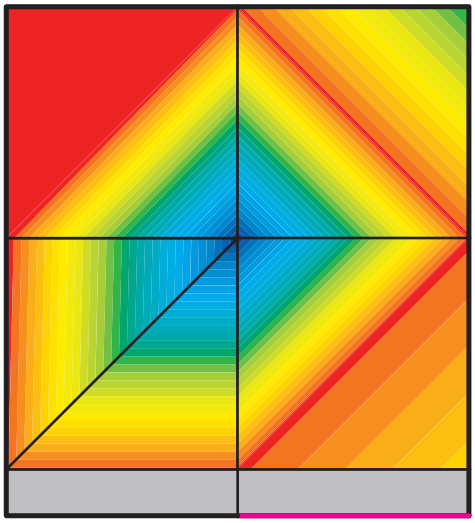
```
dfisetools -plot DF-ISE/dfisetools/3d/3dgrid.grd DF-ISE/dfisetools/3d/3dgrid.dat
```

Cut line input: [A = (1, 3, 1) - B = (1, 5, 9.36)]



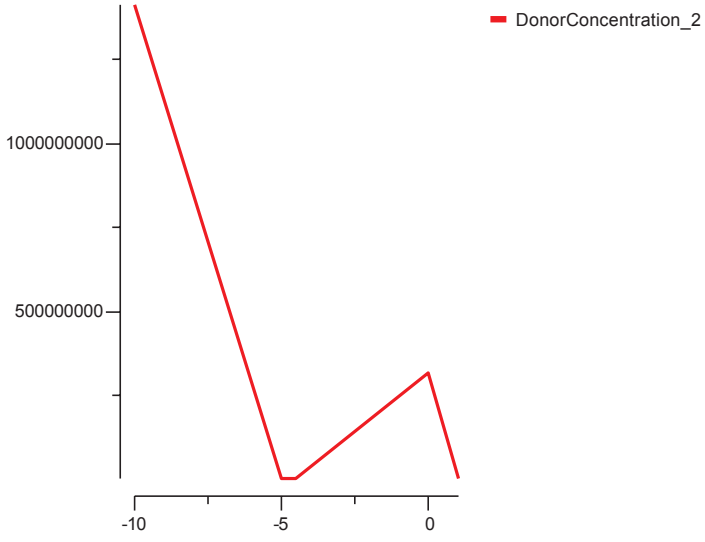
4.3.2 Example 2: Cut line through a 2D grid

Initial object



```
dfisetools -plot DF-ISE/dfisetools/2d/2grid.grd DF-ISE/dfisetools/2d/2grid.dat
```

Cut line input: [A = (0, -10) - B = (10, 1)]



4.4 Mirroring a device through a plane

Usage `dfisetools -reflect` *filename*

Validity DF–ISE layout files; 1D, 2D, 3D DF–ISE boundary files; 1D, 2D, 3D DF–ISE grid files

This function reflects an object through a given plane, and creates a new object by merging the initial object and its reflection.

NOTE If the object to reflect is a grid, the corresponding data file is accessed with the same basename as the geometry file (with the extension `.dat`).

Depending on the type of the object, one or two new DF–ISE files are created:

Layout files: *filename*`_refl_axis.lyt`

Boundary files: *filename*`_refl_axis.bnd`

Grid files: *filename*`_refl_axis.grd` and *filename*`_reflect_axis.dat`

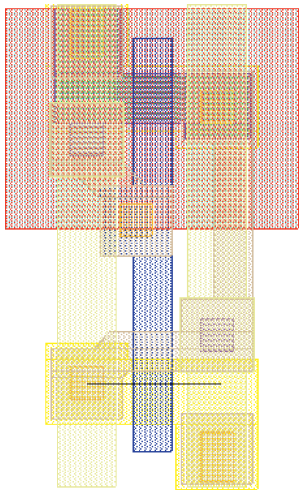
Tensor product files: *filename*`_refl_axis.ten`

There are six available mirror axis:

`x`	Vertical plane at x minimum
`X`	Vertical plane at x maximum
`y`	Vertical plane at y minimum (for 2D and 3D only)
`Y`	Vertical plane at y maximum (for 2D and 3D only)
`z`	Horizontal plane at z minimum (for 3D only)
`Z`	Horizontal plane at z maximum (for 3D only)

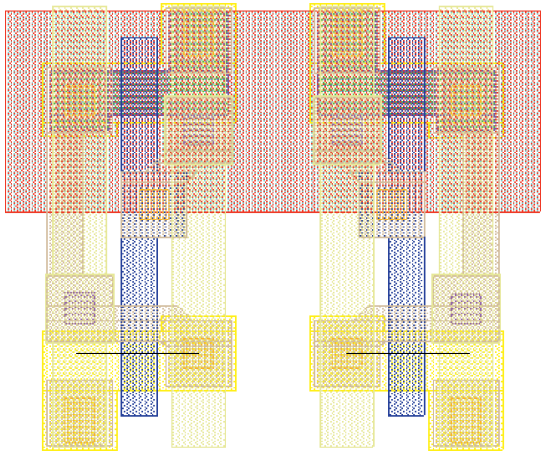
4.4.1 Example 1: Mirror a layout

Initial object



```
dfisetools -reflect DF-ISE/dfisetools/layout/layout.lyt
```

Reflect input: x



4.4.2 Example 2: Mirror a 1D grid

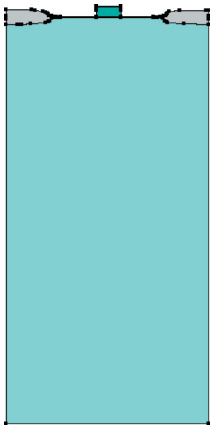
Initial object

```
dfisetools -reflect DF-ISE/dfisetools/1d/1dgrid.grd
```

Reflect input: x

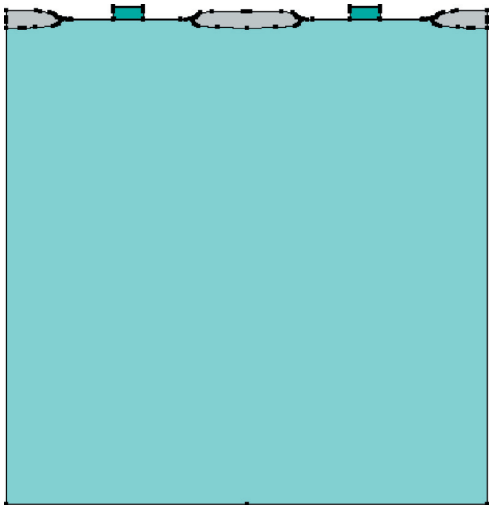
4.4.3 Example 3: Mirror a 2D boundary

Initial object



```
dfisetools -reflect DF-ISE/dfisetools/2d/2dbound.bnd
```

Reflect input: x

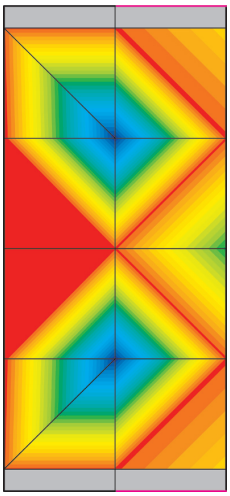


4.4.4 Example 4: Mirror a 2D grid

Initial object (see Section 4.3.2 on page 6.32)

```
dfisetools -reflect DF-ISE/dfisetools/2d/2dgrid.grd
```

Reflect input: y

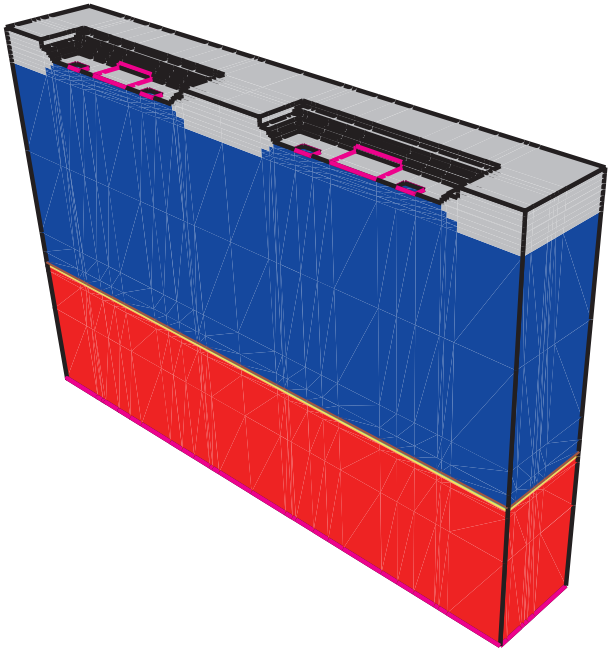


4.4.5 Example 5: Mirror a 3D boundary

Initial object (see Section 4.2.1 on page 6.29)

```
dfisetools -reflect DF-ISE/dfisetools/3d/3dbound.bnd
```

Reflect input: y

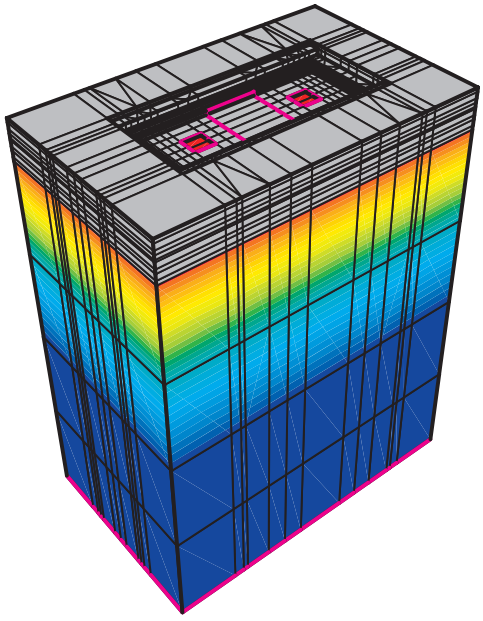


4.4.6 Example 6: Mirror a 3D grid

Initial object (see Section 4.2.2 on page 6.30)

```
dfisetools -reflect DF-ISE/dfisetools/3d/3dgrid.grd
```

Reflect input: x



4.5 Extension of 2D geometry

Usage `dfisetools -extend` *filename*

Validity 2D DF–ISE boundary files, 2D DF–ISE grid files

For a 2D boundary, the result is a DF–ISE 3D boundary (*filename*`_ext.bnd`). For a 2D grid, the result is a DF–ISE 3D grid (*filename*`_ext.grd` and *filename*`_ext.dat`) that can be examined with Tecplot-ISE.

NOTE If the object to reflect is a grid, the corresponding data file is accessed with the same base name as the geometry file (with the extension `.dat`).

The user defines the extension:

- Direction of the extension (normal to the 2D plane or opposite direction).
- Length of extension.
- Number of steps (resolution).

The extension is performed in two steps:

1. Extension of one step (length/number_of_steps).
2. Successive reflections until the chosen length is reached.

Therefore, only an odd number of steps is allowed.

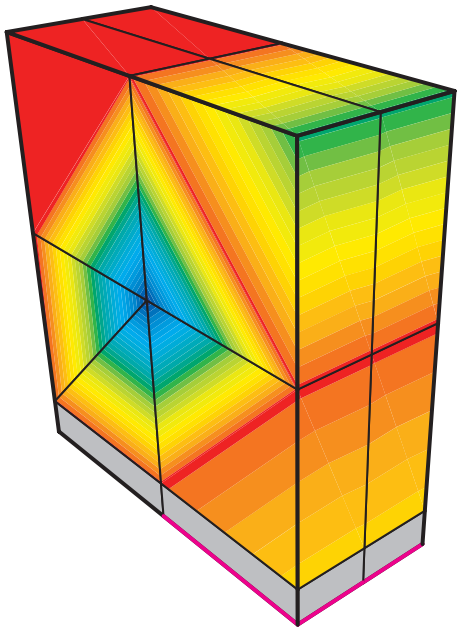
Example

Extension of a 2D grid

Initial object (see Section 4.3.2 on page 6.32)

```
dfisetools -extend DF-ISE/dfisetools/2d/2grid.grd
```

Extension input: [normal's direction, length = 4.0, step = 2]



4.6 Setting transformation on a geometry or tensor

Usage `dfisetools -set_trans` *filename*

Validity 2D, 3D DF–ISE boundary files; 2D, 3D DF–ISE grid files; 2D, 3D DF–ISE tensor files

For any type of geometry, this operation results in a modification of the translation vector and transformation matrix in the DF–ISE geometry file (*filename*`_tsf.{grd,bnd,ten}`).

Example

Set transformation of a 2D grid.

Initial Object (see Section 4.3.2 on page 6.32)

```
-> bounding box = (0, 0, -1) (10, 0, 10)
dfisetools -set_trans DF-ISE/dfisetools/2d/2grid.grd
```

Set Transformation Input:

```
[translation vector = (0, 0, 1)
transformation matrix = (0, 1, 0) (1, 0, 0) (-1, 0, 2)]
-> bounding box = (-1, -10, 2) (-1, 1, 12)
```

4.7 Applying transformation on a geometry or tensor

Usage `dfisetools -transform` *filename*

Validity 2D, 3D DF–ISE boundary files; 2D, 3D DF–ISE grid files; 2D, 3D DF–ISE tensor files

For any type of geometry, this operation consists of applying to the vertices of a DF–ISE geometry file the transformation defined in the translation vector and transformation matrix (*filename*`_tsf.{grd,bnd,ten}`). The translation vector of the output file is set to `null`, and its transformation matrix is set to `identity`.

4.8 Tessellating a geometry

Usage `dfisetools -tessellate` *filename*

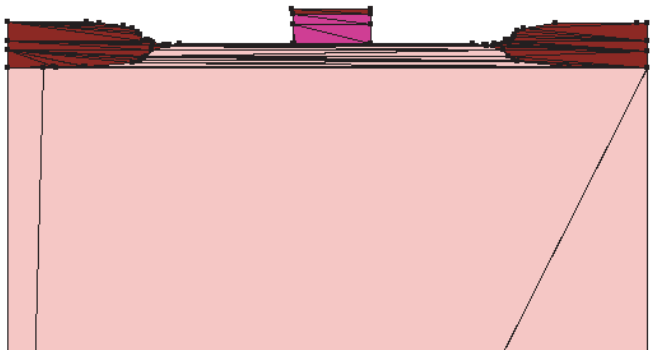
Validity 2D, 3D DF–ISE boundary files

This operation consists of triangulating each polygon that defines the input geometry. It produces a new `DF-ISE boundary` file (*filename*`_tess.bnd`).

4.8.1 Example 1: Tessellating a 2D boundary

Initial object (see Section 4.4.3 on page 6.35)

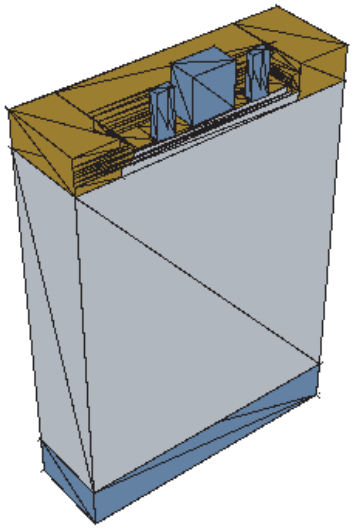
```
dfisetools -tessellate DF-ISE/dfisetools/2d/2dbound.bnd
```



4.8.2 Example 2: Tessellating a 3D boundary

Initial object (see Section 4.2.1 on page 6.29)

```
dfisetools -tessellate DF-ISE/dfisetools/3d/3dbound.bnd
```



4.9 Using DFISETOOLS with a Tcl script

All DFISETOOLS commands can be grouped into a Tcl script and then executed in sequence without any user input. In addition to the command inputs, a file extension is required. The examples presented are available in each project in the `DF-ISE/dfisetools` directory of the GENESISe example library. When DFISETOOLS is started without arguments, only a brief help message is printed, and the Tcl shell is not started automatically.

```
====================================================="
-- LAYOUT TESTS --
"====================================================="
```

Scripting option '*-resume*'

```
resume layout/layout.lyt
puts ">>> reflect layout x"
reflect layout/layout.lyt x rflx

"====================================================="
"-- 1d TESTS --"
"====================================================="
```

Scripting option '*-resume*'

```
resume 1d/1dgrid.grd
```

Scripting option '*-plot*'

```
puts ">>> plot 1d"
plot {1d/1dgrid.grd 1d/1dgrid.dat} {2 -11} plt
```

Scripting option '*-reflect*'

```
puts ">>> reflect 1d x"
reflect 1d/1dgrid.grd x rflx

"======================================================"
"-- 2d TESTS --"
"======================================================"
```

Scripting option '*-resume*'

```
resume 2d/2dgrid.grd
resume 2d/2dbound.bnd
```

Scripting option '*-plot*'

```
puts ">>> plot 2d"
plot {2d/2dgrid.grd 2d/2dgrid.dat} {0 -10 10 1} plt
```

Scripting option '*-reflect*'

```
puts ">>> reflect 2d grid x"
reflect 2d/2dgrid.grd x rflx
puts ">>> reflect 2d bound y"
reflect 2d/2dbound.bnd y rfly
```

Scripting option '*-extend*'

```
puts ">>> extend 2d"
extend 2d/2dgrid.grd {1 4 2} ext
extend 2d/2dbound.bnd {0 4 2} ext
```

Scripting option '*-set_trans*'

```
puts ">>> set_transform 2d grid"
set_trans 2d/2dgrid.grd {0 0 1 0 1 0 1 0 0 -1 0 2} tsf
```

Scripting option '*-transform*'

```
puts ">>> transform 2d bound"
transform 2d/2dbound.bnd tsf
```

Scripting option '*-tessellate*'

```
puts ">>> tessellate 2d bound"
tessellate 2d/2dbound.bnd

"======================================================"
"-- 3d TESTS --"
"======================================================"
```

Scripting option '*-resume*'

```
resume 3d/3dgrid.grd
resume 3d/3dbound.bnd
```

Scripting option '*-plot*'

```
puts ">>> plot 3d"
plot {3d/3dgrid.grd 3d/3dgrid.dat} {1 3 1 1 5 9.36} plt
```

Scripting option '*-cut*'

```
puts ">>> cut 3d grid"
cut {3d/3dgrid.grd 3d/3dgrid.dat} {0 3.8 1 0 1 0} cut
puts ">>> cut 3d bound"
cut {3d/3dbound.bnd} {0 3.8 1 0 1 0} cut
```

Scripting option '*-reflect*'

```
puts ">>> reflect 3d grid x"
reflect 3d/3dgrid.grd x rflx
puts ">>> reflect 3d bound y"
reflect 3d/3dbound.bnd y rfly
```

Scripting option '*-set_trans*'

```
puts ">>> set_transform 3d grid"
set_trans 3d/3dgrid.grd {0 0 1 0 1 0 1 0 0 -1 0 2} tsf
```

Scripting option '*-transform*'

```
puts ">>> transform 3d bound"
transform 3d/3dbound.bnd tsf
```

Scripting option '*-tessellate*'

```
puts ">>> tessellate 3d bound"
tessellate 3d/3dbound.bnd tess
```

4.10 Compressing and uncompressing

Usage `dfisetools -compress <filename>.grd <filename>.dat`

Validity 2D, 3D DF–ISE data files

This operation compresses all datasets in the file `filename.dat`.

Usage `dfisetools -expand <filename>.grd <filename>.dat`

Validity 2D, 3D DF–ISE data files

This operation uncompresses all datasets in the file `filename.dat`.

Part 6 – Utilities

5 – Interfise

Interfise allows for the conversion between the DF–ISE file format, and foreign and old file formats:

- Conversion of two common foreign SUPREM-IV-type formats to DF–ISE for 2D grids
- Conversion of DF–ISE to two common foreign SUPREM-IV-type formats for 2D grids
- Conversion of old file format to DF–ISE format
- Conversion of grid to boundary for 2D grids
- Conversion of DF–ISE boundary file to OMEGA bound format
- Conversion of OMEGA bound format to DF–ISE boundary file

NOTE Interfise is continually being extended and improved. Due to the difficulty of recording changes to file formats that are not under the control of ISE, contact ISE Support if you have problems, questions, or suggestions regarding the support of specific file formats.

5.1 Conversion between DF–ISE and SUPREM-IV format

The translation of material and species names between the SUPREM-IV-type foreign format and DF–ISE format is handled through the file `datexcodes.txt` (see Chapter 3 on page 6.25). The user can modify any already defined translation in the `datexcodes.txt` file or can add new translation entries to this file.

To add new translation entries, for example, for a given native species name, in the file `datexcodes.txt`, the entries `alter1` and `alter2` specify the translated names for the options `sup4a2ise` and `sup4b2ise`, respectively:

```
Silicon {
    ... other labels as described in the documentation of datexcodes.txt
    alter1 = Si    # used for sup4a translation
    alter2 = 3     # used for sup4b translation
}
```

These changes affect the Interfise options:

`-sup4a2ise` See Section 5.1.1 on page 6.44.

`-ise2sup4a` See Section 5.1.2 on page 6.44.

`-sup4b2ise` See Section 5.1.3 on page 6.44.

`-ise2sup4b` See Section 5.1.4 on page 6.45

In addition, for the SUPREM-4b format only, the translation has been extended to accommodate material and species defined as two or more separate variables in the foreign format (SUPREM-4b), but which correspond to a single vector field in the DF–ISE format.

For example, if two separate variables (`120` and `121`) are defined in the SUPREM-4b file and correspond to a single vector field (`"ElectricField"`) in DF–ISE, users can define such a translation in the file `datexcodes.txt`

as shown below, and Interfise (using option `-sup4b2ise`) translates it into single vector variable `ElectricField`. For this example, the entry in the `datexcodes.txt` file is:

```
ElectricField{
    ... other labels as described in the documentation of datexcodes.txt
    label  = "electric field"
    alter1 = em          # used for sup4a translation
    alter2 = {120,121}   # used for sup4b translation with dual vector components
}
```

NOTE This extension is valid only for SUPREM-4b file translation. Therefore, it is applicable only to the Interfise option `-sup4b2ise`.

5.1.1 SUPREM-IV format A to DF–ISE 2D grid file converter

Usage `interfise -sup4a2ise` *suprem_file basename*

This operation creates four files that can be used by MDRAW:

basename`.bnd` 2D DF–ISE boundary

basename`.cmd` MDRAW command file

basename`.grd` 2D DF–ISE grid

basename`.dat` 2D DF–ISE data

The generation of the boundary file for MDRAW uses the algorithms used for the boundary extraction in Section 5.3 on page 6.45 and, therefore, suffers from the same limitations. For recommendations on how to overcome these limitations, see Section 5.3 on page 6.45.

In structures that contain silicon–germanium, the material silicon is translated to silicon–germanium and the germanium concentration is converted to a mole fraction.

5.1.2 DF–ISE to SUPREM-IV format A 2D grid file converter

Usage `interfise -ise2sup4a` *basename suprem_file*

This operation reads a grid and data file (`basename.grd` and `basename.dat`) and creates a `suprem_file`.

5.1.3 SUPREM-IV format B to DF–ISE 2D grid file converter

Usage `interfise -sup4b2ise` *suprem_file basename*

This operation creates four files that can be used by MDRAW:

basename`.bnd`: 2D DF–ISE boundary

basename`.cmd`: MDRAW command file

basename`.grd`: 2D DF–ISE grid

basename`.dat`: 2D DF–ISE data

Translation of discontinuous datasets is supported. The generation of the boundary file for MDRAW uses the algorithms used for the boundary extraction in Section 5.3 and, therefore, suffers from the same limitations. For recommendations on how to overcome these limitations, see Section 5.3.

5.1.4 DF–ISE to SUPREM-IV format B 2D grid file converter

Usage `interfise -ise2sup4b` *basename suprem_file*

This operation reads a grid and data file (`basename.grd` and `basename.dat`) and creates a `suprem_file` in Format B. Translation of discontinuous datasets is not supported.

5.2 Old format to DF–ISE

Usage `interfise -old2new` *filename1 filename2 ... filenamen*

This converter is useful for converting geometry or data files obtained with an earlier version of DF–ISE or DATEX files to the DF–ISE format. It requires a list of boundary, grid, or data files (2D or 3D).

The name of each output file is automatically generated by adding the suffix *_dfise* to the name of the original file.

5.3 Boundary extraction

Usage `interfise -grd2bnd` *basename*

This option enables the extraction of the boundary of a 2D or 3D grid object. The input grid is given in the file `basename.grd`. The output boundary is written in the file `basename.bnd`. This operation is automatically called when using the SUPREM-IV-to-DF–ISE converter (see Section 5.1.1 on page 6.44).

The boundary extraction algorithm used in the option `-grd2bnd`, which is also used to generate MDRAW boundary files for the options `-sup4a2ise` and `-sup4b2ise`, may fail on geometrically complex structures with multiply connected regions, or regions with the same name and material in several places of the structure. Common cases are nitride spacers on both sides of a polygate or a polygate completely enclosed by oxide. The resulting boundary can, therefore, be faulty, which results in error messages in MDRAW or MESH. The following solutions can be used:

- If DIOS is available, the grid and dataset files produced by Interfise can be loaded into DIOS. A DIOS `save` statement with MDRAW writes both the MDRAW boundary and command files. A more stable algorithm for boundary extraction is used. Furthermore, the MDRAW command file generated by DIOS contains refinement boxes that are based on an analysis of the structure and, therefore, are optimized for the structure, instead of the dummy command file produced by Interfise.
- If DIOS is not available and MESH is used, the grid file can be copied to the boundary file name. MESH extracts the boundary from the structure with a robust algorithm. This solution does not work with MDRAW.

5.4 DF–ISE boundary to OMEGA bound format

Usage `interfise -bnd2bound` *bnd_file bound_file*

This option enables the conversion of a DF–ISE boundary format file to an OMEGA-style bound format file.

5.5 OMEGA bound format to DF–ISE boundary

Usage `interfise -bound2bnd` *bound_file bnd_file*

This option enables the conversion of an OMEGA-style bound format file to a DF–ISE boundary file. This is useful if an initial device is built manually, which is rather simple using OMEGA syntax. By using this conversion, further steps, such as meshing and device simulation, can be performed.

NOTE Interfise does not understand lines in Contact definitions of the bound syntax. It also expects all coordinates in the bound file to be in 3D, before attempting conversion to boundary file format.

5.6 Using Interfise with a Tcl script

Interfise can also be used with a Tcl script, in which case, the names of the commands are the same as the command-line options.

All Interfise commands can be grouped into a Tcl script and then executed in sequence without any user input. The contents of the Tcl script have the format shown in the following example; several commands can be put into the same file:

```
grd2bnd grd2bnd/test2d
grd2bnd grd2bnd/test3d
```

Part 6 – Utilities
6 – CURPOT

6.1 Overview

CURPOT is a small postprocessing tool for computing the current potential from the total current density for 2D simulations performed with DESSIS. The computational method is implemented after[1]:

Abstract
The conservation of the total current density in semiconductor devices implies that the current derives from a vector potential. The calculation of this current potential is a dual problem of the original device simulation problem. A simple and elegant discrete method is proposed for the 2D case, which yields the current potential for a given, numerical calculated current density. The approach is based upon a least squares principle and is consistent with the assumptions leading to the discretized formulation of the semiconductor transport equations. Accurate values of the contact currents are obtained and a simple way to generate representations of the current lines becomes available. The method has the advantage that it does not require any definition of paths for the integration of current lines.

6.2 Using CURPOT

CURPOT is used as follows:

```
curpot <grid file> <data file> <output file>
```

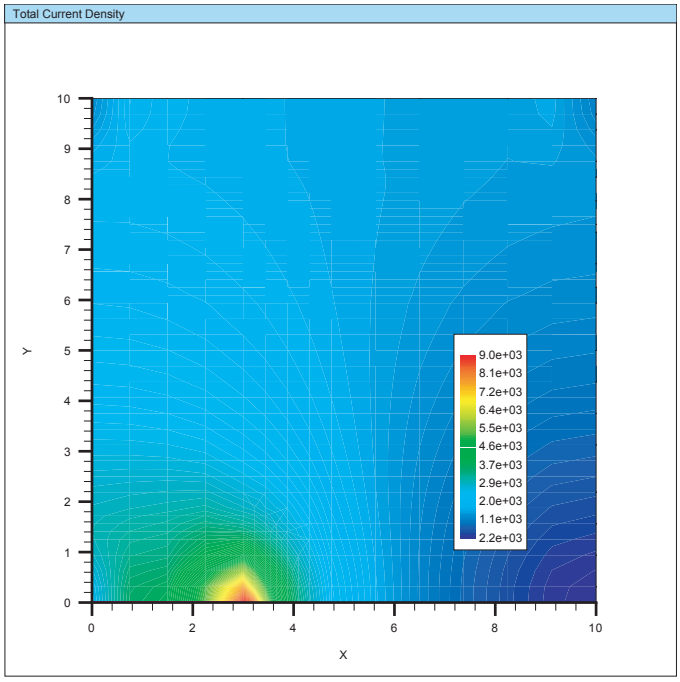
where `<grid file>` is the name of the `grid` file belonging to the simulated problem, `<data file>` is the name of the `plot` file saved by DESSIS after successfully finishing a simulation, and `<output file>` is the name of the file where the data for the current potential should be stored (in DF–ISE format).

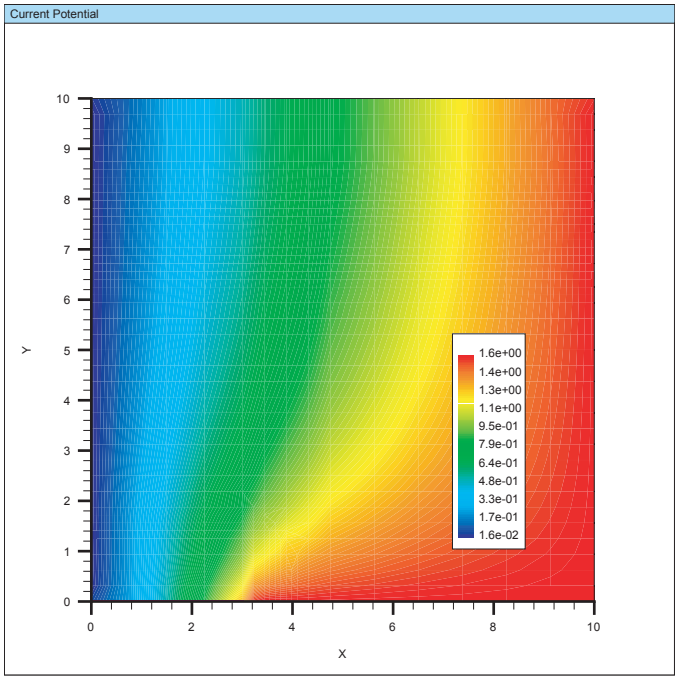
NOTE The `<data file>` should contain (at least) the plot data for the total current density, that is, `TotaCurrentDensity` should have been specified in the `Plot` statement of the DESSIS input file.

1. E. Palm and F. van de Wiele, “Current Lines and Accurate Contact Current Evaluation in 2-D Numerical Simulation of Semiconductor Devices,” *IEEE Transactions on Electron Devices*, vol. ed-32, no. 10, pp. 2052–2059, October, 1985.

6.3 Example

The following figures display plots of the current density and current potential for a simple device. The current potential was computed with CURPOT.





Part 6 – Utilities

7 – Converter for SUPREM-IV: alien2lig

7.1 Introduction

This utility allows the conversion of input decks of SUPREM-IV or its derivatives into the semiconductor process representation (SPR) language of LIGAMENT (see LIGAMENT, Section 4.8 on page 3.74). The fundamental idea behind this program is not the translation of fully calibrated process flows, but the possibility of a smooth and user-friendly transition from one format to another.

Since `alien2lig` invokes LIGAMENT after its conversion process, the user automatically obtains a command file for the required ISE process simulator:

```
SUPREM® IV => alien2lig => LIGAMENT => DIOS/FLOOPS/DEVISE
```

7.2 Main features

7.2.1 Input format

The program supports most of the input format features of SUPREM-IV, for example, macro definitions are recognized and expanded, arithmetic expressions are calculated, lines can be concatenated using a plus (+) sign, and parameter and statement abbreviations are allowed. However, we restrict ourselves to official documented input syntax.

Special characters are:

- $
- +
- ;
- ‘@{macro}’ and ‘@macro’
- %
- ^, !, #
- Arithmetic operations (+, –, /, E, e)

7.2.2 Supported commands

In general, `alien2lig` only translates process simulation and several auxiliary commands:

- Comment
- Source
- Return (interactive mode is not supported)

- Foreach (we cannot generate loops, but we take the first element of the list and treat this command as a ‘define’ statement)
- Define
- Undefine
- Line (to calculate the lateral and vertical extent of the simulation domain)
- Initialize
- Mask
- Mesh (adaptive grid algorithms are implemented in DIOS, therefore, we ignore this command)
- Deposition
- Expose
- Develop
- Etch
- Implantation
- Diffusion
- Epitaxy
- Method

and their synonyms. Of course, not all parameters of these commands are supported.

7.3 Specialities

7.3.1 Layout and masks

`alien2lig` translates LORENZO-IC® layout information into the CIF format, which can be processed by LIGAMENT and visualized by PROLYT.

Furthermore, we clearly distinguish between process flow and layout information. Therefore, we also generate CIF masks for commands such as:

```
etch nitride right p1.x=0.25
```

In SUPREM-IV, a definition of a particular mask is given as:

```
deposition thick=1 positive Photores
expose mas=nwell
develop
```

The SPR language of LIGAMENT contains one command for these three process steps:

```
pattern (layer: nwell, polarity: light_field,
thickness: 1 um, side: front, type: default)
```

Therefore, if we detect a `deposition` statement with parameter `Photores`, we store the polarity and skip this `deposition` command (see Section 7.6 on page 6.52).

7.3.2 Etching

Since DIOS cannot support pure geometric etch algorithms, a support of arbitrary trapezoidal etch steps is not feasible. Nevertheless, simple etch statements are converted (see above) and we try to detect process steps for generating ‘spacers.’

For example, in the latter case, `alien2lig` and LIGAMENT write a DIOS command file that includes `??`. This command file forces an error during syntax check:

```
! SUPREM: etch NITRIDE   THICKNESS=0.15    TRAPEZOI   ANGLE=80
Etch (time=1, Rate (mat=??nitride??, a1=150.0))
```

The user is requested to explicitly confirm this translation by editing the DIOS command file. This precaution prevents long simulation runs with wrong or incomplete command files.

7.4 Running alien2lig

If `alien2lig` is called by:

```
alien2lig
```

or:

```
alien2lig -h
```

possible command-line options with short descriptions are printed:

```
Usage of this SUPREM-IV to LIGAMENT/SPR converter, version 0.95:
supremName of SUPREM-IV process flow description

-cif          Name of the mask file (CIF format)
-spr          Name of the process flow (SPR format)
-cmd          Name of the command file for '-simulator'
-simulator    Name of the simulator (dios, floops, devise)
-depth        Specifies the depth of the simulation domain microns)
-info         Level of information (0-2)
-debug        Level of debug information (0-2)
-help         Prints this message
```

The simplest way to run `alien2lig` is to copy one SUPREM-IV input deck (`test.inp`) with all additional files (for example, `layout`) into the current working directory and to start the program by using:

```
alien2lig test.inp
```

7.5 Output

`alien2lig` generates two files, `lig.cmd` and `prl.par`, which are stored in the current working directory. The default names for these files can be changed through the command-line options:

```
alien2lig test.inp -cif test.cif -cmd test.cmd
```

Both files are processed by LIGAMENT, which finally writes a command file (for example, `test_dio.cmd`) into the directory of `test.inp`.

By default, DIOS is used as target, but the optional parameter `-simulator` enables a switch to other simulators.

7.6 Known problems

- Deposit Photoresist: As stated in Section 7.3 on page 6.50, LIGAMENT and DIOS take care of pattern operations by using a single command. This feature leads to the problem that ‘extra’ photoresist deposition statements are ignored.

- Incomplete or ‘unofficial’ SUPREM-IV input decks: Sometimes, `alien2lig` cannot resolve all macro definitions (missing files that are read by using the `source` command, unofficial syntax, and so on), in which case a message is displayed that indicates a problem.

  In this special case, load the LIGAMENT command file (`lig.cmd`) and investigate the questioned line (361). Corresponding SUPREM-IV commands are also listed in this command file.

Part 6 – Utilities

8 – DIOS to FLOOPS converter

8.1 Introduction

The DIOS to FLOOPS converter enables the conversion of DIOS command files to SPR command files with additional information for further conversion to FLOOPS command files. Conversion is performed in two steps. First, the converter front-end converts the DIOS command files to SPR command files. Second, the LIGAMENT FLOOPS back-end is used to create FLOOPS command files.

The purpose of this converter is to help users change from DIOS to FLOOPS command files by converting existing DIOS command files to FLOOPS syntax. The conversion saves a lot of manual inputting, but for various reasons, a full automatic conversion is not possible.

Only process-related DIOS commands are translated to SPR. Physical models and model parameters are not translated. Users must check the result of the conversion carefully and adapt it to their needs. Users must add mesh-related specifications.

8.2 Usage

The DIOS to FLOOPS converter is a LIGAMENT module. It can be used in two different ways.

8.2.1 From LIGAMENT

The command-line option `-input` with the argument `dios` can be used to load a DIOS command file and create a FLOOPS command file:

```
ligament -input dios dio.cmd floops.cmd
```

LIGAMENT can also be used to convert a DIOS command file to SPR for further processing by the user. When used with the command-line option `-print`, the converted command file is written to standard output in SPR format. A file can be created by redirecting the output:

```
ligament -input dios -print dio.cmd > spr.cmd
```

8.2.2 From LIGEDIT

The command-line option `-input` with the argument `dios` can be used to load a DIOS command file instead of an SPR file. The result of the conversion can be modified by the user and can be either saved in SPR format or translated to the target simulator:

```
ligedit -input dios dio.cmd
```

8.3 Supported commands

The DIOS to FLOOPS converter only supports process-specific commands. Commands that are simulation specific or simulator specific are not supported.

By default, DIOS commands that are not supported by the converter are inserted into the flow as remarks. These remarks start with the prefix `unsupported command:` and contain the command with its arguments. This behavior can be switched off (see Section 8.5 on page 6.57). The following DIOS commands are supported by the converter:

- `comment`
- `deposit`
- `diffusion`
- `etching`
- `grid`
- `implantation`
- `mask`
- `substrate`
- `title`

The names of the commands can be abbreviated as in DIOS.

The following sections list the supported parameters for each supported command. The shortest possible abbreviation for each parameter is given in parentheses.

Parameters that are not listed here are *not* supported. By default, unsupported parameters are converted to a remark with the prefix `unsupported parameter:`. This behavior can be switched off (see Section 8.5 on page 6.57).

Supported parameters that cannot be represented by SPR parameters are inserted into the `type` field using the appropriate syntax and values for DIOS and FLOOPS.

Besides these commands, the converter translates comments (lines starting with '!') to SPR remarks. Conditionals (`#if`, `#elif`, `#else`, and `#endif`) are converted to the corresponding SPR commands. See Section 8.4 on page 6.56 for details about the limitations of converting conditionals.

8.3.1 comment

DIOS `comment` commands are converted to SPR `comment`s. Double quotation marks in the comment string are converted to single quotation marks.

8.3.2 deposit

DIOS `deposit` commands are converted to SPR `deposit` commands. The supported parameters are:

- `concentration` (`con`)
- `element` (`e`)
- `material` (`m`)
- `thickness` (`th`)

8.3.3 diffusion

DIOS `diffusion` commands are converted to SPR `anneal` or `epitaxy` commands. If the parameter `thickness` has a value greater than zero, an SPR `epitaxy` command is created. For an `epitaxy` command, only one time value

and one temperature value are allowed. The full range of time, temperature, and temperature rate specifications that are possible in DIOS are supported. The supported parameters are:

- atmosphere (at)
- concentration (c)
- element (el)
- flow (f)
- h2 (h2)
- h2o (h)
- hcl (hc)
- n2 (n)
- o2 (o)
- ph2o (ph)
- po2 (po2)
- pressure (press)
- temperature (te)
- temprate (tempr)
- thickness (th)
- time (t)

8.3.4 etching

DIOS `etching` commands are converted to SPR `etch` commands. The SPR support for specifying etching rates and etching types is very limited. As there are several different possibilities in DIOS to specify etching rates and as, presently, it is not clear how this can be best translated to FLOOPS, conversion of etching rates is not supported. If no value for the parameter `remove` is specified, the etch type `strip` is used. Otherwise, the etch type is `isotropic`. The supported parameters are:

- material (m)
- remove (re)
- over (o)

8.3.5 grid

The values of the parameters `x` and `y` are used for the value of the `region` field of the SPR `environment` command. The supported parameters are:

- x (x)
- y (y)

8.3.6 implantation

DIOS `implantation` commands are converted to SPR `implant` commands. The supported parameters are:

- afactor (af)
- amorphization (amorphi)
- damage (da)
- dose (do)
- element (e)
- energy (en)
- function (fu)
- ifactor (if)
- numsplits (nu)
- revolving (rev)
- rotation (ro)
- tilt (t)
- vfactor (vf)

8.3.7 mask

DIOS `mask` commands are converted to SPR `pattern2d` commands. The supported parameters are:

- `thickness` (`th`)
- `x` (`x`)

8.3.8 substrate

DIOS `substrate` commands are converted to SPR `substrate` commands. The parameter `rho` is only used if it is specified in the input file. The supported parameters are:

- `concentration` (`co`)
- `element` (`e`)
- `orientation` (`orie`)
- `rho` (`rh`)

8.3.9 title

DIOS `title` commands are converted to SPR `environment` commands. The supported parameter is `title` (`t`).

8.4 Limitations

DIOS interpreter control commands (for example `@FILENAME`, `break`, and `set`, see DIOS, Section 3.5.2 on page 8.59) are not supported.

Comments that are placed between arguments of a DIOS command are ignored. They are not converted to SPR remarks. Comment blocks (beginning with `/*` and ending with `*/`) are not supported.

GENESISe preprocessor conditional directives (`#if`, `#elif`, `#else`, and `#endif`) must be placed between DIOS commands or comments. They cannot be placed in the list of arguments to choose between different values. Therefore, the following DIOS input cannot be converted properly:

```
diff:(
    #if @< variant == 0 >@
    moddif=Equilibrium
    #else
    moddif=PairDiffusion
    #endif
)
```

The converter displays a warning message and uses the last specified value for each argument. The previous example has to be changed to:

```
#if @< variant == 0 >@
diff:(moddif=Equilibrium)
#else
diff:(moddif=PairDiffusion)
#endif
```

or parameterized using GENESISe variables.

Other preprocessor directives (`#do` and `#enddo`) are not supported. Such lines are converted to SPR remarks.

Only simple expressions are evaluated correctly. More complicated expressions may give unexpected results. This can be either a Tcl syntax error or a wrong result. Users must check the result of any converted expression.

8.5 Customization

The converter looks for a file named `user.tcl` in the working directory, the directory given by the relative path `../input/dios` relative to the directory specified with the command-line option `-libdir`, or in the directory `$ISEROOT/tcad/$ISERELEASE/lib/ligamentlib/input/dios`. When the file is found, it is sourced. The file `user.tcl` can be used to customize and extend the converter.

By setting the variable `::InputDios::debug` to zero, unsupported commands are not converted to SPR remarks. The variable `::InputDios::unsupportedParameter` can be used in the same way to switch off the output of unsupported parameters in the form of SPR remarks.

Part 6 – Utilities

9 – MEASURE

9.1 Introduction

MEASURE is a Tcl script that extracts values from protocols with the help of templates. The generator of the protocols is not necessarily DIOS or a TCAD tool. For simplicity, this section uses the term ‘simulator’ to refer to the generator of the protocol. All that is necessary to use MEASURE is the ability to create templates and command files for it. The templates allow MEASURE to recognize a region in the protocol of the simulator from which to take the values.

MEASURE is a fully integrated tool in GENESISe.

In its simplest form, a template consists of a single line, for example, the template `thres.tmpl`:

```
threshold voltage= @<m1>@ V
```

Here, `m1` is a label that acts as a dummy argument and connects the extracted parameter to a user-specified name. The function as dummy argument is given by the special strings `@<` and `>@`. To extract a value, the other strings in the template must match exactly the content of the protocol file. The connection to a user-specified name of the extracted parameter is given by using the command file for MEASURE, for example, `thres_msr.cmd`:

```
{thres.tmpl {m1 THRESHOLD}}
```

For example, assume the protocol file of the simulator is called `n9.AUS`. A few lines from `n9.AUS` follow:

```
ETCH PO;
UTH;
threshold voltage......ug = -1.282+/-0.002 V
! UTH METHOD=0;
UTH METHOD=1;
threshold voltage= -1.43 V
PLOT FILE=N9_ID ID;
```

The command:

```
% measure -in n9.AUS -c thres_msr.cmd -out n9_msr.out
```

writes the following line into the result file `n9_msr.out`:

```
DOE: THRESHOLD -1.43
```

Here, `DOE:` is a special label used by the extractor of INSPECT to transfer the name `THRESHOLD` and the extracted value `-1.43` from `n9_msr.out` into the GENESISe Family Table.

To summarize this example, a protocol file (`n9.AUS`), a command file for MEASURE (`thres_msr.cmd`), and a template file (`thres.tmpl`) are required to run MEASURE.

9.2 Command-line options

`measure` without any argument, `measure -h`, or `measure -help` produces the output:

```
% measure -h
measure 6.0.0
usage: measure -options
```

MEASURE extracts the values located at the positions of the marks in the templates from the protocol file(s) and writes them to the result file(s) as a multicolumn table: `DOE: name_i value_i`.

Table 6.3 Command-line options

Options	Description
`-help`	Prints this message.
`-h`	Prints this message.
`-v`	Prints version number.
`-in fileName`	Name of a protocol file to be scanned for values. The metacharacter * can specify more than one protocol file.
`-out fileName`	Name of a file containing the extracted values. No specification causes the extracted values to be written only to standard output.
`-log fileName`	Name of a file to protocol the activities of MEASURE itself. If not specified, protocol only to standard output.
`-nounits`	Remove the units from the extracted values. Default is extraction with units if the value and its unit built up a word.
`-lib`	Directory containing the templates, defaults to `{'pwd', $DIOS_LIB, $ISEROOT_LIB/dioslib}`.
`-genes`	GENESISe mark for extracted values; defaults to `DOE:`.
`-info`	Level of mirroring the actions of MEASURE. Should be an integer greater than –1. Defaults to 0.
`-c fileName`	Name of a command file for MEASURE.
`-c cmdString`	String containing commands for MEASURE.
`-on string`	String(s) used to start the action of MEASURE in all protocols for all templates. Such strings should not appear in the templates.
`-off string`	String(s) used to stop the action of MEASURE in all protocols for all templates. Such strings should not appear in the templates.
`-r`	Reverse mode. The protocol file(s) are read with reversed line order (from the end to beginning). The templates are also reversed except for blocks contained in `MEASURE_REPEAT`... `MEASURE_END_REPEAT` clauses.
`-unique string`	String to be appended with a counter to repeated instances of base names. Defaults to `Q`.
`-lmb`	Left marker boundary for the template marks. Defaults to `@<`.
`-rmb`	Right marker boundary for the template marks. Defaults to `>@`.
`-b`	Batch mode. The option `-log` is ignored. No protocol of the activities of MEASURE is given. The extracted values are *always* written at least to standard output.

The interactively given command should have the form:

```
-c "{{templ1 {mark1 name1} ... {markN name1N}} \
... {templN {{mark1 nameN1} ... {markN nameNN}}}"
```

NOTE Use quotation marks; otherwise, the Tcl shell cannot parse the input. In a command file, quotation marks and backslashes must not be included. The base names of the extracted values `name11`, ..., `name1N`, ..., `nameNN` should be unequivocal. Otherwise, different numbers appear with the same base name and introduce confusion to the family tree.

If the file name following the `-in` option contains the metacharacter *, MEASURE processes all files that match that name.

The option `-c` should be followed by a MEASURE command or by the name of a file containing a MEASURE command. The templates can contain regular expressions to widen the range of matching regions in different protocols or, with other words, to decrease the number of necessary templates to perform a meaningful parameter extraction.

The output file name can be specified with the `-out` option. If this is the case, all output is written to this file. If no `-out` option is given, the extracted values are written only to standard output. They are always written to standard output, even if MEASURE works in batch mode.

MEASURE protocols its own activities to standard output and to the file specified using the `-log` option, if the latter is given. In batch mode, MEASURE does not report its own activities to standard output.

MEASURE does not overwrite protocol files from other tools with its own output. For example, if the user specifies:

```
% measure -in x_dio.log -out x_dio.log -log x_dio.log -c y_msr.cmd
```

MEASURE reports the extracted values to `x_dio.log_1` and protocol to `x_dio.log_2`. Then, if the user specifies:

```
% measure -in x_dio.log -out x_dio.log_1 -log x_dio.log_2 -c y_msr.cmd
```

MEASURE changes the name of the old `x_dio.log_1` to `x_dio.log_3` and the old `x_dio.log_2` to `x_dio.log_4` in order to write the information as far as possible to the expected files. No files should be lost with this procedure.

By using the options `-on` and `-off`, strings can be specified to search in between only in the protocol file.

NOTE A string cannot appear as an ‘on string’ and an ‘off string.’

The strings are periodically applied. For example, if two ‘on strings’ and three ‘off strings’ are given, the protocol file is scanned between `on_string1` and `off_string1`, `on_string2` and `off_string2`, `on_string1` and `off_string3`, `on_string2` and `off_string1`, and so on until the end.

Do not exclude the interested region of the protocol file. For example, if the MEASURE command is:

```
% measure.tcl -info 2 -log simplimpl.log\
-c "{impl.tmpl {m3 DOSE} {m7 ROTAT}}" \
-on {"---------------------------------"}\
-off {"----------------------------- IMPLantation -----"}
```

then, the region between the ‘off string’ and ‘on string’ is excluded, and MEASURE cannot, for example, extract the implantation dose because the implantation dose always appears in DIOS after the string given in the above example as ‘off string.’

In addition, the ‘on strings’ and ‘off strings’ must not appear in the templates; otherwise, MEASURE exits with an error.

The option `-nounits` removes the units from the extracted values. If a value does not match as a number, a warning is given and the value appears as `NaN` in the table of the extracted values.

9.3 Matching mechanism and templates

Currently, templates must be created manually, with no support from MEASURE. It is the responsibility of the user to ensure that the simulator prints the information that is of interest to the user. If the protocol does not contain at least one region that matches at least one of the specified templates, MEASURE cannot extract any value and returns an error.

There are four levels of matching between template lines and protocol lines:

- Exact or literal match
- Regular expression match
- Skip a dummy argument
- No match

White space is irrelevant in matching.

An exact match is important to determine whether a region in a protocol is represented by a template. A regular expression match gives some freedom to reduce the number of templates or to increase their applicability for extractions.

In the simplest case, a template consists of a single line, for example:

```
dopant = [a-zA-Z]+   energy = @<m7>@
```

This template could be used to extract the implantation energy from DIOS protocols. The string `[a-zA-Z]+` is a regular expression matching most possible dopant names, but it does not match `BF2`.

If the template is:

```
dopant = As          energy = @<m7>@
```

only the energies of arsenic implants can be extracted.

For example, imagine that the template:

```
dopant = [a-zA-Z]+   energy = @<m7>@
```

is stored in a file `ii.tmpl`.

An invocation of MEASURE can be:

```
% measure -in protocol_file_name -out output_file_name -c "{ii.tmpl {m7 eNeRgY}}"
```

It is important to have the same dummy argument (`m7`) in the command for `measure` (`"{ii.tmpl {m7 eNeRgY}}"`) and the template. Otherwise, the connection between the value in the protocol and the user-specified name (`eNeRgY`, in this case) cannot be established.

Therefore, the output file contains lines such as:

```
DOE: eNeRgY 160keV
DOE: eNeRgY_Q_1 120keV
DOE: eNeRgY_Q_2 180.3keV
```

if, for example, the template was matched three times in the given protocol file. The appendix `_Q_` followed by a natural number is used to make the values unequivocal in the case of more than one match of a template in a protocol.

There is one common pitfall in creating a template: Assume the protocol contains the line:

```
tilt2D = 45degree (ion beam, projected to plane.)
```

and the user wants to create a template line to match it. This cannot be performed by using:

```
tilt2D = @<m8>@ (ion beam, projected to plane.)
```

because if `(ion` encounters a Tcl string comparison engine, an error is raised because of unbalanced parentheses. This also occurs with other paired strings or ordering operators such as `'`, `"`, `()`, `{}`, `[]`. The solution is to match `(ion` and `plane.)` with regular expressions:

```
tilt2D = @<m8>@ {^\([a-zA-Z]+} beam, projected to {^[a-zA-Z.]+\)$}
```

To cope with changing layer stacks or a different number of periodic entries in the protocol, MEASURE understands the following metaclause:

```
…some template lines
MEASURE_REPEAT specifier
…some other template lines
MEASURE_END_REPEAT
…some more template lines
```

The *specifier* must appear and can be one of the following: `ZERO_OR_ONE`, `ZERO_OR_MORE`, or `ONE_OR_MORE`.

The meaning is clear: MEASURE expects zero or one, zero or more, or one or more appearances of the template lines in the metaclause to have a match. This even allows some lines of a protocol to be skipped with the help of the `ZERO_OR_ONE` specifier, for example. The metaclauses cannot be mixed up; this means that MEASURE returns an error if a template contains, for example:

```
MEASURE_REPEAT spec1
line1
line2
MEASURE_REPEAT spec2
line3
MEASURE_END_REPEAT
line4
line5
line6
MEASURE_END_REPEAT
```

The metaclauses should appear in the templates in uppercase and with exactly one period in between `MEASURE_REPEAT` and the specifier. This has been done in the expectation that no protocol contains such strings.

The `-r` option is another way to create a useful template. With this option, the protocol(s) is read from the end to the beginning (a new file, with reversed line order, is generated) and the template(s) is reversed blockwise.

For example, this means that a template:

line1	is reversed into	MEASURE_REPEAT *spec2*
line2		*line9*
		line10
MEASURE_REPEAT *spec1*		MEASURE_END_REPEAT
line3		
line4		*line8*
line5		*line7*
MEASURE_END_REPEAT		*line6*
line6		MEASURE_REPEAT *spec1*
line7		*line3*
line8		*line4*
		line5
MEASURE_REPEAT *spec2*		MEASURE_END_REPEAT
line9		
line10		*line2*
MEASURE_END_REPEAT		*line1*

This can be used to find a template for the thickness of the first oxide (if looking from the top) and the thickness of the last oxide.

Consider the following section of the protocol file `lsop_1.log`:

```
###### begin of the region #
warning: lsop_1.dio line:73 ambiguous definition of original area
warning: lsop_1.dio line:73 region fixed:
warning: lsop_1.dio line:73 boundary sort fixed
-----------------------------------------------------------print(layers, x=-2.5)
------------------------------------------- Print ----
lateral position x[um]: -2.5000
y[um]         thickness[nm]        material
2.828            1.8                  OX
2.826            348.9                NI
2.477            798.9                AL
1.678            890.0                OX
0.788            400.0                PO
0.388            289.0                NI
0.099            419.7                OX
-0.321           679.4                SI
-----------------------------------------------------
print(layers, x=-0.4)
------------------------------------------- Print ----
lateral position x[um]: -0.40000
y[um]         thickness[nm]        material
2.828            1.8                  OX
2.826            349.0                NI
2.477            799.0                AL
1.678            890.2                OX
0.788            500.1                AL
0.288            289.0                NI
```

```
-0.001           998.7               SI
-------------------------------------------------------
-------------------------------------------------------
There have been 117 warnings and errors in your job.
All error messages are contained in the protocol file.
```

Let the template be (`firstOx.tmpl`):

```
lateral position x.um.: {^(-)?[0-9]*(\.|,)[0-9]+}
y.um.   thickness.nm.   material
MEASURE_REPEAT ZERO_OR_MORE
@<m3>@     @<m4>@    {SI|PO|NI|AL|LA|MS|ME}
MEASURE_END_REPEAT
@<m2>@     @<m1>@        OX
MEASURE_REPEAT ZERO_OR_MORE
@<m5>@     @<m6>@    {OX|PO|NI|AL|LA|MS|ME}
MEASURE_END_REPEAT
```

The MEASURE command:

```
% measure -in lsop_1.log \
-c "{firstOx.tmpl {m1 first_oxide}}" \
-on {"---------------------------- Print ----"} \
-off {"----------------------------------------"}
```

gives on standard output:

```
Results from lsop_1.log
   with template /usr/k2/users7/krause/work/firstOx.tmpl:
     DOE: first_oxide 1.8

Results from lsop_1.log
   with template /usr/k2/users7/krause/work/firstOx.tmpl:
     DOE: first_oxide_Q_1 1.8
                                  measure finished
```

but the MEASURE command:

```
% measure -in lsop_1.log \
-c "{firstOx.tmpl {m1 last_oxide}}" \
-on {"---------------------------- Print ----"} \
-off {"----------------------------------------"} -r
```

gives:

```
Results from /tmp/lsop_1_reverse.log
with template /usr/k2/users7/krause/work/firstOx.tmpl:
DOE: last_oxide 890.2

Results from /tmp/lsop_1_reverse.log
with template /usr/k2/users7/krause/work/firstOx.tmpl:
DOE: last_oxide_Q_1 419.7
                                  measure finished
```

NOTE The same template was used with the `-r` option, and the 'on strings' are internally interchanged with the 'off strings.' The user should always review the protocol from beginning to end. `/tmp/lsop_1_reverse.log` is the protocol file `lsop_1.log` with reversed line order, created by MEASURE.

9.4 Using MEASURE inside GENESISe

MEASURE is available as a tool in GENESISe. The user should specify (**Edit** > **Input** > **Commands**) a command for MEASURE after selecting MEASURE as a tool in the tool flow. By default, the tool specification for MEASURE contains a run of the INSPECT extractor to copy the values from the output of MEASURE using the `DOE:` label into the GENESISe Family Table.

The user can overwrite the tool specification. To do this, create a `tooldb_$USER` file.

9.5 Using MEASURE inside DIOS

In DIOS, the user can launch a system call to MEASURE. DIOS has a built-in MEASURE command. A simple input file is:

```
title()
subs()
mask(x=0)
print(lay)
measure(template=thick,label("THICKNESS","Thickness"))
etch()
impl()
ld(rs)
measure(template=thres,label("RS","Rs","SURFACE","Surface"))
end
```

The MEASURE command in DIOS mimics a MEASURE command. The last MEASURE call in the DIOS command file translates into the following MEASURE command:

```
{thres.tmpl {RS Rs} {SURFACE Surface}}
```

Where the template `thres.tmpl` can be given as:

```
@<q1>@ @<OXIDE>@ @<bla>@
@<q3>@ @<q4>@ @<bla1>@ @<RS>@ Masetti @<SURFACE>@
```

If MEASURE is launched by a system call inside DIOS, MEASURE writes its output into the actual DIOS protocol file. If control is given back from MEASURE to DIOS, DIOS continues to write into the same protocol file. The tool specification for DIOS contains a run of the INSPECT extractor to copy the values from the DIOS protocol file using the `DOE:` label into the family table.

A full reference of the command in DIOS is given in the DIOS manual.

9.6 Limitations

Equations between the dummy arguments are not implemented. Therefore, for example, MEASURE can be used to extract numbers from a two-column table, but cannot find the values in the first column at which the second column has a relative extremum, and so on.

MEASURE performs no computations, only search operations.

Currently, there is no template creator or template checker.